Simulation-based Testing with BeamNG.tech

Tutorial @ SBST 2021

Alessio Gambi, Marc Müller, Pascale Maul

The Team

What’s BeamNG.tech?

- Some basic info
- Rebranded from BeamNG.research
- Works only on Windows (we show also Parallels)
- KVM can be used to run into a VM (advanced, requires GPU pass-through)
- Linux support coming soon!
- **Documentation link (work in progress)**

BeamNGpy

The Python API to BeamNG.tech

- FAQ: Why Python?
- FAQ: Can one use Java or some other language? What would be required to do so? Is the API “Rest” ?

Who created (and maintains) it?

Who uses it?

- Researchers
- SBST Tool competition
- Academia (Seminars)
- Add some work here

Goals of the Tutorial

It's a *Beginners* tutorial

- Get a grip on BeamNG.tech (5min)
- Basics of simulation-based tests (20min)
- Automated test generation (15min)

Disclaimer: We show only a fraction of the possibilities that BeamNG.tech enables. The simulator is designed to be extensible (via "mods" for example) so most its code is **open source**.

Part I: Getting Started

Obtaining the Simulator

Getting Started

- The simulator can be obtained **for free** after registering at https://register.beamng.tech/
- Use a valid **university** email address for the registration
- After registration you should get a confirmation email (check the spam folder!) with a `tech.key` file and the link to download the simulator.
- In this tutorial, we use the **BeamNG.tech version `v0.21.3.0`**
- Officially, the simulator can run only under Windows, but you can try it also on Mac OS using Parallels (note, this is **shareware**)

BeamNG.tech Application Form

register.beamng.tech

BeamNG.tech

BeamNG.tech software is a great solution for everyone who uses simulation in an individual and need-based manner to be used across industry especially suitable for the automotive sector. With BeamNG.tech you have direct access to a wide range of features which are exclusively available in this package.

BeamNG.tech software for commercial purposes is subject to annual maintenance and support fees. For more information, please contact us.

Academic and non-commercial licenses are provided free of charge at our discretion. The application process requires filling out the following information including your name and email address. **Please use your professional/academic email address as proof of membership.** Upon submitting the application request you will receive an email notifying you about us having received it, and an email containing out response within 5 business days. If approved, the email will include download and installation instructions.

Name:

Enter your full name

Email Address:

Enter email

Application Text:

Installing the Simulator

Getting Started

- The simulator comes as "tar-ball" (actually a zip-ball)
- The simulator does not require any specific installation, just expand it somewhere it can fit (~16.5 GB)
- **Avoid** using **special characters** and **spaces**
- We will refer to this folder as `<BNG_HOME>`

Running the Simulator

Getting Started

- The simulator can be **started manually** by double-clicking on

```
<BNG_HOME>\\Bin64\\BeamNG.tech.x64.exe
```

- Note that starting the simulator like this will **not** automatically start the Python API so you will not able to control remotely the simulator.

- However, you can use the simulator via it GUI that is useful to

  - browse around existing maps

  - create new content using the various editors

Video

Running the Simulator

Getting Started

- The simulator can be **started** from Powershell using the following command:

```
 <BNG_HOME>\\Bin64\\BeamNG.tech.x64.exe -console -rport 64256 -nosteam
-physicsfps 4000 -lua "registerCoreModule('util/researchGE')" -userpath
<BNG_USER>
```

- <BNG_USER> is the *work dir* of the simulator, you can choose any folder on your system as long as that is writable and have no spaces or special characters in its name and path,

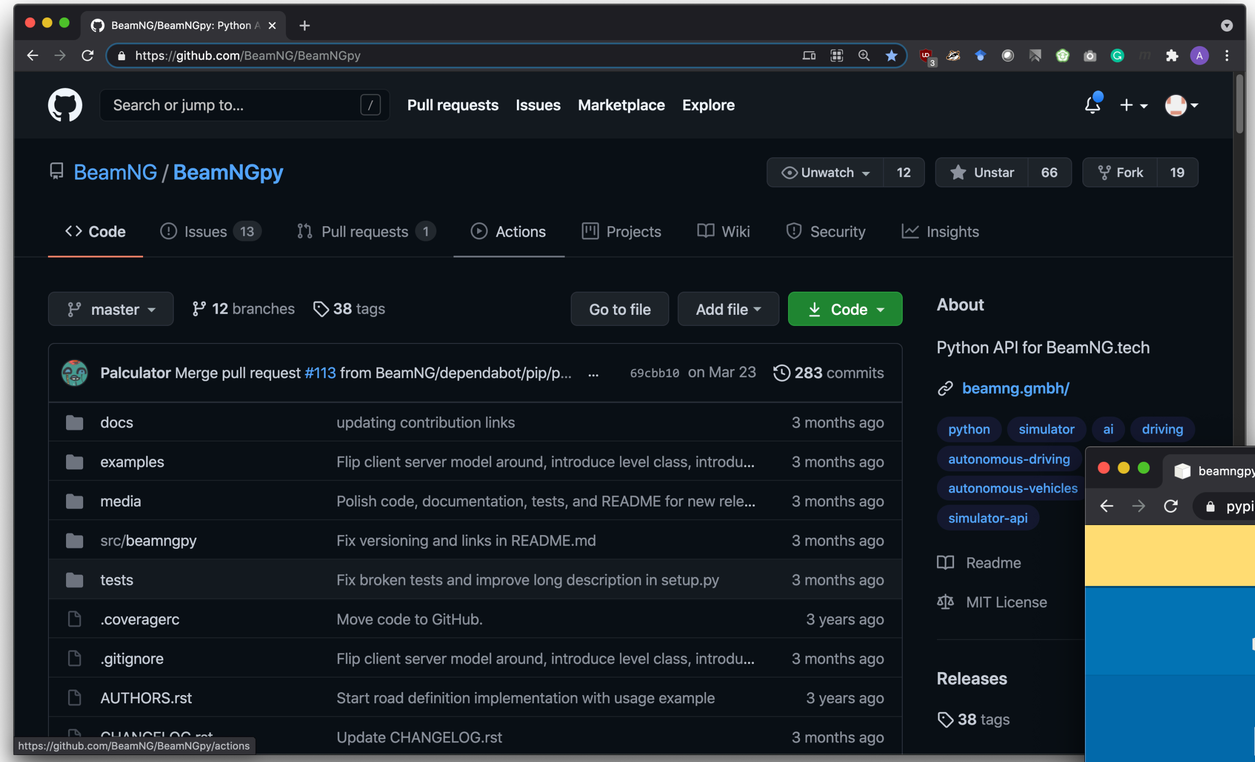
- <BNG_USER> must contain the registration key (i.e., `tech.key`) that you received after the registration, the simulator will **not** start if it cannot find this file.

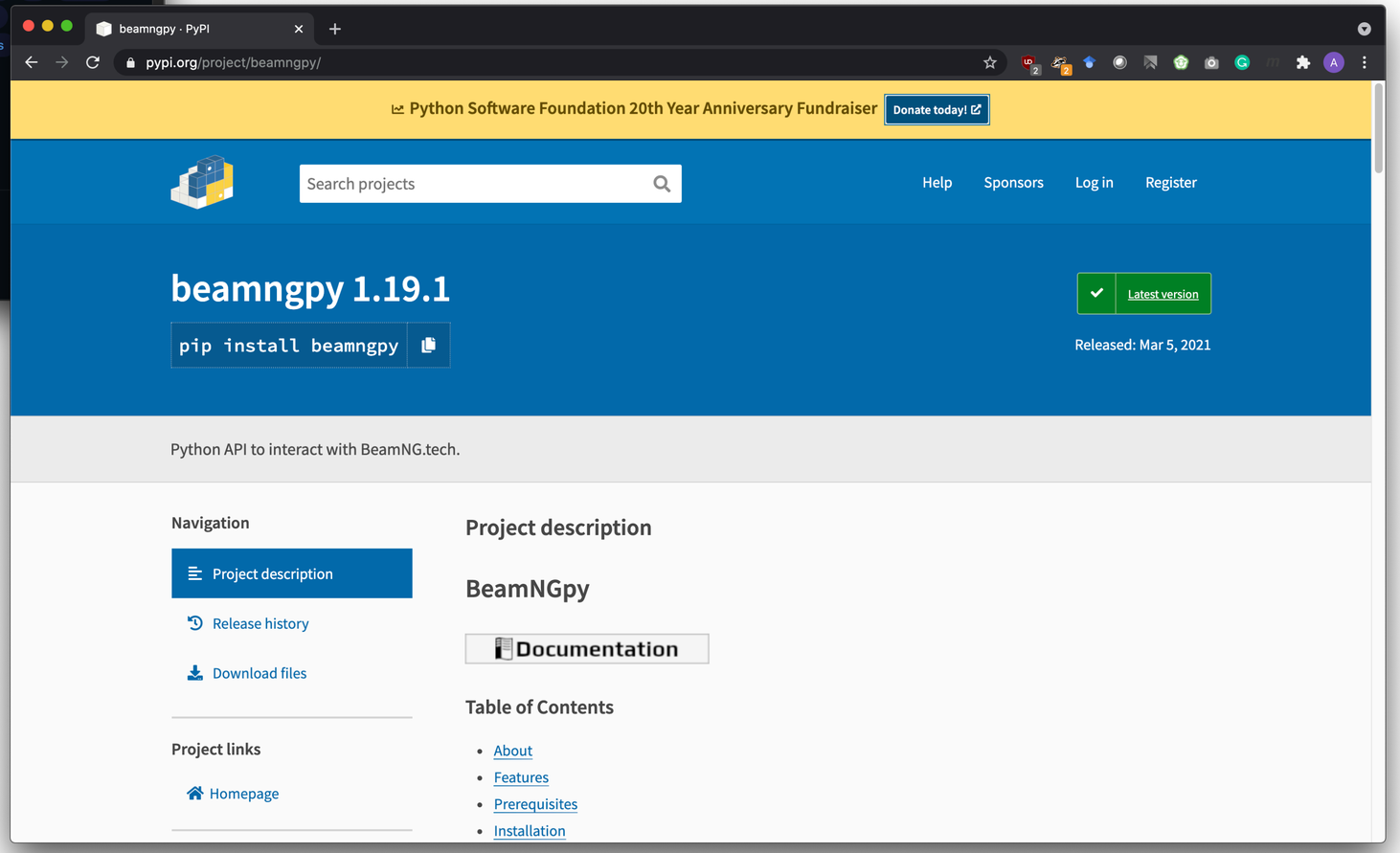
Video

Running the Simulator

Getting Started

- The simulator can be **started** (and **controlled**) using its Pyhon API: `BeamNGpy`
- `BeamNGpy` is open source and available both on GitHub and on PyPI as `beamngpy`
- Running the simulator using the Python API requires to set an env variable called `BNG_HOME` pointing to `<BNG_HOME>` or provide this value as input parameter.
- `<BNG_USER>` instead is not mandatory. If not specified, `BeamNGpy` will default its value to `~\\Documents\\BeamNG.tech_userpath`
  - Remember that there should be no special characters or empty space


BeamNG/BeamNGpy: Python
https://github.com/BeamNG/BeamNGpy
Search or jump to...
Pull requests
Issues
Marketplace
Explore
BeamNG / BeamNGpy
Unwatch 12
Unstar 66
Fork 19
Code
Issues 13
Pull requests 1
Actions
Projects
Wiki
Security
Insights
master
12 branches
38 tags
Go to file
Add file
Code
Palculator Merge pull request #113 from BeamNG/dependabot/pip/p... ... 69cbb10 on Mar 23 283 commits
docs updating contribution links 3 months ago
examples Flip client server model around, introduce level class, introdu... 3 months ago
media Polish code, documentation, tests, and README for new rele... 3 months ago
src/beamngpy Fix versioning and links in README.md 3 months ago
tests Fix broken tests and improve long description in setup.py 3 months ago
.coveragerc Move code to GitHub. 3 years ago
.gitignore Flip client server model around, introduce level class, introdu... 3 months ago
AUTHORS.rst Start road definition implementation with usage example 3 years ago
Update CHANGELOG.rst 3 months ago
About
Python API for BeamNG.tech
beamng.gmbh/
python
simulator
ai
driving
autonomous-driving
autonomous-vehicles
simulator-api
Readme
MIT License
Releases
38 tags
https://github.com/BeamNG/BeamNGpy/actions


beamngpy · PyPI
pypi.org/project/beamngpy/
Python Software Foundation 20th Year Anniversary Fundraiser
Donate today!
Search projects
Help
Sponsors
Log in
Register
beamngpy 1.19.1
pip install beamngpy
Latest version
Released: Mar 5, 2021
Python API to interact with BeamNG.tech.
Navigation
Project description
Release history
Download files
Project links
Homepage
Project description
BeamNGpy
Documentation
Table of Contents
About
Features
Prerequisites
Installation

Installing the Dependencies

- `Code\README.md` contains the verbose descriptions on how to install the Python dependencies, please refer to that if you face issues.
- The short version is:
  - create a new virtual environment: `py.exe -m venv .venv`
  - activate it: `.\.venv\Scripts\activate`
  - update pip: `py.exe -m pip install --upgrade pip`
  - update utilities: `pip install --upgrade setuptools wheel`
  - Install the library: `pip install beamngpy==1.19.1`

Running the Simulator

Assumption: BeamNGpy correctly installed

```python
from beamngpy import BeamNGpy, Scenario, Road

# Specify where BeamNG home and user are
BNG_HOME = "C:\\BeamNG.tech.v0.21.3.0"
BNG_USER = "C:\\BeamNG.tech_userpath"

beamng = BeamNGpy('localhost', 64256, home=BNG_HOME, user=BNG_USER)

# Start BeamNG by setting launch to True
bng = beamng.open(launch=True)

try:
    input('Press enter when done...')
finally:
    bng.close()
```

example1.py

Running the Simulator

Assumption: BeamNGpy correctly installed

```
from beamngpy import BeamNGpy, Scenario, Road

# Specify where BeamNG home and user are
BNG_HOME = "C:\\BeamNG.tech.v0.21.3.0"
BNG_USER = "C:\\BeamNG.tech_userpath"

# As a Python Contenxt Manager
with BeamNGpy('localhost', 64256, home=BNG_HOME, user=BNG_USER):
    input('Press enter when done...')
```

Video

Part II

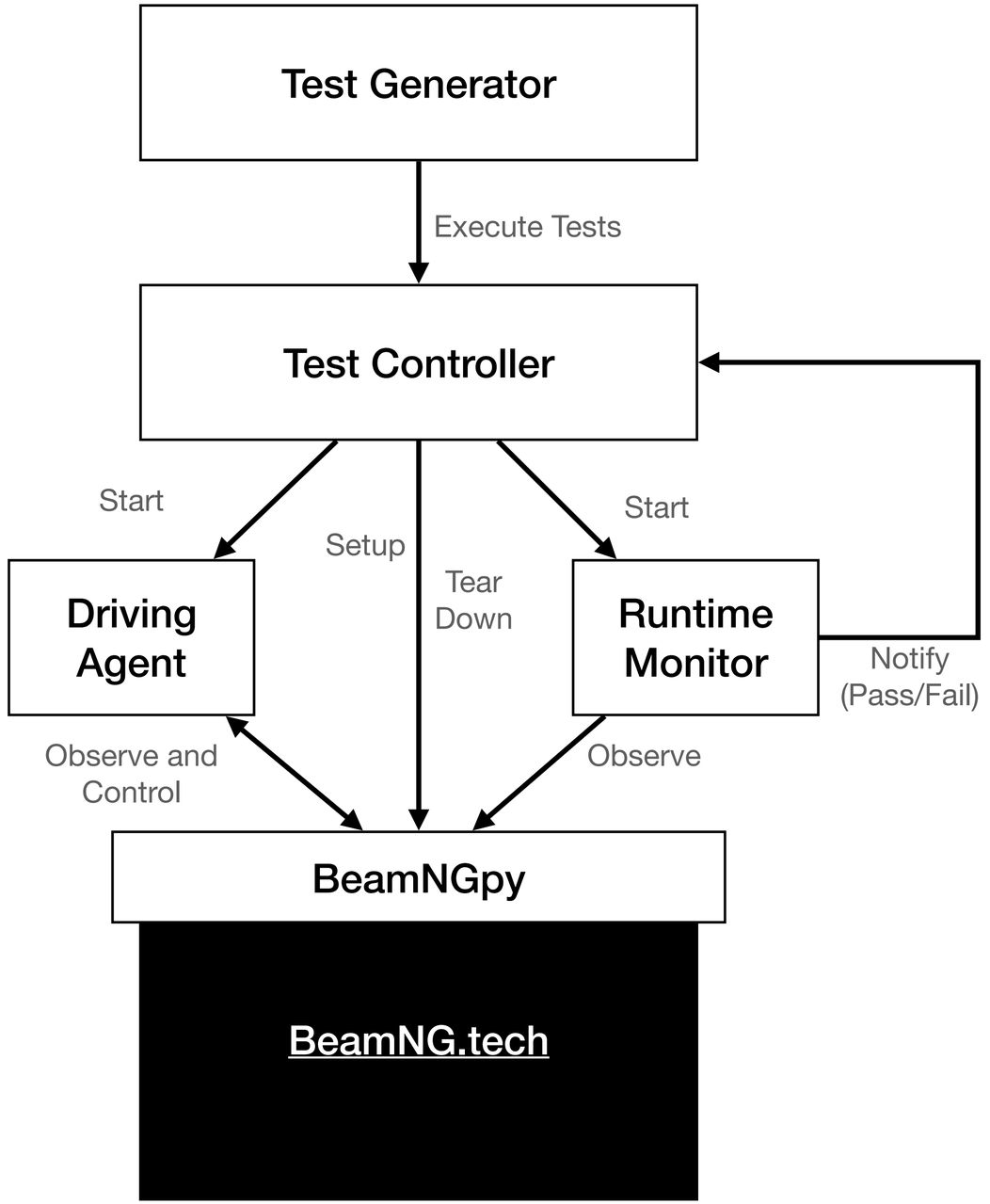
Simulation-Based Testing

Introduction

- Simulation-based testing can be used to implement **X**-in-the-loop
  - We focus on Software-in-the-loop (SIL)
  - We focus on System level testing
- Test subjects are **continuous controllers** and may be based on **ML/AI** that are notoriously difficult, ineffective, or impossible to test by traditional means
- No ROS/ROS2 integration yet

Reference Architecture

- **BeamNG.tech** the simulator (considered as black-box for the moment)
- **BeamNGpy** API to BeamNG.tech, allow for multiple simultaneous connections.
- **Runtime Monitor** monitor execution, check oracles
- **Driving Agent** test subject (get sensor data, drive virtual ego-car
- **Test Controller** start/stop simulation, start/stop tests (e.g., `pytest, unittest`)
- **Test Generator** (either human or algorithm) to generate the tests.



Anatomy of a Simulation-based Test

- **Precondition:** A running simulator
- Set the **environment** (terrain, map, roads)
- Set the test **initial conditions** (placement of vehicles, obstacles)
- Configure **runtime monitors** (positive/negative oracles)
- Configure the **test subject** (connect to ego-car, instruct about test goal/task)
- Start the **execution/simulation**

Anatomy of a Simulation-based Test

- **Precondition:** A running simulator
- Set the **environment** (terrain, map, roads)
- Set the test **initial conditions** (placement of vehicles, obstacles)
- Configure **runtime monitors** (positive/negative oracles)
- Configure the **test subject** (connect to ego-car, instruct about test goal/task)
- Start the **execution/simulation**

Fresh vs Shared Simulator Instances

- One can start a new instance of the simulator **for each test** or start only one instance of the simulator and **reuse it for all the tests**
- Starting a fresh instance may reduce flakyness/pollution, but it takes more time
- One can also run **multiple concurrent instances** of the simulation (not shown in this tutorial)

Fresh Instance

```
def do_the_sleep(for_seconds):
    print('Sleep for', for_seconds, 'seconds and stop')
    for i in reversed(range(1, for_seconds, )):
        print(i)
        sleep(1)

class StartFreshInstanceOfTheSimulator(unittest.TestCase):

    def test_that_simulation_start(self):
        with BeamNGpy('localhost', 64256, home=BNG_HOME, user=BNG_USER):
            do_the_sleep(5)

    def test_that_simulation_restart(self):
        with BeamNGpy('localhost', 64256, home=BNG_HOME, user=BNG_USER):
            do_the_sleep(3)
```

simple-test.py

Fresh Instance (2)

```
class StartFreshInstanceOfTheSimulatorUsingSetupAndTearDown(unittest.TestCase):

    def setUp(self):
        self.beamng = BeamNGpy('localhost', 64256, home=BNG_HOME, user=BNG_USER)
        self.beamng = self.beamng.open(launch=True)

    def tearDown(self):
        if self.beamng:
            self.beamng.close()

    def test_that_simulation_start(self):
        do_the_sleep(5)

    def test_that_simulation_restart(self):
        do_the_sleep(3)
```

simple-test.py

Shared Instance

```
class SharedInstance(unittest.TestCase):
    # Static Field
    beamng = None

    @classmethod
    def setUpClass(cls):
        bng = BeamNGpy('localhost', 64256, home=BNG_HOME, user=BNG_USER)
        cls.beamng = bng.open(launch=True)

    @classmethod
    def tearDownClass(cls):
        if cls.beamng:
            cls.beamng.close()
```

simple-test.py

Shared Instance

```
class SharedInstance(unittest.TestCase):
    # Static Field
    beamng = None

    @classmethod
    def setUpClass(cls):
        bng = BeamNGpy('loca
        cls.beamng = bng.open

    @classmethod
    def tearDownClass(cls):
        if cls.beamng:
            cls.beamng.close()
```

```
    def test_that_can_connect_to_simulator(self):
        client_a = BeamNGpy('localhost', 64256, home=BNG_HOME, user=BNG_USER)
        try:
            client_a.open(launch=False, deploy=False)
            do_the_sleep(2)
        finally:
            client_a.skt.close()

    def test_that_can_reconnect_to_simulator(self):
        client_b = BeamNGpy('localhost', 64256, home=BNG_HOME, user=BNG_USER)
        try:
            client_b.open(launch=False, deploy=False)
            do_the_sleep(5)
        finally:
            client_b.skt.close()
```

simple-test.py

Anatomy of a Simulation-based Test

- **Precondition:** A running simulator
- Set the **environment** (terrain, map, roads)
- Set the test **initial conditions** (placement of vehicles, obstacles)
- Configure **runtime monitors** (positive/negative oracles)
- Configure the **test subject** (connect to ego-car, instruct about test goal/task)
- Start the **execution/simulation**

Levels and Scenarios

- Simulation-based tests with BeamNG take place inside **scenarios**
- Scenarios are defined in the context of **levels**
- Levels contain terrains, maps, props, materials, and other meta-data required to execute the simulation
- Levels must be manually defined, while scenarios can be defined programmatically
- Currently, BeamNG.tech contains 19 levels (more can be downloaded)
- Levels are locate inside the `levels` folders inside `<BNG_HOME>` and `<BNG_USER>`

Drive on Existing Roads

- Almost all the **existing** levels come with large **maps** that include **roads**
- These levels can be loaded and used "as they are" or can be customized

```
with BeamNGpy('localhost', 64256, home=BNG_HOME, user=BNG_USER) as bng:
    scenario = Scenario('Utah', 'ai_test')
    scenario.make(bng)
    bng.load_scenario(scenario)
    bng.start_scenario()
    input('Press enter when done...')
```

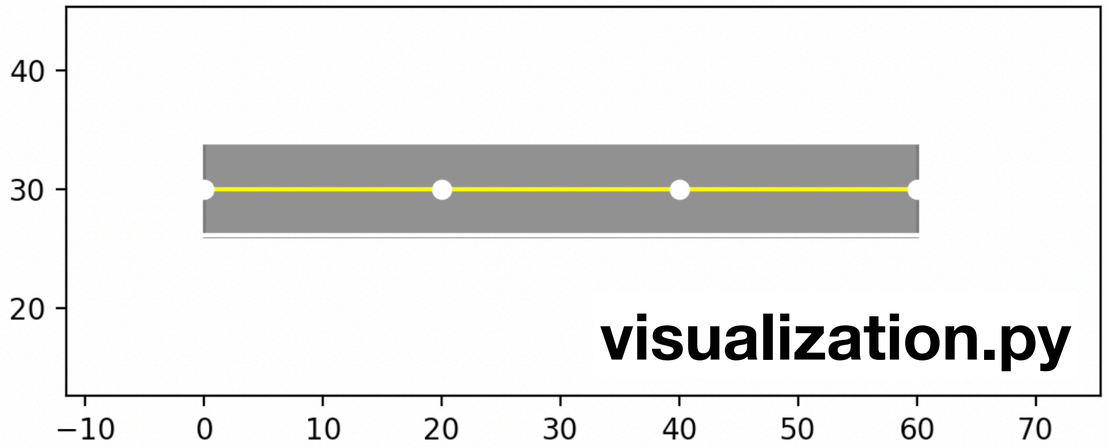
load_scenario_and_roads.py

Procedural Road Generation

- One can create "simple" levels without roads and generate roads procedurally
- (Decal) roads are defined as sequences of points `(x,y,z)` with `width`

```
road_nodes = [(0, 30, 0, 8), (20, 30, 0, 8), (40, 30, 0, 8), (60, 30, 0, 8)]
# TIG from Precrime Group, Lugano
scenario = Scenario('tig', 'test_scenario_1')
# The material is plain asphalt, no lane marking
road = Road('road_rubber_sticky', rid='road_1')
road.nodes.extend(road_nodes)
scenario.add_road(road)
scenario.make(bng)
bng.load_scenario(scenario)
bng.start_scenario()
```

load_scenario_and_roads.py



visualization.py

Procedural Road Generation (2)

- One way to generate roads is to abstract them into **sequences of segments** such as straight and turns, and then generate the road nodes from there like **AsFault** does
- Many other techniques are available implemented by tools like **DeepJanus**, **DeepHyperion**, **Deeper**, **Frenetic**, **SWAT**, and **GA-Bezier**
- Procedurally generated roads do not come with **lane markings**

trajectory_generator.py

Lane Markings

- Create a road material that *embeds* the lane marking (e.g., DeepJanus)

```
road_nodes = generate_road_nodes()

with BeamNGpy('localhost', 64256, home=BNG_HOME, user=BNG_USER) as bng:
    scenario = Scenario('tig', 'test_scenario_2')
    # This material comes already with lane markings
    road = Road('tig_road_rubber_sticky', rid='road_1')
    road.nodes.extend(road_nodes)
    scenario.add_road(road)
    scenario.make(bng)

    bng.load_scenario(scenario)
    bng.start_scenario()
```

load_scenario_and_roads.py

Lane Markings

- Stack the lane markings (thin roads) on top of the actual road (e.g., AsFault).

```
right_marking = Road('line_white', rid='right_white')
right_marking.nodes.extend(right_marking_nodes)
scenario.add_road(right_marking)

left_marking = Road('line_white', rid='left_white')
left_marking.nodes.extend(left_marking_nodes)
scenario.add_road(left_marking)

central_marking = Road('line_yellow', rid='central_yellow')
central_marking.nodes.extend(central_marking_nodes)
scenario.add_road(central_marking)

road = Road('road_rubber_sticky', rid='road_1')
road.nodes.extend(road_nodes)
scenario.add_road(road)
```

First Steps

Customize the Environment

- **Task:** Change the color of the sky, clouds, weather, Time-of-Day (not sure what else/can be done)
- (Only what it is there)

Anatomy of a Simulation-based Test

- **Precondition:** A running simulator
- Set the **environment** (terrain, map, roads)
- Set the test **initial conditions** (placement of vehicles, obstacles)
- Configure **runtime monitors** (positive/negative oracles)
- Configure the **test subject** (connect to ego-car, instruct about test goal/task)
- Start the **execution/simulation**

Simulation-based Testing

Anatomy of a (Simulation-based) Test

- **Assume** a running simulator
- **Setup** the environment (level/terrain, map, materials, roads)
- **Setup** the test initial conditions (placement of ego-car, obstacles, NPC vehicles, etc.)
- **Setup** the test subject (connect to ego-car, instruct about test goal)
- **Monitor** and **assert** behavior (runtime monitors, positive/negative oracles)
- **Run** the simulation

Place vehicles

Placement and orientation (on flat map)

- **Task:** place the ego car at the beginning of the road
- **Task:** place another vehicle on the road (in front of the ego car)
- **Task:** place another vehicle on the road in the opposite lane

Place vehicles

Placement and orientation (non-flat map)

- **Task:** place the ego car on a step road
- Show how to find where the placement his
- Use: Query for Waypoint?

Place Obstacles and Props

Procedurally generated obstacles

- **Task:** Flat road, straight segment(?), place procedurally generated obstacles
-

Place Obstacles and Props

Parked cars (if no , Lightpoles/Trafficlights

- **Task:** Flat road, straight segment(?), place procedurally obstacles from models.
- Show that the car can crash into it

Anatomy of a Simulation-based Test

- **Precondition:** A running simulator
- Set the **environment** (terrain, map, roads)
- Set the test **initial conditions** (placement of vehicles, obstacles)
- Configure **runtime monitors** (positive/negative oracles)
- Configure the **test subject** (connect to ego-car, instruct about test goal/task)
- Start the **execution/simulation**

Simulation-based Testing

Anatomy of a (Simulation-based) Test

- **Assume** a running simulator
- **Setup** the environment (level/terrain, map, materials, roads)
- **Setup** the test initial conditions (placement of ego-car, NPC vehicles, obstacles)
- **Setup** the test subject (connect to ego-car, instruct about test goal)
- **Monitor** and **assert** behavior (runtime monitors, positive/negative oracles)
- **Run** the simulation

Collect Data (to drive)

- **Task**: Start a second python process (from the test code is ok for example using multiprocessing API) that connect to a running simulation where a scenario (e.g., straight road) is already loaded. The vehicle is setup with some sensors (camera, electronic) and we visualize the values from those sensors every X steps (e.g., show the camera)

- Possible sensors to show:
  - camera, pixel annotated camera
- Mention that there are other sensors as well!

Control the EgoVehicle

- **Task**: Start a second python process that connects to a running simulation where a scenario (e.g., straight road) is already loaded. And we also send control commands: we may use a simple logic, if we are getting too close to the left/right side of the lane we steer towards the center. Or we can hardcode some control commands (steer, acc/dec). Or we drive it manually: right arrow -> steer right command.

- What commands are available to show case? steering, acc, braking? Indicators? anything else?

Instructing the EgoCar…

- This is test case specific and application specific. We can set the destination or tell the car to keep driving…

Anatomy of a Simulation-based Test

- **Precondition:** A running simulator
- Set the **environment** (terrain, map, roads)
- Set the test **initial conditions** (placement of vehicles, obstacles)
- Configure **runtime monitors** (positive/negative oracles)
- Configure the **test subject** (connect to ego-car, instruct about test goal/task)
- Start the **execution/simulation**

Simulation-based Testing

Anatomy of a (Simulation-based) Test

- **Assume** a running simulator
- **Setup** the environment (level/terrain, map, materials, roads)
- **Setup** the test initial conditions (placement of ego-car, NPC vehicles, obstacles)
- **Setup** the test subject (connect to ego-car, instruct about test goal)
- **Monitor** and **assert** behavior (runtime monitors, positive/negative oracles)
- **Run** the simulation

Simulation-based Testing

Anatomy of a (Simulation-based) Test

- **Assume** a running simulator
- **Setup** the environment (level/terrain, map, materials, roads)
- **Setup** the test initial conditions (placement of ego-car, NPC vehicles, obstacles)
- **Setup** the test subject (connect to ego-car, instruct about test goal)
- **Monitor** and **assert** behavior (runtime monitors, positive/negative oracles)
- **Run** the simulation

Runtime Monitoring

- **Task**. From a different process or from the “main” process collect data from sensors. Sensors must be attached to vehicles. Use damage sensor to observe collisions, use Timer sensors to trigger timeouts, use Position of the car to check for target/goal area and to check for out-of-lane episodes.

Offline Oracles

- **Task** Show how data can be collected/stored into a datastructure (e.g., array of states) that can be processed later to assert additional behaviors (e.g., max speed violation, passenger comfort, count how many times it press the brake pedal, etc..)

Simulation-based Testing

Anatomy of a (Simulation-based) Test

- **Assume** a running simulator
- **Setup** the environment (level/terrain, map, materials, roads)
- **Setup** the test initial conditions (placement of ego-car, NPC vehicles, obstacles)
- **Setup** the test subject (connect to ego-car, instruct about test goal)
- **Monitor** and **assert** behavior (runtime monitors, positive/negative oracles)
- **Run** the simulation

Running the simulation

- **Task** Run the simulation for some steps; wait for command; wait for the oracle triggering; resume the simulation. Repeat

Part I

Simulation Based Testing

Achieving Automation

- We show some examples of **automated test generation**
- **Parameter exploration**
  - Speed, Acceleration, Take the case of the car driving in front and change:
    - The Car model or color
    - The initial position (distance from the ego-car)
    - The speed or brake intensity or time when braking starts
- **(Random) Procedural Test Generation**.
  - Road Generation (SBST)
  - Placing obstacles (change size, placement), parked cars
  - Adding (Random) Traffic (if possible)
  - ?

Moving Beyond

- Extension of the Simulator: Fault Injection
- Collect data for training
- Record/Replay (replicating driving behaviors)
- A “drone” example? (this will be **extremely** well perceived!)