BeamNGpy

BeamNG GmbH

Dec 21, 2023

CONTENTS:

CHAPTER

ONE

BEAMNGPY

BeamNGpy is an official library providing a Python API to BeamNG.tech, the academia- and industry-oriented fork of the video game BeamNG.drive. BeamNGpy and BeamNG.tech are designed to go hand in hand, both being kept up to date to support each other's functions, meaning using the latest versions of both is recommended.

It allows remote control of the simulation, including vehicles contained in it. See *Features* or go through the Feature Overview Jupyter notebook.

1.1 Table of Contents

1.2 Features

BeamNGpy comes with a wide range of low-level functions to interact with the simulation and a few higher-level interfaces that make more complex actions easier. Some features to highlight are:

1.2.1 Remote Control of Vehicles

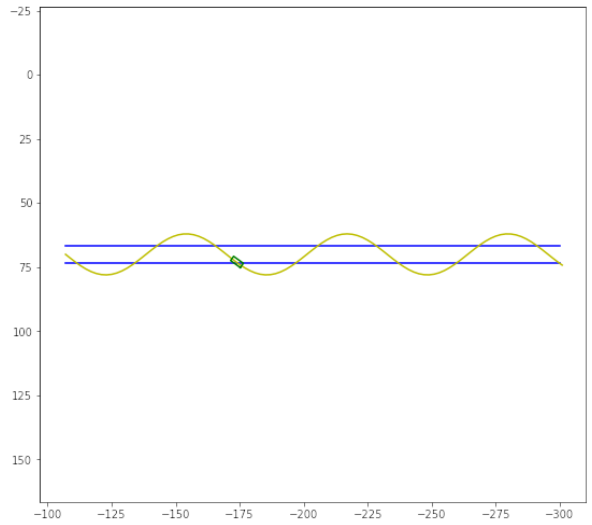
Each vehicle can be controlled individually and independently during the simulation. This includes basic steering inputs, but also controls over various lights (headlights, indicators, etc.) or gear shifting.

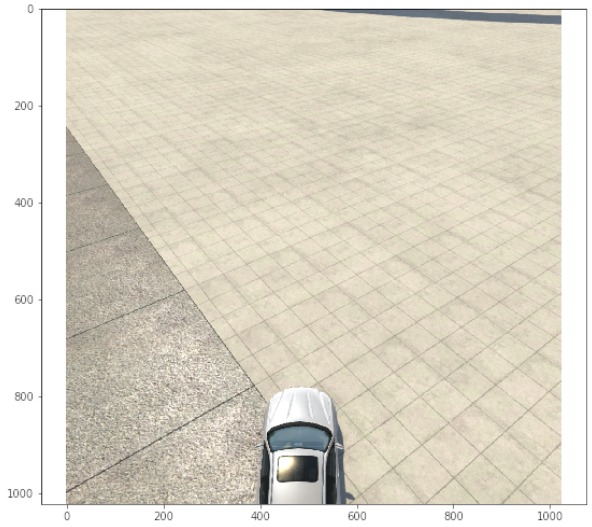
Throttle Control.webm

Steering Control.webm

1.2.2 AI-controlled Vehicles

Besides manual control, BeamNG.tech ships with its own AI to control vehicles. This AI can be configured and controlled from BeamNGpy. It can be used to make a vehicle drive to a certain waypoint, make it follow another vehicle, span the map, or follow a user-defined trajectory:

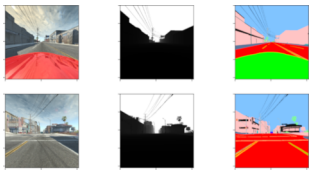




1.2.3 Dynamic Sensor Models

Vehicles and the environment can be equipped with various sensors that provide simulated sensor data. These sensors include:

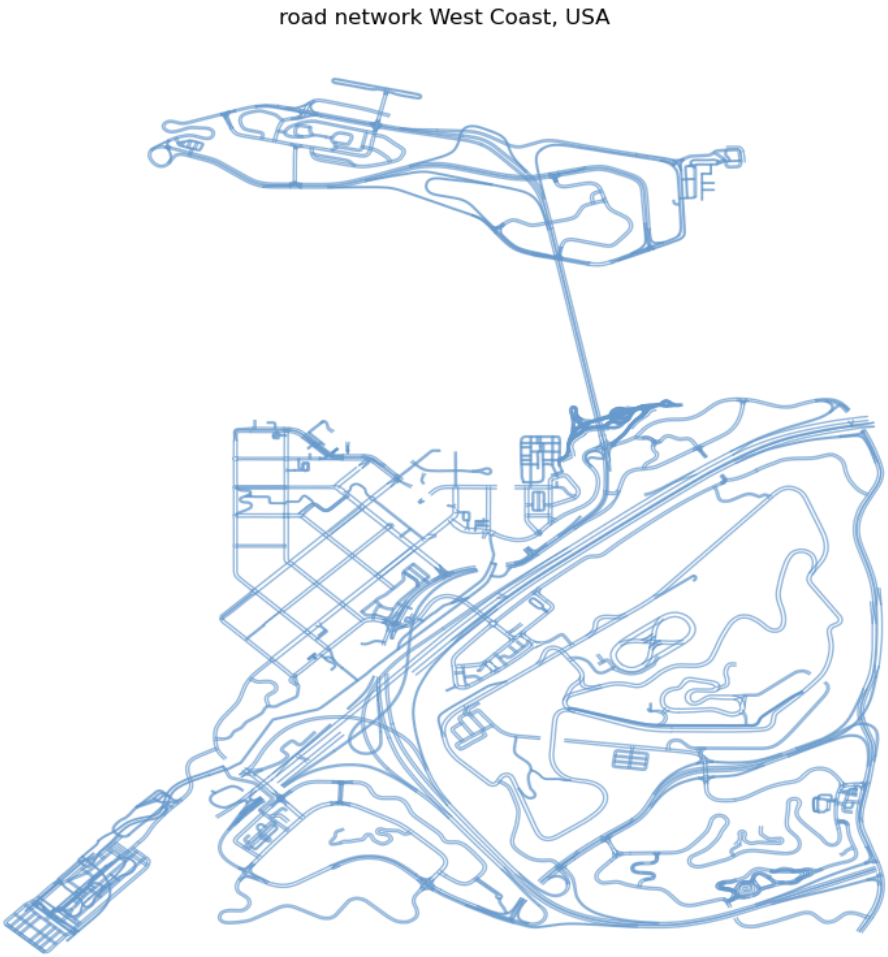
- Cameras
  - Color camera
  - Depth camera
  - Semantic and Instance annotations
- Lidars
- Inertial Measurement Units
- Ultrasonic Distance Measurements



These sensors give perfect data from the simulation by default. Therefore, some of them, like the camera and lidar sensor, can be equipped to also simulate noisy data.

1.2.4 Access to Road Network & Scenario Objects

Geometry of roads in the currently-loaded level/scenario are made available via BeamNGpy. Objects and vehicles that are currently active in the scene are also exposed, allowing for analysis of the current simulation state.



1.2.5 Multiple Clients

BeamNGpy interacts with BeamNG.tech as the client, with BeamNG.tech acting as the server. This allows for multiple BeamNGpy processes to connect to a running simulation and have each control the simulator, making it possible to, for example, run a scenario in which each vehicle is controlled by a separate client.

1.2.6 More

There is a healthy collection of usage examples in the examples/ folder of this repository. These highlight more features, but also serve as documentation, so be sure to check them out.

1.3 Prerequisites

Usage of BeamNGpy requires BeamNG.tech to be installed. For commercial use, contact us at licensing@beamng.gmbh. Builds of BeamNG.tech are made available for research and academic use upon request using this form. Once downloaded, you can use the environment variable `BNG_HOME` to where BeamNG.tech can be run from, or provide a path to the BeamNGpy library during initialization.

1.4 Installation

The library itself is available on PyPI and can therefore be installed using common methods like `pip`:

```
pip install beamngpy
```

If you use Anaconda, you can install BeamNGpy from the `conda-forge` channel by:

```
conda install beamngpy -c conda-forge
```

To upgrade, use

```
pip install --upgrade beamngpy
```

if you installed BeamNGpy using `pip` or

```
conda update beamngpy -c conda-forge --no-pin
```

if you installed it using `conda`.

1.5 Usage

DISCLAIMER: If you are using an older version of `beamngpy` and BeamNG.tech, please follow the instructions of the corresponding `README` file (for example, 1.27.1 instructions). If you are using the latest version of BeamNGpy, continue following the instructions located in this README file.

The library can be imported using `import beamngpy`. A short usage example setting up a scenario with one vehicle in the West Coast USA map that spans the area is:

```
from beamngpy import BeamNGpy, Scenario, Vehicle

# Instantiate BeamNGpy instance running the simulator from the given path,
# communicating over localhost:64256
bng = BeamNGpy('localhost', 64256, home='/path/to/bng/tech', user='/path/to/bng/tech/
↪userfolder')
# Launch BeamNG.tech
bng.open()
# Create a scenario in west_coast_usa called 'example'
scenario = Scenario('west_coast_usa', 'example')
# Create an ETK800 with the licence plate 'PYTHON'
vehicle = Vehicle('ego_vehicle', model='etk800', license='PYTHON')
# Add it to our scenario at this position and rotation
scenario.add_vehicle(vehicle, pos=(-717, 101, 118), rot_quat=(0, 0, 0.3826834, 0.
↪9238795))
# Place files defining our scenario for the simulator to read
scenario.make(bng)

# Load and start our scenario
bng.scenario.load(scenario)
bng.scenario.start()
# Make the vehicle's AI span the map
vehicle.ai.set_mode('span')
input('Hit enter when done...')
```

We have a guide helping you getting started and navigating our collection of examples and the documentation of the library is available here.

1.6 Compatibility

BeamNG.tech is not a finished product but is still under development. Thus frequent changes on the simulation side are to be expected. While the BeamNGpy library maintains compatibility between minor versions for the user, this doesn't extend to the BeamNG.tech side. Not all BeamNGpy versions are compatible with all BeamNG.tech versions. Below is a list of compatible BeamNG.tech and BeamNGpy versions. However we do not maintain minor versions: bug fixes and new features will only be available for the newest BeamNG.tech and BeamNGpy releases.

BeamNG.tech version	BeamNGpy version
0.31	1.28
0.30	1.27.1
0.28, 0.29	1.26.1
0.27	1.25.1
0.26	1.24
0.25	1.23.1
0.24	1.22
0.23	1.21.1
0.22	1.20
0.21	1.19.1

1.7 Troubleshooting

This section lists common issues with BeamNGpy in particular. Since this library is closely tied to BeamNG.tech and thus BeamNG.drive, it is also recommended to consult the documentation on BeamNG.drive here:

https://documentation.beamng.com/

1.7.1 BeamNGpy cannot establish a connection

- Be sure to complete the initial set up step described in the Usage section and to repeat it with every newly released BeamNG.tech version.
- Make sure BeamNG.tech and Python are allowed to connect to your current network in Windows Firewall.

1.7.2 BeamNG.tech quietly fails to launch

- There is a known issue where BeamNG.tech quietly crashes when there is a space in the configured userpath. Until this issue is fixed, it is recommended to either switch to a path that does not contain a space or change the userpath directly in the "startup.ini" file located in the directory of your BeamNG.tech installation.

1.8 Contributions

We always welcome user contributions, be sure to check out our contribution guidelines first, before starting your work.

1.8.1 README

1.8.1.1 BeamNGpy

BeamNGpy is an official library providing a Python API to BeamNG.tech, the academia- and industry-oriented fork of the video game BeamNG.drive. BeamNGpy and BeamNG.tech are designed to go hand in hand, both being kept up to date to support each other's functions, meaning using the latest versions of both is recommended.

It allows remote control of the simulation, including vehicles contained in it. See *Features* or go through the Feature Overview Jupyter notebook.

1.8.1.1.1 Table of Contents

1.8.1.1.2 Features

BeamNGpy comes with a wide range of low-level functions to interact with the simulation and a few higher-level interfaces that make more complex actions easier. Some features to highlight are:

Remote Control of Vehicles

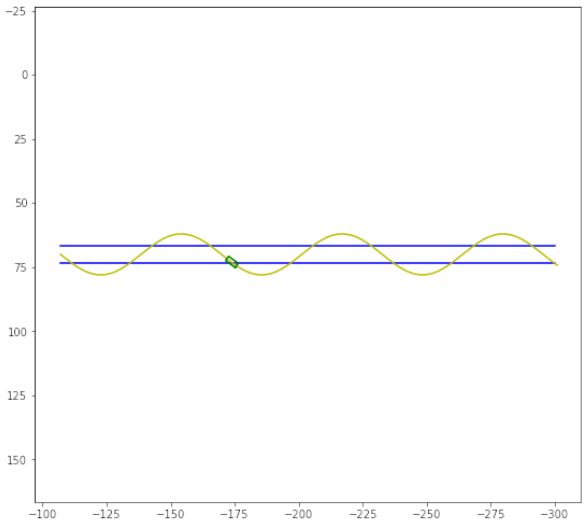
Each vehicle can be controlled individually and independently during the simulation. This includes basic steering inputs, but also controls over various lights (headlights, indicators, etc.) or gear shifting.

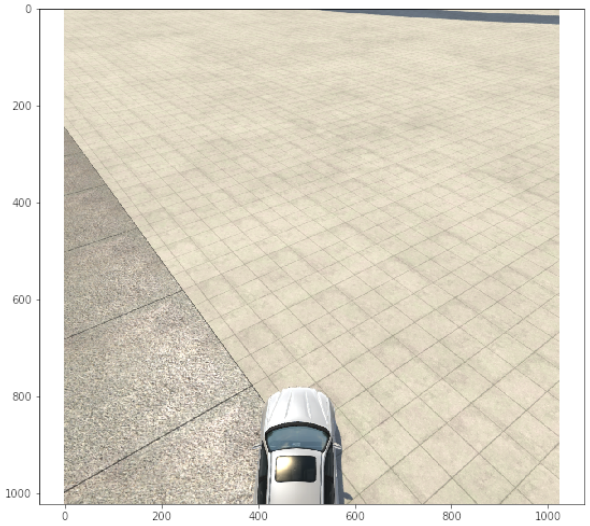
Throttle Control.webm

Steering Control.webm

AI-controlled Vehicles

Besides manual control, BeamNG.tech ships with its own AI to control vehicles. This AI can be configured and controlled from BeamNGpy. It can be used to make a vehicle drive to a certain waypoint, make it follow another vehicle, span the map, or follow a user-defined trajectory:

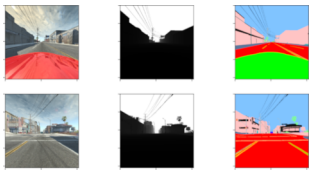




Dynamic Sensor Models

Vehicles and the environment can be equipped with various sensors that provide simulated sensor data. These sensors include:

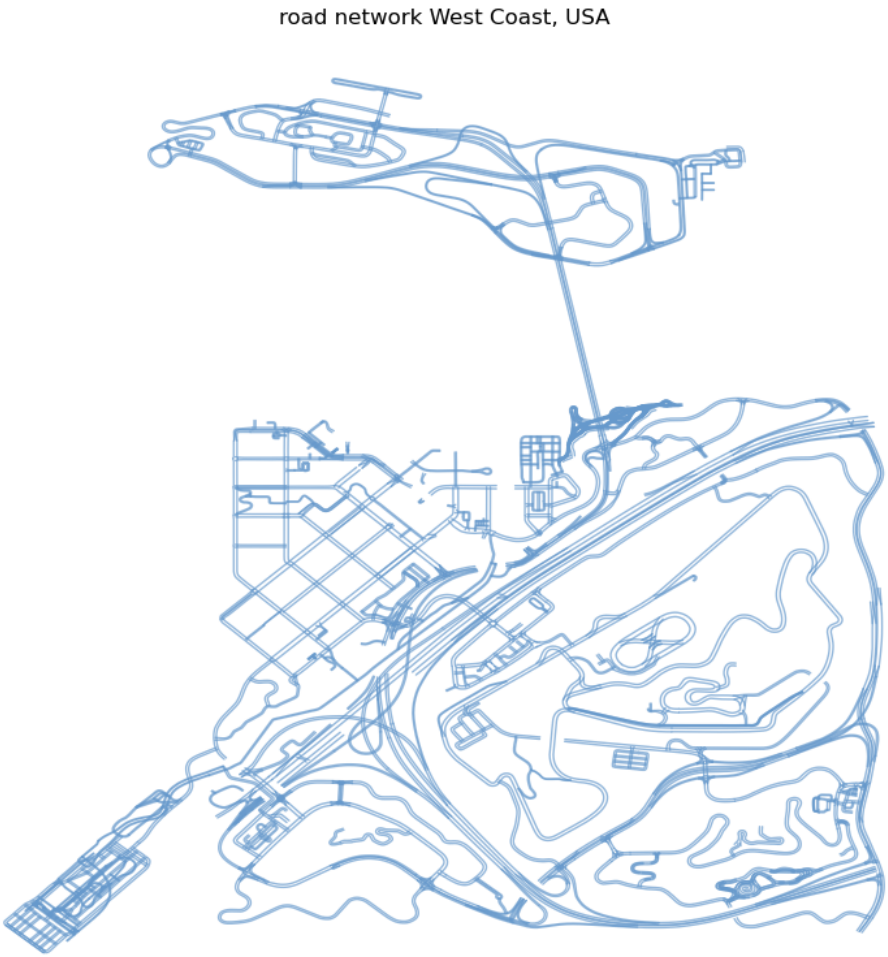
- Cameras
  - Color camera
  - Depth camera
  - Semantic and Instance annotations
- Lidars
- Inertial Measurement Units
- Ultrasonic Distance Measurements



These sensors give perfect data from the simulation by default. Therefore, some of them, like the camera and lidar sensor, can be equipped to also simulate noisy data.

Access to Road Network & Scenario Objects

Geometry of roads in the currently-loaded level/scenario are made available via BeamNGpy. Objects and vehicles that are currently active in the scene are also exposed, allowing for analysis of the current simulation state.



Multiple Clients

BeamNGpy interacts with BeamNG.tech as the client, with BeamNG.tech acting as the server. This allows for multiple BeamNGpy processes to connect to a running simulation and have each control the simulator, making it possible to, for example, run a scenario in which each vehicle is controlled by a separate client.

More

There is a healthy collection of usage examples in the examples/ folder of this repository. These highlight more features, but also serve as documentation, so be sure to check them out.

1.8.1.1.3 Prerequisites

Usage of BeamNGpy requires BeamNG.tech to be installed. For commercial use, contact us at licensing@beamng.gmbh. Builds of BeamNG.tech are made available for research and academic use upon request using this form. Once downloaded, you can use the environment variable `BNG_HOME` to where BeamNG.tech can be run from, or provide a path to the BeamNGpy library during initialization.

1.8.1.1.4 Installation

The library itself is available on PyPI and can therefore be installed using common methods like `pip`:

```
pip install beamngpy
```

If you use Anaconda, you can install BeamNGpy from the `conda-forge` channel by:

```
conda install beamngpy -c conda-forge
```

To upgrade, use

```
pip install --upgrade beamngpy
```

if you installed BeamNGpy using `pip` or

```
conda update beamngpy -c conda-forge --no-pin
```

if you installed it using `conda`.

1.8.1.1.5 Usage

DISCLAIMER: If you are using an older version of `beamngpy` and BeamNG.tech, please follow the instructions of the corresponding `README` file (for example, 1.27.1 instructions). If you are using the latest version of BeamNGpy, continue following the instructions located in this README file.

The library can be imported using `import beamngpy`. A short usage example setting up a scenario with one vehicle in the West Coast USA map that spans the area is:

```
from beamngpy import BeamNGpy, Scenario, Vehicle

# Instantiate BeamNGpy instance running the simulator from the given path,
# communicating over localhost:64256
bng = BeamNGpy('localhost', 64256, home='/path/to/bng/tech', user='/path/to/bng/tech/
↪userfolder')
```

(continues on next page)

(continued from previous page)

```
# Launch BeamNG.tech
bng.open()
# Create a scenario in west_coast_usa called 'example'
scenario = Scenario('west_coast_usa', 'example')
# Create an ETK800 with the licence plate 'PYTHON'
vehicle = Vehicle('ego_vehicle', model='etk800', license='PYTHON')
# Add it to our scenario at this position and rotation
scenario.add_vehicle(vehicle, pos=(-717, 101, 118), rot_quat=(0, 0, 0.3826834, 0.
↪9238795))
# Place files defining our scenario for the simulator to read
scenario.make(bng)

# Load and start our scenario
bng.scenario.load(scenario)
bng.scenario.start()
# Make the vehicle's AI span the map
vehicle.ai.set_mode('span')
input('Hit enter when done...')
```

We have a guide helping you getting started and navigating our collection of examples and the documentation of the library is available here.

1.8.1.1.6 Compatibility

BeamNG.tech is not a finished product but is still under development. Thus frequent changes on the simulation side are to be expected. While the BeamNGpy library maintains compatibility between minor versions for the user, this doesn't extend to the BeamNG.tech side. Not all BeamNGpy versions are compatible with all BeamNG.tech versions. Below is a list of compatible BeamNG.tech and BeamNGpy versions. However we do not maintain minor versions: bug fixes and new features will only be available for the newest BeamNG.tech and BeamNGpy releases.

BeamNG.tech version	BeamNGpy version
0.31	1.28
0.30	1.27.1
0.28, 0.29	1.26.1
0.27	1.25.1
0.26	1.24
0.25	1.23.1
0.24	1.22
0.23	1.21.1
0.22	1.20
0.21	1.19.1

1.8.1.1.7 Troubleshooting

This section lists common issues with BeamNGpy in particular. Since this library is closely tied to BeamNG.tech and thus BeamNG.drive, it is also recommended to consult the documentation on BeamNG.drive here:

https://documentation.beamng.com/

BeamNGpy cannot establish a connection

- Be sure to complete the initial set up step described in the Usage section and to repeat it with every newly released BeamNG.tech version.
- Make sure BeamNG.tech and Python are allowed to connect to your current network in Windows Firewall.

BeamNG.tech quietly fails to launch

- There is a known issue where BeamNG.tech quietly crashes when there is a space in the configured userpath. Until this issue is fixed, it is recommended to either switch to a path that does not contain a space or change the userpath directly in the "startup.ini" file located in the directory of your BeamNG.tech installation.

1.8.1.1.8 Contributions

We always welcome user contributions, be sure to check out our contribution guidelines first, before starting your work.

1.8.2 BeamNGpy Reference

1.8.2.1 BeamNGpy

class beamngpy.**BeamNGpy**(*host: str*, *port: int*, *home: str | None = None*, *binary: str | None = None*, *user: str | None = None*, *quit_on_close: bool = True*, *crash_lua_on_error: bool | None = None*)

The *BeamNGpy* class is the backbone of communication with the BeamNG simulation and offers methods of starting, stopping, connecting to, and controlling the state of the simulator.

Instantiates a BeamNGpy instance connecting to the simulator on the given host and port. The home directory of the simulator can be passed to this constructor. If `None` is given, this class tries to read a home path from the `BNG_HOME` environment variable.

Parameters

- **host** (*str*) – The host to connect to.
- **port** (*int*) – The port to connect to.
- **home** (*str | None*) – Path to the simulator's home directory.
- **binary** (*str | None*) – Optional custom path to the binary, relative to the simulator's home directory. Default is `Bin64/BeamNG.{tech/drive}.x64.exe` for Windows hosts, `BinLinux/BeamNG.{tech/drive}.x64` for Linux hosts.
- **user** (*str | None*) – Additional optional user path to set. This path can be used to set where custom files created during executions will be placed if the home folder shall not be touched.

- **quit_on_close** (*bool*) – Whether the simulator should be closed when *close()* is called. Defaults to True.
- **crash_lua_on_error** (*bool | None*) – If True, then sets BeamNG to not respond to BeamNGpy requests when a Lua error happens and prints the stacktrace instead. Is applicable only when the process is launched by this instance of BeamNGpy, as it sets a launch argument of the process. Defaults to False.

camera

The API module to control the camera in the simulator. See *CameraApi* for details.

Type
CameraApi

control

The API module to control the flow of the simulation. See *ControlApi* for details.

Type
ControlApi

debug

The API module to control debug objects. See *DebugApi* for details.

Type
DebugApi

env

The API module to control the simulation's environment. See *EnvironmentApi* for details.

Type
EnvironmentApi

scenario

The API module to control the scenarios. See *ScenarioApi* for details.

Type
ScenarioApi

settings

The API module to control the settings of the simulator. See *SettingsApi* for details.

Type
SettingsApi

system

The API module for getting information about the host system. See *SystemApi* for details.

Type
SystemApi

traffic

The API module to control the traffic. See *TrafficApi* for details.

Type
TrafficApi

vehicles

The API module to control the vehicles in the scenario. See *VehiclesApi* for details.

Type
VehiclesApi

`close`() → None

Disconnects from the simulator and kills the BeamNG.* process.

Return type
None

`disconnect`() → None

Disconnects from the BeamNG simulator.

Return type
None

`host_os`() → str | None

The operating system of the host the simulator is running on.

Return type
str | None

`open`(*extensions: Optional[List[str]] = None*, **args: str*, *launch: bool = True*, *crash_lua_on_error: bool | None = None*, *listen_ip: str = '127.0.0.1'*, ***opts: str*) → *BeamNGpy*

Starts a BeamNG.* process, opens a server socket, and waits for the spawned BeamNG.* process to connect. This method blocks until the process started and is ready.

Parameters

- **extensions** (*Optional[List[str]]*) – A list of non-default BeamNG Lua extensions to be loaded on start.
- **launch** (*bool*) – Whether to launch a new process or connect to a running one on the configured host/port. Defaults to True.
- **crash_lua_on_error** (*bool | None*) – If True, then sets BeamNG to not respond to BeamNGpy requests when a Lua error happens and prints the stacktrace instead. Is applicable only when the process is launched by this instance of BeamNGpy, as it sets a launch argument of the process. Defaults to False.
- **listen_ip** (*str*) – The IP address that the BeamNG process will be listening on. Only relevant when `launch` is True. Set to * if you want BeamNG to listen on ALL network interfaces.
- **args** (*str*) –
- **opts** (*str*) –

Return type
BeamNGpy

`tech_enabled`() → bool | None

A flag that specifies whether a BeamNG.tech features are enabled or not.

Return type
bool | None

1.8.2.1.1 API

class beamngpy.api.beamng.**Api**(*beamng:* BeamNGpy)

Bases: object

A base API class from which all the API communicating with the simulator derive.

Parameters
beamng (BeamNGpy) – An instance of the simulator.

class beamngpy.api.beamng.**CameraApi**(*beamng:* BeamNGpy)

Bases: *Api*

An API class which allows control of the in-game camera and also provides information about the semantic annotation classes.

Parameters
beamng (BeamNGpy) – An instance of the simulator.

get_annotation_classes(*annotations: Dict[str, Int3]*) → Dict[int, str]

Method to convert the annotation configuration of the simulator into a mapping of colors to the corresponding object classes.

Parameters
annotations (*Dict[str, Int3]*) – The annotation configuration of the simulator. Expected to be in the format *get_annotations()* returns.

Returns
A mapping of colors encoded as 24bit integers to object classes according to the simulator.

Return type
Dict[int, str]

get_annotations() → Dict[str, Int3]

Method to obtain the annotation configuration of the simulator.

Returns
A mapping of object classes to lists containing the [R, G, B] values of the colors objects of that class are rendered with.

Return type
Dict[str, Int3]

get_player_modes(*vehicle: str | Vehicle*) → StrDict

Retrieves information about the camera modes configured for the vehicle identified by the given ID.

Parameters
vehicle (*str | Vehicle*) – Vehicle ID of the vehicle to get camera mode information of.

Returns
A dictionary mapping camera mode names to configuration options.

Return type
StrDict

set_free(*pos: Float3*, *direction: Float3*) → None

Sets the position and direction of the free camera. The free camera is one that does not follow any particular vehicle, but can instead be put at any spot and any position on the map.

Parameters

- **pos** (*Float3*) – The position of the camera as a (x, y, z) triplet.

- **direction** (*Float3*) – The directional vector of the camera as a (x, y, z) triplet.

Return type
None

set_player_mode(*vehicle: str | Vehicle*, *mode: str*, *config: StrDict*, *custom_data: StrDict | None = None*) → None

Sets the camera mode of the vehicle identified by the given vehicle ID. The mode is given as a string that identifies one of the valid modes offered by the simulator. These modes can be queried using the *get_player_modes()* method.

The camera can be further configured with some common parameters, but it is not guaranteed the camera mode will respect all of them. These parameters include:

- `rotation`: The rotation of the camera as a triplet of Euler angles
- `fov`: The field of view angle
- `offset`: The (x, y, z) vector to offset the camera's position by
- `distance`: The distance of the camera to the vehicle

Since each camera mode is implemented as a custom Lua extension, it is not possible to automatically query the exact features of the mode. Further information can be found in the `lua/ge/extensions/core/cameraModes` files which contain the implementations of each camera mode.

Parameters

- **vehicle** (*str | Vehicle*) – Vehicle ID of the vehicle to change the mode of.
- **mode** (*str*) – Camera mode to set.
- **config** (*StrDict*) – Dictionary of further properties to set in the mode.
- **custom_data** (*StrDict | None*) – Custom data used by the specific camera mode. Defaults to None.

Return type
None

set_relative(*pos: Float3*, *dir: Float3*, *up: Float3 = (0.0, 0.0, 1.0)*) → None

Switches the camera mode for the currently-entered vehicle to the 'relative' mode in which the camera can be placed at an arbitrary point relative to the vehicle, moving along with it as it drives around.

Parameters

- **pos** (*Float3*) – (x, y, z) tuple of the camera's position relative to the vehicle.
- **dir** (*x, y, z*) – The cameras direction vector.
- **up** (*x, y, z*) – The camera up vector (optional).

Return type
None

class beamngpy.api.beamng.**ControlApi**(*beamng: BeamNGpy*)

Bases: *Api*

An API allowing control of the flow of the simulation - pausing/resuming, stepping, and also enabling support for calling custom Lua code.

Parameters
beamng (*BeamNGpy*) – An instance of the simulator.

get_gamestate() → Dict[str, str]

Retrieves the current game state of the simulator. The game state is returned as a dictionary containing a `state` entry that is either:

- `scenario` when a scenario is loaded
- `menu` otherwise

If a scenario is loaded, the resulting dictionary also contains a `scenario_state` entry whose value is `pre-running` if the scenario is currently at the start screen or `running` otherwise.

Returns
The game state as a dictionary as described above.

Return type
Dict[str, str]

pause() → None

Sends a pause request to BeamNG.*, blocking until the simulation is paused.

Return type
None

queue_lua_command(*chunk: str*) → None

Executes one lua chunk in the game engine VM.

Parameters
chunk (*str*) – lua chunk as a string

Return type
None

quit_beamng() → None

Sends the quit request to the simulator, which also closes the process.

Return type
None

resume() → None

Sends a resume request to BeamNG.*, blocking until the simulation is resumed.

Return type
None

return_to_main_menu() → None

Returns to the main menu, possibly closing the loaded scenario.

Return type
None

step(*count: int*, *wait: bool = True*) → None

Advances the simulation the given amount of steps, assuming it is currently paused. If the wait flag is set, this method blocks until the simulator has finished simulating the desired amount of steps. If not, this method resumes immediately. This can be used to queue commands that should be executed right after the steps have been simulated.

Parameters

- **count** (*int*) – The amount of steps to simulate.
- **wait** (*bool*) – Optional. Whether to wait for the steps to be simulated. Defaults to True.

Raises
BNGError – If the wait flag is set but the simulator doesn't respond appropriately.

Return type
None

class beamngpy.api.beamng.**DebugApi**(*beamng:* BeamNGpy)

Bases: *Api*

An API for drawing debug graphical objects in the simulator.

Parameters
beamng (BeamNGpy) – An instance of the simulator.

add_cylinder(*circle_positions: List[Float3]*, *radius: float*, *rgba_color: Color*) → int

Adds graphical debug cylinder to the simulator with bases at positions specified by the **circle_positions** argument.

Parameters

- **circle_positions** (*List[Float3]*) – List of two `(x, y, z)` coordinates of the circle centers.
- **radius** (*float*) – The radius of the cylinder.
- **rgba_color** (*Color*) – A single color of the points of the debug cylinder, in the format of `(R, G, B, A)`. An `A` of 1.0 means full visibility, 0.0 means full transparency. Can also be instance of any type that the *coerce_color()* function accepts.

Returns
An integer ID of the debug cylinder added. This ID can be passed to the *remove_cylinder()* function.

Return type
int

add_polyline(*coordinates: List[Float3]*, *rgba_color: Color*, *cling: bool = False*, *offset: float = 0.0*) → int

Adds graphical debug polyline to the simulator with points at positions specified by the **coordinates** argument.

The arguments **coordinates**, **radii** and **rgba_colors** have to have the same length, which is the number of the debug spheres added.

Parameters

- **coordinates** (*List[Float3]*) – List of `(x, y, z)` coordinates of the debug spheres.
- **rgba_color** (*Color*) – A single color of the points of the debug polyline, in the format of `(R, G, B, A)`. An `A` of 1.0 means full visibility, 0.0 means full transparency. Can also be instance of any type that the *coerce_color()* function accepts.
- **cling** (*bool*) – Whether or not to align the `z` coordinate of the spheres to the ground.
- **offset** (*float*) – The z-axis offset of the sphere coordinates. Can only be used together with `cling=True` to spawn spheres an exact amount above the ground.

Returns
An integer ID of the debug polyline added. This ID can be passed to the *remove_polyline()* function.

Return type
int

add_rectangle(*vertices: List[Float3]*, *rgba_color: Color*, *cling: bool = False*, *offset: float = 0.0*) → int

Adds graphical debug rectangle to the simulator with points at positions specified by the **vertices** argument.

Parameters

- **vertices** (*List[Float3]*) – List of four `(x, y, z)` coordinates of the rectangle points.
- **rgba_color** (*Color*) – A single color of the points of the debug rectangle, in the format of `(R, G, B, A)`. An `A` of 1.0 means full visibility, 0.0 means full transparency. Can also be instance of any type that the *coerce_color()* function accepts.
- **cling** (*bool*) – Whether or not to align the `z` coordinate of the rectangle points to the ground.
- **offset** (*float*) – The z-axis offset of the rectangle coordinates. Can only be used together with `cling=True` to spawn rectangle an exact amount above the ground.

Returns

An integer ID of the debug rectangle added. This ID can be passed to the *remove_rectangle()* function.

Return type

int

add_spheres(*coordinates: List[Float3]*, *radii: List[float]*, *rgba_colors: Union[List[Color], Color]*, *cling: bool = False*, *offset: float = 0.0*) → List[int]

Adds graphical debug spheres to the simulator at positions specified by the **coordinates** argument.

The arguments **coordinates**, **radii** and **rgba_colors** have to have the same length, which is the number of the debug spheres added.

Parameters

- **coordinates** (*List[Float3]*) – List of `(x, y, z)` coordinates of the debug spheres.
- **radii** (*List[float]*) – List of radii of the debug spheres in meters.
- **rgba_colors** (*Union[List[Color], Color]*) – Either a single color or list of colors of the debug spheres, in the format of `(R, G, B, A)`. An `A` of 1.0 means full visibility, 0.0 means full transparency. Can also be instances of any type that the *coerce_color()* function accepts.
- **cling** (*bool*) – Whether or not to align the `z` coordinate of the spheres to the ground.
- **offset** (*float*) – The z-axis offset of the sphere coordinates. Can only be used together with `cling=True` to spawn spheres an exact amount above the ground.

Returns

List of string IDs of the debug spheres added. This list can be passed to the *remove_spheres()* function.

Return type

List[int]

add_square_prism(*end_points: List[Float3]*, *end_point_dims: List[Float2]*, *rgba_color: Color*) → int

Adds graphical debug square prism to the simulator with the base squares at positions specified by the **end_points** argument.

Parameters

- **end_points** (*List[Float3]*) – List of two `(x, y, z)` coordinates of the square prism end points.

- **end_point_dims** (*List[Float2]*) – List of two `(width, height)` dimensions of the square prism end points.
- **rgba_color** (*Color*) – A single color of the points of the debug square prism, in the format of `(R, G, B, A)`. An `A` of 1.0 means full visibility, 0.0 means full transparency. Can also be instance of any type that the *coerce_color()* function accepts.

Returns
An integer ID of the debug square prism added. This ID can be passed to the *remove_square_prism()* function.

Return type
int

add_text(*origin: Float3*, *content: str*, *rgba_color: Color*, *cling: bool = False*, *offset: float = 0.0*) → int

Adds graphical debug text to the simulator at the position specified by the **origin** argument.

Parameters

- **origin** (*Float3*) – The position of the text as an `(x, y, z)` coordinate.
- **content** (*str*) – The text that is going to be displayed.
- **rgba_color** (*Color*) – A single color of the text, in the format of `(R, G, B, A)`. An `A` of 1.0 means full visibility, 0.0 means full transparency. Can also be instance of any type that the *coerce_color()* function accepts.
- **cling** (*bool*) – Whether or not to align the `z` coordinate of the text to the ground.
- **offset** (*float*) – The z-axis offset of the text origin. Can only be used together with `cling=True` to spawn the text an exact amount above the ground.

Returns
An integer ID of the text added. This ID can be passed to the *remove_text()* function.

Return type
int

add_triangle(*vertices: List[Float3]*, *rgba_color: Color*, *cling: bool = False*, *offset: float = 0.0*) → int

Adds graphical debug triangle to the simulator with points at positions specified by the **vertices** argument.

Parameters

- **vertices** (*List[Float3]*) – List of three `(x, y, z)` coordinates of the triangle points.
- **rgba_color** (*Color*) – A single color of the points of the debug triangle, in the format of `(R, G, B, A)`. An `A` of 1.0 means full visibility, 0.0 means full transparency. Can also be instance of any type that the *coerce_color()* function accepts.
- **cling** (*bool*) – Whether or not to align the `z` coordinate of the triangle points to the ground.
- **offset** (*float*) – The z-axis offset of the triangle coordinates. Can only be used together with `cling=True` to spawn triangle an exact amount above the ground.

Returns
An integer ID of the debug triangle added. This ID can be passed to the *remove_triangle()* function.

Return type
int

remove_cylinder(*cylinder_id: int*) → None

Removes the cylinder with the ID provided in the **cylinder_id** argument.

Parameters
cylinder_id (*int*) – An integer ID of the cylinder to be deleted.

Return type
None

remove_polyline(*line_id: int*) → None

Removes the polyline with the ID provided in the **line_id** argument.

Parameters
line_id (*int*) – An integer ID of the polyline to be deleted.

Return type
None

remove_rectangle(*rectangle_id: int*) → None

Removes the rectangle with the ID provided in the **rectangle_id** argument.

Parameters
rectangle_id (*int*) – An integer ID of the rectangle to be deleted.

Return type
None

remove_spheres(*sphere_ids: List[int]*) → None

Removes the spheres with the IDs provided in the **sphere_ids** argument.

Parameters
sphere_ids (*List[int]*) – A list of the integer IDs of the spheres to be deleted.

Return type
None

remove_square_prism(*prism_id: int*) → None

Removes the square prism with the ID provided in the **prism_id** argument.

Parameters
prism_id (*int*) – An integer ID of the prism to be deleted.

Return type
None

remove_text(*text_id: int*) → None

Removes the text with the ID provided in the **text_id** argument.

Parameters
text_id (*int*) – An integer ID of the text to be deleted.

Return type
None

remove_triangle(*triangle_id: int*) → None

Removes the triangle with the ID provided in the **triangle_id** argument.

Parameters
triangle_id (*int*) – An integer ID of the triangle to be deleted.

Return type
None

class beamngpy.api.beamng.**EnvironmentApi**(*beamng:* BeamNGpy)

Bases: *Api*

An API allowing control of the in-game environment variables, such as time, weather or gravity.

Parameters
beamng (BeamNGpy) – An instance of the simulator.

get_gravity() → float

Gets the strength of gravity in the simulator.

Returns
The gravity value of the simulator.

Return type
float

get_tod() → StrDict

Gets the current 'time of day' object. That is a dictionary with the following keys:

- `time`: Time of day on a scale from 0 to 1. 0/1 is midday, 0.5 is midnight.
- `timeStr`: Time of day as a string in the format 'HH:MM:SS'.
- `nightScale`: How fast should the night be.
- `dayScale`: How fast should the day be.
- `azimuthOverride`: Used to specify an azimuth that will stay constant throughout the day cycle.
- `startTime`: Time of day when the scenario started.
- `dayLength`: Length of the day (24 hours).

Returns
The dictionary with keys specified above.

Return type
StrDict

set_gravity(*gravity: float = -9.807*) → None

Sets the strength of gravity in the simulator.

Parameters
gravity (*float*) – The gravity value to set. The default one is that of earth (-9.807).

Return type
None

set_tod(*tod: Optional[Union[float, str]] = None*, *play: bool | None = None*, *day_scale: float | None = None*, *night_scale: float | None = None*, *day_length: float | None = None*, *azimuth_override: float | None = None*) → None

Sets the current time of day. The time of day value is given as a float between 0 and 1. How this value affects the lighting of the scene is dependant on the map's TimeOfDay object.

Parameters

- **tod** (*Optional[Union[float, str]]*) – Time of day. Can be provided as a float between 0.0 and 1.0, or as a string in the format 'HH:MM:SS'.
- **play** (*bool | None*) – False by default.
- **day_scale** (*float | None*) – How fast should the day be.

- **night_scale** (*float | None*) – How fast should the night be.
- **day_length** (*float | None*) – Length of the day (24 hours).
- **azimuth_override** (*float | None*) – Used to specify an azimuth that will stay constant throughout the day cycle.

Return type
None

set_weather_preset(*preset: str*, *time: float = 1*) → None

Triggers a change to a different weather preset. Weather presets affect multiple settings at once (time of day, wind speed, cloud coverage, etc.) and need to have been defined first. Example json objects defining weather presets can be found in BeamNG.tech's `art/weather/defaults.json` file.

Parameters

- **preset** (*str*) – The name of the preset to switch to. Needs to be defined already within the simulation.
- **time** (*float*) – Time in seconds the transition from the current settings to the preset's should take.

Return type
None

class beamngpy.api.beamng.**GEVehiclesApi**(*beamng:* BeamNGpy, *vehicle:* Vehicle)

Bases: *Api*

A vehicle API that needs a connected `BeamNGpy` instance. It is exposed at the root level (directly accessible from the `Vehicle` object).

Parameters

- **beamng** (BeamNGpy) –
- **vehicle** (Vehicle) –

annotate_parts() → None

Return type
None

get_bbox() → Dict[str, Float3]

Return type
Dict[str, Float3]

get_part_config() → StrDict

Return type
StrDict

get_part_options() → StrDict

Return type
StrDict

revert_annotations() → None

Return type
None

set_license_plate(*text: str*) → None

Parameters
text (*str*) –

Return type
None

set_part_config(*cfg: StrDict*) → None

Parameters
cfg (*StrDict*) –

Return type
None

switch()

teleport(*pos: Float3*, *rot_quat: Quat | None = None*, *reset: bool = True*) → bool

Parameters

- **pos** (*Float3*) –
- **rot_quat** (*Quat | None*) –
- **reset** (*bool*) –

Return type
bool

class beamngpy.api.beamng.**ScenarioApi**(*beamng:* BeamNGpy)

Bases: *Api*

An API gathering function for working with scenarios, levels and scenario objects.

Parameters
beamng (BeamNGpy) – An instance of the simulator.

find_objects_class(*clazz: str*) → List[*ScenarioObject*]

Scans the current environment in the simulator for objects of a certain class and returns them as a list of *ScenarioObject*.

What kind of classes correspond to what kind of objects is described in the BeamNG.drive documentation.

Parameters
clazz (*str*) – The class name of objects to find.

Returns
Found objects as a list.

Return type
List[ScenarioObject]

get_current(*connect: bool = True*) → *Scenario*

Queries the currently loaded scenario from the simulator.

Parameters
connect (*bool*) – Whether to connect the returned scenario and the currently loaded vehicles to BeamNGpy. Defaults to True. If set to False, you can still manually connect the returned scenario by running `Scenario.connect()`.

Returns
A *Scenario* instance of the currently-loaded scenario. The scenario's parent level field will be filled in accordingly.

Return type
Scenario

`get_level_scenarios`(*level: str | beamngpy.scenario.level.Level*) → List[*Scenario*]

Queries the simulator for all scenarios available in the given level.

Parameters
level (*str | beamngpy.scenario.level.Level*) – The level to get scenarios for. Can either be the name of the level as a string or an instance of *Level*.

Returns
A list of *Scenario* instances.

Return type
List[Scenario]

`get_levels`() → Dict[str, *Level*]

Queries the available levels in the simulator and returns them as a mapping of level name to *Level* instances.

Returns
A dictionary of available level names to a corresponding instance of the *Level* class.

Return type
Dict[str, Level]

`get_levels_and_scenarios`() → Tuple[Dict[str, *Level*], Dict[str, List[*Scenario*]]]

Utility method that retrieves all levels and scenarios and returns them as a tuple of (levels, scenarios).

Returns
(`get_levels()`, `get_scenarios()`)

Return type
Tuple[*Dict*[str, Level], *Dict*[str, *List*[Scenario]]]

`get_name`() → str

Retrieves the name of the currently-loaded scenario in the simulator.

Returns
The name of the loaded scenario as a string.

Return type
str

`get_road_edges`(*road: str*) → List[Dict[str, Dict[str, Float3]]]

Retrieves the edges of the road with the given name and returns them as a list of point triplets. Roads are defined by a series of lines that specify the leftmost, center, and rightmost point in the road. These lines go horizontally across the road and the series of leftmost points make up the left edge of the road, the series of rightmost points make up the right edge of the road, and the series of center points the middle line of the road.

Parameters
road (*str*) – Name of the road to get edges from.

Returns
The road edges as a list of dictionaries with (`left`, `middle`, `right`) points. Each point is an `(X, Y, Z)` coordinate triplet.

Return type
List[*Dict*[str, *Dict*[str, Float3]]]

get_roads() → StrDict

Retrieves the metadata of all DecalRoads in the current scenario. The metadata of a DecalRoad is formatted as a dictionary with the following keys:

Returns
A dict mapping DecalRoad IDs to their metadata..

Return type
StrDict

get_scenarios(*levels: Optional[Iterable[str | beamngpy.scenario.level.Level]] = None*) → Dict[str, List[*Scenario*]]

Queries the available scenarios and returns them as a mapping of paths to *Scenario* instances. The scenarios are constructed to point to their parent levels, so to avoid extra queries to the simulator about existing levels, a cache of available levels can be passed to this method. If a partial list of levels is supplied, then only scenarios for these levels will be queried and returned.

Parameters
levels (*Optional[Iterable[str | beamngpy.scenario.level.Level]]*) – A list of level names or *Level* instances to get scenarios for. If None, scenarios from all levels will be returned.

Returns
A mapping of level names to lists of *Scenario* instances.

Return type
Dict[str, *List*[Scenario]]

get_vehicle(*vehicle_id: str*) → *Vehicle* | None

Retrieves the vehicle with the given ID from the currently loaded scenario.

Parameters
vehicle_id (*str*) – The ID of the vehicle to find.

Returns
The *Vehicle* with the given ID. None if it wasn't found.

Return type
Vehicle | None

load(*scenario:* Scenario, *precompile_shaders: bool = True*, *connect_player_vehicle: bool = True*, *connect_existing_vehicles: bool = True*) → None

Loads the given scenario in the simulation and returns once loading is finished.

Parameters

- **scenario** (Scenario) – The scenario to load.
- **precompile_shaders** (*bool*) – Whether the shaders should be compiled before the start of the scenario. If False, the first load of a map will take a longer time, but disabling the precompilation can lead to issues with the Camera sensor. Defaults to True.
- **connect_player_vehicle** (*bool*) – Whether the player vehicle should be connected to this (:class:.Scenario) instance. Defaults to True.
- **connect_existing_vehicles** (*bool*) – Whether ALL vehicles spawned already in the scenario should be connected to this (:class:.Scenario) instance. Defaults to True.

Return type
None

`load_trackbuilder_track`(*path: str*)

Spawns a TrackBuilder track provided by the given path to a TrackBuilder `.json` file.

Parameters
path (*str*) – Path to a `.json` file created by TrackBuilder.

`restart`() → None

Restarts a running scenario.

Return type
None

`start`(*restrict_actions: bool = False*) → None

Starts the scenario; equivalent to clicking the "Start" button in the game after loading a scenario. This method blocks until the countdown to the scenario's start has finished.

Parameters
restrict_actions (*bool*) – Whether to keep scenario restrictions, such as limited menu options and controls. Defaults to False.

Return type
None

`stop`() → None

Stops a running scenario and returns to the main menu.

Return type
None

`teleport_object`(*scenario_object:* ScenarioObject, *pos: Float3*, *rot_quat: Quat | None = None*) → None

Teleports the given scenario object to the given position with the given rotation.

Parameters

- **scenario_object** (ScenarioObject) – The vehicle to teleport.
- **pos** (*Float3*) – The target position as an (x,y,z) tuple containing world-space coordinates.
- **rot_quat** (*Quat | None*) – Optional tuple specifying object rotation as a quaternion.

Return type
None

`class` beamngpy.api.beamng.`SettingsApi`(*beamng:* BeamNGpy)

Bases: *Api*

An API for changing the simulator settings.

Parameters
beamng (BeamNGpy) – An instance of the simulator.

`apply_graphics`() → None

Makes the game apply a graphics setting that has been changed since startup or the last time settings were applied. A call to this is required after changing settings like whether or not the game is in fullscreen or the resolution, otherwise those settings will only take effect after the next launch.

Return type
None

change(*key: str*, *value: str*) → None

Changes a setting in the game. Examples of the key and value pairs given to this method can be found in your game's settings ini files. These are usually in <userpath>/settings/game-settings.ini or <userpath>/settings/cloud/game-settings-cloud.ini.

Parameters

- **key** (*str*) – The key of the setting that is to be changed
- **value** (*str*) – The desired value.

Return type
None

remove_step_limit() → None

Removes the steps-per-second setting, making the simulation run at undefined time slices.

Return type
None

set_deterministic(*steps_per_second=None*) → None

Sets the simulator to run in deterministic mode. For this to function properly, an amount of steps per second needs to have been specified in the simulator's settings, through this function or through BeamNGpy.settings.set_steps_per_second().

Return type
None

set_nondeterministic() → None

Disables the deterministic mode of the simulator. Any steps per second setting is retained.

Return type
None

set_particles_enabled(*enabled: bool*) → None

Enable / disable visual particle emission.

Parameters
enabled (*bool*) – Whether to enable or disable effects.

Return type
None

set_steps_per_second(*sps: int*) → None

Specifies the temporal resolution of the simulation. The setting can be understood to determine into how many steps the simulation divides one second of simulation. A setting of two, for example, would mean one second is simulated in two steps. Conversely, to simulate one second, one needs to advance the simulation two steps.

Parameters
sps (*int*) – The steps per second to set.

Return type
None

class beamngpy.api.beamng.**SystemApi**(*beamng:* BeamNGpy)

Bases: *Api*

An API for getting info about the host system running the simulator.

Parameters
beamng (BeamNGpy) – An instance of the simulator.

get_info(*os: bool = True*, *cpu: bool = False*, *gpu: bool = False*, *power: bool = False*) → StrDict

Returns the information about the host's system.

Parameters

- **os** (*bool*) – Whether to include information about the operating system of the host.
- **cpu** (*bool*) – Whether to include information about the CPU of the host.
- **gpu** (*bool*) – Whether to include information about the GPU of the host.
- **power** (*bool*) – Whether to include information about the power options of the host.

Return type
StrDict

class beamngpy.api.beamng.**TrafficApi**(*beamng:* BeamNGpy)

Bases: *Api*

An API for controlling the traffic

Parameters
beamng (BeamNGpy) – An instance of the simulator.

reset() → None

Resets (force teleports) all vehicles in the traffic away from the player. Useful for resolving traffic jam situations.

Return type
None

spawn(*max_amount: int | None = None*, *police_ratio: float = 0*, *extra_amount: int | None = None*, *parked_amount: int | None = None*) → None

Enables traffic simulation with freshly spawned vehicles.

Parameters

- **max_amount** (*int | None*) – The maximum allowed vehicles to spawn. If None, defaults to in-game settings.
- **police_ratio** (*float*) – A number between 0.0 and 1.0 indicating the ratio of police vehicles in the traffic.
- **extra_amount** (*int | None*) – The maximum amount of inactive vehicles to spawn and swap in and out of the traffic system. If None, defaults to in-game settings.
- **parked_amount** (*int | None*) – The maximum amount of parked vehicles to spawn. If None, defaults to in-game settings.

Return type
None

start(*participants: List[Vehicle]*) → None

Enables traffic simulation for the given list of vehicles.

Parameters
participants (*List[Vehicle]*) – List of vehicles that will be part of the simulation. These vehicles need to be spawned beforehand and the simulation will take control of them.

Return type
None

stop(*stop: bool = False*) → None

Stops the traffic simulation.

Parameters
stop (*bool*) – Whether or not to stop the vehicles participating in traffic. If True, vehicles will come to a halt, if False, the AI will simply stop controlling the vehicle.

Return type
None

class beamngpy.api.beamng.**UiApi**(*beamng:* BeamNGpy)

Bases: *Api*

An API allowing the control of the simulator's GUI - displaying messages and hiding/showing the UI.

Parameters
beamng (BeamNGpy) – An instance of the simulator.

display_message(*msg: str*) → None

Displays a toast message in the user interface of the simulator.

Parameters
msg (*str*) – The message to display.

Return type
None

hide_hud() → None

Hides the HUD in the simulator.

Return type
None

show_hud() → None

Shows the HUD in the simulator.

Return type
None

class beamngpy.api.beamng.**VehiclesApi**(*beamng:* BeamNGpy)

Bases: *Api*

An API for vehicle manipulation in the simulator.

Parameters
beamng (BeamNGpy) – An instance of the simulator.

await_spawn(*vid: str* | beamngpy.vehicle.vehicle.Vehicle) → None

Waits for the vehicle with the given name to spawn and returns once it has.

Parameters
vid (*str | beamngpy.vehicle.vehicle.Vehicle*) – The name of the vehicle to wait for.

Return type
None

despawn(*vehicle:* Vehicle) → None

Despawns the given *Vehicle* from the simulation.

Parameters
vehicle (Vehicle) – The vehicle to despawn.

Return type
None

`get_available`() → StrDict

Retrieves a dictionary of vehicles known to the simulator that map to various properties of the vehicle and a list of pre-configured vehicle configurations.

Returns
A mapping of model names to vehicle properties & configs.

Raises
BNGError – If the game is not running to accept a request.

Return type
StrDict

`get_current`(*include_config: bool = True*) → Dict[str, *Vehicle*]

Queries the currently active vehicles in the simulator.

Parameters
include_config (*bool*) – Whether to include info about possible configurations of the vehicles.

Returns
A mapping of vehicle IDs to instances of the *Vehicle* class for each active vehicle. These vehicles are not connected to by this function.

Return type
Dict[str, Vehicle]

`get_current_info`(*include_config: bool = True*) → Dict[str, StrDict]

Queries the currently active vehicles in the simulator.

Parameters
include_config (*bool*) – Whether to include info about possible configurations of the vehicles.

Returns
A mapping of vehicle IDs to dictionaries of data needed to represent a *Vehicle*.

Return type
Dict[str, StrDict]

`get_part_annotation`(*part*)

`get_part_annotations`(*vehicle:* Vehicle)

Parameters
vehicle (Vehicle) –

`get_player_vehicle_id`() → StrDict

Queries the currently player vehicles in the simulator.

Returns
A dictionary of the active vehicle in simulator from lua. {'type': 'getPlayerVehicleID', 'id': 10455.0, 'vid': 'vehicleA'} then in python, the return will be only an int value of the 'id' and vehicle's name {'id': 10455, 'vid': 'vehicleA'} data = bng.vehicles.get_player_vehicle_id() for testing you can use the following: id_value = data['id'] vid_value = data['vid']

Return type
StrDict

get_states(*vehicles: Iterable[str]*) → Dict[str, Dict[str, Float3]]

Gets the states of the vehicles provided as the argument to this function. The returned state includes position, direction vectors and the velocities.

Parameters
vehicles (*Iterable[str]*) – A list of the vehicle IDs to query state from.

Returns
A mapping of the vehicle IDs to their state stored as a dictionary with [pos, dir, up, vel] keys.

Return type
Dict[str, *Dict*[str, Float3]]

replace(*new_vehicle:* Vehicle, *old_vehicle:* beamngpy.vehicle.vehicle.Vehicle | *str* | *None* = *None*, *connect: bool* = *True*) → None

Replaces old_vehicle with new_vehicle in the scenario. The new_vehicle keeps the position and rotation of old_vehicle. If old_vehicle is not provided, then the current player vehicle is replaced by new_vehicle.

Parameters

- **new_vehicle** (Vehicle) – The vehicle to
- **old_vehicle** (beamngpy.vehicle.vehicle.Vehicle | *str* | *None*) – The vehicle to be replaced, or its ID, or None if the currently focused vehicle should be replaced.
- **connect** (*bool*) – Whether to connect the replaced vehicle to BeamNGpy.

Return type
None

set_license_plate(*vehicle: str* | beamngpy.vehicle.vehicle.Vehicle, *text: str*) → None

Sets the text of a vehicle's license plate.

Parameters

- **vehicle** (*str* | beamngpy.vehicle.vehicle.Vehicle) – The id/name of the vehicle to teleport or the vehicle's object.
- **text** (*str*) – The vehicle plate text to be set.

Return type
None

spawn(*vehicle:* Vehicle, *pos: Float3*, *rot_quat: Quat* = *(0, 0, 0, 1)*, *cling: bool* = *True*, *connect: bool* = *True*) → bool

Spawns the given *Vehicle* instance in the simulator. This method is meant for spawning vehicles *during the simulation*. Vehicles that are known to be required before running the simulation should be added during scenario creation instead. Cannot spawn two vehicles with the same id/name.

Parameters

- **vehicle** (Vehicle) – The vehicle to be spawned.
- **pos** (*Float3*) – Where to spawn the vehicle as a (x, y, z) triplet.
- **rot_quat** (*Quat*) – Vehicle rotation in form of a quaternion
- **cling** (*bool*) – If set, the z-coordinate of the vehicle's position will be set to the ground level at the given position to avoid spawning the vehicle below ground or in the air.
- **connect** (*bool*) – Whether to connect the newly spawned vehicle to BeamNGpy.

Returns
bool indicating whether the spawn was successful or not

Return type
bool

start_connection(*vehicle: Vehicle*, *extensions: Optional[List[str]]*) → StrDict

Parameters

- **vehicle** (*Vehicle*) –
- **extensions** (*Optional[List[str]]*) –

Return type
StrDict

switch(*vehicle: str | beamngpy.vehicle.vehicle.Vehicle*) → None

Switches to the given *Vehicle*. This means that the simulator's main camera, inputs by the user, and so on will all focus on that vehicle from now on.

Parameters
vehicle (*str | beamngpy.vehicle.vehicle.Vehicle*) – The target vehicle.

Return type
None

teleport(*vehicle: str | Vehicle*, *pos: Float3*, *rot_quat: Quat | None = None*, *reset: bool = True*) → bool

Teleports the given vehicle to the given position with the given rotation.

Parameters

- **vehicle** (*str | Vehicle*) – The id/name of the vehicle to teleport or the vehicle's object.
- **pos** (*Float3*) – The target position as an (x, y, z) tuple containing world-space coordinates.
- **rot_quat** (*Quat | None*) – Optional tuple (x, y, z, w) specifying vehicle rotation as quaternion.
- **reset** (*bool*) – Specifies if the vehicle will be reset to its initial state during teleport (including its velocity).

Return type
bool

1.8.2.2 Vehicle

class beamngpy.**Vehicle**(*vid: str*, *model: str*, *port: int | None = None*, *license: str | None = None*, *color: Color | None = None*, *color2: Color | None = None*, *color3: Color | None = None*, *extensions: List[str] | None = None*, *part_config: str | None = None*, ***options: Any*)

The vehicle class represents a vehicle of the simulation that can be interacted with from BeamNGpy. This class offers methods to both control the vehicle's state as well as retrieve information about it through sensors the user can attach to the vehicle.

Creates a vehicle with the given vehicle ID. The ID must be unique within the scenario.

Parameters

- **vid** (*str*) – The vehicle's ID.
- **model** (*str*) – Model of the vehicle.

- **port** (*int | None*) – The TCP port on which the vehicle should connect. If None, a new port is requested from the simulator.
- **license** (*str | None*) – The license plate's text.
- **color** (*Color | None*) – The primary vehicle color.
- **color2** (*Color | None*) – The secondary vehicle color.
- **color3** (*Color | None*) – The tertiary vehicle color.
- **extensions** (*List[str] | None*) – A list of vehicle Lua extensions to load for the vehicle.
- **part_config** (*str | None*) – The path to the vehicle part configuration (a `.pc` file).
- **options** (*Any*) – Other possible vehicle options.

sensors

The sensors attached to the vehicle.

Type
Sensors

ai

The API module to control the AI behavior of the vehicle. See *AIApi* for details.

Type
AIApi

logging

The API module to control the logging behavior of the vehicle inside the simulator. See *LoggingApi* for details.

Type
LoggingApi

annotate_parts() → None

Triggers the process to have individual parts of a vehicle have unique annotation colors.

Return type
None

close() → None

Closes this vehicle's and its sensors' connection and detaches all sensors.

Return type
None

connect(*bng:* BeamNGpy) → None

Opens socket communication with the corresponding vehicle.

Parameters
bng (BeamNGpy) – An instance of the simulator.

Return type
None

control(*steering: float | None = None*, *throttle: float | None = None*, *brake: float | None = None*, *parkingbrake: float | None = None*, *clutch: float | None = None*, *gear: int | None = None*) → None

Sends a control message to the vehicle, setting vehicle inputs accordingly.

Parameters

- **steering** (*float | None*) – Rotation of the steering wheel, from -1.0 to 1.0.

- **throttle** (*float | None*) – Intensity of the throttle, from 0.0 to 1.0.
- **brake** (*float | None*) – Intensity of the brake, from 0.0 to 1.0.
- **parkingbrake** (*float | None*) – Intensity of the parkingbrake, from 0.0 to 1.0.
- **clutch** (*float | None*) – Clutch level, from 0.0 to 1.0.
- **gear** (*int | None*) – Gear to shift to, -1 eq backwards, 0 eq neutral, 1 to X eq nth gear

Return type
None

deflate_tire(*wheel_id: int*) → None

Deflates the tire of this vehicle with the given wheel ID.

Parameters
wheel_id (*int*) – The given wheel ID.

Return type
None

disconnect() → None

Closes socket communication with the corresponding vehicle.

Return type
None

static from_dict(*d: StrDict*) → *Vehicle*

Parameters
d (*StrDict*) –

Return type
Vehicle

get_bbox() → Dict[str, Float3]

Returns a vehicle's current bounding box as a dictionary containing eight points. The bounding box corresponds to the vehicle's location/rotation in world space, i.e. if the vehicle moves/turns, the bounding box moves accordingly. Note that the bounding box contains the min/max coordinates of the entire vehicle. This means that the vehicle losing a part like a mirror will cause the bounding box to "expand" while the vehicle moves as the mirror is left behind, but still counts as part of the box containing the vehicle.

Returns

The vehicle's current bounding box as a dictionary of eight points. Points are named following the convention that the cuboid has a "near" rectangle towards the rear of the vehicle and "far" rectangle towards the front. The points are then named like this:

- **front_bottom_left**
  Bottom left point of the front rectangle as an `(x, y, z)` triplet
- **front_bottom_right**
  Bottom right point of the front rectangle as an `(x, y, z)` triplet
- **front_top_left**
  Top left point of the front rectangle as an `(x, y, z)` triplet
- **front_top_right**
  Top right point of the front rectangle as an `(x, y, z)` triplet
- **rear_bottom_left**
  Bottom left point of the rear rectangle as an `(x, y, z)` triplet

- **rear_bottom_right**
  Bottom right point of the rear rectangle as an `(x, y, z)` triplet
- **rear_top_left**
  Top left point of the rear rectangle as an `(x, y, z)` triplet
- **rear_top_right**
  Top right point of the rear rectangle as an `(x, y, z)` triplet

Return type
Dict[str, Float3]

get_center_of_gravity(*without_wheels=False*) → Float3

Returns the vehicle's center of gravity.

Parameters
without_wheels – If True, the center of gravity is calculated without the wheels. Defaults to False.

Returns
The center of gravity as a `(x, y, z)` triplet.

Return type
Float3

get_part_config() → StrDict

Retrieves the current part configuration of the given vehicle. The configuration contains both the current values of adjustable vehicle parameters and a mapping of part types to their currently-selected part.

Returns
The current vehicle configuration as a dictionary.

Return type
StrDict

get_part_options() → StrDict

Retrieves a mapping of part slots for the given vehicle and their possible parts.

Returns
A mapping of part configuration options for the given.

Return type
StrDict

is_connected() → bool

Whether the vehicle is connected to the simulator and can be controlled from Python.

Return type
bool

queue_lua_command(*chunk: str*) → None

Executes a chunk of Lua code in the vehicle engine VM.

Parameters
chunk (*str*) – chunk of Lua code as a string

Return type
None

recover() → None

Recovers the vehicle to a drivable position and state and repairs its damage.

Return type
None

revert_annotations() → None

Reverts the given vehicle's annotations back to the object-based mode, removing the per-part annotations.

Return type
None

set_color(*rgba: Color = (1.0, 1.0, 1.0, 1.0)*) → None

Sets the color of this vehicle. Colour can be adjusted on the RGB spectrum and the "shininess" of the paint.

Parameters
rgba (*Color*) – The new colour given as a tuple of RGBA floats, where the alpha channel encodes the shininess of the paint. Also can be given in any format specified in *Color*.

Return type
None

set_license_plate(*text: str*) → None

Sets the text of the vehicle's license plate.

Parameters
text (*str*) – The vehicle plate text to be set.

Return type
None

set_lights(*left_signal: bool | None = None*, *right_signal: bool | None = None*, *hazard_signal: bool | None = None*, *headlights: int | None = None*, *fog_lights: int | None = None*, *lightbar: int | None = None*) → None

Sets the vehicle's lights to given intensity values. The lighting system features lights that are simply binary on/off, but also ones where the intensity can be varied. Binary lights include:

- `left_signal`
- `right_signal`
- `hazard_signal`

Non-binary lights vary between 0 for off, 1 for on, 2 for higher intensity. For example, headlights can be turned on with 1 and set to be more intense with 2. Non-binary lights include:

- `headlights`
- `fog_lights`
- `lightbar`

Parameters

- **left_signal** (*bool | None*) – On/off state of the left signal
- **right_signal** (*bool | None*) – On/off state of the right signal
- **hazard_signal** (*bool | None*) – On/off state of the hazard lights
- **headlights** (*int | None*) – Value from 0 to 2 indicating headlight intensity
- **fog_lights** (*int | None*) – Value from 0 to 2 indicating fog light intensity
- **lightbar** (*int | None*) – Value from 0 to 2 indicating lightbar intensity

Return type
None

Note: Not every vehicle has every type of light. For example, the *lightbar* refers to the kind of lights typically found on top of police cars. Setting values for non-existent lights will not cause an error, but also achieve no effect.

Note also that lights are not independent. For example, turning on the hazard lights will make both signal indicators blink, meaning they will be turned on as well. Opposing indicators also turn each other off, i.e. turning on the left signal turns off the right one, and turning on the left signal during

---

Raises
BNGValueError – If an invalid light value is given.

Returns
Nothing. To query light states, attach an *sensors.Electrics* sensor and poll it.

Parameters

- **left_signal** (*bool | None*) –
- **right_signal** (*bool | None*) –
- **hazard_signal** (*bool | None*) –
- **headlights** (*int | None*) –
- **fog_lights** (*int | None*) –
- **lightbar** (*int | None*) –

Return type
None

set_part_config(*cfg: StrDict*) → None

Sets the current part configuration of the given vehicle. The configuration is given as a dictionary containing both adjustable vehicle parameters and a mapping of part types to their selected parts.

Parameters
cfg (*StrDict*) – The new vehicle configuration as a dictionary.

Return type
None

Notes

Changing parts causes the vehicle to respawn, which repairs it as a side-effect.

set_shift_mode(*mode: str*) → None

Sets the shifting mode of the vehicle. This changes whether or not and how the vehicle shifts gears depending on the RPM. Available modes are:

- **realistic_manual**
  Gears have to be shifted manually by the user, including engaging the clutch.
- **realistic_manual_auto_clutch**
  Gears have to be shifted manually by the user, without having to use the clutch.

- **arcade**
  Gears shift up and down automatically. If the brake is held, the vehicle automatically shifts into reverse and accelerates backward until brake is released or throttle is engaged.
- **realistic_automatic**
  Gears shift up automatically, but reverse and parking need to be shifted to manually.

Parameters
mode (*str*) – The mode to set. Must be a string from the options listed above.

Raises
BNGValueError – If an invalid mode is given.

Return type
None

set_velocity(*velocity: float*, *dt: float = 1.0*) → None

Sets the velocity of this vehicle. The velocity is not achieved instantly, it is acquired gradually over the time interval set by the *dt* argument.

As the method of setting velocity uses physical forces, at high velocities it is important to set `dt` to an appropriately high value. The default `dt` value of 1.0 is suitable for velocities up to 30 m/s.

Parameters

- **velocity** (*float*) – The target velocity in m/s.
- **dt** (*float*) – The time interval over which the vehicle reaches the target velocity. Defaults to 1.0.

Return type
None

property state: Dict[str, Float3 | Quat]

This property contains the vehicle's current state in the running scenario. It is empty if no scenario is running or the state has not been retrieved yet. Otherwise, it contains the following key entries:

- `pos`: The vehicle's position as an `(x, y, z)` triplet
- `dir`: The vehicle's direction vector as an `(x, y, z)` triplet
- `up`: The vehicle's up vector as an `(x, y, z)` triplet
- `vel`: The vehicle's velocity along each axis in metres per second as an `(x, y, z)` triplet
- `rotation`: The vehicle's rotation as an `(x, y, z, w)` quaternion

Note that the `state` variable represents a *snapshot* of the last state. It has to be updated with a call to *Sensors.poll()* or to *Scenario.update()*.

switch() → None

Switches the simulator to this vehicle. This means that the simulator's main camera, inputs by the user, and so on will all focus on this vehicle from now on.

Return type
None

teleport(*pos: Float3*, *rot_quat: Quat | None = None*, *reset: bool = True*) → bool

Teleports the vehicle to the given position with the given rotation.

Parameters

- **pos** (*Float3*) – The target position as an (x,y,z) tuple containing world-space coordinates.

- **rot_quat** (*Quat | None*) – Optional tuple (x, y, z, w) specifying vehicle rotation as quaternion.
- **reset** (*bool*) – Specifies if the vehicle will be reset to its initial state during teleport (including its velocity).

Return type
bool

1.8.2.2.1 Sensors

class beamngpy.vehicle.**Sensors**(*vehicle:* Vehicle)

A sensor collection for a vehicle.

Parameters
vehicle (Vehicle) – The vehicle to which this object instance should belong to.

attach(*name: str*, *sensor:* Sensor) → None

Enters a sensor into this vehicle's map of known sensors and calls the attach-hook of said sensor. The sensor is identified using the given name, which has to be unique among the other sensors of the vehicle.

Parameters

- **name** (*str*) – The name of the sensor.
- **sensor** (Sensor) – The sensor to attach to the vehicle.

Return type
None

detach(*name: str*) → None

Detaches a sensor from the vehicle's map of known sensors and calls the detach-hook of said sensor.

Parameters
name (*str*) – The name of the sensor to disconnect.

Return type
None

poll(**sensor_names: str*) → None

Updates the vehicle's sensor readings.

Parameters
sensor_names (*str*) – Names of sensors to poll. If none are provided, then all attached sensors are polled.

Returns
Nothing. Use `vehicle.sensors[<sensor_id>][<data_access_id>]` to access the polled sensor data.

Return type
None

1.8.2.2.2 API

class beamngpy.api.vehicle.**AIApi**(*vehicle:* Vehicle)

Bases: *VehicleApi*

An API class gathering AI-related functionality.

Parameters
vehicle (Vehicle) – An instance of a vehicle object.

drive_in_lane(*lane: bool*) → None

Sets the drive in lane flag of the AI. If True, the AI only drives within the lane it can legally drive in.

Parameters
lane (*bool*) – Lane flag to set.

Return type
None

execute_script(*script*, *cling: bool = True*, *start_delay: float = 0.0*, *no_reset: bool = False*) → None

Parameters

- **cling** (*bool*) –
- **start_delay** (*float*) –
- **no_reset** (*bool*) –

Return type
None

get_initial_spawn_position_orientation(*script*)

set_aggression(*aggr: float*) → None

Parameters
aggr (*float*) –

Return type
None

set_line(*line: List[Dict[str, Float3 | float]]*, *cling: bool = True*) → None

Makes the AI follow a given polyline. The line is specified as a list of dictionaries where each dictionary has a `pos` entry specifying the supposed position as an `(x, y, z)` triplet and a `speed` entry specifying the speed in m/s.

Parameters

- **line** (*List[Dict[str, Float3 | float]]*) – Polyline as list of dicts as described above.
- **cling** (*bool*) – Whether or not to align the `z` coordinate of the polyline to the ground.

Return type
None

set_mode(*mode: str*) → None

Sets the desired mode of the simulator's built-in AI for this vehicle. Possible values are:

- `disabled`: Turn the AI off (default state)
- `random`: Drive from random points to random points on the map

- `span`: Drive along the entire road network of the map
- `manual`: Drive to a specific waypoint, target set separately
- `chase`: Chase a target vehicle, target set separately
- `flee`: Flee from a vehicle, target set separately
- `stopping`: Make the vehicle come to a halt (AI disables itself once the vehicle stopped.)

Note: Some AI methods automatically set appropriate modes, meaning a call to this method might be optional.

Parameters
mode (*str*) – The AI mode to set.

Return type
None

set_script(*script: List[Dict[str, float]]*, *cling: bool = True*) → None

Makes the vehicle follow a given "script" – a script being a list of timestamped positions defining where a vehicle should be at what time. This can be used to make the vehicle drive a long a polyline with speed implicitly expressed in the time between points.

Parameters

- **script** (*List[Dict[str, float]]*) – A list of nodes in the script. Each node is expected to be a dict-like that has `x`, `y`, and `z` entries for the supposed position of the vehicle, and a `t` entry for the time of the node along the path. Time values are in seconds relative to the time when script playback is started.
- **cling** (*bool*) – A flag that makes the simulator cling z-coordinates to the ground. Since computing z-coordinates in advance without knowing the level geometry can be cumbersome, this flag is used to automatically set z-coordinates in the script to the ground height. Defaults to True.

Return type
None

Notes

The AI follows the given script the best it can. It cannot drive along scripts that would be physically impossible, e.g. specifying a script with points A & B one kilometer apart and giving it a a second between those points will make the AI drive from A to B as fast as it can, but unlikely to reach it in the given time. Furthermore, if the AI falls behind schedule, it will start skipping points in the script in an effort to make up for lost time.

Raises
BNGValueError – If the script has fewer than three nodes, the minimum length of a script.

Parameters

- **script** (*List[Dict[str, float]]*) –
- **cling** (*bool*) –

Return type
None

set_speed(*speed: float*, *mode: str = 'limit'*) → None

Sets the target speed for the AI in m/s. Speed can be maintained in two modes:

- **limit: Drive speeds between 0 and the limit, as the AI** sees fit.
- set: Try to maintain the given speed at all times.

Parameters

- **speed** (*float*) – The target speed in m/s.
- **mode** (*str*) – The speed mode.

Return type
None

set_target(*target: str*, *mode: str = 'chase'*) → None

Sets the target to chase or flee. The target should be the ID of another vehicle in the simulation. The AI is automatically set to the given mode.

Parameters

- **target** (*str*) – ID of the target vehicle as a string.
- **mode** (*str*) – How the target should be treated. chase to chase the target, flee to flee from it.

Return type
None

set_waypoint(*waypoint: str*) → None

Sets the waypoint the AI should drive to in manual mode. The AI gets automatically set to manual mode when this method is called.

Parameters
waypoint (*str*) – ID of the target waypoint as a string.

Return type
None

start_recording() → None

Return type
None

stop_recording(*filename*) → None

Return type
None

class beamngpy.api.vehicle.**LoggingApi**(*vehicle:* Vehicle)

Bases: *VehicleApi*

A base API class from which all the API communicating with a vehicle derive.

Parameters
vehicle (Vehicle) – An instance of a vehicle object.

set_options_from_json(*filename: str*) → None

Updates the in game logging with the settings specified in the given file/json. The file is expected to be in the following location: <userpath>/<version_number>/<file_name>

Parameters
filename (*str*) –

Return type
None

start(*output_dir: str*) → None

Starts in game logging. Beware that any data from previous logging sessions is overwritten in the process.

Parameters
output_dir (*str*) – to avoid overwriting logging from other vehicles, specify the output directory, overwrites the output_dir set through the json. The data can be found in: <user-path>/<BeamNG version number>/<output_dir>

Return type
None

stop() → None

Stops in game logging.

Return type
None

write_options_to_json(*filename: str = 'template.json'*) → None

Writes all available options from the in-game-logger to a json file. The purpose of this functionality is to facilitate the acquisition of a valid template to adjust the options/settings of the in game logging as needed. Depending on the executable used the file can be found at the following location: <userpath>/<BeamNG version number>/<fileName>

Parameters
filename (*str*) – not the absolute file path but the name of the json

Return type
None

class beamngpy.api.vehicle.**VehicleApi**(*vehicle:* Vehicle)

Bases: object

An API class for in-game logging of vehicle data.

Parameters
vehicle (Vehicle) – An instance of a vehicle object.

1.8.2.3 Scenario

class beamngpy.**Scenario**(*level: str | beamngpy.scenario.level.Level*, *name: str*, *path: str | None = None*, *human_name: str | None = None*, *description: str | None = None*, *difficulty: int = 0*, *authors: str = 'BeamNGpy'*, ***options: Any*)

The scenario class contains information for setting up and executing simulation scenarios along with methods to extract data during their execution.

Instantiates a scenario instance with the given name taking place in the given level.

Parameters

- **level** (*str | Level*) – Either the name of the level this scenario takes place in as a string or as an instance of *Level*
- **name** (*str*) – The name of this scenario. Should be unique for the level it's taking place in to avoid file collisions.

- **path** (*str | None*) – The path to an already existing scenario file (relative to the home folder / user folder). If set, then *Scenario.make()* should not be called, as the scenario is already made.
- **human_name** (*str | None*) – The human-readable name of the scenario. If None, it will be set to name.
- **description** (*str | None*) – The description of the scenario displayed in the simulator.
- **difficulty** (*int*) – The difficulty of the scenario displayed in the simulator.
- **authors** (*str*) – Names of the authors. Defaults to BeamNGpy.
- **options** (*Any*) – Other pptions of the scenario object, not used at the moment.

add_checkpoints(*positions: List[Float3]*, *scales: List[Float3]*, *ids: Optional[List[str]] = None*) → None

Adds checkpoints to the scenario.

Parameters

- **positions** (*List[Float3]*) – Positions (tuple of length 3) of the individual points.
- **scales** (*List[Float3]*) – Scales (tuple of length 3) of the individual points
- **ids** (*Optional[List[str]]*) – Optional, names of the individual points.

Return type
None

add_mesh_road(*road:* MeshRoad) → None

Adds a *MeshRoad* to this scenario.

Parameters
road (MeshRoad) – Mesh road to be added to the scenario.

Return type
None

add_object(*obj:* ScenarioObject) → None

Adds an extra object to be placed in the prefab. Objects are expected to be *ScenarioObject* instances with additional, type- specific properties in that class's opts dictionary.

Parameters
obj (ScenarioObject) –

Return type
None

add_procedural_mesh(*mesh:* ProceduralMesh) → None

Adds a *ProceduralMesh* to be placed in world to the scenario.

Parameters
mesh (ProceduralMesh) – The mesh to place.

Return type
None

add_road(*road:* Road) → None

Adds a *Road* to this scenario.

Parameters
road (Road) – Road to be added to the scenario.

Return type
None

add_vehicle(*vehicle: Vehicle*, *pos: Float3 = (0, 0, 0)*, *rot_quat: Quat = (0, 0, 0, 1)*, *cling: bool = True*) → None

Adds a *Vehicle*: to this scenario at the given position with the given orientation.

Parameters

- **vehicle** (*Vehicle*) – The vehicle to spawn.
- **pos** (*Float3*) – `(x, y, z)` tuple specifying the position of the vehicle.
- **rot_quat** (*Quat*) – `(x, y, z, w)` tuple specifying the rotation as quaternion.
- **cling** (*bool*) – If True, the z-coordinate of the vehicle's position will be set to the ground level at the given position to avoid spawning the vehicle below ground or in the air.

Return type
None

close() → None

Closes open connections and allocations of the scenario.

Raises
BNGError – If the scenario has not been loaded.

Return type
None

connect(*bng: BeamNGpy*, *connect_player: bool = True*, *connect_existing: bool = True*) → None

Connects this scenario to the simulator.

Parameters

- **bng** (*BeamNGpy*) – The BeamNGpy instance to generate the scenario for.
- **connect_player** (*bool*) – Whether the player vehicle should be connected to this (:class:`.Scenario`) instance. Defaults to True.
- **connect_existing** (*bool*) – Whether ALL vehicles spawned already in the scenario should be connected to this (:class:`.Scenario`) instance. Defaults to True.

Return type
None

delete(*bng: BeamNGpy*) → None

Deletes files created by this scenario from the given *BeamNGpy*'s home/user path.

Parameters
bng (*BeamNGpy*) –

Return type
None

find(*bng: BeamNGpy*) → str | None

Looks for the files of an existing scenario and returns the path to the info file of this scenario, iff one is found.

Parameters
bng (*BeamNGpy*) – The BeamNGpy instance to look for the scenario in.

Returns
The path to the information file of his scenario found in the simulator as a string. None if it could not be found.

Return type
str | None

`find_procedural_meshes()` → List[*ScenarioObject*]

Finds procedural meshes placed in the world right now.

Returns
A list of `ScenarioObject` containing procedural meshes found in the world.

Raises
BNGError – If the scenario is not currently loaded.

Return type
List[ScenarioObject]

`find_static_objects()` → List[*ScenarioObject*]

Finds static objects placed in the world right now.

Returns
A list of `ScenarioObject` containing statically placed objects found in the world.

Raises
BNGError – If the scenario is not currently loaded.

Return type
List[ScenarioObject]

`find_waypoints()` → List[*ScenarioObject*]

Finds waypoints placed in the world right now.

Returns
A list of `ScenarioObject` containing waypoints found in the world.

Raises
BNGError – If the scenario is not currently loaded.

Return type
List[ScenarioObject]

`static from_dict(`*d: StrDict*`)` → *Scenario*

Parameters
d (*StrDict*) –

Return type
Scenario

`get_vehicle(`*vehicle_id: str*`)` → *beamngpy.vehicle.vehicle.Vehicle* | None

Retrieves the vehicle with the given ID from this scenario.

Parameters
vehicle_id (*str*) – The ID of the vehicle to find.

Returns
The `Vehicle` with the given ID. `None` if it wasn't found.

Return type
beamngpy.vehicle.vehicle.Vehicle | None

make(*bng:* BeamNGpy) → None

Generates necessary files to describe the scenario in the simulation and outputs them to the simulator.

Parameters
bng (BeamNGpy) – The BeamNGpy instance to generate the scenario for.

Raises
BNGError – If the scenario already has set its info .json file included.

Return type
None

remove_procedural_mesh(*mesh:* ProceduralMesh) → None

Removes a *ProceduralMesh* that was placed in the world.

Parameters
mesh (ProceduralMesh) – The mesh to remove.

Raises
BNGError – If the mesh to remove was not found.

Return type
None

remove_vehicle(*vehicle:* Vehicle) → None

Removes the given *Vehicle*: from this scenario. If the scenario is currently loaded, the vehicle will be despawned.

Parameters
vehicle (Vehicle) – The vehicle to remove.

Return type
None

restart() → None

Restarts this scenario. Requires the scenario to be loaded into a running *BeamNGpy* instance first.

Notes

If any vehicles have been added during the scenario after it has been started, they will be removed as the scenario is reset to its original state.

Raises
BNGError – If the scenario has not been loaded.

Return type
None

scenetree_classes: Dict[str, Callable[[StrDict], SceneObject]] = {'DecalRoad': <function Scenario.<lambda>>, 'MissionGroup': <function Scenario.<lambda>>}

set_initial_focus(*vehicle_id: str*) → None

Defines which vehicle has the initial focus.

Parameters
vehicle_id (*str*) – Vehicle id of focused vehicle

Return type
None

sync_scene() → None

Retrieves the current scene tree of the scenario from the simulator, converting them into the most appropriate known (sub)class of `SceneObject`. The result is not returned but rather stored in the `scene` field of this class.

Return type
None

update() → None

Synchronizes object states of this scenario with the simulator. This is used to update the *Vehicle.state* fields of each vehicle in the scenario.

Raises
BNGError – If the scenario is currently not loaded.

Return type
None

class beamngpy.**Level**(*name: str*, *size: Int2*, *path: str | None*, ***props: Any*)

Represents a level in the simulator, listing various properties like the level's name, size, and available scenarios.

Parameters

- **name** (*str*) –
- **size** (*Int2*) –
- **path** (*str | None*) –
- **props** (*Any*) –

static from_dict(*d: StrDict*) → *Level*

Parameters
d (*StrDict*) –

Return type
Level

class beamngpy.**ScenarioObject**(*oid: str*, *name: str | None*, *otype: str*, *pos: Float3*, *scale: Float3*, *rot_quat: Quat | None = None*, ***options: str*)

This class is used to represent objects in the simulator's environment. It contains basic information like the object type, position, rotation, and scale.

Creates a scenario object with the given parameters.

Parameters

- **oid** (*str*) – name of the asset
- **name** (*str | None*) – asset id
- **otype** (*str*) – type of the object according to the BeamNG classification
- **pos** (*Float3*) – x, y, and z coordinates
- **scale** (*Float3*) – defining the scale along the x,y, and z axis.
- **rot_quat** (*Quat | None*) – Quaternion describing the initial orientation. Defaults to None.
- **options** (*str*) –

static from_game_dict(*d: StrDict*) → *ScenarioObject*

Parameters
d (*StrDict*) –

Return type
ScenarioObject

remove(*bng:* BeamNGpy) → None

Parameters
bng (BeamNGpy) –

Return type
None

1.8.2.3.1 Procedural Objects

class beamngpy.**ProceduralMesh**(*pos: Float3*, *name: str*, *material: str | None*, *rot_quat: Quat | None = None*)

Bases: *ScenarioObject*

Parameters

- **pos** (*Float3*) –
- **name** (*str*) –
- **material** (*str | None*) –
- **rot_quat** (*Quat | None*) –

class beamngpy.**ProceduralCylinder**(*pos: Float3*, *radius: float*, *height: float*, *name: str*, *rot_quat: Quat | None = None*, *material: str | None = None*)

Bases: *ProceduralMesh*

Creates a procedurally generated cylinder mesh with the given radius and height at the given position and rotation. The material can optionally be specified and a name can be assigned for later identification.

Parameters

- **pos** (*Float3*) – (X, Y, Z) coordinate triplet specifying the cylinder's position.
- **radius** (*float*) – The radius of the cylinder's base circle.
- **height** (*float*) – The between top and bottom circles of the cylinder.
- **name** (*str*) – Name for the mesh. Should be unique.
- **rot_quat** (*Quat | None*) – Quaternion specifying the cylinder's rotation
- **material** (*str | None*) – Optional material name to use as a texture for the mesh.

class beamngpy.**ProceduralBump**(*pos: Float3*, *width: float*, *length: float*, *height: float*, *upper_length: float*, *upper_width: float*, *name: str*, *rot_quat: Quat | None = None*, *material: str | None = None*)

Bases: *ProceduralMesh*

Creates a procedurally generated bump with the given properties at the given position and rotation. The material can optionally be specified and a name can be assigned for later identification.

Parameters

- **pos** (*Float3*) – (X, Y, Z) coordinate triplet specifying the cylinder's position.

- **width** (*float*) – The width of the bump, i.e. its size between left and right edges.
- **length** (*float*) – The length of the bump, i.e. the distances from up and downward slopes.
- **height** (*float*) – The height of the tip.
- **upper_length** (*float*) – The length of the tip.
- **upper_width** (*float*) – The width of the tip.
- **name** (*str*) – Name for the mesh. Should be unique.
- **rot_quat** (*Quat | None*) – Quaternion specifying the bump's rotation
- **material** (*str | None*) – Optional material name to use as a texture for the mesh.

class beamngpy.**ProceduralCone**(*pos: Float3*, *radius: float*, *height: float*, *name: str*, *rot_quat: Quat | None = None*, *material: str | None = None*)

Bases: *ProceduralMesh*

Creates a procedurally generated cone with the given properties at the given position and rotation. The material can optionally be specified and a name can be assigned for later identification.

Parameters

- **pos** (*Float3*) – (X, Y, Z) coordinate triplet specifying the cylinder's position.
- **radius** (*float*) – Radius of the base circle.
- **height** (*float*) – Distance of the tip to the base circle.
- **name** (*str*) – Name for the mesh. Should be unique.
- **rot_quat** (*Quat | None*) – Quaternion specifying the cone's rotation
- **material** (*str | None*) – Optional material name to use as a texture for the mesh.

class beamngpy.**ProceduralCube**(*pos: Float3*, *size: Float3*, *name: str*, *rot_quat: Quat | None = None*, *material: str | None = None*)

Bases: *ProceduralMesh*

Creates a procedurally generated cube with the given properties at the given position and rotation. The material can optionally be specified and a name can be assigned for later identification.

Parameters

- **pos** (*Float3*) – (X, Y, Z) coordinate triplet specifying the cylinder's position.
- **size** (*Float3*) – A triplet specifying the (length, width, height) of the cuboid.
- **name** (*str*) – Name for the mesh. Should be unique.
- **rot_quat** (*Quat | None*) – Quaternion specifying the cube's rotation
- **material** (*str | None*) – Optional material name to use as a texture for the mesh.

class beamngpy.**ProceduralRing**(*pos: Float3*, *radius: float*, *thickness: float*, *name: str*, *rot_quat: Quat | None = None*, *material: str | None = None*)

Bases: *ProceduralMesh*

Creates a procedurally generated ring with the given properties at the given position and rotation. The material can optionally be specified and a name can be assigned for later identification.

Parameters

- **pos** (*Float3*) – (X, Y, Z) coordinate triplet specifying the cylinder's position.

- **radius** (*float*) – Radius of the circle encompassing the ring.
- **thickness** (*float*) – Thickness of the rim.
- **name** (*str*) – Name for the mesh. Should be unique.
- **rot_quat** (*Quat | None*) – Quaternion specifying the ring's rotation
- **material** (*str | None*) – Optional material name to use as a texture for the mesh.

1.8.2.3.2 Roads

class beamngpy.**Road**(*material: str*, *rid: str | None = None*, *interpolate: bool = True*, *default_width: float = 10.0*, *drivability: int = 1*, *one_way: bool = False*, *flip_direction: bool = False*, *over_objects: bool = True*, *looped: bool = False*, *smoothness: float = 0.5*, *break_angle: float = 3*, *texture_length: int = 5*, *render_priority: int = 10*)

This class represents a DecalRoad in the environment. It contains information about the road's material, direction-ness of lanes, and geometry of the edges that make up the road.

Creates a new road instance using the given material name. The material name needs to match a material that is part of the level in the simulator this road will be placed in.

Parameters

- **material** (*str*) – Name of the material this road uses. This affects how the road looks visually and needs to match a material that's part of the level this road is placed in.
- **rid** (*str | None*) – Optional string setting this road's name. If specified, needs to be unique with respect to other roads in the level/scenario.
- **interpolate** (*bool*) – Whether to apply Catmull-Rom spline interpolation to smooth transition between the road's nodes.
- **default_width** (*float*) – Default width of the road nodes.
- **drivability** (*int*) –
- **one_way** (*bool*) –
- **flip_direction** (*bool*) –
- **over_objects** (*bool*) –
- **looped** (*bool*) –
- **smoothness** (*float*) –
- **break_angle** (*float*) –
- **texture_length** (*int*) –
- **render_priority** (*int*) –

add_nodes(**nodes: Float3 | Float4*) → None

Adds a list of nodes to this decal road.

Parameters

nodes (*Float3 | Float4*) – List of `(x, y, z)` or `(x, y, z, width)` tuples of the road's nodes.

Return type

None

class beamngpy.**MeshRoad**(*top_material: str*, *bottom_material: str | None = None*, *side_material: str | None = None*, *rid: str | None = None*, *default_width: float = 10.0*, *default_depth: float = 5.0*, *texture_length: float = 5*, *break_angle: float = 3*, *width_subdivisions: int = 0*)

This class represents a MeshRoad in the environment. It contains information about the road's materials, direction-ness of lanes, and geometry of the edges that make up the road.

Creates a new road instance using the given material name. The material name needs to match a material that is part of the level in the simulator this road will be placed in.

Parameters

- **top_material** (*str*) – Name of the material this road uses for the top part. This affects how the road looks visually and needs to match a material that's part of the level this road is placed in.
- **bottom_material** (*str | None*) – Name of the material this road uses for the bottom part. Defaults to `top_material`.
- **side_material** (*str | None*) – Name of the material this road uses for the side part. Defaults to `top_material`.
- **rid** (*str | None*) – Optional string setting this road's name. If specified, needs to be unique with respect to other roads in the level/scenario.
- **default_width** (*float*) – Default width of the road nodes.
- **default_depth** (*float*) – Default depth of the road nodes.
- **texture_length** (*float*) –
- **break_angle** (*float*) –
- **width_subdivisions** (*int*) –

add_nodes(**nodes: Float3 | Float4 | Float5*) → None

Adds a list of nodes to this decal road.

Parameters
nodes (*Float3 | Float4 | Float5*) – List of `(x, y, z)`, `(x, y, z, width)` or `(x, y, z, width, depth)` tuples of the road's nodes.

Return type
None

1.8.2.4 Sensors

1.8.2.4.1 Automated Sensors

Camera

class beamngpy.sensors.**Camera**(*name: str*, *bng:* BeamNGpy, *vehicle:* Vehicle *| None = None*, *requested_update_time: float = 0.1*, *update_priority: float = 0.0*, *pos: Float3 = (0, 0, 3)*, *dir: Float3 = (0, -1, 0)*, *up: Float3 = (0, 0, 1)*, *resolution: Int2 = (512, 512)*, *field_of_view_y: float = 70*, *near_far_planes: Float2 = (0.05, 100.0)*, *is_using_shared_memory: bool = False*, *is_render_colours: bool = True*, *is_render_annotations: bool = True*, *is_render_instance: bool = False*, *is_render_depth: bool = True*, *is_depth_inverted: bool = False*, *is_visualised: bool = False*, *is_streaming: bool = False*, *is_static: bool = False*, *is_snapping_desired: bool = False*, *is_force_inside_triangle: bool = False*)

An interactive, automated camera sensor, which can produce regular colour images, depth images, or annotation images. This sensor can be attached to a vehicle, or can be fixed to a position in space. The dir and up parameters are used to set the local coordinate system. A requested update rate can be provided, to tell the simulator how often to read measurements for this sensor. If a negative value is provided, the sensor will not update automatically at all. However, ad-hoc polling requests can be sent at any time, even for non-updating sensors.

Parameters

- **name** (*str*) – A unique name for this camera sensor.
- **bng** (BeamNGpy) – The BeamNGpy instance, with which to communicate to the simulation.
- **vehicle** (Vehicle | *None*) – The vehicle to which this sensor should be attached, if any.
- **requested_update_time** (*float*) – The time which should pass between sensor reading updates, in seconds. This is just a suggestion to the manager.
- **update_priority** (*float*) – The priority which the sensor should ask for new readings. lowest -> 0, highest -> 1.
- **pos** (*Float3*) – (X, Y, Z) Coordinate triplet specifying the position of the sensor, in world space.
- **dir** (*Float3*) – (X, Y, Z) Coordinate triplet specifying the forward direction of the sensor.
- **up** (*Float3*) – (X, Y, Z) Coordinate triplet specifying the up direction of the sensor.
- **resolution** (*Int2*) – (X, Y) The resolution of the sensor images.
- **field_of_view_y** (*float*) – The sensor vertical field of view parameters.
- **near_far_planes** (*Float2*) – (X, Y) The sensor near and far plane distances.
- **is_using_shared_memory** (*bool*) – A flag which indicates if we should use shared memory to send/recieve the sensor readings data.
- **is_render_colours** (*bool*) – A flag which indicates if this sensor should render colour data.
- **is_render_annotations** (*bool*) – A flag which indicates if this sensor should render semantic annotation data.
- **is_render_instance** (*bool*) – A flag which indicates if this sensor should render instance annotation data.
- **is_render_depth** (*bool*) – A flag which indicates if this sensor should render depth data.
- **is_depth_inverted** (*bool*) – A flag which indicates if the depth values should be shown white->black or black->white, as distance increases.
- **is_visualised** (*bool*) – A flag which indicates if this LiDAR sensor should appear visualised or not.
- **is_streaming** (*bool*) – Whether or not to stream the data directly to shared memory (no poll required, for efficiency - BeamNGpy won't block.)
- **is_static** (*bool*) – A flag which indicates whether this sensor should be static (fixed position), or attached to a vehicle.
- **is_snapping_desired** (*bool*) – A flag which indicates whether or not to snap the sensor to the nearest vehicle triangle (not used for static sensors).
- **is_force_inside_triangle** (*bool*) – A flag which indicates if the sensor should be forced inside the nearest vehicle triangle (not used for static sensors).

collect_ad_hoc_poll_request(*request_id: int*) → StrDict

Collects a previously-issued ad-hoc polling request, if it has been processed.

Parameters
request_id (*int*) – The unique Id number of the ad-hoc request. This was returned from the simulator upon sending the ad-hoc polling request.

Returns
The readings data.

Return type
StrDict

static draw_bounding_boxes(*bounding_boxes: List[StrDict]*, *colour: PIL.Image.Image*, *width: int = 3*, *font: str = 'arial.ttf'*, *font_size: int = 14*) → PIL.Image.Image

Draws the given list of bounding boxes onto the given image. The boxes are drawn with the given width of outlines in pixels and the given font and size configuration. NOTE: The given image is not directly modified and the boxes are drawn onto a copy.

Parameters

- **bounding_boxes** (*List[StrDict]*) – List of bounding boxes to draw.
- **colour** (*PIL.Image.Image*) – The image to draw the bounding boxes on.
- **width** (*int*) – The width of bounding box outlines in pixels.
- **font** (*str*) – A string specifying the font which bounding box labels will have.
- **font_size** (*int*) – The font size used when drawing labels.

Returns
An `Image` that is a copy of the given image with bounding boxes drawn onto it.

Return type
PIL.Image.Image

static export_bounding_boxes_xml(*bounding_boxes: List[StrDict]*, *folder: str | None = None*, *filename: str | None = None*, *path: str | None = None*, *database: str | None = None*, *size: Int3 | None = None*) → str

Exports the given list of bounding boxes to the Pascal-VOC XML standard. Additional properties to this function correspond to tags in the Pascal-VOC standard.

Parameters

- **bounding_boxes** (*List[StrDict]*) – The list of bounding boxes to export.
- **folder** (*str | None*) – Contents of the 'folder' tag.
- **filename** (*str | None*) – Contents of the 'filename' tag.
- **path** (*str | None*) – Contents of the 'path' tag.
- **database** (*str | None*) – Contents of the 'database' tag.
- **size** (*Int3 | None*) – Contents of the 'size tag. It's expected to be a tuple of the image width, height, and depth.

Returns
XML string encoding of the given list of bounding boxes according to Pascal-VOC.

Return type
str

static extract_bounding_boxes(*semantic_image: PIL.Image.Image*, *instance_image: PIL.Image.Image*, *classes: StrDict*) → List[StrDict]

Analyzes the given semantic annotation and instance annotation images for its object bounding boxes. The identified objects are returned as a list of dictionaries containing their bounding box corners, class of object according to the corresponding colour in the semantic annotations and the given class mapping, and the colour of the object in the instance annotation.

Parameters

- **semantic_image** (*PIL.Image.Image*) – The image containing semantic annotation information.
- **instance_image** (*PIL.Image.Image*) – The image containing instance annotation information.
- **classes** (*StrDict*) – A mapping of colours to their class names to identify object types based on the semantic annotation information. The keys in this dictionary are the respective colours expressed as a 24-bit integer, i.e. [r * 256^2 + g * 256 + b].

Returns

'bbox': [min_x, min_y, max_x, max_y], 'color': [233, 11, 15], 'class': ['CAR'], where min_x, min_y, max_x, max_y mark the corners of the bounding box, colour contains the RGB colour of the object in the instance annotations, and class the object type identified through the given class mapping.

Return type

A list of bounding boxes specified as dictionaries. Example

get_direction() → Float3

Gets the current forward direction vector of this sensor.

Returns

The sensor direction.

Return type

Float3

get_full_poll_request() → StrDict

Gets a full camera request (semantic annotation and instance annotation data included). NOTE: this function blocks the simulation until the data request is completed.

Returns

The camera data, as images

Return type

StrDict

get_max_pending_requests() → int

Gets the current 'max pending requests' value for this sensor. This is the maximum number of polling requests which can be issued at one time.

Returns

The max pending requests value.

Return type

int

get_position() → Float3

Gets the current world-space position of this sensor.

Returns
The sensor position.

Return type
Float3

`get_requested_update_time()` → float

Gets the current 'requested update time' value for this sensor.

Returns
The requested update time.

Return type
float

`get_update_priority()` → float

Gets the current 'update priority' value for this sensor, in range [0, 1], with priority going 0 –> 1, highest to lowest.

Returns
The update priority value.

Return type
float

`is_ad_hoc_poll_request_ready`(*request_id: int*) → bool

Checks if a previously-issued ad-hoc polling request has been processed and is ready to collect.

Parameters
request_id (*int*) – The unique Id number of the ad-hoc request. This was returned from the simulator upon sending the ad-hoc polling request.

Returns
A flag which indicates if the ad-hoc polling request is complete.

Return type
bool

`poll()` → Dict[str, Image.Image | None]

Gets the most-recent readings for this sensor as processed images. Note: if this sensor was created with a negative update rate, then there may have been no readings taken.

Returns

A dictionary with the values as processed images and the following keys

- `colour`: The colour data.
- `annotation`: The semantic annotation data.
- `depth`: The depth camera data.

Return type
Dict[str, Image.Image | None]

`poll_raw()` → Dict[str, bytes | None]

Gets the most-recent readings for this sensor as unprocessed bytes. Note: if this sensor was created with a negative update rate, then there may have been no readings taken.

Returns

A dictionary with values being the unprocessed bytes representing the RGBA data from the sensors and the following keys

- `colour`: The colour data.
- `annotation`: The semantic annotation data.
- `depth`: The depth camera data.

Return type
Dict[str, bytes | None]

poll_shmem_annotation()

poll_shmem_colour()

poll_shmem_depth()

remove() → None

Removes this sensor from the simulation.

Return type
None

send_ad_hoc_poll_request() → int

Sends an ad-hoc polling request to the simulator. This will be executed by the simulator immediately, but will take time to process, so the result can be queried after some time has passed. To check if it has been processed, we first call the is_ad_hoc_poll_request_ready() function, then call the collect_ad_hoc_poll_request() function to retrieve the sensor reading.

Returns
A unique Id number for the ad-hoc request.

Return type
int

set_direction(*dir: Float3*) → None

Sets the current forward direction vector of this sensor.

Parameters
dir (*Float3*) – The new forward direction vector.

Return type
None

set_max_pending_requests(*max_pending_requests: int*) → None

Sets the current 'max pending requests' value for this sensor. This is the maximum number of polling requests which can be issued at one time.

Parameters
max_pending_requests (*int*) – The new max pending requests value.

Return type
None

set_position(*pos: Float3*) → None

Sets the current world-space position for this sensor.

Parameters
pos (*Float3*) – The new position.

Return type
None

set_requested_update_time(*requested_update_time: float*) → None

Sets the current 'requested update time' value for this sensor.

Parameters
requested_update_time (*float*) – The new requested update time.

Return type
None

set_up(*up: Float3*) → None

Sets the current up vector of this sensor.

Parameters

- **pos** – The new up vector.
- **up** (*Float3*) –

Return type
None

set_update_priority(*update_priority: float*) → None

Sets the current 'update priority' value for this sensor, in range [0, 1], with priority going 0 –> 1, , highest to lowest.

Parameters
update_priority (*float*) – The new update priority value.

Return type
None

stream() → Dict[str, Image.Image | None]

Gets the most-recent readings for this sensor as processed images without sending a request to the simulator. Can only be called in the case that the Camera sensor was constructed with `is_streaming=True`. Note: if this sensor was created with a negative update rate, then there may have been no readings taken.

Returns

A dictionary with the values as processed images and the following keys

- `colour`: The colour data.
- `annotation`: The semantic annotation data.
- `depth`: The depth camera data.

Return type
Dict[str, Image.Image | None]

stream_annotation(*size*)

stream_colour(*size*)

stream_depth(*size*)

stream_raw() → Dict[str, bytes]

Gets the most-recent readings for this sensor as unprocessed bytes without sending a request to the simulator. Can only be called in the case that the Camera sensor was constructed with `is_streaming=True`. Note: if this sensor was created with a negative update rate, then there may have been no readings taken.

Returns

A dictionary with values being the unprocessed bytes representing the RGBA data from the sensors and the following keys

- `colour`: The colour data.
- `annotation`: The semantic annotation data.
- `depth`: The depth camera data.

Return type
Dict[str, bytes]

world_point_to_pixel(*point: Float3*) → Int2

Converts a 3D point in world space to the 2D pixel coordinate at which it is represented on this camera. NOTE: The pixel does not have to actually be visible on the camera image itself in order to retrieve a value; it can be obscured by geometry which is closer, or it can be run without respect to the near and far plane values of the camera.

Parameters
point (*Float3*) – The given 3D point, in world space coordinates.

Returns
The 2D pixel value which represents the given 3D point, on this camera.

Return type
Int2

Lidar

class beamngpy.sensors.**Lidar**(*name: str*, *bng:* BeamNGpy, *vehicle:* Vehicle | *None = None*, *requested_update_time: float = 0.1*, *update_priority: float = 0.0*, *pos: Float3 = (0, 0, 1.7)*, *dir: Float3 = (0, -1, 0)*, *up: Float3 = (0, 0, 1)*, *vertical_resolution: int = 64*, *vertical_angle: float = 26.9*, *rays_per_second: float = 2200000*, *frequency: float = 20*, *horizontal_angle: float = 360*, *max_distance: float = 120*, *is_using_shared_memory: bool = True*, *is_visualised: bool = True*, *is_streaming: bool = False*, *is_annotated: bool = False*, *is_static: bool = False*, *is_snapping_desired: bool = False*, *is_force_inside_triangle: bool = False*)

An interactive, automated LiDAR sensor, which produces regular LiDAR point clouds, ready for further processing. This sensor can be attached to a vehicle, or can be fixed to a position in space. The dir and up parameters are used to set the local coordinate system. A requested update rate can be provided, to tell the simulator how often to read measurements for this sensor. If a negative value is provided, the sensor will not update automatically at all. However, ad-hoc polling requests can be sent at any time, even for non-updating sensors.

Parameters

- **name** (*str*) – A unique name for this LiDAR sensor.
- **bng** (BeamNGpy) – The BeamNGpy instance, with which to communicate to the simulation.
- **vehicle** (Vehicle | *None*) – The vehicle to which this sensor should be attached, if any.
- **requested_update_time** (*float*) – The time which should pass between sensor reading updates, in seconds. This is just a suggestion to the manager.
- **update_priority** (*float*) – The priority which the sensor should ask for new readings. lowest -> 0, highest -> 1.
- **pos** (*Float3*) – (X, Y, Z) coordinate triplet specifying the position of the sensor, in world space.
- **dir** (*Float3*) – (X, Y, Z) Coordinate triplet specifying the forward direction of the sensor.

- **up** (*Float3*) – (X, Y, Z) Coordinate triplet specifying the up direction of the sensor.
- **vertical_resolution** (*int*) – The vertical resolution of this LiDAR sensor.
- **vertical_angle** (*float*) – The vertical angle of this LiDAR sensor, in degrees.
- **rays_per_second** (*float*) – The number of LiDAR rays per second which this sensor should emit.
- **frequency** (*float*) – The frequency of this LiDAR sensor.
- **horizontal_angle** (*float*) – The horizontal angle of this LiDAR sensor.
- **max_distance** (*float*) – The maximum distance which this LiDAR sensor will detect, in metres.
- **is_using_shared_memory** (*bool*) – A flag which indicates if we should use shared memory to send/recieve the sensor readings data.
- **is_visualised** (*bool*) – A flag which indicates if this LiDAR sensor should appear visualised or not.
- **is_streaming** (*bool*) – Whether or not to stream the data directly to shared memory (no poll required, for efficiency - BeamNGpy won't block.)
- **is_annotated** (*bool*) – A flag which indicates if this LiDAR sensor should return annotation data instead of distance data.
- **is_static** (*bool*) – A flag which indicates whether this sensor should be static (fixed position), or attached to a vehicle.
- **is_snapping_desired** (*bool*) – A flag which indicates whether or not to snap the sensor to the nearest vehicle triangle (not used for static sensors).
- **is_force_inside_triangle** (*bool*) – A flag which indicates if the sensor should be forced inside the nearest vehicle triangle (not used for static sensors).

collect_ad_hoc_poll_request(*request_id: int*) → StrDict

Collects a previously-issued ad-hoc polling request, if it has been processed.

Parameters
request_id (*int*) – The unique Id number of the ad-hoc request. This was returned from the simulator upon sending the ad-hoc polling request.

Returns
A dictionary containing the LiDAR point cloud and colour data.

Return type
StrDict

get_direction() → Float3

Gets the current direction vector of this sensor.

Returns
The sensor direction.

Return type
Float3

get_is_annotated() → bool

Gets a flag which indicates if this LiDAR sensor is annotated or not.

Returns
A flag which indicates if this LiDAR sensor is annotated or not.

Return type
bool

`get_is_visualised`() → bool

Gets a flag which indicates if this LiDAR sensor is visualised or not.

Returns
A flag which indicates if this LiDAR sensor is visualised or not.

Return type
bool

`get_max_pending_requests`() → int

Gets the current 'max pending requests' value for this sensor. This is the maximum number of polling requests which can be issued at one time.

Returns
The max pending requests value.

Return type
int

`get_position`() → Float3

Gets the current world-space position of this sensor.

Returns
The sensor position.

Return type
Float3

`get_requested_update_time`() → float

Gets the current 'requested update time' value for this sensor.

Returns
The requested update time.

Return type
float

`get_update_priority`() → float

Gets the current 'update priority' value for this sensor, in range [0, 1], with priority going 0 –> 1, highest to lowest.

Returns
The update priority value.

Return type
float

`is_ad_hoc_poll_request_ready`(*request_id: int*) → bool

Checks if a previously-issued ad-hoc polling request has been processed and is ready to collect.

Parameters
request_id (*int*) – The unique Id number of the ad-hoc request. This was returned from the simulator upon sending the ad-hoc polling request.

Returns
A flag which indicates if the ad-hoc polling request is complete.

Return type
bool

poll() → StrDict

Gets the most-recent readings for this sensor. Note: if this sensor was created with a negative update rate, then there may have been no readings taken.

Returns
The LiDAR point cloud and colour data.

Return type
StrDict

remove() → None

Removes this sensor from the simulation.

Return type
None

send_ad_hoc_poll_request() → int

Sends an ad-hoc polling request to the simulator. This will be executed by the simulator immediately, but will take time to process, so the result can be queried after some time has passed. To check if it has been processed, we first call the is_ad_hoc_poll_request_ready() function, then call the collect_ad_hoc_poll_request() function to retrieve the sensor reading.

Returns
A unique Id number for the ad-hoc request.

Return type
int

set_is_annotated(*is_annotated: bool*) → None

Sets whether this LiDAR sensor is to be annotated or not. This means it will return annotation data instead of distances.

Parameters
is_annotated (*bool*) – A flag which indicates if this LiDAR sensor is to be annotated or not.

Return type
None

set_is_visualised(*is_visualised: bool*) → None

Sets whether this LiDAR sensor is to be visualised or not.

Parameters
is_visualised (*bool*) – A flag which indicates if this LiDAR sensor is to be visualised or not.

Return type
None

set_max_pending_requests(*max_pending_requests: int*) → None

Sets the current 'max pending requests' value for this sensor. This is the maximum number of polling requests which can be issued at one time.

Parameters
max_pending_requests (*int*) – The new max pending requests value.

Return type
None

set_requested_update_time(*requested_update_time: float*) → None

Sets the current 'requested update time' value for this sensor.

Parameters

- **update_priority** – The new requested update time.
- **requested_update_time** (*float*) –

Return type
None

set_update_priority(*update_priority: float*) → None

Sets the current 'update priority' value for this sensor, in range [0, 1], with priority going 0 –> 1, , highest to lowest.

Parameters
update_priority (*float*) – The new update priority value.

Return type
None

stream() → StrDict

Gets the streamed LiDAR point cloud data from the associated shared memory location.

Returns
The LiDAR point cloud data.

Return type
StrDict

Ultrasonic Sensor

class beamngpy.sensors.**Ultrasonic**(*name: str*, *bng:* BeamNGpy, *vehicle:* Vehicle | *None = None*, *requested_update_time: float = 0.1*, *update_priority: float = 0.0*, *pos: Float3 = (0, 0, 1.7)*, *dir: Float3 = (0, -1, 0)*, *up: Float3 = (0, 0, 1)*, *resolution: Int2 = (200, 200)*, *field_of_view_y: float = 5.7*, *near_far_planes: Float2 = (0.1, 5.1)*, *range_roundness: float = -1.15*, *range_cutoff_sensitivity: float = 0.0*, *range_shape: float = 0.3*, *range_focus: float = 0.376*, *range_min_cutoff: float = 0.1*, *range_direct_max_cutoff: float = 5.0*, *sensitivity: float = 3.0*, *fixed_window_size: float = 10*, *is_visualised: bool = True*, *is_streaming: bool = False*, *is_static: bool = False*, *is_snapping_desired: bool = False*, *is_force_inside_triangle: bool = False*)

An interactive, automated ultrasonic sensor, which produces regular distance measurements, ready for further processing. This sensor can be attached to a vehicle, or can be fixed to a position in space. The dir and up parameters are used to set the local coordinate system. A requested update rate can be provided, to tell the simulator how often to read measurements for this sensor. If a negative value is provided, the sensor will not update automatically at all. However, ad-hoc polling requests can be sent at any time, even for non-updating sensors.

Parameters

- **name** (*str*) – A unique name for this ultrasonic sensor.
- **bng** (BeamNGpy) – The BeamNGpy instance, with which to communicate to the simulation.
- **vehicle** (Vehicle | *None*) – The vehicle to which this sensor should be attached, if any.

- **requested_update_time** (*float*) – The time which should pass between sensor reading updates, in seconds. This is just a suggestion to the manager.
- **update_priority** (*float*) – The priority which the sensor should ask for new readings. lowest -> 0, highest -> 1.
- **pos** (*Float3*) – (X, Y, Z) Coordinate triplet specifying the position of the sensor, in world space.
- **dir** (*Float3*) – (X, Y, Z) Coordinate triplet specifying the forward direction of the sensor.
- **up** (*Float3*) – (X, Y, Z) Coordinate triplet specifying the up direction of the sensor.
- **size** – (X, Y) The resolution of the sensor (the size of the depth buffer image in the distance measurement computation).
- **field_of_view_y** (*float*) – The sensor vertical field of view parameters.
- **near_far_planes** (*Float2*) – (X, Y) The sensor near and far plane distances.
- **range_roundness** (*float*) – the general roudness of the ultrasonic sensor range-shape. Can be negative.
- **range_cutoff_sensitivity** (*float*) – a cutoff sensitivity parameter for the ultrasonic sensor range-shape.
- **range_shape** (*float*) – the shape of the ultrasonic sensor range-shape in [0, 1], from conical to circular.
- **range_focus** (*float*) – the focus parameter for the ultrasonic sensor range-shape.
- **range_min_cutoff** (*float*) – the minimum cut-off distance for the ultrasonic sensor range-shape. Nothing closer than this will be detected.
- **range_direct_max_cutoff** (*float*) – the maximum cut-off distance for the ultrasonic sensor range-shape. This parameter is a hard cutoff - nothing further than this will be detected, although other parameters can also control the max distance.
- **sensitivity** (*float*) – an ultrasonic sensor sensitivity parameter.
- **fixed_window_size** (*float*) – an ultrasonic sensor sensitivity parameter.
- **is_visualised** (*bool*) – Whether or not to render the ultrasonic sensor points in the simulator.
- **is_streaming** (*bool*) – Whether or not to stream the data directly to shared memory (no poll required, for efficiency - BeamNGpy won't block.)
- **is_static** (*bool*) – A flag which indicates whether this sensor should be static (fixed position), or attached to a vehicle.
- **is_snapping_desired** (*bool*) – A flag which indicates whether or not to snap the sensor to the nearest vehicle triangle (not used for static sensors).
- **is_force_inside_triangle** (*bool*) – A flag which indicates if the sensor should be forced inside the nearest vehicle triangle (not used for static sensors).
- **resolution** (*Int2*) –

collect_ad_hoc_poll_request(*request_id: int*) → StrDict

Collects a previously-issued ad-hoc polling request, if it has been processed.

Parameters

request_id (*int*) – The unique Id number of the ad-hoc request. This was returned from the simulator upon sending the ad-hoc polling request.

Returns
The readings data.

Return type
StrDict

`get_direction()` → Float3

Gets the current direction vector of this sensor.

Returns
The sensor direction.

Return type
Float3

`get_is_visualised()` → bool

Gets a flag which indicates if this ultrasonic sensor is visualised or not.

Returns
A flag which indicates if this ultrasonic sensor is visualised or not.

Return type
bool

`get_max_pending_requests()` → int

Gets the current 'max pending requests' value for this sensor. This is the maximum number of polling requests which can be issued at one time.

Returns
The max pending requests value.

Return type
int

`get_position()` → Float3

Gets the current world-space position of this sensor.

Returns
The sensor position.

Return type
Float3

`get_requested_update_time()` → float

Gets the current 'requested update time' value for this sensor.

Returns
The requested update time.

Return type
(float)

`get_update_priority()` → float

Gets the current 'update priority' value for this sensor, in range [0, 1], with priority going 0 –> 1, highest to lowest.

Returns
The update priority value.

Return type
float

is_ad_hoc_poll_request_ready(*request_id: int*) → bool

Checks if a previously-issued ad-hoc polling request has been processed and is ready to collect.

Parameters
request_id (*int*) – The unique Id number of the ad-hoc request. This was returned from the simulator upon sending the ad-hoc polling request.

Returns
A flag which indicates if the ad-hoc polling request is complete.

Return type
bool

poll() → StrDict

Gets the most-recent readings for this sensor. Note: if this sensor was created with a negative update rate, then there may have been no readings taken.

Returns
A dictionary containing the distance measurement and the window (min and mix values) in which it was computed.

Return type
StrDict

remove()

Removes this sensor from the simulation.

send_ad_hoc_poll_request() → int

Sends an ad-hoc polling request to the simulator. This will be executed by the simulator immediately, but will take time to process, so the result can be queried after some time has passed. To check if it has been processed, we first call the is_ad_hoc_poll_request_ready() function, then call the collect_ad_hoc_poll_request() function to retrieve the sensor reading.

Returns
A unique Id number for the ad-hoc request.

Return type
int

set_is_visualised(*is_visualised: bool*) → None

Sets whether this ultrasonic sensor is to be visualised or not.

Parameters
is_visualised (*bool*) – A flag which indicates if this ultrasonic sensor is to be visualised or not.

Return type
None

set_max_pending_requests(*max_pending_requests: int*) → None

Sets the current 'max pending requests' value for this sensor. This is the maximum number of polling requests which can be issued at one time.

Parameters
max_pending_requests (*int*) – The new max pending requests value.

Return type
None

set_requested_update_time(*requested_update_time: float*)

Sets the current 'requested update time' value for this sensor.

Parameters
requested_update_time (*float*) – The new requested update time.

set_update_priority(*update_priority: float*) → None

Sets the current 'update priority' value for this sensor, in range [0, 1], with priority going 0 –> 1, , highest to lowest.

Parameters
update_priority (*float*) – The new update priority

Return type
None

stream()

Gets the latest Ultrasonic distance reading from shared memory (which is being streamed directly).

Returns
The latest Ultrasonic distance reading from shared memory.

Powertrain Sensor

class beamngpy.sensors.**PowertrainSensor**(*name: str*, *bng:* BeamNGpy, *vehicle:* Vehicle, *gfx_update_time: float = 0.0*, *physics_update_time: float = 0.01*, *is_send_immediately: bool = False*)

An interactive, automated powertrain sensor, which produces regular readings directly from a vehicle's powertrain. A requested update rate can be provided, to tell the simulator how often to read measurements for this sensor. If a negative value is provided, the sensor will not update automatically at all. However, ad-hoc polling requests can be sent at any time, even for non-updating sensors. We can set this sensor to poll the send data back in two modes: i) immediate mode: data is sent back as soon as it is available (single readings arrive instantly) - this method is suitable when working with tightly-coupled systems requiring fast feedback, or ii) post-processing mode: we can set it to send the data back in bulk on the simulations graphics step - this method is appropriate for the case when the user wishes simply to post-process the data (such as for plotting graphs etc) and is also more efficient. In this case, the returned data will contain all the individual samples which were measured in the simulations physics step, so the data is the same as in mode i); it just arrives later, in bulk.

Parameters

- **name** (*str*) – A unique name for this powertrain sensor.
- **bng** (BeamNGpy) – The BeamNGpy instance, with which to communicate to the simulation.
- **vehicle** (Vehicle) – The vehicle to which this sensor should be attached. Note: a vehicle must be provided for the powertrain sensor.
- **gfx_update_time** (*float*) – The gfx-step time which should pass between sensor reading updates to the user, in seconds.
- **physics_update_time** (*float*) – The physics-step time which should pass between actual sampling the sensor, in seconds.
- **is_send_immediately** (*bool*) – A flag which indicates if the readings should be sent back as soon as available or upon graphics step updates, as bulk.

collect_ad_hoc_poll_request(*request_id: int*) → StrDict

Collects a previously-issued ad-hoc polling request, if it has been processed.

Parameters
request_id (*int*) – The unique Id number of the ad-hoc request. This was returned from the simulator upon sending the ad-hoc polling request.

Returns
The readings data.

Return type
StrDict

is_ad_hoc_poll_request_ready(*request_id: int*) → bool

Checks if a previously-issued ad-hoc polling request has been processed and is ready to collect.

Parameters
request_id (*int*) – The unique Id number of the ad-hoc request. This was returned from the simulator upon sending the ad-hoc polling request.

Returns
A flag which indicates if the ad-hoc polling request is complete.

Return type
bool

poll() → StrDict

Gets the most-recent readings for this sensor. Note: if this sensor was created with a negative update rate, then there may have been no readings taken.

Returns
A dictionary containing the sensor readings data.

Return type
StrDict

remove() → None

Removes this sensor from the simulation.

Return type
None

send_ad_hoc_poll_request() → int

Sends an ad-hoc polling request to the simulator. This will be executed by the simulator immediately, but will take time to process, so the result can be queried after some time has passed. To check if it has been processed, we first call the is_ad_hoc_poll_request_ready() function, then call the collect_ad_hoc_poll_request() function to retrieve the sensor reading.

Returns
A unique Id number for the ad-hoc request.

Return type
int

set_requested_update_time(*requested_update_time: float*) → None

Sets the current 'requested update time' value for this sensor.

Parameters
requested_update_time (*float*) – The new requested update time.

Return type
None

Advanced IMU

class beamngpy.sensors.**AdvancedIMU**(*name: str*, *bng: BeamNGpy*, *vehicle: Vehicle*, *gfx_update_time: float = 0.0*, *physics_update_time: float = 0.01*, *pos: Float3 = (0, 0, 1.7)*, *dir: Float3 = (0, -1, 0)*, *up: Float3 = (0, 0, 1)*, *accel_window_width: float | None = None*, *gyro_window_width: float | None = None*, *accel_frequency_cutoff: float | None = None*, *gyro_frequency_cutoff: float | None = None*, *is_send_immediately: bool = False*, *is_using_gravity: bool = False*, *is_visualised: bool = True*, *is_snapping_desired: bool = False*, *is_force_inside_triangle: bool = False*)

An interactive, automated IMU sensor, which produces regular acceleration and gyroscopic measurements in a local coordinate space. This sensor must be attached to a vehicle; it cannot be fixed to a position in space. The dir and up parameters are used to set the local coordinate system. A requested update rate can be provided, to tell the simulator how often to read measurements for this sensor. If a negative value is provided, the sensor will not update automatically at all. However, ad-hoc polling requests can be sent at any time, even for non-updating sensors.

Parameters

- **name** (*str*) – A unique name for this advanced IMU sensor.
- **bng** (BeamNGpy) – The BeamNGpy instance, with which to communicate to the simulation.
- **vehicle** (Vehicle) – The vehicle to which this sensor should be attached. Note: a vehicle must be provided for the advanced IMU sensor.
- **gfx_update_time** (*float*) – The gfx-step time which should pass between sensor reading updates to the user, in seconds.
- **physics_update_time** (*float*) – The physics-step time which should pass between actual sampling the sensor, in seconds.
- **pos** (*Float3*) – (X, Y, Z) Coordinate triplet specifying the position of the sensor, in world space.
- **dir** (*Float3*) – (X, Y, Z) Coordinate triplet specifying the forward direction of the sensor.
- **up** (*Float3*) – (X, Y, Z) Coordinate triplet specifying the up direction of the sensor.
- **accel_window_width** (*float | None*) – The width of the window used in smoothing the acceleration data, if required.
- **accel_frequency_cutoff** (*float | None*) – The filtering cutoff frequency to be used for acceleration (instead of a window width), if required.
- **gyro_window_width** (*float | None*) – The width of the window used in smoothing the gyroscopic data, if required.
- **gyro_frequency_cutoff** (*float | None*) – The filtering cutoff frequency to be used for gyroscopic (instead of a window width), if required.
- **is_send_immediately** (*bool*) – A flag which indicates if the readings should be sent back as soon as available or upon graphics step updates, as bulk.
- **is_using_gravity** (*bool*) – A flag which indicates whether this sensor should consider acceleration due to gravity in its computations, or not.
- **is_visualised** (*bool*) – Whether or not to render the ultrasonic sensor points in the simulator.

- **is_snapping_desired** (*bool*) – A flag which indicates whether or not to snap the sensor to the nearest vehicle triangle.
- **is_force_inside_triangle** (*bool*) – A flag which indicates if the sensor should be forced inside the nearest vehicle triangle.

collect_ad_hoc_poll_request(*request_id: int*) → StrDict

Collects a previously-issued ad-hoc polling request, if it has been processed.

Parameters
request_id (*int*) – The unique Id number of the ad-hoc request. This was returned from the simulator upon sending the ad-hoc polling request.

Returns
The readings data.

Return type
StrDict

is_ad_hoc_poll_request_ready(*request_id: int*) → bool

Checks if a previously-issued ad-hoc polling request has been processed and is ready to collect.

Parameters
request_id (*int*) – The unique Id number of the ad-hoc request. This was returned from the simulator upon sending the ad-hoc polling request.

Returns
A flag which indicates if the ad-hoc polling request is complete.

Return type
bool

poll() → StrDict

Gets the most-recent readings for this sensor. Note: if this sensor was created with a negative update rate, then there may have been no readings taken.

Returns
A dictionary containing the sensor readings data.

Return type
StrDict

remove() → None

Removes this sensor from the simulation.

Return type
None

send_ad_hoc_poll_request() → int

Sends an ad-hoc polling request to the simulator. This will be executed by the simulator immediately, but will take time to process, so the result can be queried after some time has passed. To check if it has been processed, we first call the is_ad_hoc_poll_request_ready() function, then call the collect_ad_hoc_poll_request() function to retrieve the sensor reading.

Returns
A unique Id number for the ad-hoc request.

Return type
int

set_is_using_gravity(*is_using_gravity: bool*) → None

Sets whether this sensor is to include gravity in the computation or not.

Parameters
is_using_gravity (*bool*) – A flag which indicates if this sensor is to use gravity in the computation or not.

Return type
None

set_is_visualised(*is_visualised: bool*) → None

Sets whether this sensor is to be visualised or not.

Parameters
is_visualised (*bool*) – A flag which indicates if this sensor is to be visualised or not.

Return type
None

set_requested_update_time(*requested_update_time: float*) → None

Sets the current 'requested update time' value for this sensor.

Parameters
requested_update_time (*float*) – The new requested update time.

Return type
None

Radar

class beamngpy.sensors.**Radar**(*name: str*, *bng:* BeamNGpy, *vehicle:* Vehicle | *None = None*, *requested_update_time: float = 0.1*, *update_priority: float = 0.0*, *pos: Float3 = (0, 0, 1.7)*, *dir: Float3 = (0, -1, 0)*, *up: Float3 = (0, 0, 1)*, *range_bins: int = 200*, *azimuth_bins: int = 200*, *vel_bins: int = 200*, *range_min: float = 0.1*, *range_max: float = 100.0*, *vel_min: float = -50.0*, *vel_max: float = 50.0*, *half_angle_deg: float = 30.0*, *resolution: Int2 = (200, 200)*, *field_of_view_y: float = 70*, *near_far_planes: Float2 = (0.1, 150.0)*, *range_roundess: float = -2.0*, *range_cutoff_sensitivity: float = 0.0*, *range_shape: float = 0.23*, *range_focus: float = 0.12*, *range_min_cutoff: float = 0.5*, *range_direct_max_cutoff: float = 150.0*, *is_visualised: bool = True*, *is_streaming: bool = False*, *is_static: bool = False*, *is_snapping_desired: bool = False*, *is_force_inside_triangle: bool = False*)

An interactive, automated RADAR sensor, which produces regular RADAR measurements. This sensor can be attached to a vehicle, or can be fixed to a position in space. The dir and up parameters are used to set the local coordinate system. A requested update rate can be provided, to tell the simulator how often to read measurements for this sensor. If a negative value is provided, the sensor will not update automatically at all. However, ad-hoc polling requests can be sent at any time, even for non-updating sensors.

Parameters

- **name** (*str*) – A unique name for this RADAR sensor.
- **bng** (BeamNGpy) – The BeamNGpy instance, with which to communicate to the simulation.
- **vehicle** (Vehicle | *None*) – The vehicle to which this sensor should be attached, if any.
- **requested_update_time** (*float*) – The time which should pass between sensor reading updates, in seconds. This is just a suggestion to the manager.

- **update_priority** (*float*) – The priority which the sensor should ask for new readings. lowest -> 0, highest -> 1.
- **pos** (*Float3*) – (X, Y, Z) Coordinate triplet specifying the position of the sensor, in world space.
- **dir** (*Float3*) – (X, Y, Z) Coordinate triplet specifying the forward direction of the sensor.
- **up** (*Float3*) – (X, Y, Z) Coordinate triplet specifying the up direction of the sensor.
- **range_bins** (*int*) – The number of bins to use in the range dimension, for RADAR post-processing (the images returned from the simulator).
- **azimuth_bins** (*int*) – The number of bins to use in the azimuth dimension, for RADAR post-processing (PPI plots).
- **vel_bins** (*int*) – The number of bins to use in the velocity dimension, for RADAR post-processing (range-Doppler plots).
- **range_min** (*float*) – The minimum range to display in the post-processing.
- **range_max** (*float*) – The maximum range to display in the post-processing.
- **vel_min** (*float*) – The minimum velocity to display in the post-processing (range-Doppler images), in m/s.
- **vel_max** (*float*) – The maximum velocity to display in the post-processing (range-Doppler images), in m/s.
- **half_angle_deg** (*float*) – On the PPI plot, this is half the azimuthal range (angle between the vertical and cone edge), in degrees.
- **size** – (X, Y) The resolution of the sensor (the size of the depth buffer image in the distance measurement computation).
- **field_of_view_y** (*float*) – The sensor vertical field of view parameter.
- **near_far_planes** (*Float2*) – (X, Y) The sensor near and far plane distances.
- **range_roundness** – the general roudness of the RADAR sensor range-shape. Can be negative.
- **range_cutoff_sensitivity** (*float*) – a cutoff sensitivity parameter for the RADAR sensor range-shape.
- **range_shape** (*float*) – the shape of the RADAR sensor range-shape in [0, 1], from conical to circular.
- **range_focus** (*float*) – the focus parameter for the RADAR sensor range-shape.
- **range_min_cutoff** (*float*) – the minimum cut-off distance for the RADAR sensor range-shape. Nothing closer than this will be detected.
- **range_direct_max_cutoff** (*float*) – the maximum cut-off distance for the RADAR sensor range-shape. This parameter is a hard cutoff - nothing further than this will be detected, although other parameters can also control the max distance.
- **is_visualised** (*bool*) – Whether or not to render the RADAR sensor points in the simulator.
- **is_streaming** (*bool*) – Whether or not to stream the data directly to shared memory (no poll required, for efficiency - BeamNGpy won't block.)
- **is_static** (*bool*) – A flag which indicates whether this sensor should be static (fixed position), or attached to a vehicle.

- **is_snapping_desired** (*bool*) – A flag which indicates whether or not to snap the sensor to the nearest vehicle triangle (not used for static sensors).
- **is_force_inside_triangle** (*bool*) – A flag which indicates if the sensor should be forced inside the nearest vehicle triangle (not used for static sensors).
- **resolution** (*Int2*) –
- **range_roundess** (*float*) –

collect_ad_hoc_poll_request(*request_id: int*)

Collects a previously-issued ad-hoc polling request, if it has been processed.

Parameters
request_id (*int*) – The unique Id number of the ad-hoc request. This was returned from the simulator upon sending the ad-hoc polling request.

Returns
The readings data.

get_direction() → Float3

Gets the current direction vector of this sensor.

Returns
The sensor direction.

Return type
Float3

get_max_pending_requests() → int

Gets the current 'max pending requests' value for this sensor. This is the maximum number of polling requests which can be issued at one time.

Returns
The max pending requests value.

Return type
int

get_position() → Float3

Gets the current world-space position of this sensor.

Returns
The sensor position.

Return type
Float3

get_ppi()

Gets the latest RADAR PPI (plan position indicator) image from shared memory.

Returns
The latest RADAR PPI (plan position indicator) image from shared memory.

get_range_doppler()

Gets the latest RADAR Range-Doppler image from shared memory.

Returns
The latest RADAR Range-Doppler image from shared memory.

`get_requested_update_time()` → float

Gets the current 'requested update time' value for this sensor.

Returns
The requested update time.

Return type
(float)

`get_update_priority()` → float

Gets the current 'update priority' value for this sensor, in range [0, 1], with priority going 0 –> 1, highest to lowest.

Returns
The update priority value.

Return type
float

`is_ad_hoc_poll_request_ready`(*request_id: int*) → bool

Checks if a previously-issued ad-hoc polling request has been processed and is ready to collect.

Parameters
request_id (*int*) – The unique Id number of the ad-hoc request. This was returned from the simulator upon sending the ad-hoc polling request.

Returns
A flag which indicates if the ad-hoc polling request is complete.

Return type
bool

`plot_data`(*readings_data*, *resolution*, *field_of_view_y*, *range_min*, *range_max*, *range_bins: int = 200*, *azimuth_bins: int = 200*)

Plot the RADAR readings data. The data plots are: B-Scope, PPI (Plan Position Indicator), RCS (Radar Cross Section), and SNR (Signal-to-Noise Ratio). The data is used to populate bins, where each bin represents one pixel on the images, and contains a weighted average of the data at that location. If data exists outside of the given distance/angle ranges, it will be snapped to the nearest bin, so this should be avoided by providing accurate limits for these.

Parameters

- **readings_data** – The readings data structure obtained from polling the RADAR sensor.
- **resolution** – (X, Y) The resolution of the sensor (the size of the depth buffer image in the distance measurement computation).
- **field_of_view_y** – The vertical field of view of the RADAR, in degrees.
- **range_min** – The minimum range of the sensor, in metres.
- **range_max** – The maximum range of the sensor, in metres.
- **range_bins** (*int*) – The number of bins to use for the range dimension, in the data plots.
- **azimuth_bins** (*int*) – The number of bins to use for the azimuth dimension, in the data plots.

`plot_velocity_data`(*velocity_data*, *resolution*, *field_of_view_y*, *range_min: float = 0.0*, *range_max: float = 100.0*, *range_bins: int = 200*, *azimuth_bins: int = 200*)

Plot the RADAR Doppler velocities.

Parameters

- **velocity_data** – The 2D velocity array obtained from the RADAR sensor.
- **resolution** – (X, Y) The resolution of the sensor (the size of the depth buffer image in the distance measurement computation).
- **field_of_view_y** – The vertical field of view of the RADAR, in degrees.
- **range_min** (*float*) – The minimum range of the sensor, in metres.
- **range_max** (*float*) – The maximum range of the sensor, in metres.
- **range_bins** (*int*) – The number of bins to use for the range dimension, in the data plots.
- **azimuth_bins** (*int*) – The number of bins to use for the azimuth dimension, in the data plots.

poll()

Gets the most-recent raw readings for this RADAR sensor, if they exist. Note: if this sensor was created with a negative update rate, then there may have been no readings taken.

Returns
A 6D point cloud of raw RADAR data, where each entry is (range, doppler velocity, azimuth angle, elevation angle, radar cross section, signal to noise ratio).

remove()

Removes this sensor from the simulation.

send_ad_hoc_poll_request() → int

Sends an ad-hoc polling request to the simulator. This will be executed by the simulator immediately, but will take time to process, so the result can be queried after some time has passed. To check if it has been processed, we first call the is_ad_hoc_poll_request_ready() function, then call the collect_ad_hoc_poll_request() function to retrieve the sensor reading.

Returns
A unique Id number for the ad-hoc request.

Return type
int

set_max_pending_requests(*max_pending_requests: int*) → None

Sets the current 'max pending requests' value for this sensor. This is the maximum number of polling requests which can be issued at one time.

Parameters
max_pending_requests (*int*) – The new max pending requests value.

Return type
None

set_requested_update_time(*requested_update_time: float*)

Sets the current 'requested update time' value for this sensor.

Parameters
requested_update_time (*float*) – The new requested update time.

set_update_priority(*update_priority: float*) → None

Sets the current 'update priority' value for this sensor, in range [0, 1], with priority going 0 –> 1, , highest to lowest.

Parameters
update_priority (*float*) – The new update priority

Return type
None

`stream_ppi()`

Gets the latest RADAR PPI image from shared memory (which is being streamed directly).

Returns
The latest RADAR PPI image from shared memory.

`stream_range_doppler()`

Gets the latest RADAR Range-Doppler image from shared memory (which is being streamed directly).

Returns
The latest RADAR Range-Doppler image from shared memory.

Ideal Radar

`class beamngpy.sensors.IdealRadar(name: str, bng: BeamNGpy, vehicle: Vehicle, gfx_update_time: float = 0.0, physics_update_time: float = 0.01, is_send_immediately: bool = False)`

This automated sensor provides the user with data relating to vehicles within a close proximity to its position. Quantities such as velocity and acceleration are available for these vehicles, in a reference frame local the sensor. These sensors can be attached to any vehicle, or to any fixed point on the map.

Parameters

- **name** (*str*) – A unique name for this ideal RADAR sensor.
- **bng** (BeamNGpy) – The BeamNGpy instance, with which to communicate to the simulation.
- **vehicle** (Vehicle) – The vehicle to which this sensor should be attached. Note: a vehicle must be provided for the ideal RADAR sensor.
- **gfx_update_time** (*float*) – The gfx-step time which should pass between sensor reading updates to the user, in seconds.
- **physics_update_time** (*float*) – The physics-step time which should pass between actual sampling the sensor, in seconds.
- **is_send_immediately** (*bool*) – A flag which indicates if the readings should be sent back as soon as available or upon graphics step updates, as bulk.

`collect_ad_hoc_poll_request(request_id: int)` → StrDict

Collects a previously-issued ad-hoc polling request, if it has been processed.

Parameters
request_id (*int*) – The unique Id number of the ad-hoc request. This was returned from the simulator upon sending the ad-hoc polling request.

Returns
The readings data.

Return type
StrDict

`is_ad_hoc_poll_request_ready(request_id: int)` → bool

Checks if a previously-issued ad-hoc polling request has been processed and is ready to collect.

Parameters
request_id (*int*) – The unique Id number of the ad-hoc request. This was returned from the simulator upon sending the ad-hoc polling request.

Returns
A flag which indicates if the ad-hoc polling request is complete.

Return type
bool

poll() → StrDict

Gets the most-recent readings for this sensor. Note: if this sensor was created with a negative update rate, then there may have been no readings taken.

Returns
A dictionary containing the sensor readings data.

Return type
StrDict

remove() → None

Removes this sensor from the simulation.

Return type
None

send_ad_hoc_poll_request() → int

Sends an ad-hoc polling request to the simulator. This will be executed by the simulator immediately, but will take time to process, so the result can be queried after some time has passed. To check if it has been processed, we first call the is_ad_hoc_poll_request_ready() function, then call the collect_ad_hoc_poll_request() function to retrieve the sensor reading.

Returns
A unique Id number for the ad-hoc request.

Return type
int

set_requested_update_time(*requested_update_time: float*) → None

Sets the current 'requested update time' value for this sensor.

Parameters
requested_update_time (*float*) – The new requested update time.

Return type
None

Mesh Sensor

class beamngpy.sensors.**Mesh**(*name: str*, *bng:* BeamNGpy, *vehicle:* Vehicle, *gfx_update_time: float = 0.0*, *groups_list=[]*, *is_track_beams=True*)

An automated 'sensor' to retrieve mesh data in real time.

Parameters

- **name** (*str*) – A unique name for this mesh sensor.
- **bng** (BeamNGpy) – The BeamNGpy instance, with which to communicate to the simulation.

- **vehicle** (Vehicle) – The vehicle to which this sensor should be attached. Note: a vehicle must be provided for the mesh sensor.
- **gfx_update_time** (*float*) – The gfx-step time which should pass between sensor reading updates to the user, in seconds.
- **groups_list** – A list of mesh groups which are to be considered. Optional. If empty, we include all mesh nodes/beams.
- **is_track_beams** – A flag which indicates if we should keep updating the beam to node maps. This will track broken beams over time, but is slower.

collect_ad_hoc_poll_request(*request_id: int*) → StrDict

Collects a previously-issued ad-hoc polling request, if it has been processed.

Parameters
request_id (*int*) – The unique Id number of the ad-hoc request. This was returned from the simulator upon sending the ad-hoc polling request.

Returns
The readings data.

Return type
StrDict

compute_beam_line_segments()

force_direction_plot(*data*)

force_distribution_plot(*data*)

get_node_positions()

is_ad_hoc_poll_request_ready(*request_id: int*) → bool

Checks if a previously-issued ad-hoc polling request has been processed and is ready to collect.

Parameters
request_id (*int*) – The unique Id number of the ad-hoc request. This was returned from the simulator upon sending the ad-hoc polling request.

Returns
A flag which indicates if the ad-hoc polling request is complete.

Return type
bool

mass_distribution_plot(*data*)

mesh_plot()

poll() → StrDict

Gets the most-recent readings for this sensor. Note: if this sensor was created with a negative update rate, then there may have been no readings taken.

Returns
A dictionary containing the sensor readings data.

Return type
StrDict

remove() → None

Removes this sensor from the simulation.

Return type
None

send_ad_hoc_poll_request() → int

Sends an ad-hoc polling request to the simulator. This will be executed by the simulator immediately, but will take time to process, so the result can be queried after some time has passed. To check if it has been processed, we first call the is_ad_hoc_poll_request_ready() function, then call the collect_ad_hoc_poll_request() function to retrieve the sensor reading.

Returns
A unique Id number for the ad-hoc request.

Return type
int

set_requested_update_time(*requested_update_time: float*) → None

Sets the current 'requested update time' value for this sensor.

Parameters
requested_update_time (*float*) – The new requested update time.

Return type
None

velocity_direction_plot(*data*)

velocity_distribution_plot(*data*)

GPS

class beamngpy.sensors.**GPS**(*name: str*, *bng:* BeamNGpy, *vehicle:* Vehicle, *gfx_update_time: float = 0.0*, *physics_update_time: float = 0.01*, *pos: Float3 = (0, 0, 1.7)*, *ref_lon: float = 0.0*, *ref_lat: float = 0.0*, *is_send_immediately: bool = False*, *is_visualised: bool = True*, *is_snapping_desired: bool = False*, *is_force_inside_triangle: bool = False*)

This automated sensor provides GPS readings (position) in spherical coordinates (lattitude, longitude). It can be attached to any point on or relative to the vehicle.

Parameters

- **name** (*str*) – A unique name for this ideal RADAR sensor.
- **bng** (BeamNGpy) – The BeamNGpy instance, with which to communicate to the simulation.
- **vehicle** (Vehicle) – The vehicle to which this sensor should be attached.
- **gfx_update_time** (*float*) – The gfx-step time which should pass between sensor reading updates to the user, in seconds.
- **physics_update_time** (*float*) – The physics-step time which should pass between actual sampling the sensor, in seconds.
- **ref_lon** (*float*) – A reference longitude value, which tells the sensor where the origin point of the map is on the (longitude, lattitude) system.
- **ref_lat** (*float*) – A reference lattitude value, which tells the sensor where the origin point of the map is on the (longitude, lattitude) system.

- **is_send_immediately** (*bool*) – A flag which indicates if the readings should be sent back as soon as available or upon graphics step updates, as bulk.
- **is_visualised** (*bool*) – Whether or not to render the ultrasonic sensor points in the simulator.
- **is_snapping_desired** (*bool*) – A flag which indicates whether or not to snap the sensor to the nearest vehicle triangle.
- **is_force_inside_triangle** (*bool*) – A flag which indicates if the sensor should be forced inside the nearest vehicle triangle.
- **pos** (*Float3*) –

collect_ad_hoc_poll_request(*request_id: int*) → StrDict

Collects a previously-issued ad-hoc polling request, if it has been processed.

Parameters
request_id (*int*) – The unique Id number of the ad-hoc request. This was returned from the simulator upon sending the ad-hoc polling request.

Returns
The readings data.

Return type
StrDict

is_ad_hoc_poll_request_ready(*request_id: int*) → bool

Checks if a previously-issued ad-hoc polling request has been processed and is ready to collect.

Parameters
request_id (*int*) – The unique Id number of the ad-hoc request. This was returned from the simulator upon sending the ad-hoc polling request.

Returns
A flag which indicates if the ad-hoc polling request is complete.

Return type
bool

poll() → StrDict

Gets the most-recent readings for this sensor. Note: if this sensor was created with a negative update rate, then there may have been no readings taken.

Returns
A dictionary containing the sensor readings data.

Return type
StrDict

remove() → None

Removes this sensor from the simulation.

Return type
None

send_ad_hoc_poll_request() → int

Sends an ad-hoc polling request to the simulator. This will be executed by the simulator immediately, but will take time to process, so the result can be queried after some time has passed. To check if it has been processed, we first call the is_ad_hoc_poll_request_ready() function, then call the collect_ad_hoc_poll_request() function to retrieve the sensor reading.

Returns
A unique Id number for the ad-hoc request.

Return type
int

`set_is_visualised`(*is_visualised: bool*) → None

Sets whether this sensor is to be visualised or not.

Parameters
is_visualised (*bool*) – A flag which indicates if this sensor is to be visualised or not.

Return type
None

`set_requested_update_time`(*requested_update_time: float*) → None

Sets the current 'requested update time' value for this sensor.

Parameters
requested_update_time (*float*) – The new requested update time.

Return type
None

Roads Sensor

`class beamngpy.sensors.RoadsSensor`(*name: str*, *bng:* BeamNGpy, *vehicle:* Vehicle, *gfx_update_time: float = 0.0*, *physics_update_time: float = 0.01*, *is_send_immediately: bool = False*)

A sensor which gives geometric and semantic data of the road; this data is the parametric cubic equations for the left and right roadedge and the centerline, as well as 4 points of the centerline.

Parameters

- **name** (*str*) – A unique name for this roads sensor.
- **bng** (BeamNGpy) – The BeamNGpy instance, with which to communicate to the simulation.
- **vehicle** (Vehicle) – The vehicle to which this sensor should be attached. Note: a vehicle must be provided for the roads sensor.
- **gfx_update_time** (*float*) – The gfx-step time which should pass between sensor reading updates to the user, in seconds.
- **physics_update_time** (*float*) – The physics-step time which should pass between actual sampling the sensor, in seconds.
- **is_send_immediately** (*bool*) – A flag which indicates if the readings should be sent back as soon as available or upon graphics step updates, as bulk.

`collect_ad_hoc_poll_request`(*request_id: int*) → StrDict

Collects a previously-issued ad-hoc polling request, if it has been processed.

Parameters
request_id (*int*) – The unique Id number of the ad-hoc request. This was returned from the simulator upon sending the ad-hoc polling request.

Returns
The readings data.

Return type
StrDict

`is_ad_hoc_poll_request_ready`(*request_id: int*) → bool

Checks if a previously-issued ad-hoc polling request has been processed and is ready to collect.

Parameters
request_id (*int*) – The unique Id number of the ad-hoc request. This was returned from the simulator upon sending the ad-hoc polling request.

Returns
A flag which indicates if the ad-hoc polling request is complete.

Return type
bool

`poll`() → StrDict

Gets the most-recent readings for this sensor. Note: if this sensor was created with a negative update rate, then there may have been no readings taken.

Returns
A dictionary containing the sensor readings data.

Return type
StrDict

`remove`() → None

Removes this sensor from the simulation.

Return type
None

`send_ad_hoc_poll_request`() → int

Sends an ad-hoc polling request to the simulator. This will be executed by the simulator immediately, but will take time to process, so the result can be queried after some time has passed. To check if it has been processed, we first call the is_ad_hoc_poll_request_ready() function, then call the collect_ad_hoc_poll_request() function to retrieve the sensor reading.

Returns
A unique Id number for the ad-hoc request.

Return type
int

`set_requested_update_time`(*requested_update_time: float*) → None

Sets the current 'requested update time' value for this sensor.

Parameters
requested_update_time (*float*) – The new requested update time.

Return type
None

1.8.2.4.2 Classical Sensors

Sensor

class beamngpy.sensors.**Sensor**

Sensor meta-class declaring methods common to them.

attach(*vehicle:* Vehicle, *name: str*) → None

Called when the sensor is attached to a *Vehicle* instance. Used to perform sensor setup code before the simulation is started. This is called *after* the sensor has been entered into the vehicle's map of sensors under the given name.

Parameters

- **vehicle** (Vehicle) – The vehicle instance the sensor is being attached to.
- **name** (*str*) – The name the sensor is known under to the vehicle.

Return type
None

connect(*bng:* BeamNGpy, *vehicle:* Vehicle) → None

Called when the attached vehicle is being initialised in the simulation. This method is used to perform setup code that requires the simulation to be running.

Parameters

- **bng** (BeamNGpy) –
- **vehicle** (Vehicle) –

Return type
None

decode_response(*resp: StrDict*) → StrDict

Called to do post-processing on sensor data obtained from the simulation. This method is called after raw simulation data is received and the resulting processed data is considered the result of a sensor request.

Parameters
resp (*StrDict*) –

Return type
StrDict

detach(*vehicle:* Vehicle, *name: str*) → None

Called when the sensor is detached from a *Vehicle* instance. Used to perform sensor teardown code after the simulation is finished. This is called *after* the sensor has been removed from the vehicle's map of sensors under the given name.

Parameters

- **vehicle** (Vehicle) – The vehicle instance the sensor is being detached from.
- **name** (*str*) – The name the sensor was known under to the vehicle.

Return type
None

disconnect(*bng:* BeamNGpy, *vehicle:* Vehicle) → None

Called when the attached vehicle is being removed from simulation. This method is used to perform teardown code after the simulation.

Parameters

- **bng** (BeamNGpy) –
- **vehicle** (Vehicle) –

Return type
None

encode_engine_request() → StrDict | None

Called to retrieve this sensor's data request to the engine as a dictionary. The dictionary returned by this method will be bundled along the vehicle's other sensors' requests as a SensorRequest to the simulator's engine.

Note: Sensors require corresponding code in the simulator to handle requests.

Returns
The request to send to the engine as a dictionary.

Return type
StrDict | None

encode_vehicle_request() → StrDict

Called to retrieve this sensor's request to the vehicle as a dictionary. The dictionary returned by this method will be bundled along the vehicle's other sensors' requests as a SensorRequest to the attached vehicle.

Note: Sensors require corresponding code in the simulator to handle requests.

Returns
The request to send to the vehicle as a dictionary.

Return type
StrDict

State

class beamngpy.sensors.**State**

Bases: *Sensor*

The state sensor monitors general stats of the vehicle, such as position, direction, velocity, etc.

It contains the following:

- `pos`: The vehicle's position as an `(x, y, z)` triplet
- `dir`: The vehicle's direction vector as an `(x, y, z)` triplet
- `up`: The vehicle's up vector as an `(x, y, z)` triplet
- `vel`: The vehicle's velocity along each axis in metres per second as an `(x, y, z)` triplet
- `rotation`: The vehicle's rotation as an `(x, y, z, w)` quaternion

Electrics

class beamngpy.sensors.**Electrics**

Bases: *Sensor*

This sensor is used to retrieve various values made available by the car's eletrics systems. These values include:

TODO: List all the electrics.lua values. - abs (int): ABS state - abs_active (bool): - airspeed (float): Airspeed - airflowspeed (float): - altitude (float): Z axis position - avg_wheel_av (float): - brake (int): Brake value - brake_lights (int): - brake_input (int): Brake input value - check_engine (bool): Check engine light state. - clutch (int): Clutch value - clutch_input (int): Clutch input value - clutch_ratio (int): - driveshaft (float): Driveshaft - engine_load (float): - engine_throttle (int): Engine throttle state - esc (int): ESC state. 0 = not present/inactive, 1 = disabled, Blink = active - esc_active (bool): - exhaust_flow (float): - fog_lights (int): Fog light state - fuel (float): Percentage of fuel remaining. - fuel_capacity (int): Total Fuel Capacity [L]. - fuel_volume (float): - gear (int): - gear_a (int): Gear selected in automatic mode. - gear_index (int): - gear_m (int): Gear selected in manual mode. - hazard (int): Hazard light state - hazard_signal (bool): - headlights (int): - highbeam (int): High beam state - horn (int): - ignition (bool): Engine state - left_signal (bool): - lightbar (int): Lightbar state - lights (int): General light state. 1 = low, 2 = high - lowbeam (int): Low beam state - lowfuel (bool): Low fuel indicator - lowhighbeam (int): Low-high beam state - lowpressure (int): Low fuel pressure indicator - oil (int): - oil_temperature (float): Oil temperature [C]. - parking (int): Parking lights on/off (not implemented yet) - parkingbrake (float): Parking brake state. 0.5 = halfway on - parkingbrake_input (int): Parking brake input state - radiator_fan_spin (int): - reverse (int): Reverse gear state - right_signal (bool): - rpm (float): Engine RPM - rpmspin (float): - rpm_tacho (float): - running (bool): Engine running state - signal_l (int): Left signal state. 0.5 = halfway to full blink - signal_r (int): Right signal state. 0.5 = halfway to full blink - steering (float): Angle of the steering wheel in degrees. - steering_input (int): Steering input state - tcs (int): TCS state. 0 = not present/inactive, 1 = disabled, Blink = active - tcs_active (bool): - throttle (int): Throttle state - throttle_factor (int): - throttle_input (int): Throttle input state - turnsignal (int): Turn signal value. -1 = Left, 1 = Right, gradually 'fades' between values. Use "signal_L" and "signal_R" for flashing indicators. - two_step (bool): - water_temperature (float): Water temperature [C]. - wheelspeed (float): Wheel speed [m/s].

IMU

class beamngpy.sensors.**IMU**(*pos: Float3 | None = None*, *node: int | None = None*, *name: str | None = None*, *debug: bool = False*)

Bases: *Sensor*

An IMU measures forces and rotational acceleration at a certain point on a vehicle. This can be used to analyze forces acting on certain areas of the car (like the driver's position) or estimate the trajectory of a vehicle from its rotation and acceleration.

Parameters

- **pos** (*Float3 | None*) –
- **node** (*int | None*) –
- **name** (*str | None*) –
- **debug** (*bool*) –

Timer

class beamngpy.sensors.**Timer**

Bases: *Sensor*

The timer sensor keeps track of the time that has passed since the simulation started. It provides that information in seconds relative to the scenario start and does not represent something like a day time or date. It properly handles pausing the simulation, meaning the value of the timer sensor does not progress while the simulation is paused.

When polled, this sensor provides the time in seconds since the start of the scenario in a dictionary under the `time` key.

Damage

class beamngpy.sensors.**Damage**

Bases: *Sensor*

The damage sensor retrieves information about how damaged the structure of the vehicle is. It's important to realise that this is a sensor that has no analogue in real life as it returns a perfect knowledge overview of how deformed the vehicle is. It's therefore more of a ground truth than simulated sensor data.

GForces

class beamngpy.sensors.**GForces**

Bases: *Sensor*

This sensor is used to obtain the GForces acting on a vehicle.

TODO: GForce sensor for specific points on/in the vehicle

1.8.2.5 Logging

exception beamngpy.logging.**BNGDisconnectedError**

Exception class for BeamNGpy being disconnected when it shouldn't.

exception beamngpy.logging.**BNGError**

Generic BeamNG error.

exception beamngpy.logging.**BNGValueError**

Value error specific to BeamNGpy.

beamngpy.logging.**config_logging**(*handlers: List[Handler]*, *replace: bool = True*, *level: int = 10*, *redirect_warnings: bool = True*, *log_communication: bool = False*) → None

Function to configure logging.

Parameters

- **handlers** (*List[Handler]*) – list of already configured logging.Handler objects
- **replace** (*bool*) – whether to replace existing list of handlers with new ones or whether to add them, optional

- **level** (*int*) – log level of the beamngpy logger object, optional. Defaults to `logging.DEBUG`.
- **redirect_warnings** (*bool*) – whether to redirect warnings to the logger. Beware that this modifies the warnings settings.
- **log_communication** (*bool*) – whether to log the BeamNGpy protocol messages between BeamNGpy and BeamNG.tech, optional

Return type
None

beamngpy.logging.**create_warning**(*msg: str*, *category: Any = None*) → None

Helper function for BeamNGpy modules to create warnings.

Parameters

- **msg** (*str*) – message to be displayed
- **category** (*Any*) – Category of warning to be issued. See *warnings* documentation for more details. Defaults to None.

Return type
None

beamngpy.logging.**set_up_simple_logging**(*log_file: str | None = None*, *redirect_warnings: bool = True*, *level: int = 20*, *log_communication: bool = False*) → None

Helper function that provides high-level control over beamng logging. For low-level control over the logging system use *config_logging()*. Sets up logging to `sys.stderr` and optionally to a given file. Existing log files are moved to `<log_file>.1`. By default beamngpy logs warnings and errors to `sys.stderr`, so this function is only of use, if the log output should additionaly be written to a file, or if the log level needs to be adjusted.

Parameters

- **log_file** (*str | None*) – log filename, optional
- **redirect_warnings** (*bool*) – Whether to redirect warnings to the logger. Beware that this modifies the warnings settings.
- **level** (*int*) – log level of handler that is created for the log file. Defaults to `logging.INFO`.
- **log_communication** (*bool*) – whether to log the BeamNGpy protocol messages between BeamNGpy and BeamNG.tech, optional

Return type
None

1.8.2.6 Tools

class beamngpy.tools.**OpenDriveExporter**

A class for exporting BeamNG road network data to OpenDrive (.xodr) format.

static compute_roads_and_junctions(*navigraph_data*, *path_segments*)

Computes a collection of individual road sections and junctions, both indexed by a unique Id. This function produces all the relevant data ready to be exported to OpenDrive (.xodr) format.

static export(*name*, *bng*)

Exports the road network data to OpenDrive (.xodr) format. The export contains all road sections, some basic lane data, and some junction connectivity data.

Parameters

- **name** – The path/filename at which to save the .xodr file.
- **bng** – The BeamNG instance.

class beamngpy.tools.**OpenStreetMapExporter**

static **export**(*name*, *bng*)

Exports the road network data to OpenStreetMap (.osm) format. The export contains all road sections, some basic lane data, and some junction connectivity data.

Parameters

- **name** – The path/filename by which to save the .osm file.
- **bng** – The BeamNG instance.

class beamngpy.tools.**SumoExporter**

static **export**(*name*, *bng*)

Exports the road network data to Sumo (.nod.xml and .edg.xml) format. The export contains all road sections, some basic lane data, and some junction connectivity data. This function will generate both .xml files required to generate the Sumo road network. The user should then type the following into the command prompt: *netconvert –node-files=<NAME>.nod.xml –edge-files=<NAME>.edg.xml -o converted.net.xml* which will then generate the final road network, which can be loaded with the sumo applications.

Parameters

- **name** – the filename prefix, by which to save the sumo road network (the .nod.xml and .edg.xml extensions will be appended to the end of this name).
- **bng** – The BeamNG instance.

class beamngpy.tools.**OpenDriveImporter**

static **FresnelCS**(*y*)

static **GeneralizedFresnelCS**(*a*, *b*, *c*)

static **add_lateral_offset**(*roads*)

static **adjust_elevation**(*roads*, *min_elev=5.0*)

static **combine_geometry_data**(*lines*, *arcs*, *spirals*, *polys*, *cubics*, *elevations*, *widths*, *lane_offsets*)

static **compute_width_sum**(*s*, *q*, *width_data*, *lane_offset*)

static **evalClothoid**(*x0*, *y0*, *theta0*, *kappa*, *dkappa*, *s*)

static **evalXYaLarge**(*a*, *b*)

static **evalXYaSmall**(*a*, *b*)

static **evalXYazero**(*b*)

static **extract_road_data**(*filename*)

static **get_elevation_profile**(*s*, *profiles*)

static import_xodr(*filename*, *scenario:* Scenario)

Parameters
scenario (Scenario) –

static rLommel(*mu*, *nu*, *b*)

class beamngpy.tools.**OpenStreetMapImporter**

static extract_road_data(*filename*)

static import_osm(*filename*, *scenario:* Scenario)

Parameters
scenario (Scenario) –

class beamngpy.tools.**SumoImporter**

static extract_edge_data(*filename*)

static extract_node_data(*filename*)

static import_sumo(*prefix*, *scenario:* Scenario)

Parameters
scenario (Scenario) –

static remove_duplicate_edges(*edges*)

1.8.2.7 Miscellaneous

1.8.2.7.1 Colors

beamngpy.misc.colors.**coerce_color**(*color: Color*, *alpha=0.0*) → Float4

Tries to coerce a color to a 4-tuple of floats.

Parameters

- **color** (*Color*) – A vehicle color.
- **alpha** – The alpha (transparency) value of the color. Defaults to 0.0.

Returns
An `(R, G, B, A)` tuple of floats.

Return type
Float4

beamngpy.misc.colors.**rgba_to_str**(*color: Float4*) → str

Converts an `(R, G, B, A)` tuple of floats to a string format parsable by BeamNG.

Returns
The converted string of format 'R G B A'.

Parameters
color (*Float4*) –

Return type
str

1.8.2.7.2 Quaternions

beamngpy.misc.quat.**angle_to_quat**(*angle: Float3*) → Quat

Converts an euler angle to a quaternion.

Parameters
angle (*Float3*) – Euler angle in degrees.

Returns
Quaternion with the order `(x, y, z, w)` with `w` representing the real component.

Return type
Quat

beamngpy.misc.quat.**compute_rotation_matrix**(*quat: Quat*) → numpy.ndarray

Calculates the rotation matrix for the given quaternion to be used in a scenario prefab.

Parameters
quat (*Quat*) – Quaternion with the order `(x, y, z, w)` with `w` representing the real component.

Returns
The rotation matrix as a `NumPy` array.

Return type
numpy.ndarray

beamngpy.misc.quat.**flip_y_axis**(*q: Quat*) → Quat

Returns a rotation with a flipped y-axis.

Parameters
q (*Quat*) – Quaternion with the order `(x, y, z, w)` with `w` representing the real component.

Returns
The flipped quaternion.

Return type
Quat

beamngpy.misc.quat.**normalize**(*q: Quat*) → Quat

Normalizes the given quaternion.

Parameters
q (*Quat*) – Quaternion with the order `(x, y, z, w)` with `w` representing the real component.

Returns
The normalized quaternion.

Return type
Quat

beamngpy.misc.quat.**quat_as_rotation_mat_str**(*quat: Quat*, *delimiter: str = ' '*) → str

For a given quaternion, the function computes the corresponding rotation matrix and converts it into a string.

Parameters

- **quat** (*Quat*) – Quaternion with the order `(x, y, z, w)` with `w` representing the real component.
- **delimiter** (*str*) – The string with which the elements of the matrix are divided.

Returns
Rotation matrix as a string.

Return type
str

beamngpy.misc.quat.**quat_multiply**(*a: Quat*, *b: Quat*) → Quat

Multiplies two quaternions.

Parameters

- **a** (*Quat*) – Quaternion with the order `(x, y, z, w)` with `w` representing the real component.
- **b** (*Quat*) – Quaternion with the order `(x, y, z, w)` with `w` representing the real component.

Returns
The product of `a` and `b` as a quaternion.

Return type
Quat

1.8.2.7.3 Vec3

class beamngpy.misc.**vec3**(*x*, *y*, *z=0.0*)

Bases: `object`

A class for storing vectors in R^3. Contains functions for operating within that vector space. Can also be used as a vec2 class, since the z component is optional.

cross(*b*)

The cross product between this vector and a given vector.

Parameters
b – The given vector.

Returns
The cross product between the two vectors (a vector value)

distance(*b*) → float

The L^2 (Euclidean) distance between this vector and a given vector. AKA the distance formula.

Parameters
b – The given vector.

Returns
The L^2 (Euclidean) distance between the two vectors (a scalar value).

Return type
float

distance_sq(*b*) → float

The L^1 (squared) distance between this vector and a given vector. AKA the distance formula.

Parameters
b – The given vector.

Returns
The squared distance between the two vectors (a scalar value).

Return type
float

dot(*b*) → float

The dot product between this vector and a given vector.

Parameters
b – The given vector.

Returns
The dot product between the two vectors (a scalar value).

Return type
float

length() → float

The length (magnitude) of this vector. [ie $length := |vector|$]

Returns
The length of this vector (a scalar value).

Return type
float

normalize()

Normalizes this vector so that it becomes unit length ($length = 1$).

Returns
The normalized vector.

1.8.2.7.4 Types

beamngpy.types.**Color**

Vehicle color. Can be either:

- `(R, G, B)` tuple of floats between 0.0 and 1.0,
- `(R, G, B, A)` tuple of floats between 0.0 and 1.0,
- string of format `'R G B'`, where `R`, `G`, and `B` are floats between 0.0 and 1.0,
- string of format `'R G B A'`, where `R`, `G`, `B`, and `A` are floats between 0.0 and 1.0,
- a common color name (parsable by `matplotlib.colors`).

alias of `Union[Tuple[float, float, float], Tuple[float, float, float, float], str]`

beamngpy.types.**Float2**

alias of `Tuple[float, float]`

beamngpy.types.**Float3**

alias of `Tuple[float, float, float]`

beamngpy.types.**Float4**

alias of `Tuple[float, float, float, float]`

beamngpy.types.**Float5**

alias of `Tuple[float, float, float, float, float]`

beamngpy.types.**Int2**

alias of `Tuple[int, int]`

beamngpy.types.**Int3**
alias of Tuple[int, int, int]

beamngpy.types.**Quat**
alias of Tuple[float, float, float, float]

beamngpy.types.**StrDict**
alias of Dict[str, Any]

1.8.2.7.5 Connection

class beamngpy.connection.**CommBase**(*bng:* BeamNGpy, *vehicle:* Vehicle | *None*)
Communication helper base class to make the socket communication easier to implement for derived classes.

Parameters

- **bng** (BeamNGpy) –
- **vehicle** (Vehicle | *None*) –

send_ack_ge(*type: str*, *ack: str*, ***kwargs: Any*) → None
Sends a request to the GE Lua with the provided type and data, and receives the acknowledgement.

Parameters

- **type** (*str*) – Type of the request to send.
- **ack** (*str*) – Type of the acknowledgement to be received.
- **kwargs** (*Any*) – The other data being sent.

Returns
The response of the simulator.

Return type
None

send_ack_veh(*type: str*, *ack: str*, ***kwargs: Any*) → None
Sends a request to the Vehicle Lua with the provided type and data, and receives the acknowledgement.

Parameters

- **type** (*str*) – Type of the request to send.
- **ack** (*str*) – Type of the acknowledgement to be received.
- **kwargs** (*Any*) – The other data being sent.

Returns
The response of the simulator.

Return type
None

send_recv_ge(*type: str*, ***kwargs: Any*) → StrDict
Sends a request to the GE Lua with the provided type and data, receives the answer and returns it.

Parameters

- **type** (*str*) – Type of the request to send.
- **kwargs** (*Any*) – The other data being sent.

Returns
The response of the simulator.

Return type
StrDict

send_recv_veh(*type: str*, ***kwargs: Any*) → StrDict

Sends a request to the Vehicle Lua with the provided type and data, receives the answer and returns it.

Parameters

- **type** (*str*) – Type of the request to send.
- **kwargs** (*Any*) – The other data being sent.

Returns
The response of the simulator.

Return type
StrDict

class beamngpy.connection.**Connection**(*host: str*, *port: int | None = None*)

The class for handling socket communication between BeamNGpy and the simulator, including establishing connections to both the simulator and to its vehicles individually, and for sending and recieving data across these sockets.

Instantiates an instance of the Connection class, creating an unconnected socket ready to be connected when required.

Parameters

- **host** (*str*) – The host to connect to.
- **port** (*int | None*) – The port to connect to.

PROTOCOL_VERSION = 'v1.22'

connect_to_beamng(*tries: int = 25*, *log_tries: bool = True*) → bool

Sets the socket of this connection instance and attempts to connect to the simulator over the host and port configuration set in this class. Upon failure, connections are re-attempted a limited amount of times.

Parameters

- **tries** (*int*) – The number of connection attempts.
- **log_tries** (*bool*) – True if the connection logs should be propagated to the caller. Defaults to True.

Returns
True if the connection was successful, False otherwise.

Return type
bool

connect_to_vehicle(*vehicle: Vehicle*, *tries: int = 25*) → None

Sets the socket of this Connection instance, and attempts to connect it to the given vehicle. Upon failure, connections are re-attempted a limited amount of times.

Parameters

- **vehicle** (*Vehicle*) – The vehicle instance to be connected.
- **tries** (*int*) – The number of connection attempts.

Return type
None

disconnect() → None

Closes socket communication for this Connection instance.

Return type
None

hello() → None

First function called after connections. Exchanges the protocol version with the connected simulator and raises an error upon mismatch.

Return type
None

message(*req: str*, ***kwargs: Any*) → Any

Generic message function which is parameterized with the type of message to send and all parameters that are to be embedded in the request. Responses are expected to have the same type as the request. If this is not the case, an error is raised.

Parameters

- **req** (*str*) – The request type.
- **kwargs** (*Any*) –

Returns
The response received from the simulator as a dictionary.

Return type
Any

recv(*req_id: int*) → StrDict | *BNGError* | *BNGValueError*

Parameters
req_id (*int*) –

Return type
StrDict | *BNGError* | *BNGValueError*

send(*data: StrDict*) → *Response*

Encodes the given data using Messagepack and sends the resulting bytes over the socket of this Connection instance. NOTE: messages are prefixed by the message length value.

Parameters
data (*StrDict*) – The data to encode and send

Return type
Response

class beamngpy.connection.**Response**(*connection:* Connection, *req_id: int*)

Parameters

- **connection** (Connection) –
- **req_id** (*int*) –

ack(*ack_type: str*) → None

Parameters
ack_type (*str*) –

Return type
None

recv(*type: str | None = None*) → StrDict

Parameters
type (*str | None*) –

Return type
StrDict

1.8.3 BeamNG ROS Integration

To support the interoperability between BeamNG.tech and ROS we published the BeamNG ROS Integration. It is an independent ROS package that translates a range of BeamNGpy features to the ROS framework. beamng-ros-integration is an repository contains packages to support the interoperability between BeamNG.tech and ROS 1 distributions Melodic Morenia and Noetic Ninjemys.

Basic ROS functionality are included i.e., sensors streaming, Rviz simulation, direct keyboard control (Teleop). ROS topics for Sensor Suite: multiple filters of the camera (Annotated, instance, Depth, and RGB), 3D Lidar, Ultrasonic, IMU, and vehicle electrics (speed, fuel, temperature, gear, signals, lights, etc).

- Installation Prerequisites:

you must have the following softwares/packages installed i.e., BeamNG.Tech, BeamNGpy,and WSL2.

1.8.3.1 ROS packages

- beamng_agent: for the control of a driving agent used for Teloep movement of the beamng_teleop_keyboard package, also used for enable/disable keyboard remote control to the BeamNG.Tech simulation platform.
- beamng_control: loading the ROS-BeamNG.Tech bridge and the scenario details (vehicle, environment, sensors, location, etc.).
- beamng_msgs: Defind the custom messages of the BeamNG.Tech simulator to be readable by ROS-standards.
- beamng_teleop_keyboard: keyboard remote control of the BeamNG.Tech simulation platform through ROS bridge.

1.8.3.2 Compatibility

Running the BeamNG ROS integration requires three individual software components, here is a list of compatible versions.

BeamNG.tech	BeamNGpy	BeamNG ROS Integration
0.30	1.26.1	0.1.4
0.28	1.26	0.1.3
0.27	1.25.1	0.1.2
0.26	1.24	0.1.1
0.25	1.23.1	0.1.0

1.8.3.3 WSL2 setup

ROS1 integration is built on top of Windows Subsystem for Linux (WSL2). The recommended linux version is ubuntu focal 20.04, and The recommended ROS 1 distribution is Noetic. The ROS bridge is made through python API support from BeamNGpy.

1.8.3.4 ROS setup

setup catkin_ws

Install and build ROS bridge

```
git@github.com:BeamNG/beamng-ros-integration.git
```

```
cd ~/catkin_ws/ && catkin_make
```

WSL2 dependencies:

```
sudo apt install python3-rosdep2
```

```
sudo apt install python3-pip
```

```
pip install beamngpy
```

```
sudo apt-get install ros-noetic-rostest
```

```
sudo apt-get install ros-noetic-actionlib
```

```
python3 -m pip install -U scikit-image
```

```
sudo apt install python3-rosservice
```

1.8.3.5 Getting started

BeamNG-ROS bridge needs to be configured to contain the correct IPv4 address of the machine hosting the simulation. Using it will start up a node that connects to the simulation and starts up a scenario as defined in the *beamng_control/config/scenario/{scenario}.json*. Other scenario specifications are available in the same directory.

- Scenarios are defined through JSON objects, here is a list of possible keys and values.

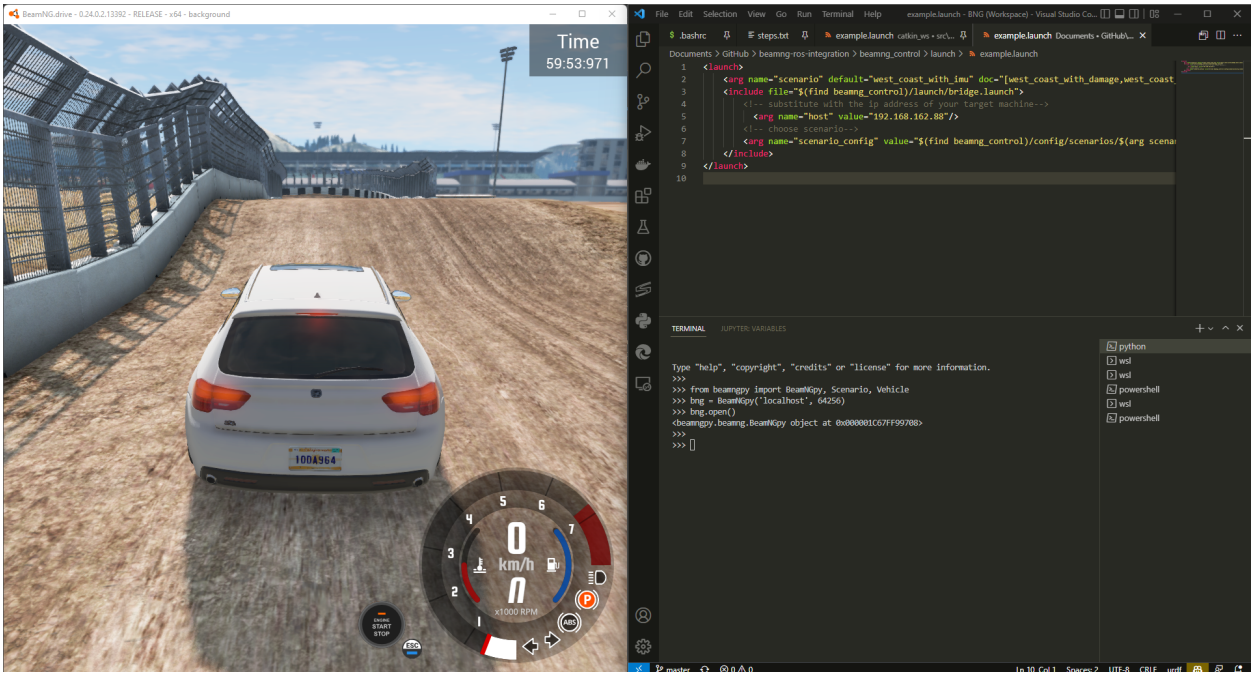
Key	Value Type	Value Specification	Entry Type
`version`	String	BeamnG ROS Integration version, f.ex. `1`	Mandatory
`level`	String	BeamNG.tech level name, f.ex. `west_coast_usa`	Mandatory
`mode`	String	Value	Optional
`vehicles`	Array	At least one vehicle needs to be specified in order to obtain a valid scenario. See the table below for the Specification.	Mandatory
`name`	String	Name of the level.	Mandatory
`time_of_day`	Float	Value between `0` and `1` where the range `[0, .5]` corresponds to the times between 12 a.m. and 12 p.m. and `[.5], 1]` corresponds to the time range between 12 p.m. and 12 a.m.	Optional
`weather_presets`	String	Weather presets are level specific, **ToDo**	Optional

- Vehicles are also defined as JSON objectsin *beamng_control/config/vehicles/{vehicle}.json*.

Key	Value Type	Value Specification	Entry Type
`name`	String	Name of the vehicle, used for identification	Mandatory
`model`	String	Name of the vehicle type, f.ex. `etk800`	Mandatory
`position`	Array	Array of 3 floats, specifying the `x`, `y`, and `x` position of the vehicle.	Mandatory
`rotation`	Array	Array of 4 floats, specifying the vehicle rotation quaternion.	Mandatory
`sensors_classical`	Array	Array of JSON objects, specifying the vehicles sensor parameters i.e., electrics, IMU, damage, gforce, and time sensor	Optional
`sensors_automation`	Array	Array of JSON objects, specifying the ad-hoc_sensors parameters i.e., Lidar, camera, and Ultrasonic sensor	Optional

1.8.3.6 Running BeamNG.Tech

After installing BeamNGpy, and setup BeamNG.Tech, you can run BeamNG.py from the Powershell as shown in the picture below.



1.8.3.7 Running the ROS-bridge

- Loading beamng_control node for loading the scenarios components i.e., level, vehicle, environemnt and sensors from *example.launch* file in the *beamng_control* package through the command:

```
roslaunch beamng_control example.launch
```

1.8.3.8 Running beamng_agent

- **Loading beamng_agent node for enabling the control from ROS side:**
  `roslaunch beamng_agent example.launch`

The folloing topics for move/stop the vehicle in simulation and enable/disable keybard control from the simulation side; using an array of commands as following [steering throttle brake parkingbrake clutch gear], here's some exmaples of the `VehicleControl` :

- **Driving:**
  `rostopic pub --once control beamng_msgs/VehicleControl 0 1 0 0 0 1`
- **Stopping:**
  `rostopic pub --once control beamng_msgs/VehicleControl 0 0 1 0 0 1`
- **Release:**
  `rostopic pub --once control beamng_msgs/VehicleControl 0 0 0 0 0 1`

1.8.3.9 Calling ROS-services for controlling the Simulation

To Dis-/Enables user keyboard and mouse control of the BeamNG.Tech simulator

Name	Type	Purpose
`/beamng_control/pause`	`bng_msgs.srv.ChangeSmulationState`	Pause the simulation.
`/beamng_control/resume`	`bng_msgs.srv.ChangeSmulationState`	Resume the simulation.

- Disable user keyboard and mouse control of the BeamNG.Tech simulator: `rosservice call /beamng_control/pause "{}"`
- terminal feedback should be:

  ```
  success:  True
  ```

- Enable user keyboard and mouse control of the BeamNG.Tech simulator: `rosservice call /beamng_control/resume "{}"`
- terminal feedback should be:

  ```
  success:  True
  ```

1.8.3.10 Vehicle Creation and Control

Various services to control the state of the simulation are available.

Name	Type	Purpose
/beamng_control/ get_scenario_state	bng_msgs.srv. GetScenarioState	Determining the current state of thescenario.
/beamng_control/ spawn_vehicle	beamng_msgs.srv.SpawnVehicle	Spawn new vehicle.
/beamng_control/ teleport_vehicle	beamng_msgs.srv. TeleportVehicle	Teleport vehicle.
/beamng_control/ start_scenario	bng_msgs.srv.StartScenario	Starting a loaded scenario.
/beamng_control/ get_current_vehicles	beamng_msgs.srv. GetCurrentVehiclesInfo	Get the spawned vehicle information.

- **Clone a new vehicle:**
  `rosservice call /beamng_control/spawn_vehicle 'ros' [0,5,10] [0,0,0,1] "/config/vehicles/etk800.json"`
- **Load a new scenario:**
  `rosservice call /beamng_control/start_scenario "/config/scenarios/west_coast.json"`
- **Reposition the current vehicle in west coast:**
  `rosservice call /beamng_control/teleport_vehicle "ego_vehicle" [568.908,13.422,148.565] [0,0,0,1]`
- **Getting the scenario state:**
  `rosservice call /beamng_control/get_scenario_state`
- **Getting the get_current_vehicles:**
  `rosservice call /beamng_control/get_current_vehicles`
- **Getting the get_loggers:**
  `rosservice call /beamng_control/get_loggers`

1.8.3.11 Note

- if you got a feedback *success: False* for *resume* or *pause* services, that means your *beamng_agent* node isn't active, and you will getting the following error message in the terminal of *beamng_control* node:

1.8.3.12 List of ROS-topics

- **ROS-visualization tool (Rviz) map:**
  `/beamng_control/<vehicle_id>/marker`
- Camera:

Contrary to other sensors, the Camera sensor may publish to multiple topics. If the camera sensor is configured to collect color, depth, annotation, and instance data, it is published to the respective topics:

`/beamng_control/<vehicle_id>/<camera_id>/color`

`/beamng_control/<vehicle_id>/<camera_id>/depth`

`/beamng_control/<vehicle_id>/<camera_id>/annotation`

`/beamng_control/<vehicle_id>/<camera_id>/instance`

The message type for all topics is *sensor_msgs.msg.Image*. Note that although the bounding_box option is given, this feature is still under development and will automatically be disabled.

- LiDAR:

Message type: ***sensor_msgs.msg.PointCloud2***
`/beamng_control/<vehicle_id>/<lidar_id>`

Key	Value Type	Value Specification	Entry Type
`type`	String	`Lidar.default`	Mandatory
`name`	String	Unique sensor id.	Mandatory
`position`	Array	Array of 3 floats, specifying the `x`, `y`, and `x` position of the sensor.	Mandatory
`rotation`	Array	Array of 3 floats, specifying the vehicle rotation quaternion	Mandatory
`vertical_resolution`	Integer	Vertical resolution, i.e. how many lines are sampled vertically	Optional
`vertical_angle`	Float	The vertical LiDAR sensor angle, in degrees.	Optional
`frequency`	Integer	The frequency of this LiDAR sensor.	Optional
`rays_per_second`	Integer	The rays per second emmited by the LiDAR sensor	Optional
`is_visualised`	Boolean	Dis-/Enable in-simulation visualization.	Optional

- Ultrasonic sensor :

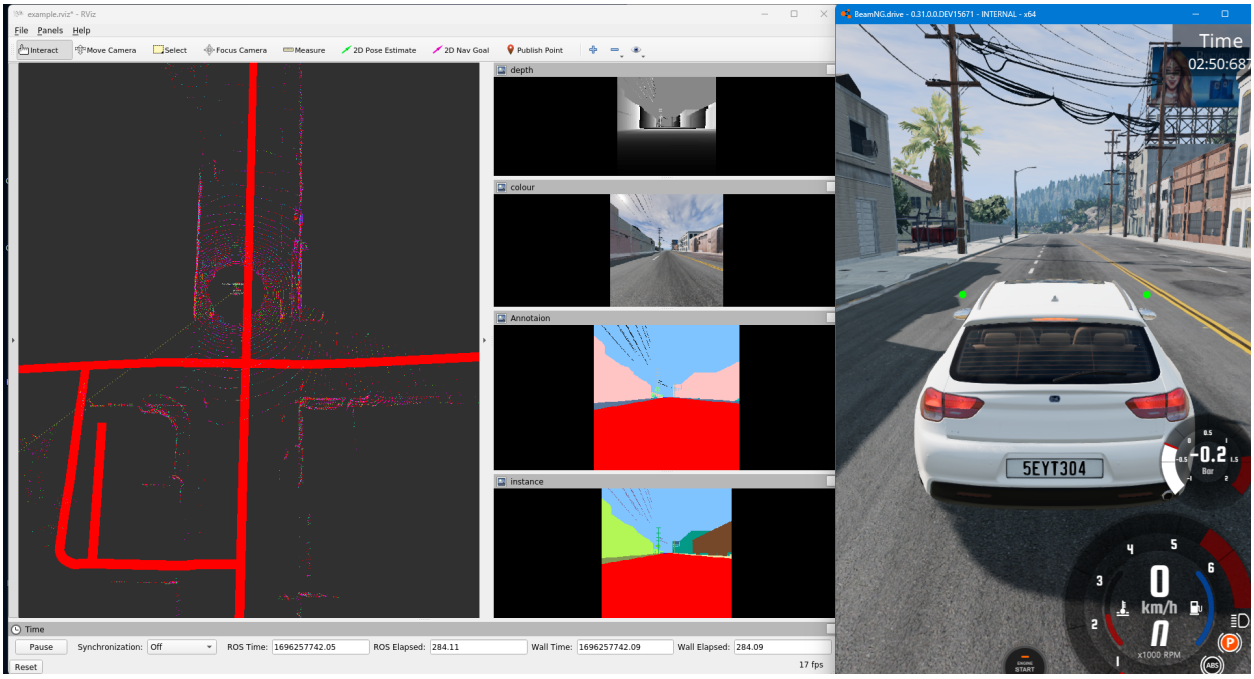
Message type: ***sensor_msgs.msg.Range***
`/beamng_control/<vehicle_id>/<ultrasonic_sensor_name>`

Key	Value Type	Value Specification	Entry Type
`type`	String	`Ultrasonic.smallrange`,and/or `Ultrasonic.midrange`,and/or `Ultrasonic.largerange`	Mandatory
`name`	String	Unique sensor id.	Mandatory
`position`	Array	Array of 3 floats, specifying the `x`, `y`, and `x` position of the sensor.	Mandatory
`rotation`	Array	Array of 3 floats, specifying the vehicle rotation quaternion	Mandatory
`is_visualised`	Boolean	Dis-/Enable in-simulation visualization.	Optional

- Damage:

Message type: ***beamng_msgs.msg.DamagSensor***
`/beamng_control/<vehicle_id>/<damage_sensor_id>`

Key	Value Type	Value Specification	Entry Type
`type`	String	`Damage`	Mandatory
`name`	String	Unique sensor id.	Mandatory



- time:

Message type: ***beamng_msgs.msg.TimeSensor***
`/beamng_control/<vehicle_id>/<time_sensor_id>`

Key	Value Type	Value Specification	Entry Type
`type`	String	`Timer`	Mandatory
`name`	String	Unique sensor id.	Mandatory

- Gforces:

Message type: ***beamng_msgs.msg.GForceSensor***
`/beamng_control/<vehicle_id>/<gforce_sensor_id>`

Key	Value Type	Value Specification	Entry Type
`type`	String	`GForces`	Mandatory
`name`	String	Unique sensor id.	Mandatory

- Electrics:

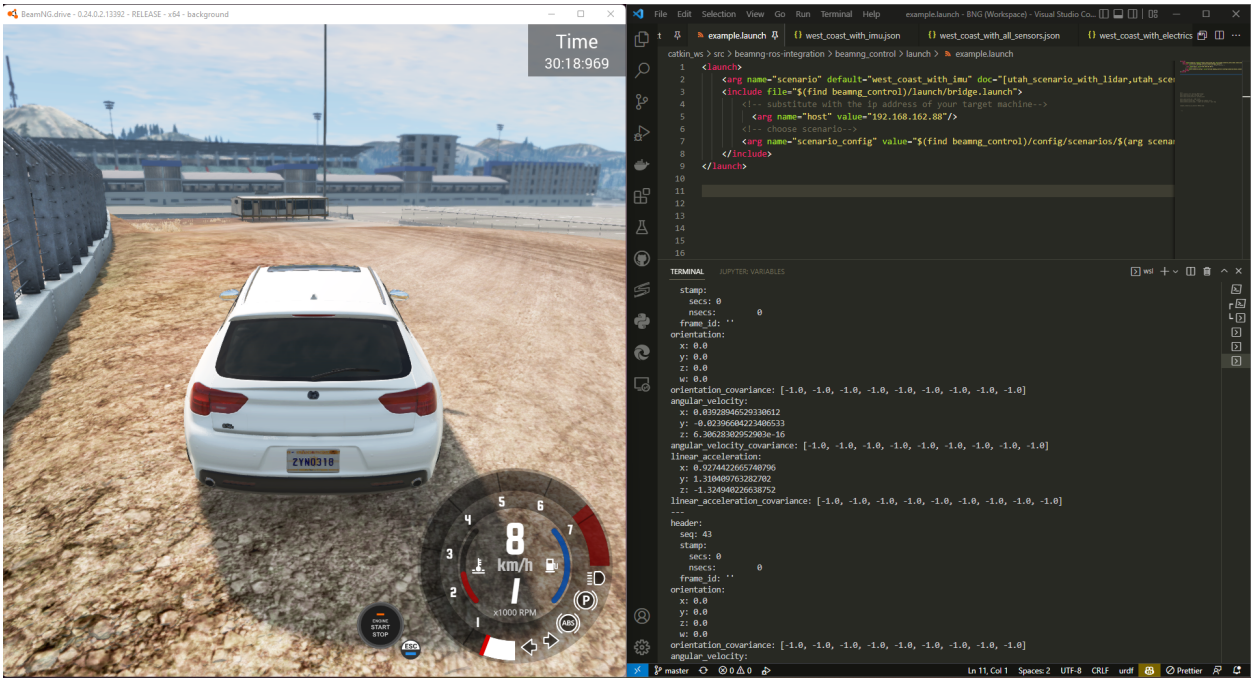
Message type: ***beamng_msgs.msg.ElectricsSensor***
`/beamng_control/<vehicle_id>/<electrics_sensor_id>`

Key	Value Type	Value Specification	Entry Type
`type`	String	`Electrics`	Mandatory
`name`	String	Unique sensor id.	Mandatory

- Imu pose:

Message type: ***sensor_msgs.msg.Imu***
`/beamng_control/<vehicle_id>/<imu_sensor_id>`

Key	Value Type	Value Specification	Entry Type
`type`	String	`IMU`	Mandatory
`name`	String	Unique sensor id.	Mandatory
`position`	Array	Array of 3 floats, specifying the `x`, `y`, and `x` position of the sensor.	Mandatory



- Vehicle state:

Message type: ***beamng_msgs.msg.StateSensor***
`/beamng_control/<vehicle_id>/state`

1.8.3.13 Teleop_control

beamng_teleop_keyboard is a generic Keyboard Packages is built for teleoperating ROS robots using Twist message from geometry_messages.

1.8.3.14 Running beamng_teleop_keyboard

- **Loading BeamNG-ROS bridge:**
  `roslaunch beamng_control example.launch`
- **Calling Twist_message converter node:**
  `rosrun beamng_teleop_keyboard converter`
- **Calling Teleop node:**
  `rosrun beamng_teleop_keyboard teleop_key`
- **Loading beamng_agent node:**
  `roslaunch beamng_agent example.launch`

1.8.4 BeamNG MATLAB integration

1.8.4.1 Overview

We are excited to announce that the highly requested feature of bridging BeamNG.tech and MATLAB is here. MATLAB, with its long history as an academic engineering and mathematical tool, is a programming and numeric computing platform used to analyse data, develop algorithms, and create models. The newly created bridge with MATLAB will enable you to run, control, and interact with the BeamNG.tech simulation. We have integrated five main scripts for your convenience in making use of annotations, bounding boxes, multi-shot camera, object placement, vehicle state plotting and creation of simple scenarios on our East Coast USA map.

1.8.4.2 Prequest

you must have the following softwares/packages installed:

1. Compatible Python
2. BeamNGpy
3. BeamNG.Tech

The BeamNG-MATLAB-integration bridge is depending on BeamNG.Tech and BeamNGpy. Make sure that you have the license for BeamNG.Tech. The Github repository of the BeamNG-MATLAB has some basic examples of scripts that run a vehicle with some sensors ex. Lidar, Camera, and state sensor.

1.8.4.2.1 Compatibility

Running the BeamNG ROS integration requires three individual software components, here is a list of compatible versions.

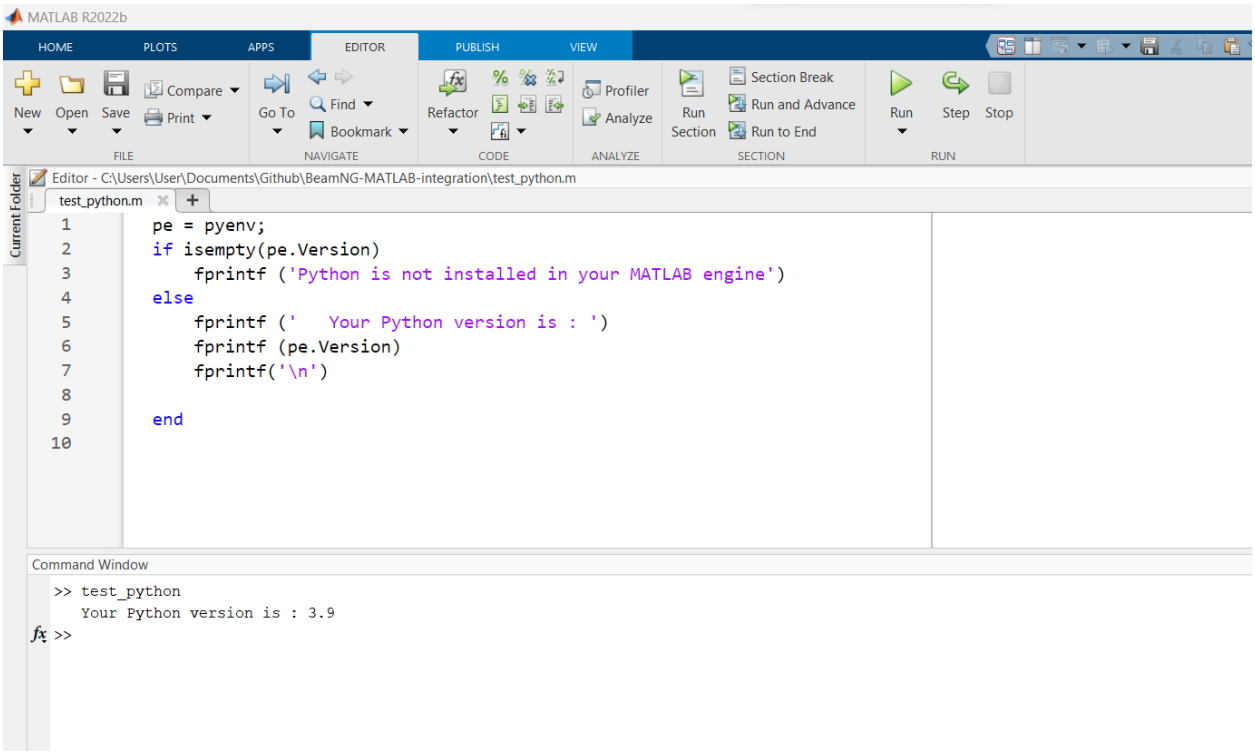
BeamNG.tech	BeamNGpy	BeamNG MATLAB integration	MATLAB	Python
0.28	1.26	0.1.1	R2023a	3.9
0.27	1.25.1	0.1.0	R2022b	3.9

1.8.4.2.2 1. Setup a compatible python version

After installing the compatible python version with MATLAB, make sure to include the path of excutable python file (exe) in your in "path" variable of "environment variables" as explained here.

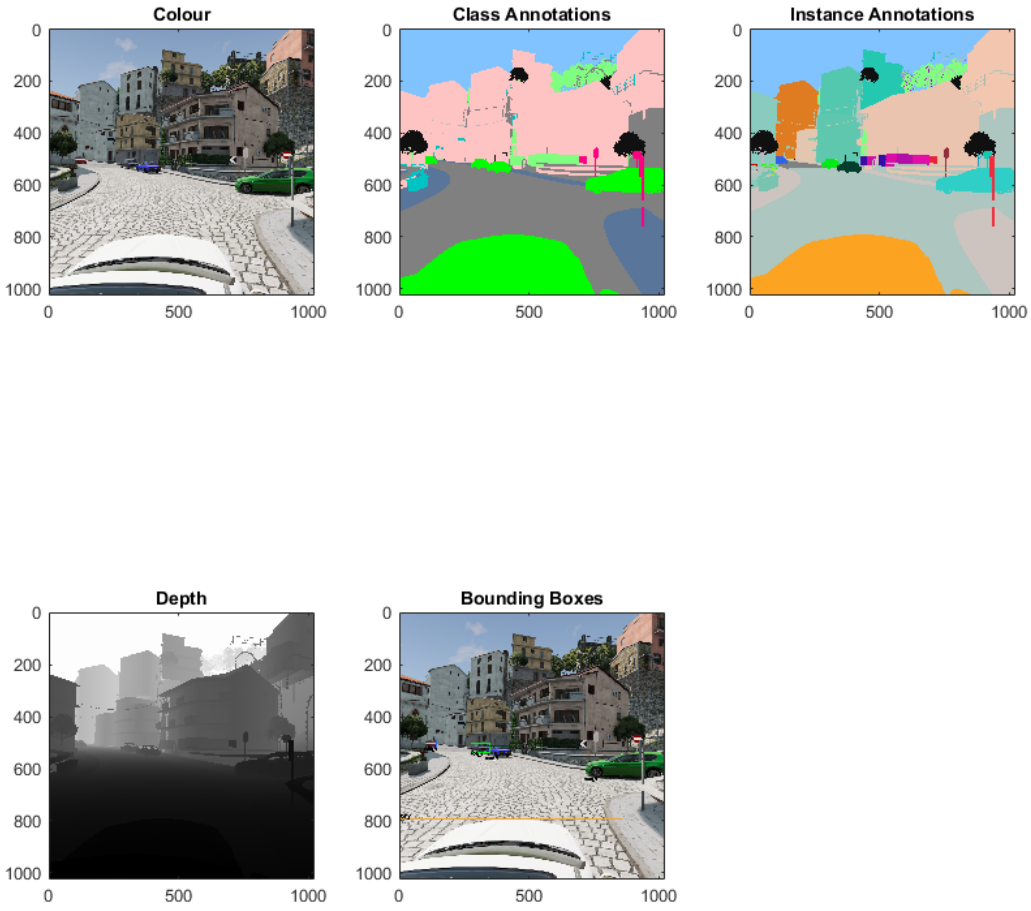
1.8.4.2.3 2. Run python engine in MATLAB

Run the test_python.m to make sure that python engine is connected to your MATLAB engine as shown in the picture below.



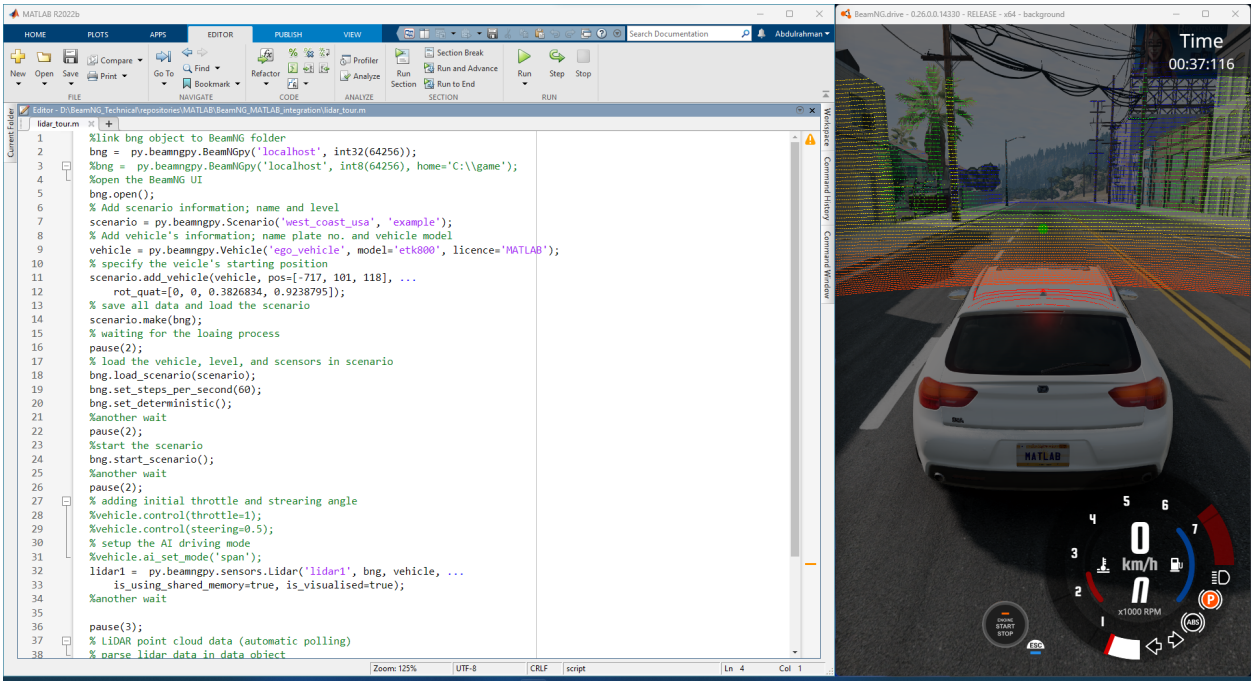
1.8.4.3 Vehicle State Plotting

Use the state sensor to plot some graphs of the vehicle position, wheel speed and direction, throttle, and brake.



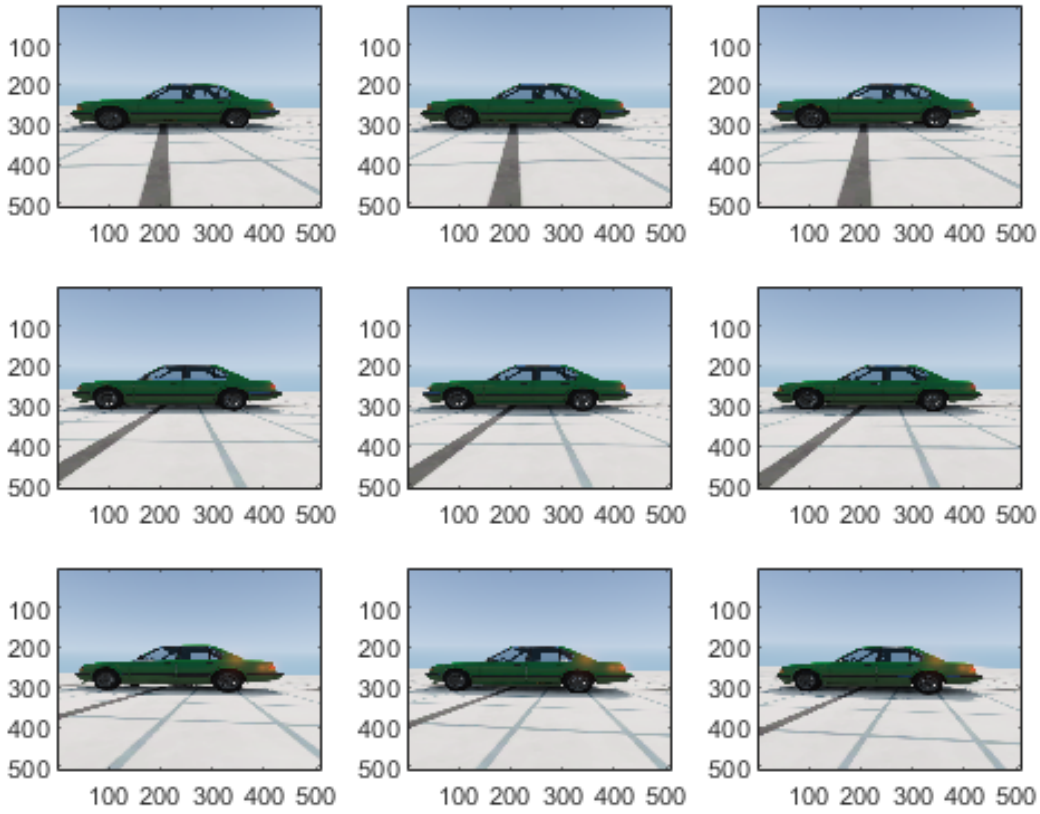
1.8.4.4 Running Lidar sensor, and AI control.

1. Create a simple scenario
2. Use the simulator's AI with BeamNGpy



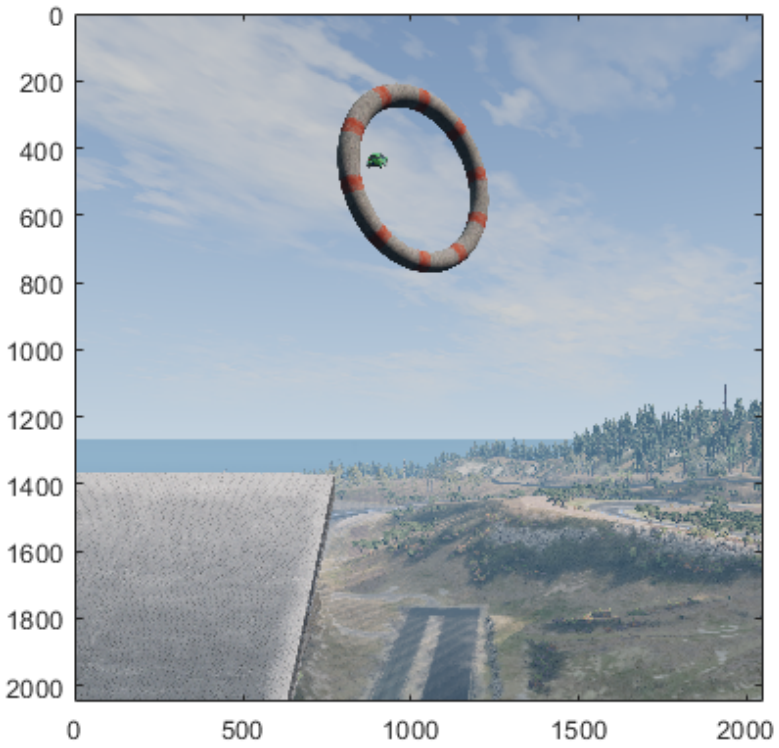
1.8.4.5 Multi-shot Camera

Change the position and rotation of a camera



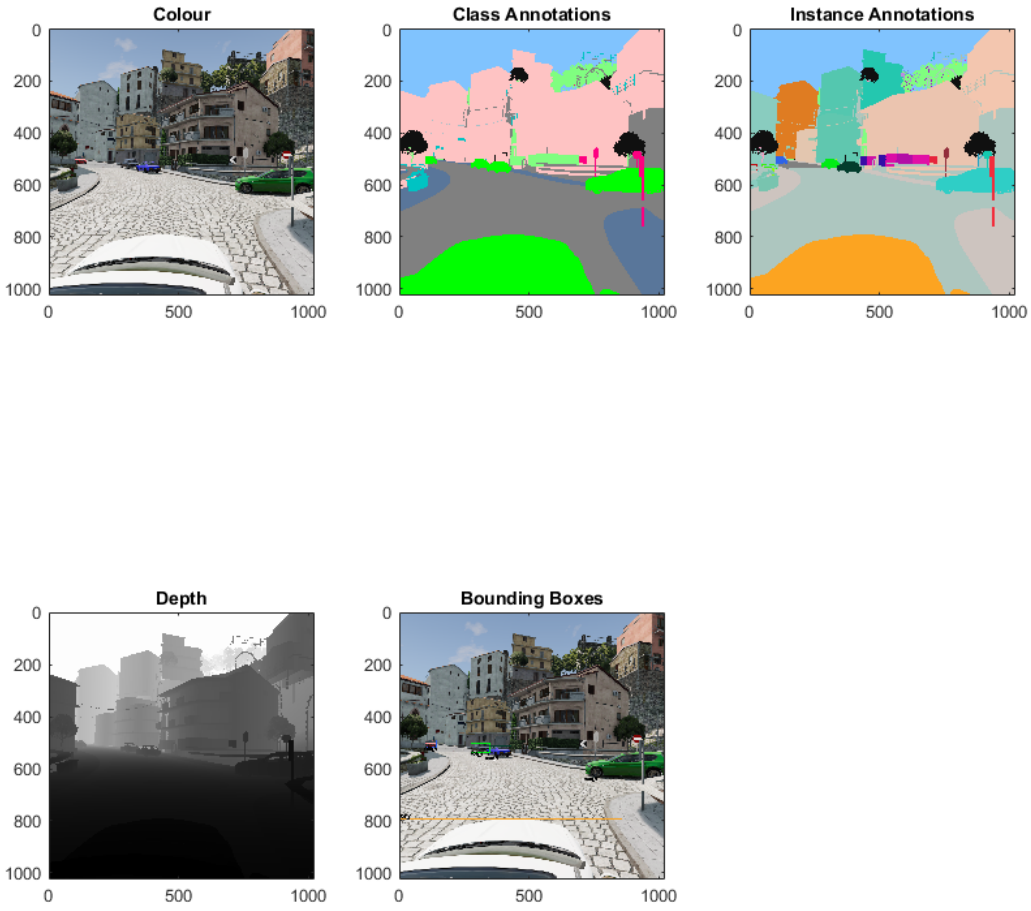
1.8.4.6 Object Placement

1. Define a custom scenario for a given map
2. Generate procedural content, i.e. simple meshes



1.8.4.7 Annotation and Bounding Boxes

1. Get semantic annotations
2. Get instance annotations
3. Graw bounding boxes (note that this feature is not ready for use yet)



1.8.5 BeamNG Simulink generic interface

1.8.5.1 About

We are excited to announce that the highly requested feature of interfacing BeamNG.Tech in Simulink is here. The user can connect the BeamNG simulator with a specially designed Simulink S-function, which will allow management of a tightly coupled two-way communication between these two environments. The major focus here is on allowing native Simulink code to control and query a vehicle in BeamNG. This includes various powertrain properties, including wheel torques (both drive and brake), or pedal inputs.

The integration is available at https://github.com/BeamNG/BeamNG-Simulink_generic_interface.

1.8.6 Changelog

1.8.6.1 Version 1.28

- Functionality added to allow the import of heightmaps (from 2D Python arrays).
- Optimized network communication by removing extra acknowledgement messages.
- The way of launching BeamNG.tech from BeamNGpy has changed. If you are launching BeamNG.tech without BeamNGpy and want to connect BeamNGpy later, you should change the command-line arguments you are using to:

  ```
  BeamNG.tech.x64.exe -console -nosteam -tcom-listen-ip 127.0.0.1 -lua extensions.
  load('tech/techCore');tech_techCore.openServer(64256)
  ```

- Added scenarios on `IdealRADAR` sensor use, to plot radar data and track objects.
- Added scenario on road profile plotting.
- Bugfixes
  - The Vehicle.logging module has been fixed and is usable again.
  - Fixed OpenStreetMap importer to manage mixed data as input in some cases.

1.8.6.2 Version 1.27.1

- Camera sensor improvements
  - Added the Camera.stream function for easier retrieval of camera images being streamed through shared memory
  - Added the Camera.poll_raw and Camera.stream_raw functions for getting raw bytes from the simulator, the conversion to a bitmap image is skipped
  - Added the `camera_streaming.py` example to showcase these functions

1.8.6.3 Version 1.27

- New features
  - GPS sensor added
    - check the documentation or the `GPS_trajectory.py` example script for more information on usage
  - RoadsSensor sensor added
  - IdealRadar sensor added
  - RADAR sensor now reads the Doppler velocity from vehicles in the simulation as well as static objects.
  - BeamNGpy now fully supports loading existing missions and Flowgraph scenarios. Look into the Scenario Control example notebook to learn more.
  - Beam stresses added as a mode to the `AdvancedIMU` sensor.
  - Camera, Lidar, and Radar sensor readings can now be streamed directly to shared memory in BeamNGpy, using dedicated `stream()` functions now found in the respective BeamNGpy sensor classes. This represents an alternative to the polling method used previously.
- BeamNGpy projects updated for latest BeamNG.tech version

  - Impactgen: A script to generate various vehicle impact scenarios and output surround views of the affected vehicle in color and semantically annotated images.
  - BeamNG.gym: A collection of Gymnasium environments that cover various driving tasks simulated in BeamNG.tech.
- API changes
  - Relative camera interface changed to use vectors instead of quaternions.
  - Changed the input and output types of the `BeamNGpy.scenario.get_scenarios` function:
    * the `levels` argument is now a list of level names or instances of the Level class to get scenarios for
    * the return value is now a dictionary where the keys are the level names, and the values are lists of scenarios for the given level
  - Removed the `level` argument of `BeamNGpy.scenario.get_current`, as the level information is now queried from the simulator.
  - Function added to the `Vehicle` class to deflate vehicle tires, e.g. to simulate tire blowout.
- Bugfixes
  - Fixed a bug where loading a BeamNGpy scenario could cause an infinite-loading screen glitch.
  - Fixed the `Mesh` sensor not working.
  - Part annotations for vehicles are working again.
  - Bug fixed when using multiple ultrasonic sensors, where the first sensor would not update in simulator.
  - Bug fixed when using ultrasonic sensor, relating to failure to detect at some angles to surfaces
  - Bug fixed with ultrasonic sensor, relating to typos in parameter names, rendering some parameters unusable from BeamNGpy.
  - Bug fixed with `AdvancedIMU` sensor, when using gravity. Did not work from BeamNGpy before.
  - Bug fixed with `AdvancedIMU` sensor, relating to the smoothing not working from BeamNGpy.
  - Bug fixed with the relative camera, which was not operating correctly.
- Miscellaneous
  - The physics update rate of BeamNG.tech launched from BeamNGpy is being changed from 4000 to 2000 times per second to be consistent with the default for the simulator. To change the physics update rate to a different value, you can pass the `-physicsfps <DESIRED_VALUE>` argument to the simulator binary.
  - Scenarios created using BeamNGpy are now using the JSON format for prefab generation instead of the old TorqueScript format.
  - BeamNG.tech connection to the simulator is now by default listening on the local interface only (`127.0.0.1`). You can change it to listen on other IP addresses by using the `listen_ip` argument in the `BeamNGpy.open` function, or the `-tcom-listen-ip` command-line argument, if you are not launching BeamNG.tech using BeamNGpy.
  - Optimized Python processing of the depth camera image (thanks for the contribution!)

1.8.6.4 Version 1.26.1

- New features
  - OpenDrive (.xodr) importer added, and new example created in Examples folder.
  - OpenStreetMap (.osm) importer and exporter added, and new examples created in Examples folder.
  - Eclipse Sumo (.nod.xml and .edg.xml) importer and exporter added, and new examples created in Examples folder.
- BeamNGpy fixes / improvements
  - Improved/added documentation
    - `Scenario` class now has all parameters documented.
    - `BeamNGpy.debug` API methods are now documented
    - `BeamNGpy.env` now contains more information about the 'time of day' object
    - Added documentation for RADAR and Mesh sensors
  - `Vehicle.set_part_config` now does not recreate the existing connection to the simulator, as it was not needed
  - Small refactor of unit tests, the automated sensor scripts are now also runnable under the `pytest` framework
  - Invalid vehicle and scene object names produced error in the simulation, now the validation is done on BeamNGpy side
    - name cannot start with the `%` character or a digit
    - name cannot contain the `/` character
  - Added new options to `BeamNGpy.scenario.load` called `connect_player_vehicle` and `connect_existing_vehicles`
    - `connect_player_vehicle` is `True` by default and it connects the player vehicle to the simulation after scenario load
    - `connect_existing_vehicles` is `True` by default and it connects all the already existing vehicles to the simulation after scenario load
    - setting these options to `False` can reduce the loading time by skipping the connection-establishing part, and these vehicles can still be connected manually using `Vehicle.connect`
  - Added `crash_lua_on_error` option to the BeamNGpy constructor
    - behaves in the same way as the option of the same name in `BeamNGpy.open`

1.8.6.5 Version 1.26

- RADAR sensor
  - Sensor currently works with static scenery but not vehicles. Will be added in later update.
  - Sensor comes with standard Lua API and BeamNGpy API.
  - Example scripts provided in BeamNGpy.
- Vehicle meshes now available in BeamNGpy
  - Can provide data up to 2000 times per second.
  - Vehicle nodes and physics triangle data available in BeamNGpy, including for individual vehicle wheels.

  - Comes with standard Lua API and BeamNGpy API.
  - Post-processing written in BeamNGpy to compute mesh connectivity data and analyse the mesh data (position, mass, force, velocity).
  - Example scripts provided in BeamNGpy.
- IMU sensor
  - Added ability to filter gyroscopic readings (as well as acceleration readings). Separate data filtering is used for each.
- Sensor suite bug fixes
  - Fix: problem when changing the requested update times/priority parameters after various sensors were already created, sensor would not update correctly/quickly.
  - Fix: gravity vector was not being applied correctly in IMU sensor.
  - Fix: camera images from static sensors were being rendered upside down.
  - Fix: LiDAR sensor was not returning the whole point cloud in BeamNGpy
- Export BeamNG maps as .xodr files (OpenDrive)
  - BeamNGpy now provides the option to export our map road networks as .xodr files (OpenDrive). The exported road networks contain elevation and road wideness data, along with junction connectivity. On top of this, BeamNGpy also includes a new class with which to analyse the road network data oneself, and process it as required.
- BeamNGpy fixes / improvements
  - Optimized the speed of depth camera processing
  - Added new API:
    * `BeamNGpy.env.get_tod` for getting the information about the time of day
    * `BeamNGpy.env.set_tod` for setting the time-of-day information, allowing to control the day/night cycle from Python
    * `BeamNGpy.env.get_gravity` for getting the current value of the strength of gravity in the simulator.
    * `Vehicle.get_center_of_gravity` for getting the center of gravity of a vehicle.
  - Added option to remove procedural meshes
  - Added new option to `BeamNGpy.open` called `crash_lua_on_error`
    * If `False` (the default), then Lua crashes in the simulator will not break the connection between BeamNG.tech and BeamNGpy. Set to `True` for getting proper stacktraces and easier debugging.
  - Added new option to `BeamNGpy.scenario.load` called `precompile_shaders`
    * If `True` (the default), asynchronous shader compilation is disabled. That means the first loading of a map will take longer time, but all parts of the map will be preloaded. If `False`, the camera sensor can have issues shortly after starting the scenario.
  - Better handling of errors and crashes in the BeamNGpy TCP protocol.
  - Fixed `vehicle.control` with zero integer arguments being ignored.
  - Re-added `BeamNGpy.scenario.get_vehicle` (removed by accident in the last release).
  - `BeamNGpy.settings.set_deterministic` and `BeamNGpy.settings.set_steps_per_second` are not persistent anymore and are applied only for a single run of the simulation.

1.8.6.6 Version 1.25.1

- fixed in BeamNG.tech v0.27.1.0: converted all vehicle rotations sent to BeamNGpy to be consistent with each other - if the rotation you are using is 180° rotated across the Y axis, you can use the `beamngpy.quat.flip_y_axis` function to flip it
- fixed `BeamNGpy.vehicles.replace` to respect vehicle color and license plate text

1.8.6.7 Version 1.25

- Added type hints to the whole BeamNGpy codebase
- Updated documentation to be more readable
- Modularized BeamNGpy API
  - The functions on the BeamNGpy object are now split into modules for easier navigation:
    - `BeamNGpy.camera` - configuring the in-game camera
    - `BeamNGpy.control` - controlling the simulator state (pausing, stepping, quitting the simulator)
    - `BeamNGpy.debug` - drawing debug objects
    - `BeamNGpy.env` - controlling the environment state (time of day, gravity)
    - `BeamNGpy.scenario` - loading/starting/stopping a BeamNG scenario
    - `BeamNGpy.settings` - changing the simulator's settings
    - `BeamNGpy.system` - info about the host system
    - `BeamNGpy.traffic` - controlling the traffic
    - `BeamNGpy.ui` - controlling the GUI elements of the simulator
    - `BeamNGpy.vehicles` - controlling vehicles
  - Some of the functions on the `Vehicle` object are also moved into modules for easier navigation:
    - `Vehicle.ai` - controlling the AI of the vehicle
    - `Vehicle.logging` - controlling the in-game logging
  - the previous, not modularized API is still available for backwards compatibility reasons
  - see more in the documentation
- Advanced IMU sensor
  - replaces the accelerometer sensor from last release
  - improves upon the existing IMU sensor by using a more advanced algorithm, and provides readings at up to 2000 Hz
- Powertrain sensor
  - new sensor for analysing powertrain properties at high frequency (up to 2000 Hz)
  - new test/demo scripts are available to show execution of this sensor
- New BeamNGpy functionality
  - added support for a custom binary name in BeamNGpy constructor
  - `BeamNGpy.traffic.spawn` to spawn traffic without a set of predefined vehicles
  - `BeamNGpy.traffic.reset` to reset all traffic vehicles from the player (teleport them away).

  - `Vehicle.teleport` now supports changing rotation without resetting the vehicle
  - `BeamNGpy.open` now always tries to connect to already running simulator no matter the value of the launch argument
  - `Vehicle.switch`, `Vehicle.focus` to switch the simulator's focus to the selected vehicle
  - `BeamNGpy.vehicles.spawn` now has a new argument `connect` to allow for not connecting the newly spawned vehicle to BeamNGpy
  - `Vehicle.recover` to repair a vehicle and teleport it to a drivable position
  - `BeamNGpy.vehicles.replace` to replace a vehicle with another one at the same position
  - `beamngpy.quat.quat_multiply` utility function to multiply two quaternions
  - optimized the `Camera` sensor decoding to be faster
  - updated the required Python packages to newer versions
  - `Vehicle.set_license_plate` to set a license plate text for a vehicle
  - `Vehicle.sensors.poll` now allows also polling only a specified list of sensor names
  - `BeamNGpy.disconnect` to disconnect from the simulator without closing it
  - changed `Camera` sensor default parameters to not include annotation and depth data (for faster polling)
  - added the optional `steps_per_second` parameter to `BeamNGpy.settings.set_deterministic`
  - `BeamNGpy.control.return_to_main_menu` to exit the currently loaded scenario
  - added the parameter `quit_on_close` to the BeamNGpy constructor. If set to `False`, `BeamNGpy.close` will keep the simulator running.
- Bugfixes
  - `Vehicle.state['rotation']` now returns vehicle rotation consistent with the rest of the simulator. Previously, this rotation was rotated 180° around the Y axis.
    * if you are using `Vehicle.state['rotation']` in your existing scripts, you may need to flip it back for your intended use. You can use `beamngpy.quat.quat_multiply((0, 0, 1, 0), <your_rotation>)` for that purpose.
  - fixed the issue with BeamNGpy scenarios sometimes resetting and not working properly after loading
  - fixed `Camera.extract_bounding_boxes` not to crash on non-Windows systems
  - fixed `beamng.scenario.start()` not working when the simulator was paused with `beamng.control.pause()` before
  - fixed vehicle color and license plate text not being applied to dynamically spawned vehicles
- BeamNGpy protocol: added support for out-of-order protocol messages
- Deprecations
  - the `remote` argument of the `BeamNGpy` class is not used anymore

1.8.6.8 Version 1.24

- Major changes to the protocol communicating between BeamNG.tech and BeamNGpy
  - Be aware that versions of BeamNG.tech older than 0.26 are not compatible with BeamNGpy 1.24 and older versions of BeamNGpy will not work with BeamNG.tech 0.26.
- Major updates to BeamNGpy sensor suite and its API
  - The public API of the `Camera`, `Lidar` and `Ultrasonic` sensors changed heavily, please see the `examples` folder to see their usage.
- Accelerometer sensor now available
- Add support for loading TrackBuilder tracks
- Add support for loading Flowgraph scenarios
- Fix: multiple vehicles now do not share color in instance annotations
- Add `Vehicle.teleport` helper function which allows to teleport a vehicle directly through its instance
- `BeamNGpy.open` now tries to (re)connect to already running local instance
- Removed deprecated BeamNGpy functionality
  - `setup_logging` (superseded by `set_up_simple_logging` and `config_logging`)
  - `rot` argument used for setting rotation of objects and vehicles in Euler angles, use `rot_quat` which expects quaternions (you can use the helper function `angle_to_quat` to convert Euler angles to quaternions)
  - `update_vehicle` function is removed
  - the `requests` argument in `Vehicle.poll_sensors` is removed
  - `poll_sensors` now does not return a value
  - the `deploy` argument of `BeamNGpy.open` is removed

1.8.6.9 Version 1.23.1

- Add Feature Overview notebook
- Add argument checking to the IMU sensor
- Add support for Mesh Roads
- Add option to log BeamNGpy protocol messages
- Fix duplicate logging when calling `config_logging` multiple times

1.8.6.10 Version 1.23

- Fix semantic annotations (supported maps are Italy and ECA)
- Add option to teleport vehicle without resetting its physics state
- Add option to set velocity of a vehicle by applying force to it
- Support for updated ultrasonic sensor
- New sensor API - LiDAR, ultrasonic sensor
- Fix camera sensor creating three shared memories even when not needed

- Add BeamNGpy feature overview example notebook
- Remove research mod deployment and `setup-workspace` phase of setup
- (Experimental) Support for Linux BeamNG.tech servers

1.8.6.11 Version 1.22

- Hide menu on a scenario start
- Do not detach the state sensor on disconnecting a vehicle, as this disallows the reuse of vehicle objects
- Fix camera sensor logging error
- Fix 'Using mods with BeamNGpy' demo notebook

1.8.6.12 Version 1.21.1

- Fix example notebooks

1.8.6.13 Version 1.21

- Fix and restructure `logging` usage
- Add more verbose logging
- Fix message chunking in networking
- Update examples/tests to address GridMap being gone
- Improve handling of userpath discovery and mod deployment

1.8.6.14 Version 1.20

- Adjust userpath handling according to changes in BeamNG.drive from 0.22 onwards
- Overhaul documentation style and structure
- Add function to set up userpath for BeamNG.tech usage
- Add multicam test
- Fix issue when multiple functions are waiting in researchGE.lua
- Fix instance annotations always being rendered even when not desired

1.8.6.15 Version 1.19.1

- Swap client/server model to allow multiple BeamNGpy instances to connect to one running simulator simultaneously
- Add `Level` class representing a level in the simulation
- Change `Scenario` class to point to `Level` it is in
- Add `get_levels`, `get_scenarios`, `get_level_scenarios`, `get_levels_and_scenarios` methods to `BeamNGpy` class to query available content
- Add `get_current_scenario` method to `BeamNGpy` class to query running scenario

- Add `get_current_vehicles` method to `BeamNGpy` class to query active vehicles
- Add `SceneObject` class to the `scenario` module as a basis for the various types of objects in a scene in BeamNG.tech, currently including `DecalRoad`
- Add `get_scenetree` and `get_scene_object` methods to `BeamNGpy` class to enable querying objects in the active scene
- Add `add_debug_spheres`, `add_debug_polyline`, `add_debug_cylinder`, `add_debug_triangle`, `add_debug_rectangle`, `add_debug_text`, `add_debug_square_prism` methods to `BeamNGpy` class to visualize 3D gizmos in the simulator
- Add Inertial Measurement Unit sensors
- Add Ultrasonic Distance Measurement sensor
- Add noise module to randomize sensor data for cameras and lidars
- Add instance annotation option to `Camera` sensor including methods to `extract_bboxes`, `export_bbox_xml`, and `draw_bboxes` for bounding-box-related operations based on semantic and instance annotations (limited to vehicles right now)
- Add options to use only socket-based communication for `Camera` and `Lidar` sensor
- Add methods to configure BeamNG.tech's Vehicle Stats Logger from BeamNGpy
- Add FAQ to README
- Add Contributor License Agreement and guidelines
- Fix stray dependency on PyScaffold
- Fix lidar points being visible in camera sensor images

1.8.6.16 Version 1.18

- Add function to switch current viewport to the relative camera mode with options to control the position of the camera
- Add function to display debug lines in the environment
- Add function to send Lua commands to be executed inside the simulation

1.8.6.17 Version 1.17.1

- Fix deterministic mode ignoring user-defined steps per second

1.8.6.18 Version 1.17

- Add `change_setting` and `apply_graphics_setting` methods including a usage example
- Add option to specify rotations as quaternions where appropriate
- Add example for querying the road network

1.8.6.19 Version 1.16.5

- Fix prefab compilation

1.8.6.20 Version 1.16.4

- Add `teleport_scenario_object` method to `BeamNGpy` class
- Update vehicle state example
- Fix decal road positioning
- Fix `spawn_vehicle` not setting color and license plate correctly
- Fix `spawn_vehicle` rotation in degrees

1.8.6.21 Version 1.16.3

- Fix lidar visualizer using wrong buffer types in newer PyOpenGL version

1.8.6.22 Version 1.16.2

- Update values of *Electrics* sensor not following our naming conventions
- Fix camera orientation issue
- Add example for using the *Camera* sensor like a multishot camera

1.8.6.23 Version 1.16.1

- Fix spaces in vehicle names breaking the scenario prefab

1.8.6.24 Version 1.16

- Make BeamNGpy ship required Lua files and deploy them as a mod on launch
- Add traffic controls
- Add option to specify additional Lua extensions to load per vehicle
- Add `set_lights` method to vehicle class
- Add test for setting lights
- Add test for vehicle bounding box
- Add `over_objects` field to `Road` class
- Fix lack of *__version__*
- Fix electrics sensor not returning values directly
- Fix *ai_set_script* teleporting vehicle

1.8.6.25 Version 1.15

- Add option to pass additional Lua extensions to be loaded on startup
- Fix waiting for vehicle spawn after changing parts to hang infinitely

1.8.6.26 Version 1.13

- Add option to disable node interpolation on roads
- Add *get_bbox()* method to *Vehicle* class

1.8.6.27 Version 1.12

- Add option to specify road ID for placed DecalRoads

1.8.6.28 Version 1.11

- Add `StaticObject` class to scenario module that allows placement of static meshes
- Add option for visualization to the Lidar sensor
- Add helper functions to query scenario for certain objects in the world
- Add example notebook showcasing procedural mesh and static mesh placement including a scenario camera
- Fix vehicle state not being synchronized properly
- Fix scenario unloading glitch
- Fix `ai_drive_in_lane` not updating GUI state correctly
- Fix camera sensor showing residual head-/taillight flare

1.8.6.29 Version 1.10

- Add functions to spawn/despawn vehicles during a scenario
- Add script AI function to vehicle and update AI line example accordingly
- Add function to change AI aggression
- Add functions to place procedurally generated primitives in the environment
- Add unit tests for sensors, scenarios, and vehicles
- Fix scenario not being cleared when BeamNG instance is closed

1.8.6.30 Version 1.9.1

- Make scenario generation & loading respect user path setting

1.8.6.31 Version 1.9

- Add function to switch active vehicle
- Add function to set position & orientation of the ingame camera

1.8.6.32 Version 1.8

- Add vehicle teleporting function to `BeamNGpy` class
- Add time of day control
- Add function to switch weather presets
- Add function to await vehicle spawns
- Expose part configuration options of vehicles
- Expose current part configuration of vehicles
- Add function to change part configuration of vehicles
- Add function to change vehicle colour
- Add more documentation

1.8.6.33 Version 1.7.1

- Make ai methods switch to appropriate modes

1.8.6.34 Version 1.7

- Add manual gear control
- Add shift mode control

1.8.6.35 Version 1.6

- Add option to set target waypoint for builtin vehicle AI
- Make shmem handle unique OS-wide

1.8.6.36 Version 1.5

- Add `get_gamestate()` to `BeamNGpy` class
- Make vehicle state being synched upon initial connection
- Fix vehicle state not being updated on poll if only gameengine-specific sensors were attached.

1.8.6.37 Version 1.4

- Add vehicle-level state updates
- Rework code to work with existing scenarios/vehicles

1.8.6.38 Version 1.3

- Add support to specify polyline with per-vertex speed to the AI

1.8.6.39 Version 1.2

- Add wait option to step function in `beamng.py`

1.8.6.40 Version 1.1

- Add basic Lidar point cloud visualiser
- Add AI control to vehicles
- Add option to attach cameras to scenarios to render frames relative to world space

1.8.6.41 Version 1.0

- Restructure code to offer modular sensor model
- Implement scenario class to specify and generate BeamNG scenarios
- Implement vehicle class that offers control over vehicles and ways to dynamically de-/attach sensors
- Implement shared memory communication to boost performance
- Add Camera sensor with colour, depth, and annotation data
- Add multi-cam support
- Add lidar sensor
- Add G-Force sensor
- Add damage sensor
- Add electrics sensor
- Add control over simulation timescale and stepping through simulation at fixed rates
- Add example code demonstrating scenario specification with control of a vehicle that has various sensors attached

1.8.6.42 Version 0.4

- Add `move_vehicle()` method.

1.8.6.43 Version 0.3.6

- Pass configured host and port to BeamNG.drive process.

1.8.6.44 Version 0.3.5

- Fix `close()` in `BeamNGPy` not checking if there's even a process to be killed.

1.8.6.45 Version 0.3.4

- Fix messages being split incorrectly when the message happened to contain a newline through msgpack encoding.

1.8.6.46 Version 0.3.3

- Make `BeamNGPy` class take `**options` and add `console` as one to allow running BeamNG.drive with the console flag.

1.8.6.47 Version 0.3.2

- Make BeamNGpy assume a running instance if binary is set to `None`
- Add option to change vehicle cursor

1.8.6.48 Version 0.3.1

- Add `restart_scenario` method to restart a running scenario

1.8.6.49 Version 0.3

- Add method to pause simulation
- Add method to resume simulation

1.8.6.50 Version 0.2

- Add option to specify image size when requesting vehicle state
- Add blocking method to get vehicle state
- Add method to set relative camera
- Add methods to hide/show HUD
- Default to realistic gearbox behaviour
- Add `gear` property to vehicle state
- Add `gear` as an option to vehicle input representing the gear the vehicle is supposed to shift to.

1.8.6.51 Version 0.1.2

- Remove fstrings from documentation
- Add option to override BeamNG.drive binary being called

1.8.6.52 Version 0.1

- Basic IPC and example functions

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W