

CARLA Simulator

Open-source simulator for autonomous driving research

CARLA Team

January 7, 2021

Contents

0.1	CARLA Documentation	3
1	Getting started	4
1.1	CARLA	4
1.2	Quick start	6
2	Building CARLA	8
2.1	Linux build	8
2.2	Windows build	14
2.3	Requirements	15
2.4	Necessary software	15
2.5	CARLA build	16
2.6	Update CARLA	17
2.7	Build system	19
2.8	Running CARLA in a Docker	21
2.9	F.A.Q.	22
2.10	Running CARLA	26
2.11	Other	26
3	First steps	27
3.1	Core concepts	27
3.2	1st. World and client	28
3.3	The world	30
3.4	2nd. Actors and blueprints	32
3.5	3rd. Maps and navigation	37
3.6	4th. Sensors and data	41
4	Advanced steps	44
4.1	OpenDRIVE standalone mode	44
4.2	PTV-Vissim co-simulation	46
4.3	Recorder	48
4.4	Rendering options	55
4.5	RSS	58
4.6	SUMO co-simulation	60
4.7	Synchrony and time-step	62
4.8	Traffic Manager	65
5	References	73
5.1	Code recipes	120
5.2	C++ Reference	141
5.3	Recorder Binary File Format	142
5.4	Sensors reference	154
6	Plugins	185
6.1	carlaviz	185

6.2	Run carlaviz	186
6.3	Utilities	187
7	ROS bridge	188
7.1	ROS bridge installation	188
7.2	CARLA messages reference	192
7.3	Launchfiles reference	203
8	Tutorials (general)	209
8.1	How to add friction triggers	209
8.2	How to control vehicle physics	210
8.3	Walker Bone Control	211
8.4	Generate maps with OpenStreetMap	213
8.5	Retrieve simulation data	216
9	Tutorials (assets)	243
9.1	Add a new map	243
9.2	Add a new vehicle	262
9.3	Add new props	268
9.4	Create distribution packages for assets	271
9.5	Map customization tools	271
9.6	Material customization	277
9.7	How to model vehicles	290
10	Tutorials (developers)	291
10.1	How to upgrade content	291
10.2	How to add a new sensor	291
10.3	Customize vehicle suspension	299
10.4	Generate detailed colliders	301
10.5	How to make a release	306
10.6	How to generate the pedestrian navigation info	306
11	Contributing	309
11.1	Contributing to CARLA	309
11.2	Contributor Covenant Code of Conduct	311
11.3	Coding standard	312
11.4	Documentation Standard	312

0.1 CARLA Documentation

Welcome to the CARLA documentation.

This home page contains an index with a brief description of the different sections in the documentation. Feel free to read in whatever order preferred. In any case, here are a few suggestions for newcomers.

- **Install CARLA.** Either follow the Quick start installation to get a CARLA release or make the build for a desired platform.
- **Start using CARLA.** The section titled First steps is an introduction to the most important concepts.
- **Check the API.** there is a handy Python API reference to look up the classes and methods available.

The CARLA forum is available to post any doubts or suggestions that may arise during the reading.



This documentation refers to CARLA 0.9.0 or later. To read about previous versions, check the [stable branch](<https://carla.readthedocs.io/en/stable/>).

Getting started

Introduction — What to expect from CARLA. **Quick start** — Get the CARLA releases.

Building CARLA

Linux build — Make the build on Linux. **Windows build** — Make the build on Windows. **Update CARLA** — Get up to date with the latest content. **Build system** — Learn about the build and how it is made. **Running in a Docker** — Run CARLA using a container solution. **F.A.Q.** — Some of the most frequent installation issues.

First steps

Core concepts — Overview of the basic concepts in CARLA. **1st. World and client** — Manage and access the simulation. **2nd. Actors and blueprints** — Learn about actors and how to handle them. **3rd. Maps and navigation** — Discover the different maps and how do vehicles move around. **4th. Sensors and data** — Retrieve simulation data using sensors.

Advanced steps

OpenDRIVE standalone mode — Use any OpenDRIVE file as a CARLA map. **PTV-Vissim co-simulation** — Run a synchronous simulation between CARLA and PTV-Vissim. **Recorder** — Register the events in a simulation and play it again. **Rendering options** — From quality settings to no-render or off-screen modes. **RSS** — An implementation of RSS in the CARLA client library. **SUMO co-simulation** — Run a synchronous simulation between CARLA and SUMO. **Synchrony and time-step** — Client-server communication and simulation time. **Traffic Manager** — Simulate urban traffic by setting vehicles to autopilot mode.

References

Python API reference — Classes and methods in the Python API. **Code recipes** — Some code fragments commonly used. **Blueprint library** — Blueprints provided to spawn actors. **C++ reference** — Classes and methods in CARLA C++. **Recorder binary file format** — Detailed explanation of the recorder file format. **Sensors reference** — Everything about sensors and the data they retrieve.

Plugins

carlaviz — **web visualizer** — Plugin that listens the simulation and shows the scene and some simulation data in a web browser.

ROS bridge

ROS bridge installation — The different ways to install the ROS bridge. **CARLA messages reference** — Contains explanations and fields for every type of CARLA message available in ROS. **Launchfiles reference** — Lists the launchfiles and nodes provided, and the topics being consumed and published.

Tutorials — General

Add friction triggers — Define dynamic box triggers for wheels. **Control vehicle physics** — Set runtime changes on a vehicle physics. **Control walker skeletons** — Animate walkers using skeletons. **Retrieve simulation data** — A step by step guide to properly gather data using the recorder.

Tutorials — Assets

Add a new map — Create and ingest a new map. **Add a new vehicle** — Prepare a vehicle to be used in CARLA. **Add new props** — Import additional props into CARLA. **Create standalone packages** — Generate and handle standalone packages for assets. **Map customization** — Edit an existing map. **Material customization** — Edit vehicle and building materials. **Vehicle modelling** — Create a new vehicle for CARLA.

Tutorials — Developers

Contribute with new assets — Add new content to CARLA. **Create a sensor** — Develop a new sensor to be used in CARLA. **Customize vehicle suspension** — Modify the suspension system of a vehicle. **Make a release** — For developers who want to publish a release. **Generate detailed colliders** — Create detailed colliders for vehicles. **Generate pedestrian navigation** — Obtain the information needed for walkers to move around.

Contributing

Contribution guidelines — The different ways to contribute to CARLA. **Code of conduct** — Standard rights and duties for contributors. **Coding standard** — Guidelines to write proper code. **Documentation standard** — Guidelines to write proper documentation.

1 Getting started

1.1 CARLA



Figure 1: Welcome to CARLA



This documentation refers to the latest development versions of CARLA, 0.9.0 or later. There is another documentation for the stable version 0.8 [here](https://carla.readthedocs.io/en/stable/getting_started/), though it should only be used for specific queries.

CARLA is an open-source autonomous driving simulator. It was built from scratch to serve as a modular and flexible API to address a range of tasks involved in the problem of autonomous driving. One of the main goals of CARLA is to help democratize autonomous driving R&D, serving as a tool that can be easily accessed and customized by users. To do so, the simulator has to meet the requirements of different use cases within the general problem of driving (e.g. learning driving policies, training perception algorithms, etc.). CARLA is grounded on Unreal Engine to run the simulation and uses the OpenDRIVE standard (1.4 as today) to define roads and urban settings. Control over the simulation is granted through an API handled in Python and C++ that is constantly growing as the project does.

In order to smooth the process of developing, training and validating driving systems, CARLA evolved to become an ecosystem of projects, built around the main platform by the community. In this context, it is important to understand

some things about how does CARLA work, so as to fully comprehend its capabilities.

The simulator

The CARLA simulator consists of a scalable client-server architecture. The server is responsible for everything related with the simulation itself: sensor rendering, computation of physics, updates on the world-state and its actors and much more. As it aims for realistic results, the best fit would be running the server with a dedicated GPU, especially when dealing with machine learning. The client side consists of a sum of client modules controlling the logic of actors on scene and setting world conditions. This is achieved by leveraging the CARLA API (in Python or C++), a layer that mediates between server and client that is constantly evolving to provide new functionalities.

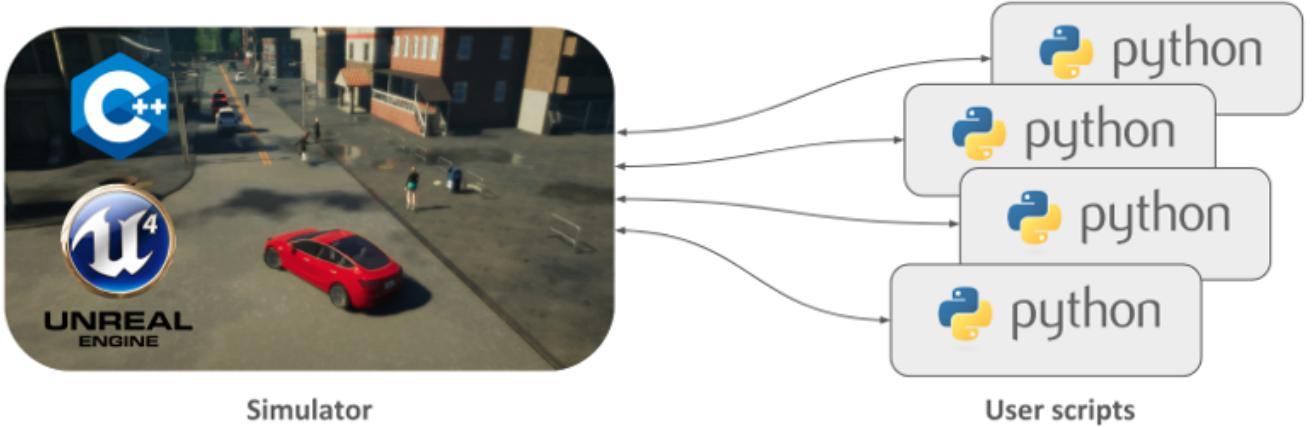


Figure 2: CARLA Modules

That summarizes the basic structure of the simulator. Understanding CARLA though is much more than that, as many different features and elements coexist within it. Some of these are listed hereunder, as to gain perspective on the capabilities of what CARLA can achieve.

- **Traffic manager.** A built-in system that takes control of the vehicles besides the one used for learning. It acts as a conductor provided by CARLA to recreate urban-like environments with realistic behaviours.
- **Sensors.** Vehicles rely on them to dispense information of their surroundings. In CARLA they are a specific kind of actor attached to the vehicle and the data they receive can be retrieved and stored to ease the process. Currently the project supports different types of these, from cameras to radars, lidar and many more.
- **Recorder.** This feature is used to reenact a simulation step by step for every actor in the world. It grants access to any moment in the timeline anywhere in the world, making for a great tracing tool.
- **ROS bridge and Autoware implementation.** As a matter of universalization, the CARLA project ties knots and works for the integration of the simulator within other learning environments.
- **Open assets.** CARLA facilitates different maps for urban settings with control over weather conditions and a blueprint library with a wide set of actors to be used. However, these elements can be customized and new can be generated following simple guidelines.
- **Scenario runner.** In order to ease the learning process for vehicles, CARLA provides a series of routes describing different situations to iterate on. These also set the basis for the CARLA challenge, open for everybody to test their solutions and make it to the leaderboard.

The project

CARLA grows fast and steady, widening the range of solutions provided and opening the way for the different approaches to autonomous driving. It does so while never forgetting its open-source nature. The project is transparent, acting as a white box where anybody is granted access to the tools and the development community. In that democratization is where CARLA finds its value. Talking about how CARLA grows means talking about a community of developers who dive together into the thorough question of autonomous driving. Everybody is free to explore with CARLA, find their own solutions and then share their achievements with the rest of the community.

This documentation will be a companion along the way. The next page contains Quick start instructions for those eager to install a CARLA release. There is also a build guide for Linux and Windows. This will make CARLA from repository and allow to dive full-length into its features.

Welcome to CARLA.

1.2 Quick start

- [Installation summary](#)
- [Requirements](#)
- [CARLA installation](#) A. *Debian CARLA installation* B. Package installation
- [Import additional assets](#)
- [Running CARLA](#) *Command-line options
- [Updating CARLA](#)
- [Follow-up](#)

Installation summary

Show command line summary for the quick start installation

```
1 # Install required modules Pygame and Numpy
2 pip install --user pygame numpy
3
4 # There are two different ways to install CARLA
5
6 # Option A) Debian package installation
7 # This repository contains CARLA 0.9.10 and later. To install previous CARLA versions, change to a
     previous version of the docs using the pannel in the bottom right part of the window
8 sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 1AF1527DE64CB8D9
9 sudo add-apt-repository "deb [arch=amd64] http://dist.carla.org/carla $(lsb_release -sc) main"
10 sudo apt-get update
11 sudo apt-get install carla-simulator # Install the latest CARLA version or update the current
     installation
12 sudo apt-get install carla-simulator=0.9.10-1 # install a specific CARLA version
13 cd /opt/carla-simulator
14 ./CarlaUE4.sh
15
16 # Option B) Package installation
17 # Go to: https://github.com/carla-simulator/carla/blob/master/Docs/download.md
18 # Download the desired package and additional assets
19 # Extract the package
20 # Extract the additional assets in `Import`
21 # Run CARLA (Linux).
22 ./CarlaUE.sh
23 # Run CARLA (Windows)
24 > CarlaUE4.exe
25
26 # Run a script to test CARLA.
27 cd PythonAPI/examples
28 python3 spawn_npc.py # Support for Python2 was provided until 0.9.10 (not included)
29
30 # The PythonAPI can be compiled for Python2 when using a Linux build from source
```

Requirements

The quick start installation uses a pre-packaged version of CARLA. The content is comprised in a bundle that can run automatically with no build installation needed. The API can be accessed fully but advanced customization and development options are unavailable. The requirements are simpler than those for the build installation.

- **Server side.** A 4GB minimum GPU will be needed to run a highly realistic environment. A dedicated GPU is highly advised for machine learning.
- **Client side.** Python is necessary to access the API via command line. Also, a good internet connection and two TCP ports (2000 and 2001 by default).
- **System requirements.** Any 64-bits OS should run CARLA. However, since release 0.9.9, **CARLA cannot run in 16.04 Linux systems with default compilers**. These should be upgraded to work with CARLA.
- **Other requirements.** Two Python modules: Pygame to create graphics directly with Python, and Numpy for great calculus.

To install both modules using pip, run the following commands.

```
1 pip install --user pygame numpy
```

CARLA installation

The **Debian installation** is the easiest way to get the latest release in Linux. **Download the GitHub repository** to get either a specific release or the Windows version of CARLA.

A. Debian CARLA installation

Set up the Debian repository in the system.

```
1 sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 1AF1527DE64CB8D9
2 sudo add-apt-repository "deb [arch=amd64] http://dist.carla.org/carla $(lsb_release -sc) main"
```

Install CARLA and check for the installation in the `/opt/` folder.

```
1 sudo apt-get update # Update the Debian package index
2 sudo apt-get install carla-simulator # Install the latest CARLA version, or update the current
                                         installation
3 cd /opt/carla-simulator # Open the folder where CARLA is installed
```

This repository contains CARLA 0.9.10 and later versions. To install a specific version add the version tag to the installation command.

```
1 sudo apt-get install carla-simulator=0.9.10-1 # In this case, "0.9.10" refers to a CARLA version, and
                                              "1" to the Debian revision
```



To install CARLA versions prior to 0.9.10, change to a previous version of the documentation using the pannel in the bottom right corner of the window, and follow the old instructions.

B. Package installation

The repository contains different versions of the simulator available. *Development* and *stable* sections list the packages for the different official releases. The later the version the more experimental it is. The *nightly build* is the current development version as today and so, the most unstable.

There may be many files per release. The package is a compressed file named as **CARLA_version.number**.

Download and extract the release file. It contains a precompiled version of the simulator, the Python API module and some scripts to be used as examples.

Import additional assets

For every release there are other packages containing additional assets and maps, such as **Additional_Maps_0.9.9.2** for CARLA 0.9.9.2, which contains **Town06**, **Town07**, and **Town10**. These are stored separately to reduce the size of the build, so they can only be run after these packages are imported.

Download and move the package to the *Import* folder, and run the following script to extract them.

```
1 > cd ~/carla
2 > ./ImportAssets.sh
```



On Windows, directly extract the package on the root folder.

Running CARLA

Open a terminal in the main CARLA folder. Run the following command to execute the package file and start the simulation:

```
1 # Linux:
2 > ./CarlaUE4.sh
3 # Windows:
4 > CarlaUE4.exe
```



In the deb installation , ‘CarlaUE4.sh‘ will be in ‘/opt/carla-simulator/bin/‘, instead of the main ‘carla/‘ folder where it normally is.

A window containing a view over the city will pop up. This is the *spectator view*. To fly around the city use the mouse and WASD keys (while clicking). The server simulator is now running and waiting for a client to connect and interact with the world. Now it is time to start running scripts. The following example will spawn some life into the city:

```
1 # Go to the folder containing example scripts
2 > cd PythonAPI/examples
3
4 > python3 spawn_npc.py # Support for Python2 was provided until 0.9.10 (not included)
```



The PythonAPI can be compiled for Python2 when using aLinux build from source.

Command-line options There are some configuration options available when launching CARLA.

- `-carla-rpc-port=N` Listen for client connections at port N. Streaming port is set to N+1 by default.
- `-carla-streaming-port=N` Specify the port for sensor data streaming. Use 0 to get a random unused port. The second port will be automatically set to N+1.
- `-quality-level={Low,Epic}` Change graphics quality level. Find out more in rendering options.
- **Full list of UE4 command-line arguments.** There is a lot of options provided by UE. However, not all of these will be available in CARLA.

```
1 > ./CarlaUE4.sh -carla-rpc-port=3000
```

The script PythonAPI/util/config.py provides for more configuration options.

```
1 > ./config.py --no-rendering      # Disable rendering
2 > ./config.py --map Town05        # Change map
3 > ./config.py --weather ClearNoon # Change weather
4
5 > ./config.py --help # Check all the available configuration options
```

Updating CARLA

The packaged version requires no updates. The content is bundled and thus, tied to a specific version of CARLA. Everytime there is a release, the repository will be updated. To run this latest or any other version, delete the previous and install the one desired.

Follow-up

Thus concludes the quick start installation process. In case any unexpected error or issue occurs, the CARLA forum is open to everybody. There is an *Installation issues* category to post this kind of problems and doubts.

So far, CARLA should be operative in the desired system. Terminals will be used to contact the server via script, interact with the simulation and retrieve data. To do so, it is essential to understand the core concepts in CARLA. Read the **First steps** section to learn on those. Additionally, all the information about the Python API regarding classes and its methods can be accessed in the Python API reference.

2 Building CARLA

2.1 Linux build

- **Linux build command summary**
- **Requirements**
 - System specifics
 - Dependencies
- **GitHub**
- **Unreal Engine**
- **CARLA build**

- Clone repository
- Get assets
- Set the environment variable
- make CARLA

The build process can be quite long and tedious. The *F.A.Q.* page offers solution for the most common complications. Alternatively, use the CARLA forum to post any unexpected issues that may occur.

Linux build command summary

Show command lines to build on Linux

```

1 # Make sure to meet the minimum requirements
2 # Read the complete documentation to understand each step
3
4 # Install dependencies
5 sudo apt-get update &&
6 sudo apt-get install wget software-properties-common &&
7 sudo add-apt-repository ppa:ubuntu-toolchain-r/test &&
8 wget -O - https://apt.llvm.org/llvm-snapshot.gpg.key|sudo apt-key add - &&
9 sudo apt-add-repository "deb http://apt.llvm.org/$(lsb_release -c --short)/
    llvm-toolchain-$(lsb_release -c --short)-8 main" &&
10 sudo apt-get update
11
12 # Additional dependencies for Ubuntu 18.04
13 sudo apt-get install build-essential clang-8 lld-8 g++-7 cmake ninja-build libvulkan1 python
    python-pip python-dev python3-dev python3-pip libpng-dev libtiff5-dev libjpeg-dev tzdata sed curl
    unzip autoconf libtool rsync libxml2-dev &&
14 pip2 install --user setuptools &&
15 pip3 install --user -Iv setuptools==47.3.1
16
17 # Additional dependencies for previous Ubuntu versions
18 sudo apt-get install build-essential clang-8 lld-8 g++-7 cmake ninja-build libvulkan1 python
    python-pip python-dev python3-dev python3-pip libpng16-dev libtiff5-dev libjpeg-dev tzdata sed
    curl unzip autoconf libtool rsync libxml2-dev &&
19 pip2 install --user setuptools &&
20 pip3 install --user -Iv setuptools==47.3.1 &&
21 pip2 install --user distro &&
22 pip3 install --user distro
23
24 # Change default clang version
25 sudo update-alternatives --install /usr/bin/clang++ clang++ /usr/lib/llvm-8/bin/clang++ 180 &&
26 sudo update-alternatives --install /usr/bin/clang clang /usr/lib/llvm-8/bin/clang 180
27
28 # Get a GitHub and a UE account, and link both
29 # Install git
30
31 # Download Unreal Engine 4.24
32 git clone --depth=1 -b 4.24 https://github.com/EpicGames/UnrealEngine.git ~/UnrealEngine_4.24
33 cd ~/UnrealEngine_4.24
34
35 # Download and install the UE patch
36 wget https://carla-releases.s3.eu-west-3.amazonaws.com/Linux/UE_Patch/430667-13636743-patch.txt
    430667-13636743-patch.txt
37 patch --strip=4 < 430667-13636743-patch.txt
38
39 # Build UE
40 ./Setup.sh && ./GenerateProjectFiles.sh && make
41
42 # Open the UE Editor to check everything works properly
43 cd ~/UnrealEngine_4.24/Engine/Binaries/Linux && ./UE4Editor
44

```

```

45 # Clone the CARLA repository
46 git clone https://github.com/carla-simulator/carla
47
48 # Get the CARLA assets
49 cd ~/carla
50 ./Update.sh
51
52 # Set the environment variable
53 export UE4_ROOT=~/UnrealEngine_4.24
54
55 # make the CARLA client and the CARLA server
56 make PythonAPI
57 make launch
58
59 # Press play in the Editor to initialize the server
60 # Run example scripts to test CARLA
61 # Terminal A
62 cd PythonAPI/examples
63 python3 spawn_npc.py
64 # Terminal B
65 cd PythonAPI/examples
66 python3 spawn_npc.py # Support for Python2 was provided until 0.9.10 (not included)
67 python3 dynamic_weather.py # Support for Python2 was provided until 0.9.10 (not included)
68
69 # Optionally, to compile the PythonAPI for Python2, run the following command in the root CARLA
       directory
70 make PythonAPI ARGS="--python-version=2"

```

Requirements

System specifics

- **Ubuntu 18.04.** CARLA provides support for previous Ubuntu versions up to 16.04. However proper compilers are needed for UE to work properly. Dependencies for Ubuntu 18.04 and previous versions are listed separately below. Make sure to install the ones corresponding to your system.
- **30GB disk space.** The complete build will require quite a lot of space, especially Unreal Engine. Make sure to have around 30/50GB of free disk space.
- **An adequate GPU.** CARLA aims for realistic simulations, so the server needs at least a 4GB GPU. A dedicated GPU is highly recommended for machine learning.
- **Two TCP ports and good internet connection.** 2000 and 2001 by default. Be sure neither the firewall nor any other application block these.

Dependencies CARLA needs many dependencies to run. Some of them are built automatically during this process, such as *Boost.Python*. Others are binaries that should be installed before starting the build (*cmake*, *clang*, different versions of *Python* and much more). In order to do so, run the commands below in a terminal window.

```

1 sudo apt-get update &&
2 sudo apt-get install wget software-properties-common &&
3 sudo add-apt-repository ppa:ubuntu-toolchain-r/test &&
4 wget -O - https://apt.llvm.org/llvm-snapshot.gpg.key|sudo apt-key add - &&
5 sudo apt-add-repository "deb http://apt.llvm.org/xenial/ llvm-toolchain-xenial-8 main" &&
6 sudo apt-get update

```



The following commands depend on your Ubuntu version. Make sure to choose accordingly.

Ubuntu 18.04.

```

1 sudo apt-get install build-essential clang-8 lld-8 g++-7 cmake ninja-build libvulkan1 python
      python-pip python-dev python3-dev python3-pip libpng-dev libtiff5-dev libjpeg-dev tzdata sed curl

```

```
1 unzip autoconf libtool rsync libxml2-dev &&
2 pip2 install --user setuptools &&
3 pip3 install --user -Iv setuptools==47.3.1 &&
4 pip2 install --user distro &&
5 pip3 install --user distro
```

Previous Ubuntu versions.

```
1 sudo apt-get install build-essential clang-8 lld-8 g++-7 cmake ninja-build libvulkan1 python
    python-pip python-dev python3-dev python3-pip libpng16-dev libtiff5-dev libjpeg-dev tzdata sed
    curl unzip autoconf libtool rsync libxml2-dev &&
2 pip2 install --user setuptools &&
3 pip3 install --user -Iv setuptools==47.3.1 &&
4 pip2 install --user distro &&
5 pip3 install --user distro
```

All Ubuntu systems. To avoid compatibility issues between Unreal Engine and the CARLA dependencies, use the same compiler version and C++ runtime library to compile everything. The CARLA team uses clang-8 and LLVM's libc++. Change the default clang version to compile Unreal Engine and the CARLA dependencies.

```
1 sudo update-alternatives --install /usr/bin/clang++ clang++ /usr/lib/llvm-8/bin/clang++ 180 &&
2 sudo update-alternatives --install /usr/bin/clang clang /usr/lib/llvm-8/bin/clang 180
```

GitHub

1. Create a GitHub account. CARLA is organized in different GitHub repositories, so an account will be needed to clone said repositories.
2. Install git to manage the repositories via terminal.
3. Create an Unreal Engine account to access the Unreal Engine repositories, which are set to private.
4. Connect both your GitHub and Unreal Engine accounts. Go to your personal settings in there is a section in Unreal Engine's website. Click on **Connections > Accounts**, and link both accounts. Here is a brief explanation just in case.



New Unreal Engine accounts need activation. After creating the account, a verification mail will be sent. Check it out, or the UE repository will be shown as non-existent in the following steps.

Unreal Engine

The current version of CARLA runs on **Unreal Engine 4.24** only. In this guide, the installation will be done in `~/UnrealEngine_4.24`, but the path can be changed. If your path is different, change the following commands accordingly.



Alternatively to this section, there is another [guide](<https://docs.unrealengine.com/en-US/Platforms/Linux/BeginnerLinuxDeveloper/SettingUpAnUnrealWorkflow/index.html>) to build UE on Linux. When consulting it, remember that CARLA will need the 4.24 release , not the latest.

1. Clone the content for Unreal Engine 4.24 in your local computer.

```
1 git clone --depth=1 -b 4.24 https://github.com/EpicGames/UnrealEngine.git ~/UnrealEngine_4.24
```

2. Get into the directory where UE4.24 has been cloned.

```
1 cd ~/UnrealEngine_4.24
```

3. Download a patch for Unreal Engine. This patch fixes some Vulkan visualization issues that may occur when changing the map. Download and install it with the following commands.

```
1 wget https://carla-releases.s3.eu-west-3.amazonaws.com/Linux/UE\_Patch/430667-13636743-patch.txt
      430667-13636743-patch.txt
2 patch --strip=4 < 430667-13636743-patch.txt
```



If UE has already been built, install the patch, and make the build again.

4. Make the build. This may take an hour or two depending on your system.

```
1 ./Setup.sh && ./GenerateProjectFiles.sh && make
```

5. Open the Editor to check that UE has been properly installed.

```
1 cd ~/UnrealEngine_4.24/Engine/Binaries/Linux && ./UE4Editor
```

Any issues this far are related with Unreal Engine. There is not much CARLA can do about it. However, the build documentation provided by Unreal Engine may be helpful.

CARLA build

The system should be ready to start building CARLA. Just for clarity, a brief summary so far.

- Minimum technical requirements to run CARLA are suitable.
- Dependencies have been properly installed.
- GitHub account is ready.
- Unreal Engine 4.24 runs smooth.



Downloading aria2 with ‘sudo apt-get install aria2‘ will speed up the following commands.

Clone repository The official repository of the project. Either download and extract it, or clone the repository with the following command line.

```
1 git clone https://github.com/carla-simulator/carla
```

Now the latest state of the simulator, known as `master` branch in the repository, has been copied in local. Here is brief introduction to the most relevant branches of the repository. Remember that you can change and check your branches with the command `git branch`.

- **master branch** — Latest fixes and features that have been tested. These will be featured in the next CARLA release.
- **dev branch** — Latest fixes and features still in development and testing. This branch will be merged with `master` when the time for a new release comes.
- **stable branch** — Latest version of CARLA tagged as stable. Previous CARLA versions also have their own branch.

Get assets Download the assets, as they are necessary to run CARLA. These are stored in a separated package to reduce the size of the build. A script downloads and extracts the latest stable assets automatically. The package is >3GB, so the download may take some time.

1. Get into your root carla directory. The path should correspond with the repository just cloned. `sh cd ~/carla`
2. Run the script to get the assets. `sh ./Update.sh`



To download the assets currently in development, visit [Update CARLA](#) and read [Get development assets](#).

Set the environment variable This is necessary for CARLA to find the Unreal Engine 4.24 installation folder.

```
1 export UE4_ROOT=~/UnrealEngine_4.24
```

The variable should be added to `~/.bashrc` or `~/.profile` to be set persistently session-wide. Otherwise, it will only be accessible from the current shell. To do this, follow these steps.

1. Open `~/.bashrc`.

```
1 gedit ~/.bashrc
```

2. Write the environment variable in the `~/.bashrc` file: `export UE4_ROOT=~/UnrealEngine_4.24`
3. Save the file and reset the terminal.

make CARLA The last step is to finally build CARLA. There are different `make` commands to build the different modules. All of them run in the root CARLA folder.



Make sure to run ‘make PythonAPI’ to prepare the client and ‘make launch’ for the server. Alternatively ‘make LibCarla’ will prepare the CARLA library to be imported anywhere.

- **make PythonAPI** compiles the API client, necessary to grant control over the simulation. It is only needed the first time. Remember to run it again when updating CARLA. Scripts will be able to run after this command is executed.

```
1 make PythonAPI
```

- **make launch** compiles the server simulator and launches Unreal Engine. Press **Play** to start the spectator view and close the editor window to exit. Camera can be moved with WASD keys and rotated by clicking the scene while moving the mouse around.

```
1 make launch
```

The project may ask to build other instances such as `UE4Editor-Carla.dll` the first time. Agree in order to open the project. During the first launch, the editor may show warnings regarding shaders and mesh distance fields. These take some time to be loaded and the city will not show properly until then.

Finally, let’s test the simulator. Inside `PythonAPI/examples` and `PythonAPI/util` there are some example scripts that may be especially useful for starters. The following commands will spawn some life into the town, and create a weather cycle. Each script should be run in one terminal

```
1 # Support for Python2 was provided until 0.9.10 (not included)
2 # Terminal A
3 cd PythonAPI/examples
4 python3 spawn_npc.py
5 # Terminal B
6 cd PythonAPI/examples
7 python3 dynamic_weather.py
```



If the simulation is running at very low FPS rates, go to ‘Edit/Editor preferences/Performance’ in the UE editor and disable Use less CPU when in background .

Optionally, to compile the PythonAPI for Python2, run the following command in the root CARLA directory.

```
1 make PythonAPI ARGS="--python-version=2"
```

Now CARLA is ready to go. Here is a brief summary of the most useful `make` commands available.

Command

Description

`make help`

Prints all available commands.

`make launch`

Launches CARLA server in Editor window.

`make PythonAPI`

Builds the CARLA client.

`make package`

Builds CARLA and creates a packaged version for distribution.

`make clean`

Deletes all the binaries and temporals generated by the build system.

`make rebuild`

make clean and make launch both in one command.

Read the *F.A.Q.* page or post in the CARLA forum for any issues regarding this guide.

Some recommendations after finishing the build. Learn how to update the CARLA build or take your first steps in the simulation, and learn some core concepts.

2.2 Windows build

- **Windows build command summary**
- **Requirements**
 - System specifics
- **Necessary software**
 - Minor installations (CMake, git, make, Python3 x64)
 - Visual Studio 2017
 - Unreal Engine (4.24)
- **CARLA build**
 - Clone repository
 - Get assets
 - Set the environment variable
 - make CARLA

The build process can be quite long and tedious. The *F.A.Q.* page offers solution for the most common complications. Alternatively, use the CARLA forum to post any unexpected issues that may occur.

Windows build command summary

Show command lines to build on Windows To execute the make commands below, you must use the Visual Studio 2017 native console x64 with administrator rights, otherwise you may be getting permission errors.



To execute the ““make““ commands below, you must use the Visual Studio 2017 native console x64 with administrator rights, otherwise you may be getting permission errors.

```
1 ## Make sure to meet the minimum requirements
2
3 ## Necessary software:
4 ##   CMake
5 ##   Git
6 ##   Make
7 ##   Python3 x64
8 ##   Unreal Engine 4.24
9 ##   Visual Studio 2017 with Windows 8.1 SDK and x64 Visual C++ Toolset
10
11 ## Set environment variables for the software
12
13 ## Clone the CARLA repository
14 git clone https://github.com/carla-simulator/carla
15
16 ## make the CARLA client and the CARLA server
17 make PythonAPI
18 make launch
19
20 ## Press play in the Editor to initialize the server
21 ## Run example scripts to test CARLA
22 ## Terminal A
23 cd PythonAPI/examples
24 python3 spawn_npc.py
```

```

25 ## Terminal B
26 cd PythonAPI/examples
27 python3 dynamic_weather.py
28 ## The PythonAPI will be built based on the installed Python version
29 ## The docs will use Python3, as support for Python2 was provided until 0.9.10 (not included)

```

2.3 Requirements

System specifics

- **x64 system.** The simulator should run in any 64 bits Windows system.
- **30GB disk space.** Installing all the software needed and CARLA will require quite a lot of space. Make sure to have around 30/50GB of free disk space.
- **An adequate GPU.** CARLA aims for realistic simulations, so the server needs at least a 4GB GPU. A dedicated GPU is highly recommended for machine learning.
- **Two TCP ports and good internet connection.** 2000 and 2001 by default. Be sure neither the firewall nor any other application are blocking these.

2.4 Necessary software

Minor installations

- **CMake** generates standard build files from simple configuration files.
- **Git** is a version control system to manage CARLA repositories.
- **Make** generates the executables.
- **Python3 x64** is the main script language in CARLA. Having a x32 version installed may cause conflict, so it is highly advisable to have it uninstalled.



Be sure that these programs are added to the [environment path](<https://www.java.com/en/download/help/path.xml>). Remember that the path added leads to the bin directory.

Visual Studio 2017

Get the 2017 version from here. **Community** is the free version. Use the *Visual Studio Installer* to install two additional elements.

- **Windows 8.1 SDK.** Select it in the *Installation details* section on the right.
- **x64 Visual C++ Toolset.** In the *Workloads* section, choose **Desktop development with C++**. This will enable a x64 command prompt that will be used for the build. Check it up by pressing the **Win** button and searching for x64. Be careful to **not open a x86_x64 prompt**.



Other Visual Studio versions may cause conflict. Even if these have been uninstalled, some registers may persist. To completely clean Visual Studio from the computer, go to ‘Program Files (x86)

Microsoft
Visual Studio

Installer

resources

app

layout‘ and
run ‘

InstallCleanup.exe
-full‘

Unreal Engine

Go to Unreal Engine and download the *Epic Games Launcher*. In **Engine versions/Library**, download **Unreal Engine 4.24.x**. Make sure to run it in order to check that everything was properly installed.



Having VS2017 and UE4.24 installed, a Generate Visual Studio project files option should appear when doing right-click on .uproject files. If this is not available, something went wrong with the UE4.24 installation. Create a UE project to check it out and reinstall if necessary.

2.5 CARLA build



Lots of things have happened so far. It is highly advisable to restart the computer.

Clone repository

The official repository of the project. Either download and extract it or clone it using the following command line in a **x64 terminal**.

```
1 git clone https://github.com/carla-simulator/carla
```

Now the latest content for the project, known as `master` branch in the repository, has been copied in local.



The ‘master’ branch contains the latest fixes and features. Stable code is inside the ‘stable’ and previous CARLA versions have their own branch. Always remember to check the current branch in git with the command ‘git branch’.

Get assets

Only the assets package is yet to be downloaded. `\Util\ContentVersions.txt` contains the links to the assets for CARLA releases. These must be extracted in `Unreal\CarlaUE4\Content\Carla`. If the path doesn’t exist, create it.

Download the **latest** assets to work with the current version of CARLA.

Set the environment variable

This is necessary for CARLA to find the Unreal Engine installation folder. By doing so, users can choose which specific version of Unreal Engine is to be used. If no environment variable is specified, the CARLA will look up in the directories and use the last version in search order.

1. Open the Windows Control Panel and go to **Advanced System Settings**. **2. On the Advanced panel open Environment Variables....** **3. Click New...** to create the variable. **4. Name the variable as UE4_ROOT** and choose the path to the installation folder of the desire UE4 installation.

make CARLA

The last step is to finally build CARLA. There are different `make` commands to build the different modules. All of them run in the root CARLA folder.



Make sure to run ‘make PythonAPI’ to prepare the client and ‘make launch’ for the server. Alternatively ‘make LibCarla’ will prepare the CARLA library to be imported anywhere.

- **make PythonAPI** compiles the API client, necessary to grant control over the simulation. It is only needed the first time. Remember to run it again when updating CARLA. Scripts will be able to run after this command is executed.

```
1 make PythonAPI
```

- **make launch** compiles the server simulator and launches Unreal Engine. Press **Play** to start the spectator view and close the editor window to exit. Camera can be moved with WASD keys and rotated by clicking the scene while moving the mouse around.

```
1 make launch
```

The project may ask to build other instances such as `UE4Editor-Carla.dll` the first time. Agree in order to open the project. During the first launch, the editor may show warnings regarding shaders and mesh distance fields. These take some time to be loaded and the city will not show properly until then.

Finally, let's test the simulator. Inside `PythonAPI/examples` and `PythonAPI/util` there are some example scripts that may be especially useful for starters. The following commands will spawn some life into the town, and create a weather cycle. Each script should be run in one terminal

```
1 ## Terminal A
2 cd PythonAPI/examples
3 python3 spawn_npc.py
4 ## Terminal B
5 cd PythonAPI/examples
6 python3 dynamic_weather.py
7 ## The PythonAPI will be built based on the installed Python version
8 ## The docs will use Python3, as support for Python2 was provided until 0.9.10 (not included)
```



If the simulation is running at very low FPS rates, go to 'Edit/Editor preferences/Performance' in the UE editor and disable Use less CPU when in background .

Now CARLA is ready to go. Here is a brief summary of the most useful `make` commands available.

Command

Description

`make help`

Prints all available commands.

`make launch`

Launches CARLA server in Editor window.

`make PythonAPI`

Builds the CARLA client.

`make package`

Builds CARLA and creates a packaged version for distribution.

`make clean`

Deletes all the binaries and temporals generated by the build system.

`make rebuild`

`make clean` and `make launch` both in one command.

Read the `*F.A.Q.*` page or post in the CARLA forum for any issues regarding this guide.

Some recommendations after finishing the build. Learn how to update the CARLA build or take your first steps in the simulation, and learn some core concepts.

2.6 Update CARLA

- **Update commands summary**
- **Get the lastest binary release**
- **Update Linux and Windows build**
 - Clean the build
 - Pull from origin

- Download the assets
- Launch the server
- **Get development assets**

To post unexpected issues, doubts or suggestions, feel free to login in the CARLA forum.

Update commands summary

Show command lines to update CARLA

```

1 # Update a CARLA packaged release.
2 #   1. Delete the current one.
3 #   2. Follow the Quick start installation to get the one desired.
4
5
6 # Update Linux build.
7 git checkout master
8 make clean
9 git pull origin master
10 ./Update.sh
11
12
13 # Update Windows build.
14 git checkout master
15 make clean
16 git pull origin master
17 # Erase the content in `Unreal\CarlaUE4\Content\Carla`.
18 # Go to `Util\ContentVersions.txt`.
19 # Download the latest content.
20 # Extract the new content in `Unreal\CarlaUE4\Content\Carla`.
21
22
23 # Get development assets.
24 # Delete the `/Carla` folder containing previous assets.
25 # Go to the main carla folder.
26 git clone https://bitbucket.org/carla-simulator/carla-content Unreal/CarlaUE4/Content/Carla

```

Get latest binary release

Binary releases are prepackaged and thus, tied to a specific version of CARLA. To get the latest, erase the previous and follow the quick start installation to get the one desired.

Releases are listed in **Development** in the CARLA repository. There is also a highly experimental **Nightly build** containing the current state of CARLA up to date.

Update Linux and Windows build

Make sure to be in the local `master` branch before the update. Then, merge or rebase the changes to other branches and solve possible conflicts.

```
1 git checkout master
```

Clean the build Go to the main CARLA folder and delete binaries and temporals generated by the previous build.

```
1 make clean
```

Pull from origin Get the current version from `master` in the CARLA repository.

```
1 git pull origin master
```

Download the assets Linux.

```
1 ./Update.sh
```

Windows.

1. Erase the previous content in `Unreal\CarlaUE4\Content\Carla`.
2. Go to `\Util\ContentVersions.txt`.
3. Download the content for `latest`.
4. Extract the new content in `Unreal\CarlaUE4\Content\Carla`.



In order to work with that the CARLA team is developing, go to get development assets below.

Launch the server Run the server in spectator view to make sure that everything worked properly.

```
1 make launch
```

Get development assets

The CARLA team works with assets still in development. These models and maps have a public git repository where the CARLA team regularly pushes latest updates. Assets are still unfinished, using them is only recommended for developers.

In order to handle this repository it is advised to install git-lfs. The repository is modified regularly, and git-lfs works faster with large binary files.

To clone the repository, **go to the main CARLA directory** and run the following command.

```
1 git clone https://bitbucket.org/carla-simulator/carla-content Unreal/CarlaUE4/Content/Carla
```



Delete the ‘/Carla’ folder containing the assets before cloning the repository. Otherwise, an error will show.

2.7 Build system

Setup LibCarla CarlaUE4 and Carla plugin PythonAPI

This document is a work in progress, only the Linux build system is taken into account here.

The most challenging part of the setup is to compile all the dependencies and modules to be compatible with a) Unreal Engine in the server-side, and b) Python in the client-side.

The goal is to be able to call Unreal Engine’s functions from a separate Python process.

In Linux, we compile CARLA and all the dependencies with clang-8.0 and C++14 standard. We however link against different runtime C++ libraries depending on where the code going to be used, since all the code that is going to be linked with Unreal Engine needs to be compiled using `libc++`.

Setup

Command

```
1 make setup
```

Get and compile dependencies

- llvm-8 (libc++ and libc++abi)
- rpcLib-2.2.1 (twice, with libstdc++ and libc++)
- boost-1.72.0 (headers and boost_python for libstdc++)
- googletest-1.8.1 (with libc++)

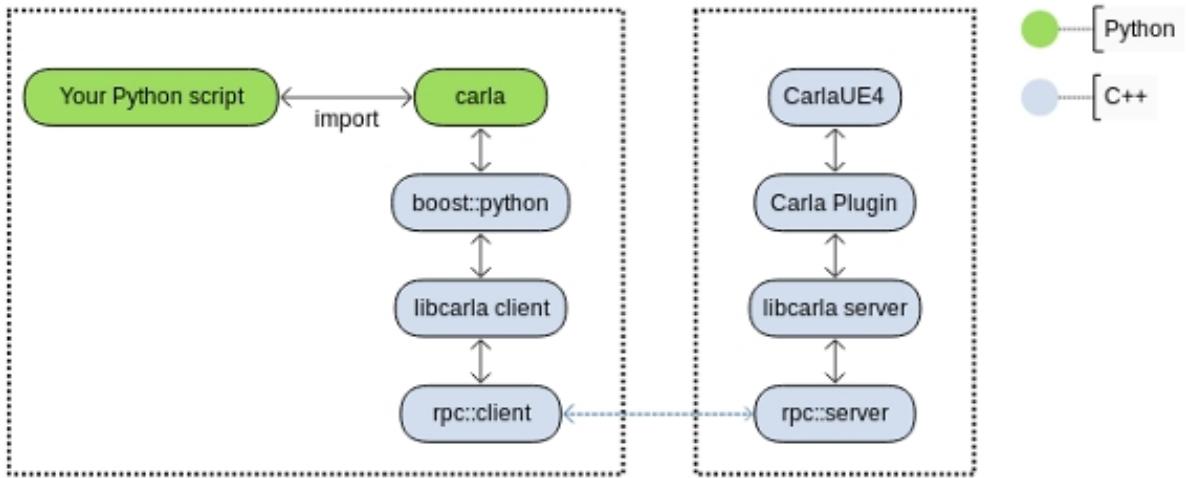


Figure 3: modules

LibCarla

Compiled with CMake (minimum version required CMake 3.9).

Command

```
1 make LibCarla
```

Two configurations:

Server

Client

Unit tests

Yes

No

Requirements

rpclib, gtest, boost

rpclib, boost

std runtime

LLVM's libc++

Default libstdc++

Output

headers and test exes

ibcarla_client.a

Required by

Carla plugin

PythonAPI

CarlaUE4 and Carla plugin

Both compiled at the same step with Unreal Engine build tool. They require the `UE4_ROOT` environment variable set.

Command

```
1 make CarlaUE4Editor
```

To launch Unreal Engine's Editor run

```
1 make launch
```

PythonAPI

Compiled using Python's `setuptools` ("setup.py"). Currently requires the following to be installed in the machine: Python, libpython-dev, and libboost-python-dev; both for Python 2.7 and 3.5.

Command

```
1 make PythonAPI
```

It creates two "egg" packages

- `PythonAPI/dist/carla-X.X.X-py2.7-linux-x86_64.egg`
- `PythonAPI/dist/carla-X.X.X-py3.7-linux-x86_64.egg`

This package can be directly imported into a Python script by adding it to the system path.

```
1 #!/usr/bin/env python
2
3 import sys
4
5 sys.path.append(
6     'PythonAPI/dist/carla-X.X.X-py%d.%d-linux-x86_64.egg' % (sys.version_info.major,
7                                                               sys.version_info.minor))
8
9 import carla
10
11 # ...
```

Alternatively, it can be installed with `easy_install`

```
1 easy_install2 --user --no-deps PythonAPI/dist/carla-X.X.X-py2.7-linux-x86_64.egg
2 easy_install3 --user --no-deps PythonAPI/dist/carla-X.X.X-py3.7-linux-x86_64.egg
```

2.8 Running CARLA in a Docker

- **Docker installation**
 - Docker CE
 - NVIDIA-Docker2
- **Running CARLA container**

This tutorial is designed for:

- People that want to run CARLA without needing to install all dependencies.
- Recommended solution to run multiple CARLA servers and perform GPU mapping.
- People who don't need to render the full simulation (the server is headless).

This tutorial was tested in Ubuntu 16.04 and using NVIDIA 396.37 drivers. This method requires a version of NVIDIA drivers $\geq=390$.

Docker Installation



Docker requires sudo to run. Follow this guide to add users to the docker sudo group
<https://docs.docker.com/install/linux/linux-postinstall/>

Docker CE For our tests we used the Docker CE version. To install Docker CE we recommend using this tutorial

NVIDIA-Docker2 To install nvidia-docker-2 we recommend using the “Quick Start” section from the nvidia-dockers github.

Running CARLA container

Pull the CARLA image.

```
1 docker pull carlasim/carla:version
```

For selecting a version, for instance, version 0.8.2 (stable), do:

```
1 docker pull carlasim/carla:0.8.2
```

Running CARLA under docker.

```
1 docker run -p 2000-2002:2000-2002 --runtime=nvidia --gpus all carlasim/carla:0.8.4
```

The `-p 2000-2002:2000-2002` argument is to redirect host ports for the docker container. Use `--gpus "device=<gpu_01>,<gpu_02>"` to specify which GPUs should run CARLA. Take a look at this NVIDIA documentation to learn other syntax options.

You can also pass parameters to the CARLA executable. With this you can chose the town and select the port that is going to be used.

```
1 docker run -p 2000-2002:2000-2002 --runtime=nvidia -e NVIDIA_VISIBLE_DEVICES=0 carlasim/carla:0.8.4  
    /bin/bash CarlaUE4.sh < Your list of parameters >
```

At the list of parameters do not forget to add `-world-port=<port_number>` so that CARLA runs on server mode listening to the `<port_number>`.

2.9 F.A.Q.

Some of the most common issues regarding CARLA installation and builds are listed here. Some more can be found in the GitHub issues for the project. In case you don't find your doubt listed here, have a look in the forum and feel free to ask there.

System requirements

Expected disk space to build CARLA. Recommended hardware to run CARLA.

Linux build

“CarlaUE4.sh” script does not appear when downloading from GitHub. “make launch” is not working on Linux. *Cloning the Unreal Engine repository shows an error.* AttributeError: module ‘carla’ has no attribute ‘Client’ when running a script. *Cannot run example scripts or “RuntimeError: rpc::rpc_error during call in function version”.

Windows build

“CarlaUE4.exe” does not appear when downloading from GitHub. CarlaUE4 could not be compiled. Try rebuilding it from source manually. *CMake error shows even though CMake is properly installed.* Error C2440, C2672: compiler version. *“make launch” is not working on Windows.* Make is missing libintl3.dll or/and libiconv2.dll. *Modules are missing or built with a different engine version.

Running Carla

Low FPS rate when running the server in Unreal Editor. Can't run a script. *Connect to the simulator while running within Unreal Editor.* Can't run CARLA neither binary nor source build. *ImportError: DLL load failed: The specified module could not be found.* ImportError: DLL load failed while importing libcarla: %1 is not a valid Win32 app.

Other

Fatal error: ‘version.h’ has been modified since the precompiled header. Create a binary version of CARLA.

System requirements

Expected disk space to build CARLA.

It is advised to have at least 30-50GB free. Building CARLA requires about 25GB of disk space, plus Unreal Engine, which is similar in size.

Unreal Engine on Linux requires much more disk space as it keeps all the intermediate files. This thread discusses the matter.

Recommended hardware to run CARLA.

CARLA is a performance demanding software. At the very minimum it requires a 4GB GPU or, even better, a dedicated GPU capable of running Unreal Engine.

Take a look at Unreal Engine's recommended hardware.

Linux build

“CarlaUE4.sh” script does not appear when downloading from GitHub.

There is no `CarlaUE4.sh` script in the source version of CARLA. Follow the build instructions to build CARLA from source.

To run CARLA using `CarlaUE4.sh`, follow the quick start installation.

“make launch” is not working on Linux.

Many different issues can be dragged out during the build installation and will manifest themselves like this. Here is a list of the most likely reasons why:

- **Run Unreal Engine 4.24.** Something may have failed when building Unreal Engine. Try running UE editor on its own and check that it is the 4.24 release.
- **Download the assets.** The server will not be able to run without the visual content. This step is mandatory.
- **UE4_ROOT is not defined.** The environment variable is not set. Remember to make it persistent session-wide by adding it to the `~/.bashrc` or `~/.profile`. Otherwise it will need to be set for every new shell. Run `export UE4_ROOT=~/UnrealEngine_4.24` to set the variable this time.
- **Check dependencies.** Make sure that everything was installed properly. Maybe one of the commands was skipped, unsuccessful or the dependencies were not suitable for the system.
- **Delete CARLA and clone it again.** Just in case something went wrong. Delete CARLA and clone or download it again.
- **Meet system requirements.** Ubuntu version should be 16.04 or later. CARLA needs around 15GB of disk space and a dedicated GPU (or at least one with 4GB) to run.

Other specific reasons for a system to show conflicts with CARLA may occur. Please, post these on the forum so the team can get to know more about them.

Cloning the Unreal Engine repository shows an error.

1. **Is the Unreal Engine account activated?** The UE repository is private. In order to clone it, create the UE account, activate it (check the verification mail), and link your GitHub account.

2. **Is git properly installed?** Sometimes an error shows incompatibilities with the `https` protocol. It can be solved easily by uninstalling and reinstalling git. Open a terminal and run the following commands:

```
1 $ sudo apt-get remove git #Uninstall git  
2 $ sudo apt install git-all #install git
```

AttributeError: module ‘carla’ has no attribute ‘Client’ when running a script.

Run the following command.

```
1 $ pip3 install -Iv setuptools==47.3.1
```

And build the PythonAPI again.

```
1 $ make PythonAPI
```

Try to build the docs to test if everything is running properly. A successful message should show.

```
1 $ make PythonAPI.docs
```

Cannot run example scripts or “RuntimeError: rpc::rpc_error during call in function version”.

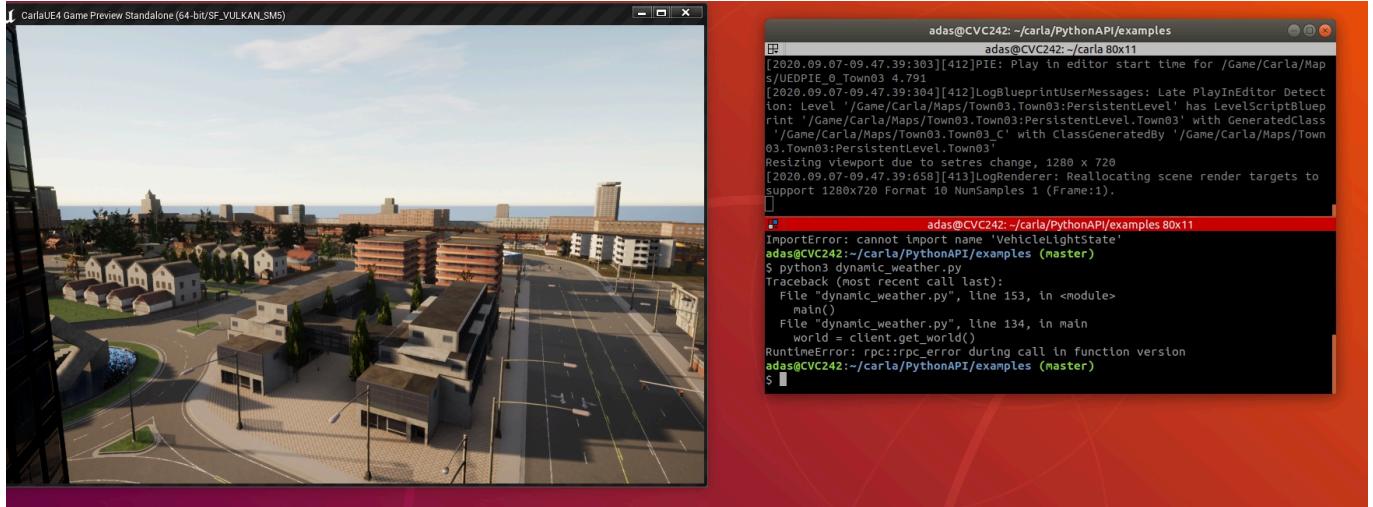


Figure 4: faq_rpc_error

If running a script returns an output similar to this, there is a problem with the .egg file in the PythonAPI.

First of all, open `<root_carla>/PythonAPI/carla/dist`. There should be an .egg file for the corresponding CARLA and Python version you are using (similar to `carla-0.X.X-pyX.X-linux-x86_64.egg`). Make sure the file matches the Python version you are using. To check your Python version use the following command.

```
1 $ python3 --version # CARLA no longer provides support for Python2, so we are dismissing it here
```

If either the file is missing or you think it could be corrupted, try rebuilding again.

```
1 $ make clean  
2 $ make PythonAPI  
3 $ make launch
```

Now try one of the example scripts again.

```
1 $ cd PythonAPI/examples  
2 $ python3 dynamic_weather.py
```

If the error persists, the problem is probably related with your PythonPATH. These scripts automatically look for the .egg file associated with the build, so maybe there is another .egg file in your PythonPATH interfering with the process. Show the content of the PythonPATH with the following command.

```
1 $ echo $PYTHONPATH
```

Look up in the output for other instances of .egg files in a route similar to `PythonAPI/carla/dist`, and get rid of these. They probably belong to other instances of CARLA installations. For example, if you also installed CARLA via `apt-get`, you can remove it with the following command, and the PythonPATH will be cleaned too.

```
1 $ sudo apt-get purge carla-simulator
```

Ultimately there is the option to add the .egg file of your build to the PythonPATH using the `~/.bashrc`. This is not the recommended way. It would be better to have a clear PythonPATH and simply add the path to the necessary .egg files in the scripts.

First, open `~/.bashrc`.

```
1 $ gedit ~/.bashrc
```

Add the following lines to `~/.bashrc`. These store the path to the build .egg file, so that Python can automatically find it. Save the file, and reset the terminal for changes to be effective.

```

1 export PYTHONPATH=$PYTHONPATH:"${CARLA_ROOT}/PythonAPI/carla/dist/$(ls
    ${CARLA_ROOT}/PythonAPI/carla/dist | grep py3.)"
2 export PYTHONPATH=$PYTHONPATH:${CARLA_ROOT}/PythonAPI/carla

```

After cleaning the PythonPATH or adding the path to the build .egg file, all the example scripts should work properly.

Windows build

“CarlaUE4.exe” does not appear when downloading from GitHub.

There is no **CarlaUE4.exe** executable in the source version of CARLA. Follow the build instructions to build CARLA from source. To directly get the **CarlaUE4.exe**, follow the quick start instructions.

CarlaUE4 could not be compiled. Try rebuilding it from source manually.

Something went wrong when trying to build CARLA. Rebuild using Visual Studio to discover what happened.

1. Go to `carla/Unreal/CarlaUE4` and right-click the `CarlaUE4.uproject`. **2.** Click on **Generate Visual Studio project files**. **3.** Open the file generated with Visual Studio 2017. **4.** Compile the project with Visual Studio. The shortcut is F7. The build will fail, but the issues found will be shown below.

Different issues may result in this specific error message. The user [@tamakoji](<https://github.com/tamakoji>) solved a recurrent case where the source code hadn't been cloned properly and the CARLA version could not be set (when downloading this as a .zip from git).

- **Check the Build/CMakeLists.txt.in.** If it shows as `set(CARLA_VERSION)` do the following:

1. Go to `Setup.bat` line 198.

2. Update the line from:

```
1 for /f %%i in ('git describe --tags --dirty --always') do set carla_version=%%i
```

to:

```
1 for /f %%i in ('git describe --tags --dirty --always') do set carla_version="0.9.9"
```

CMake error shows even though CMake is properly installed.

This issue occurs when trying to use the `make` command either to build the server or the client. Even if CMake is installed, updated and added to the environment path. There may be a conflict between Visual Studio versions.

Leave only VS2017 and completely erase the rest.

Error C2440, C2672: compiler version.

The build is not using the 2017 compiler due to conflicts with other Visual Studio or Microsoft Compiler versions. Uninstall these and rebuild again.

Visual Studio is not good at getting rid of itself. To completely clean Visual Studio from the computer go to `Program Files (x86)\Microsoft Visual Studio\Installer\resources\app\layout` and run `.\InstallCleanup.exe -full`. This may need admin permissions.

To keep other Visual Studio versions, edit `%appdata%\Unreal Engine\UnrealBuildTool\BuildConfiguration.xml` by adding the following lines:

```

1 <VCProjectFileGenerator>
2   <Version>VisualStudio2017</Version>
3 </VCProjectFileGenerator>

1 <WindowsPlatform>
2   <Compiler>VisualStudio2017</Compiler>
3 </WindowsPlatform>

```

“make launch” is not working on Windows.

Many different issues can be dragged out during the build installation and manifest themselves like this. Here is a list of the most likely reasons why:

- **Restart the computer.** There is a lot going on during the Windows build. Restart and make sure that everything is updated properly.
- **Run Unreal Engine 4.24.** Something may have failed when building Unreal Engine. Run the Editor and check that version 4.24 is being used.
- **Download the assets.** The server will not be able to run without the visual content. This step is mandatory.
- **Visual Studio 2017.** If there are other versions of Visual Studio installed or recently uninstalled, conflicts may arise. To completely clean Visual Studio from the computer go to `Program Files (x86)\Microsoft Visual Studio\Installer\resources\app\layout` and run `.\InstallCleanup.exe -full`.
- **Delete CARLA and clone it again.** Just in case something went wrong. Delete CARLA and clone or download it again.
- **Meet system requirements.** CARLA needs around 30-50GB of disk space and a dedicated GPU (or at least one with 4GB) to run.

Other specific reasons for a system to show conflicts with CARLA may occur. Please, post these on the forum so the team can get to know more about them.

Make is missing libintl3.dll or/and libiconv2.dll.

Download the dependencies and extract the `bin` content into the `make` installation path.

Modules are missing or built with a different engine version.

Click on **Accept** to rebuild them.

2.10 Running CARLA

Low FPS rate when running the server in Unreal Editor.

UE4 Editor goes to a low performance mode when out of focus.

Go to `Edit/Editor Preferences/Performance` in the editor preferences, and disable the “Use Less CPU When in Background” option.

Can't run a script.

Some scripts have requirements. These are listed in files named `Requirements.txt`, in the same path as the script itself. Be sure to check these in order to run the script. The majority of them can be installed with a simple `pip` command.

Sometimes on Windows, scripts cannot run with just `> script_name.py`. Try adding `> python3 script_name.py`, and make sure to be in the right directory.

Connect to the simulator while running within Unreal Editor.

Click on **Play** and wait until the scene is loaded. At that point, a Python client can connect to the simulator as with the standalone simulator.

Can't run CARLA neither binary nor source build.

NVIDIA drivers may be outdated. Make sure that this is not the case. If the issue is still unresolved, take a look at the forum and post the specific issue.

`ImportError: DLL load failed: The specified module could not be found.`

One of the libraries needed has not been properly installed. As a work around, go to `carla\Build\zlib-source\build`, and copy the file named `zlib.dll` in the directory of the script.

`ImportError: DLL load failed while importing libcarla: %1 is not a valid Win32 app.`

A 32-bit Python version is creating conflicts when trying to run a script. Uninstall it and leave only the Python3 x64 required.

2.11 Other

Fatal error: ‘version.h’ has been modified since the precompiled header.

This happens from time to time due to Linux updates. There is a special target in the Makefile for this issue. It takes a long time but fixes the issue:

```
1 $ make hard-clean  
2 $ make CarlaUE4Editor
```

Create a binary version of CARLA.

In Linux, run `make package` in the project folder. The package will include the project, and the Python API modules.

Alternatively, it is possible to compile a binary version of CARLA within Unreal Editor. Open the CarlaUE4 project, go to the menu `File/Package Project`, and select a platform. This may take a while.

3 First steps

3.1 Core concepts

This page introduces the main features and modules in CARLA. Detailed explanations of the different subjects can be found in their corresponding page.

In order to learn about the different classes and methods in the API, take a look at the Python API reference. Besides, the `Code recipes` reference contains some common code chunks, specially useful during these first steps.

- **First steps**

- 1st- World and client
- 2nd- Actors and blueprints
- 3rd- Maps and navigation
- 4th- Sensors and data

- **Advanced steps**



This documentation refers to CARLA 0.9.X . The API changed significantly from previous versions (0.8.X). There is another documentation regarding those versions that can be found [here](https://carla.readthedocs.io/en/stable/getting_started/).

First steps

1st- World and client **The client** is the module the user runs to ask for information or changes in the simulation. A client runs with an IP and a specific port. It communicates with the server via terminal. There can be many clients running at the same time. Advanced mult-client managing requires thorough understanding of CARLA and synchrony.

The world is an object representing the simulation. It acts as an abstract layer containing the main methods to spawn actors, change the weather, get the current state of the world, etc. There is only one world per simulation. It will be destroyed and substituted for a new one when the map is changed.

2nd- Actors and blueprints An actor is anything that plays a role in the simulation.

- Vehicles.
- Walkers.
- Sensors.
- The spectator.
- Traffic signs and traffic lights.

Blueprints are already-made actor layouts necessary to spawn an actor. Basically, models with animations and a set of attributes. Some of these attributes can be customized by the user, others don't. There is a Blueprint library containing all the blueprints available as well as information on them.

3rd- Maps and navigation **The map** is the object representing the simulated world, the town mostly. There are eight maps available. All of them use OpenDRIVE 1.4 standard to describe the roads.

Roads, lanes and junctions are managed by the Python API to be accessed from the client. These are used along with the **waypoint** class to provide vehicles with a navigation path.

Traffic signs and traffic lights are accessible as `carla.Blueprint` objects that contain information about their OpenDRIVE definition. Additionally, the simulator automatically generates stops, yields and traffic light objects when running using the information on the OpenDRIVE file. These have bounding boxes placed on the road. Vehicles become aware of them once inside their bounding box.

4th- Sensors and data Sensors wait for some event to happen, and then gather data from the simulation. They call for a function defining how to manage the data. Depending on which, sensors retrieve different types of **sensor data**.

A sensor is an actor attached to a parent vehicle. It follows the vehicle around, gathering information of the surroundings. The sensors available are defined by their blueprints in the Blueprint library.

- Cameras (RGB, depth and semantic segmentation).
- Collision detector.
- Gnss sensor.
- IMU sensor.
- Lidar raycast.
- Lane invasion detector.
- Obstacle detector.
- Radar.
- RSS.

Advanced steps

CARLA offers a wide range of features that go beyond the scope of this introduction to the simulator. Here are listed some of the most remarkable ones. However, it is highly encouraged to read the whole **First steps** section before starting with the advanced steps.

- **OpenDRIVE standalone mode.** Generates a road mesh using only an OpenDRIVE file. Allows to load any OpenDRIVE map into CARLA without the need of creating assets.
- **PTV-Vissim co-simulation.** Run a synchronous simulation between CARLA and PTV-Vissim traffic simulator.
- **Recorder.** Saves snapshots of the simulation state to reenact a simulation with exact precision.
- **Rendering options.** Graphics quality settings, off-screen rendering and a no-rendering mode.
- **RSS.** Integration of the C++ Library for Responsibility Sensitive Safety to modify a vehicle's trajectory using safety checks.
- **Simulation time and synchrony.** Everything regarding the simulation time and server-client communication.
- **SUMO co-simulation.** Run a synchronous simulation between CARLA and SUMO traffic simulator.
- **Traffic manager.** This module is in charge of every vehicle set to autopilot mode. It simulates traffic in the city for the simulation to look like a real urban environment.

That is a wrap on the CARLA basics. The next step takes a closer look to the world and the clients connecting to it.

Keep reading to learn more. Visit the forum to post any doubts or suggestions that have come to mind during this reading.

3.2 1st. World and client

The client and the world are two of the fundamentals of CARLA, a necessary abstraction to operate the simulation and its actors.

This tutorial goes from defining the basics and creation of these elements, to describing their possibilities. If any doubt or issue arises during the reading, the CARLA forum is there to solve them.

- **The client**
 - Client creation
 - World connection
 - Other client utilities
- **The world**
 - Actors
 - Weather
 - Lights
 - Debugging
 - World snapshots
 - World settings

The client

Clients are one of the main elements in the CARLA architecture. They connect to the server, retrieve information, and command changes. That is done via scripts. The client identifies itself, and connects to the world to then operate

with the simulation.

Besides that, clients are able to access advanced CARLA modules, features, and apply command batches. Only command batches will be covered in this section. These are useful for basic things such as spawning lots of actors. The rest of features are more complex, and they will be addressed in their respective pages in **Advanced steps**.

Take a look at `carla.Client` in the Python API reference to learn on specific methods and variables of the class.

Client creation Two things are needed. The **IP** address identifying it, and **two TCP ports** to communicate with the server. An optional third parameter sets the amount of working threads. By default this is set to all (0). This code recipe shows how to parse these as arguments when running the script.

```
1 client = carla.Client('localhost', 2000)
```

By default, CARLA uses local host IP, and port 2000 to connect but these can be changed at will. The second port will always be `n+1`, 2001 in this case.

Once the client is created, set its **time-out**. This limits all networking operations so that these don't block the client forever. An error will be returned if connection fails.

```
1 client.set_timeout(10.0) # seconds
```

It is possible to have many clients connected, as it is common to have more than one script running at a time. Working in a multiclient scheme with advanced CARLA features, such as the traffic manager, is bound to make communication more complex.



Client and server have different 'libcarla' modules. If the versions differ, issues may arise. This can be checked using the 'get client version()' and 'get server version()' methods.

World connection A client can connect and retrieve the current world fairly easily.

```
“‘py world = client.get_world()
```

The client can also get a list of available maps to change the current one. This will destroy the current world and create a new one.

```
1
2 print(client.get_available_maps())
3 ...
4 world = client.load_world('Town01')
5 ## client.reload_world() creates a new instance of the world with the same map.
```

Every world object has an `id` or `episode`. Everytime the client calls for `load_world()` or `reload_world()` the previous one is destroyed. A new one is created from scratch with a new episode. Unreal Engine is not rebooted in the process.

Using commands

Commands are adaptations of some of the most-common CARLA methods, that can be applied in batches. For instance, the `command.SetAutopilot` is equivalent to `Vehicle.set_autopilot()`, enables the autopilot for a vehicle. However, using the methods `Client.apply_batch` or `Client.apply_batch_sync()`, a list of commands can be applied in one single simulation step. This becomes extremely useful for methods that are usually applied to even hundreds of elements.

The following example uses a batch to destroy a list of vehicles all at once.

```
1 client.apply_batch([carla.command.DestroyActor(x) for x in vehicles_list])
```

All the commands available are listed in the latest section of the Python API reference.

Other client utilities

The main purpose of the client object is to get or change the world, and apply commands. However, it also provides access to some additional features.

- **Traffic manager.** This module is in charge of every vehicle set to autopilot to recreate urban traffic.
- **Recorder.** Allows to reenact a previous simulation. Uses snapshots summarizing the simulation state per frame.

3.3 The world

The major ruler of the simulation. Its instance should be retrieved by the client. It does not contain the model of the world itself, that is part of theMap class. Instead, most of the information, and general settings can be accessed from this class.

- Actors in the simulation and the spectator.
- Blueprint library.
- Map.
- Simulation settings.
- Snapshots.
- Weather and light manager.

Some of its most important methods are *getters*, precisely to retrieve information or instances of these elements. Take a look at carla.World to learn more about it.

Actors

The world has different methods related with actors that allow for different functionalities.

- Spawn actors (but not destroy them).
- Get every actor on scene, or find one in particular.
- Access the blueprint library.
- Access the spectator actor, the simulation's point of view.
- Retrieve a random location that is fitting to spawn an actor.

Spawning will be explained in 2nd. Actors and blueprints. It requires some understanding on the blueprint library, attributes, etc.

Weather

The weather is not a class on its own, but a set of parameters accessible from the world. The parametrization includes sun orientation, cloudiness, wind, fog, and much more. The helper class carla.WeatherParameters is used to define a custom weather.

```
1 weather = carla.WeatherParameters(  
2     cloudiness=80.0,  
3     precipitation=30.0,  
4     sun_altitude_angle=70.0)  
5  
6  
7 world.set_weather(weather)  
8  
9  
10 print(world.get_weather())
```

There are some weather presets that can be directly applied to the world. These are listed in carla.WeatherParameters and accessible as an enum.

```
1 world.set_weather(carla.WeatherParameters.WetCloudySunset)
```

The weather can also be customized using two scripts provided by CARLA.

- **environment.py** (*in PythonAPI/util*) — Provides access to weather and light parameters so that these can be changed in real time.

Optional arguments in environment.py

```
1 -h, --help          show this help message and exit  
2 --host H            IP of the host server (default: 127.0.0.1)  
3 -p P, --port P      TCP port to listen to (default: 2000)  
4 --sun SUN           Sun position presets [sunset | day | night]  
5 --weather WEATHER   Weather condition presets [clear | overcast | rain]  
6 --altitude A, -alt A Sun altitude [-90.0, 90.0]  
7 --azimuth A, -azm A Sun azimuth [0.0, 360.0]  
8 --clouds C, -c C    Clouds amount [0.0, 100.0]
```

```

9 --rain R, -r R      Rain amount [0.0, 100.0]
10 --puddles Pd, -pd Pd Puddles amount [0.0, 100.0]
11 --wind W, -w W     Wind intensity [0.0, 100.0]
12 --fog F, -f F      Fog intensity [0.0, 100.0]
13 --fogdist Fd, -fd Fd Fog Distance [0.0, inf)
14 --wetness Wet, -wet Wet
15                           Wetness intensity [0.0, 100.0]

```

- **dynamic_weather.py** (*in PythonAPI/examples*) — Enables a particular weather cycle prepared by developers for each CARLA map.

Optional arguments in `dynamic_weather.py`

```

1 -h, --help           show this help message and exit
2 --host H             IP of the host server (default: 127.0.0.1)
3 -p P, --port P       TCP port to listen to (default: 2000)
4 -s FACTOR, --speed FACTOR
5                           rate at which the weather changes (default: 1.0)

```



Changes in the weather do not affect physics. They are only visuals that can be captured by the camera sensors.

Night mode starts when `sun_altitude_angle < 0`, which is considered sunset. This is when lights become especially relevant.

Lights

- **Street lights** automatically turn on when the simulation enters night mode. The lights are placed by the developers of the map, and accessible as `carla.Light` objects. Properties such as color and intensity can be changed at will. The variable `light_state` of type `carla.LightState` allows setting all of these in one call. Street lights are categorized using their attribute `light_group`, of type `carla.LightGroup`. This allows to classify lights as street lights, building lights... An instance of `carla.LightManager` can be retrieved to handle groups of lights in one call.

```

1 ## Get the light manager and lights
2 lmanager = world.get_light_manager()
3 mylights = lmanager.get_all_lights()
4
5 ## Custom a specific light
6 light01 = mylights[0]
7 light01.turn_on()
8 light01.set_intensity(100.0)
9 state01 = carla.LightState(200.0,red,carla.LightGroup.Building,True)
10 light01.set_light_state(state01)
11
12 ## Custom a group of lights
13 my_lights = lmanager.get_light_group(carla.LightGroup.Building)
14 lmanager.turn_on(my_lights)
15 lmanager.set_color(my_lights,carla.Color(255,0,0))
16 lmanager.set_intensities(my_lights,list_of_intensities)

```

- **Vehicle lights** have to be turned on/off by the user. Each vehicle has a set of lights listed in `carla.VehicleLightState`. So far, not all vehicles have lights integrated. Here is a list of those that are available by the time of writing.
 - **Bikes.** All of them have a front and back position light.
 - **Motorcycles.** Yamaha and Harley Davidson models.
 - **Cars.** Audi TT, Chevrolet, Dodge (the police car), Etron, Lincoln, Mustang, Tesla 3S, Volkswagen T2 and the new guests coming to CARLA.

The lights of a vehicle can be retrieved and updated anytime using the methods `carla.Vehicle.get_light_state` and `carla.Vehicle.set_light_state`. These use binary operations to customize the light setting.

```
1 ## Turn on position lights
2 current_lights = carla.VehicleLightState.NONE
3 current_lights |= carla.VehicleLightState.Position
4 vehicle.set_light_state(current_lights)
```



Lights can also be set in real time using the ‘environment.py’ described in the weather section.

Debugging

World objects have a carla.DebugHelper object as a public attribute. It allows for different shapes to be drawn during the simulation. These are used to trace the events happening. The following example would draw a red box at an actor’s location and rotation.

```
1 debug = world.debug
2 debug.draw_box(carla.BoundingBox(actor_snapshot.get_transform().location,carla.Vector3D(0.5,0.5,2)),actor_snapshot
    0.05, carla.Color(255,0,0,0),0)
```

This example is extended in a code recipe to draw boxes for every actor in a world snapshot.

World snapshots

Contains the state of every actor in the simulation at a single frame. A sort of still image of the world with a time reference. The information comes from the same simulation step, even in asynchronous mode.

```
1 ## Retrieve a snapshot of the world at current frame.
2 world_snapshot = world.get_snapshot()
```

A carla.WorldSnapshot contains a carla.Timestamp and a list of carla.ActorSnapshot. Actor snapshots can be searched using the `id` of an actor. A snapshot lists the `id` of the actors appearing in it.

```
1 timestamp = world_snapshot.timestamp # Get the time reference
2
3 for actor_snapshot in world_snapshot: # Get the actor and the snapshot information
4     actual_actor = world.get_actor(actor_snapshot.id)
5     actor_snapshot.get_transform()
6     actor_snapshot.get_velocity()
7     actor_snapshot.get_angular_velocity()
8     actor_snapshot.get_acceleration()
9
10 actor_snapshot = world_snapshot.find(actual_actor.id) # Get an actor's snapshot
```

World settings

The world has access to some advanced configurations for the simulation. These determine rendering conditions, simulation time-steps, and synchrony between clients and server. They are accessible from the helper class carla.WorldSettings.

For the time being, default CARLA runs with the best graphics quality, a variable time-step, and asynchronously. To dive further in this matters take a look at the **Advanced steps** section. The pages on synchrony and time-step, and rendering options could be a great starting point.

That is a wrap on the world and client objects. The next step takes a closer look into actors and blueprints to give life to the simulation.

Keep reading to learn more. Visit the forum to post any doubts or suggestions that have come to mind during this reading.

3.4 2nd. Actors and blueprints

Actors not only include vehicles and walkers, but also sensors, traffic signs, traffic lights, and the spectator. It is crucial to have a full understanding on how to operate on them.

This section will cover spawning, destruction, types, and how to manage them. However, the possibilities are almost endless. Experiment, take a look at the **tutorials** in this documentation and share doubts and ideas in the CARLA forum.

- **Blueprints**
 - Managing the blueprint library
- **Actor life cycle**
 - Spawning
 - Handling
 - Destruction
- **Types of actors**
 - Sensors
 - Spectator
 - Traffic signs and traffic lights
 - Vehicles
 - Walkers

Blueprints

These layouts allow the user to smoothly incorporate new actors into the simulation. They are already-made models with animations and a series of attributes. Some of these are modifiable and others are not. These attributes include, among others, vehicle color, amount of channels in a lidar sensor, a walker's speed, and much more.

Available blueprints are listed in the blueprint library, along with their attributes.

Managing the blueprint library The carla.BlueprintLibrary class contains a list of carla.ActorBlueprint elements. It is the world object who can provide access to it.

```
1 blueprint_library = world.get_blueprint_library()
```

Blueprints have an ID to identify them and the actors spawned with it. The library can be read to find a certain ID, choose a blueprint at random, or filter results using a wildcard pattern.

```
1 # Find a specific blueprint.  
2 collision_sensor_bp = blueprint_library.find('sensor.other.collision')  
3 # Choose a vehicle blueprint at random.  
4 vehicle_bp = random.choice(blueprint_library.filter('vehicle.*.*'))
```

Besides that, each carla.ActorBlueprint has a series of carla.ActorAttribute that can be get and set . py is
`bike = [vehicle.get_attribute('number of wheels') == 2] if(is bike) vehicle.set_attribute('color', '255,0,0')`



Some of the attributes cannot be modified. Check it out in the blueprint library.

Attributes have an carla.ActorAttributeType variable. It states its type from a list of enums. Also, modifiable attributes come with a **list of recommended values**.

```
1 for attr in blueprint:  
2     if attr.is_modifiable:  
3         blueprint.set_attribute(attr.id, random.choice(attr.recommended_values))
```



Users can create their own vehicles. Check the Tutorials (assets) to learn on that. Contributors can [add their new content to CARLA](tuto D contribute assets.md).

Actor life cycle



This section mentions different methods regarding actors. The Python API provides for [commands](python api.md command.SpawnActor) to apply batches of the most common ones, in just one frame.

Spawning The world object is responsible of spawning actors and keeping track of these. Spawning only requires a blueprint, and a carla.Transform stating a location and rotation for the actor.

The world has two different methods to spawn actors.

- `spawn_actor()` raises an exception if the spawning fails.
- `try_spawn_actor()` returns `None` if the spawning fails.

```
1 transform = Transform(Location(x=230, y=195, z=40), Rotation(yaw=180))
2 actor = world.spawn_actor(blueprint, transform)
```



CARLA uses the [Unreal Engine coordinates system](<https://carla.readthedocs.io/en/latest/python/api/carlarotation>). Remember that `[‘carla.Rotation’](https://carla.readthedocs.io/en/latest/python/api/carlarotation)` constructor is defined as ‘(pitch, yaw, roll)’, that differs from Unreal Engine Editor ‘(roll, pitch, yaw)’.

The actor will not be spawned in case of collision at the specified location. No matter if this happens with a static object or another actor. It is possible to try avoiding these undesired spawning collisions.

- `map.get_spawn_points() for vehicles`. Returns a list of recommended spawning points.

```
1 spawn_points = world.get_map().get_spawn_points()
```

- `world.get_random_location() for walkers`. Returns a random point on a sidewalk. This same method is used to set a goal location for walkers.

```
1 spawn_point = carla.Transform()
2 spawn_point.location = world.get_random_location_from_navigation()
```

An actor can be attached to another one when spawned. Actors follow the parent they are attached to. This is specially useful for sensors. The attachment can be rigid (proper to retrieve precise data) or with an eased movement according to its parent. It is defined by the helper class `carla.AttachmentType`.

The next example attaches a camera rigidly to a vehicle, so their relative position remains fixed.

```
1 camera = world.spawn_actor(camera_bp, relative_transform, attach_to=my_vehicle,
                             carla.AttachmentType.Rigid)
```



When spawning attached actors, the transform provided must be relative to the parent actor.

Once spawned, the world object adds the actors to a list. This can be easily searched or iterated on.

```
1 actor_list = world.get_actors()
2 # Find an actor by id.
3 actor = actor_list.find(id)
4 # Print the location of all the speed limit signs in the world.
5 for speed_sign in actor_list.filter('traffic.speed_limit.*'):
6     print(speed_sign.get_location())
```

Handling `carla.Actor` mostly consists of `get()` and `set()` methods to manage the actors around the map.

```
1 print(actor.get_acceleration())
2 print(actor.get_velocity())
3
4 location = actor.get_location()
5 location.z += 10.0
6 actor.set_location(location)
```

The actor’s physics can be disabled to freeze it in place.

```
1 actor.set_simulate_physics(False)
```

Besides that, actors also have tags provided by their blueprints. These are mostly useful for semantic segmentation sensors.



Most of the methods send requests to the simulator asynchronously. The simulator has a limited amount of time each update to parse them. Flooding the simulator with `set()` methods will accumulate a significant lag.

Destruction Actors are not destroyed when a Python script finishes. They have to explicitly destroy themselves.

```
1 destroyed_sucessfully = actor.destroy() # Returns True if successful
```



Destroying an actor blocks the simulator until the process finishes.

Types of actors

Sensors Sensors are actors that produce a stream of data. They have their own section, 4th. Sensors and data. For now, let's just take a look at a common sensor spawning cycle.

This example spawns a camera sensor, attaches it to a vehicle, and tells the camera to save the images generated to disk.

```
1 camera_bp = blueprint_library.find('sensor.camera.rgb')
2 camera = world.spawn_actor(camera_bp, relative_transform, attach_to=my_vehicle)
3 camera.listen(lambda image: image.save_to_disk('output/%06d.png' % image.frame))
```

- Sensors have blueprints too. Setting attributes is crucial.
- Most of the sensors will be attached to a vehicle to gather information on its surroundings.
- Sensors **listen** to data. When data is received, they call a function described with a **Lambda expression** (6.13 in the link provided).

Spectator Placed by Unreal Engine to provide an in-game point of view. It can be used to move the view of the simulator window. The following example would move the spectator actor, to point the view towards a desired vehicle.

```
1 spectator = world.get_spectator()
2 transform = vehicle.get_transform()
3 spectator.set_transform(carla.Transform(transform.location + carla.Location(z=50),
4 carla.Rotation(pitch=-90)))
```

Traffic signs and traffic lights Only stops, yields and traffic lights are considered actors in CARLA so far. The rest of the OpenDRIVE signs are accessible from the API as **carla.Landmark**. Their information is accessible using these instances, but they do not exist in the simulation as actors. Landmarks are explained more in detail in the following step, **3rd. Maps and navigation**.

When the simulation starts, stop, yields and traffic light are automatically generated using the information in the OpenDRIVE file. **None of these can be found in the blueprint library** and thus, cannot be spawned.



CARLA maps do not have traffic signs nor lights in the OpenDRIVE file. These are manually placed by developers.

Traffic signs are not defined in the road map itself, as explained in the following page. Instead, they have a `carla.BoundingBox` to affect vehicles inside of it.

```
1 #Get the traffic light affecting a vehicle
2 if vehicle_actor.is_at_traffic_light():
3     traffic_light = vehicle_actor.get_traffic_light()
```

Traffic lights are found in junctions. They have their unique ID, as any actor, but also a `group` ID for the junction. To identify the traffic lights in the same group, a `pole` ID is used.

The traffic lights in the same group follow a cycle. The first one is set to green while the rest remain frozen in red. The active one spends a few seconds in green, yellow and red, so there is a period of time where all the lights are red. Then, the next traffic light starts its cycle, and the previous one is frozen with the rest.

The state of a traffic light can be set using the API. So does the seconds spent on each state. Possible states are described with carla.TrafficLightState as a series of enum values.

```
1 #Change a red traffic light to green
2 if traffic_light.get_state() == carla.TrafficLightState.Red:
3     traffic_light.set_state(carla.TrafficLightState.Green)
4     traffic_light.set_set_green_time(4.0)
```



Vehicles will only be aware of a traffic light if the light is red.

Vehicles carla.Vehicle are a special type of actor. They are remarkable for having better physics. This is achieved applying four types of different controls.

- carla.VehicleControl provides input for driving commands such as throttle, steering, brake, etc.

```
1 vehicle.apply_control(carla.VehicleControl(throttle=1.0, steer=-1.0))
```

- carla.VehiclePhysicsControl defines physical attributes of the vehicle. Besides many different attribute, this controller contains two more controllers. carla.GearPhysicsControl for the gears. The other is a list of carla.WheelPhysicsControl, that provide specific control over the different wheels.

```
1 vehicle.apply_physics_control(carla.VehiclePhysicsControl(max_rpm = 5000.0, center_of_mass =
    carla.Vector3D(0.0, 0.0, 0.0), torque_curve=[[0,400], [5000,400]]))
```

In order to apply physics and detect collisions, vehicles have a carla.BoundingBox encapsulating them.

```
1 box = vehicle.bounding_box
2 print(box.location)           # Location relative to the vehicle.
3 print(box.extent)            # XYZ half-box extents in meters.
```

Vehicles include other functionalities unique to them.

- The **autopilot mode** will subscribe them to theTraffic manager, and simulate real urban conditions. This module is hard-coded, not based on machine learning.

```
1 vehicle.set_autopilot(True)
```

- **Vehicle lights** have to be turned on/off by the user. Each vehicle has a set of lights listed in carla.VehicleLightState. So far, not all vehicles have lights integrated. Here is a list of those that are available by the time of writing.
 - **Bikes.** All of them have a front and back position light.
 - **Motorcycles.** Yamaha and Harley Davidson models.
 - **Cars.** Audi TT, Chevrolet, Dodge (the police car), Etron, Lincoln, Mustang, Tesla 3S, Volkswagen T2 and the new guests coming to CARLA.

The lights of a vehicle can be retrieved and updated anytime using the methods carla.Vehicle.get_light_state and carla.Vehicle.set_light_state. These use binary operations to customize the light setting.

```
1 # Turn on position lights
2 current_lights = carla.VehicleLightState.NONE
3 current_lights |= carla.VehicleLightState.Position
4 vehicle.set_light_state(current_lights)
```

Walkers carla.Walker work in a similar way as vehicles do. Control over them is provided by controllers.

- carla.WalkerControl moves the pedestrian around with a certain direction and speed. It also allows them to jump.
- carla.WalkerBoneControl provides control over the 3D skeleton. This tutorial explains how to control it.

Walkers can be AI controlled. They do not have an autopilot mode. The `carla.WalkerAIController` actor moves around the actor it is attached to.

```
1 walker_controller_bp = world.get_blueprint_library().find('controller.ai.walker')
2 world.SpawnActor(walker_controller_bp, carla.Transform(), parent_walker)
```



The AI controller is bodiless and has no physics. It will not appear on scene. Also, location '(0,0,0)' relative to its parent will not cause a collision.

Each AI controller needs initialization, a goal and, optionally, a speed. Stopping the controller works in the same manner.

```
1 ai_controller.start()
2 ai_controller.go_to_location(world.get_random_location_from_navigation())
3 ai_controller.set_max_speed(1 + random.random()) # Between 1 and 2 m/s (default is 1.4 m/s).
4 ...
5 ai_controller.stop()
```

When a walker reaches the target location, they will automatically walk to another random point. If the target point is not reachable, walkers will go to the closest point from their current location.

This recipe uses batches to spawn a lot of walkers and make them wander around.



To destroy AI pedestrians , stop the AI controller and destroy both, the actor, and the controller.

That is a wrap as regarding actors in CARLA. The next step takes a closer look into the map, roads and traffic in CARLA.

Keep reading to learn more or visit the forum to post any doubts or suggestions that have come to mind during this reading.

3.5 3rd. Maps and navigation

After discussing about the world and its actors, it is time to put everything into place and understand the map and how do the actors navigate it.

- **The map**
 - Changing the map
 - Landmarks
 - Lanes
 - Junctions
 - Waypoints
- **Navigation in CARLA**
 - Navigating through waypoints
 - Generating a map navigation
- **CARLA maps**

The map

A map includes both the 3D model of a town and its road definition. Every map is based on an OpenDRIVE file describing the road layout fully annotated. The way the OpenDRIVE standard 1.4 defines roads, lanes, junctions, etc. is extremely important. It determines the possibilities of the API and the reasoning behind decisions made.

The Python API makes for a high level querying system to navigate these roads. It is constantly evolving to provide a wider set of tools.

Changing the map To change the map, the world has to change too. Everything will be rebooted and created from scratch, besides the Unreal Editor itself. There are two ways to do so.

- `reload_world()` creates a new instance of the world with the same map.

- `load_world()` changes the current map and creates a new world.

```
1 world = client.load_world('Town01')
```

The client can get a list of available maps. Each map has a `name` attribute that matches the name of the currently loaded city, e.g. `Town01`.

```
1 print(client.get_available_maps())
```

Landmarks The traffic signs defined in the OpenDRIVE file are translated into CARLA as landmark objects that can be queried from the API. In order to facilitate their manipulation, there have been several additions to it.

- **carla.Landmark** objects represent the OpenDRIVE signals. The attributes and methods describe the landmark, and where it is effective.
 - **carla.LandmarkOrientation** states the orientation of the landmark with regards of the road's geometry definition.
 - **carla.LandmarkType** contains some common landmark types, to ease translation to OpenDRIVE types.
- A **carla.Waypoint** can get landmarks located a certain distance ahead of it. The type of landmark can be specified.
- The **carla.Map** retrieves sets of landmarks. It can return all the landmarks in the map, or those having an ID, type or group in common.
- The **carla.World** acts as intermediary between landmarks, and the *carla.TrafficSign* and *carla.TrafficLight* that embody them in the simulation.

```
1 my_waypoint.get_landmarks(200.0,True)
```

Lanes The lane types defined by OpenDRIVE standard 1.4 are translated to the API in **carla.LaneType** as a series of enum values.

The lane markings surrounding a lane can be accessed through **carla.LaneMarking**. These are defined with a series of variables.

- **carla.LaneMarkingType** are enum values according to OpenDRIVE standards.
- **carla.LaneMarkingColor** are enum values to determine the color of the marking.
- **width** to state thickness of the marking.
- **carla.LaneChange** to state permissions to perform lane changes.

Waypoints use these to acknowledge traffic permissions.

```
1 # Get the lane type where the waypoint is.
2 lane_type = waypoint.lane_type
3
4 # Get the type of lane marking on the left.
5 left_lanemarking_type = waypoint.left_lane_marking.type()
6
7 # Get available lane changes for this waypoint.
8 lane_change = waypoint.lane_change
```

Junctions A **carla.Junction** represents an OpenDRIVE junction. This class provides for a bounding box to state whereas lanes or vehicles are inside of it.

The most remarkable method of this class returns a pair of waypoints per lane inside the junction. Each pair is located at the starting and ending point of the junction boundaries.

```
1 waypoints_junc = my_junction.get_waypoints()
```

Waypoints A **carla.Waypoint** is a 3D-directed point. These are prepared to mediate between the world and the openDRIVE definition of the road. Everything related with waypoints happens on the client-side, so there no communication with the server is needed.

Each waypoint contains a **carla.Transform**. This states its location on the map and the orientation of the lane containing it. The variables `road_id`,`section_id`,`lane_id` and `s` translate this transform to the OpenDRIVE road. These combined, create the `id` of the waypoint.



Due to granularity, waypoints closer than 2cm within the same road share the same ‘id’.

A waypoint also contains some information regarding the **lane** containing it. Specifically its left and right **lane markings**, and a boolean to determine if it is inside a junction.

```
1 # Examples of a waypoint accessing to lane information
2 inside_junction = waypoint.is_junction()
3 width = waypoint.lane_width
4 right_lm_color = waypoint.right_lane_marking.color
```

Navigation in CARLA

Navigation in CARLA is managed via the waypoint API. This consists of a summary of methods in carla.Waypoint and carla.Map.

All the queries happen on the client-side. The client only communicates with the server when retrieving the map object that will be used for the queries. There is no need to retrieve the map (`world.get_map()`) more than once.

Navigating through waypoints Waypoints have a set of methods to connect with others and create a road flow. All of these methods follow traffic rules to determine only places where the vehicle can go.

- `next(d)` creates a list of waypoints at an approximate distance **d in the direction of the lane**. The list contains one waypoint for each deviation possible.
- `previous(d)` creates a list of waypoints waypoint at an approximate distance **d on the opposite direction of the lane**. The list contains one waypoint for each deviation possible.
- `next_until_lane_end(d)` and `previous_until_lane_start(d)` returns a list of waypoints a distance **d** apart. The list goes from the current waypoint to the end and start of its lane, respectively.
- `get_right_lane()` and `get_left_lane()` return the equivalent waypoint in an adjacent lane, if any. A lane change maneuver can be made by finding the next waypoint to the one on its right/left lane, and moving to it.

```
1 # Disable physics, in this example the vehicle is teleported.
2 vehicle.set_simulate_physics(False)
3 while True:
4     # Find next waypoint 2 meters ahead.
5     waypoint = random.choice(waypoint.next(2.0))
6     # Teleport the vehicle.
7     vehicle.set_transform(waypoint.transform)
```

Generating a map navigation The instance of the map is provided by the world. It will be useful to create routes and make vehicles roam around the city and reach goal destinations.

The following method asks the server for the XODR map file, and parses it to a carla.Map object. It only needs to be called once. Maps can be quite heavy, and successive calls are unnecessary and expensive.

```
1 map = world.get_map()

    • Get recommended spawn points for vehicles pointed by developers. There is no assurance that these spots will be free.

1 spawn_points = world.get_map().get_spawn_points()

    • Get the closest waypoint to a specific location or to a certain road_id, lane_id and s in OpenDRIVE.

1 # Nearest waypoint on the center of a Driving or Sidewalk lane.
2 waypoint01 = map.get_waypoint(vehicle.get_location(), project_to_road=True,
    lane_type=(carla.LaneType.Driving | carla.LaneType.Sidewalk))
3
4
5 #Nearest waypoint but specifying OpenDRIVE parameters.
6 waypoint02 = map.get_waypoint_xodr(road_id, lane_id, s)

    • Generate a collection of waypoints to visualize the city lanes. Creates waypoints all over the map, for every road and lane. All of them will be an approximate distance apart.
```

```
1 waypoint_list = map.generate_waypoints(2.0)
```

- **Generate road topology.** Returns a list of pairs (tuples) of waypoints. For each pair, the first element connects with the second one and both define the starting and ending point of each lane in the map.

```
1 waypoint_tuple_list = map.get_topology()
```

- **Convert simulation point to geographical coordinates.** Transforms a certain location to a carla.GeoLocation with latitude and longitude values.

```
1 my_geolocation = map.transform_to_geolocation(vehicle.transform)
```

- **Save road information.** Converts the road information to OpenDRIVE format, and saves it to disk.

```
1 info_map = map.to_opendrive()
```

CARLA maps

So far there are seven different maps available. Each one has unique features and is useful for different purposes. Hereunder is a brief sum up on them.



Users can [customize a map](tuto A map customization.md) or even [create a new map](tuto A add map.md) to be used in CARLA.

Town

Summary

Town01

A basic town layout with all “T junctions”.

Town02

Similar to Town01, but smaller.

Town03

The most complex town, with a 5-lane junction, a roundabout, unevenness, a tunnel, and much more. Essentially a medley.

Town04

An infinite loop with a highway and a small town.

Town05

Squared-grid town with cross junctions and a bridge. It has multiple lanes per direction. Useful to perform lane changes.

Town06

Long highways with many highway entrances and exits. It also has a .

Town07

A rural environment with narrow roads, barely non traffic lights and barns.

Town10

A city environment with with different environments such as an avenue or a promenade, and more realistic textures.

Town01

Town02

```
1 
2 <div class="text">Town03</div>
```

```
1 
2 <div class="text">Town04</div>
```

```
1 
2 <div class="text">Town05</div>
```

```
1 
2 <div class="text">Town06</div>
```

```
1 
2 <div class="text">Town07</div>
```

```
1 
2 <div class="text">Town10</div>
```

That is a wrap as regarding maps and navigation in CARLA. The next step takes a closer look into sensors types, and the data they retrieve.

Keep reading to learn more or visit the forum to post any doubts or suggestions that have come to mind during this reading.

3.6 4th. Sensors and data

Sensors are actors that retrieve data from their surroundings. They are crucial to create learning environment for driving agents.

This page summarizes everything necessary to start handling sensors. It introduces the types available and a step-by-step guide of their life cycle. The specifics for every sensor can be found in [thesensors reference](#).

Sensors step-by-step Setting * Spawning * Listening * Data **Types of sensors** Cameras * Detectors * Other

Sensors step-by-step

The class carla.Sensor defines a special type of actor able to measure and stream data.

- **What is this data?** It varies a lot depending on the type of sensor. All the types of data are inherited from the general carla.SensorData.
- **When do they retrieve the data?** Either on every simulation step or when a certain event is registered. Depends on the type of sensor.
- **How do they retrieve the data?** Every sensor has a `listen()` method to receive and manage the data.

Despite their differences, all the sensors are used in a similar way.

Setting As with every other actor, find the blueprint and set specific attributes. This is essential when handling sensors. Their attributes will determine the results obtained. These are detailed in [thesensors reference](#).

The following example sets a dashboard HD camera.

```
1 # Find the blueprint of the sensor.
2 blueprint = world.get_blueprint_library().find('sensor.camera.rgb')
3 # Modify the attributes of the blueprint to set image resolution and field of view.
4 blueprint.set_attribute('image_size_x', '1920')
5 blueprint.set_attribute('image_size_y', '1080')
6 blueprint.set_attribute('fov', '110')
7 # Set the time in seconds between sensor captures
8 blueprint.set_attribute('sensor_tick', '1.0')
```

Spawning `attachment_to` and `attachment_type`, are crucial. Sensors should be attached to a parent actor, usually a vehicle, to follow it around and gather the information. The attachment type will determine how its position is updated regarding said vehicle.

- **Rigid attachment.** Movement is strict regarding its parent location. This is the proper attachment to retrieve data from the simulation.
- **SpringArm attachment.** Movement is eased with little accelerations and decelerations. This attachment is only recommended to record videos from the simulation. The movement is smooth and “hops” are avoided when updating the cameras’ positions.

```
1 transform = carla.Transform(carla.Location(x=0.8, z=1.7))
2 sensor = world.spawn_actor(blueprint, transform, attach_to=my_vehicle)
```



When spawning with attachment, location must be relative to the parent actor.

Listening Every sensor has a `listen()` method. This is called every time the sensor retrieves data.

The argument `callback` is a lambda function. It describes what should the sensor do when data is retrieved. This must have the data retrieved as an argument.

```
1 # do_something() will be called each time a new image is generated by the camera.
2 sensor.listen(lambda data: do_something(data))
3
4 ...
5
6 # This collision sensor would print everytime a collision is detected.
7 def callback(event):
8     for actor_id in event:
9         vehicle = world_ref().get_actor(actor_id)
10        print('Vehicle too close: %s' % vehicle.type_id)
11
12 sensor02.listen(callback)
```

Data Most sensor data objects have a function to save the information to disk. This will allow it to be used in other environments.

Sensor data differs a lot between sensor types. Take a look at `thesensors` reference to get a detailed explanation. However, all of them are always tagged with some basic information.

Sensor data attribute

Type

Description

frame

int

Frame number when the measurement took place.

timestamp

double

Timestamp of the measurement in simulation seconds since the beginning of the episode.

transform

World reference of the sensor at the time of the measurement.



‘is listening’ is a sensor attribute that enables/disables data listening at will. ‘sensor tick’ is a blueprint attribute that sets the simulation time between data received.

Types of sensors

Cameras Take a shot of the world from their point of view. The helper class `carla.ColorConverter` will modify said image to represent different information.

- **Retrieve data** every simulation step.

Sensor

Output

Overview

Depth

Renders the depth of the elements in the field of view in a gray-scale map.

RGB

Provides clear vision of the surroundings. Looks like a normal photo of the scene.

Semantic segmentation

Renders elements in the field of view with a specific color according to their tags.

Detectors Retrieve data when the object they are attached to registers a specific event.

- **Retrieve data** when triggered.

Sensor

Output

Overview

Collision

Retrieves collisions between its parent and other actors.

Lane invasion

Registers when its parent crosses a lane marking.

Obstacle

Detects possible obstacles ahead of its parent.

Other Different functionalities such as navigation, measurement of physical properties and 2D/3D point maps of the scene.

- **Retrieve data** every simulation step.

Sensor

Output

Overview

GNSS

Retrieves the geolocation of the sensor.

IMU

Comprises an accelerometer, a gyroscope, and a compass.

LIDAR

A rotating LIDAR. Generates a 4D point cloud with coordinates and intensity per point to model the surroundings.

Radar

2D point map modelling elements in sight and their movement regarding the sensor.

RSS

Modifies the controller applied to a vehicle according to safety checks. This sensor works in a different manner than the rest, and there is specific for it.

Semantic LIDAR

A rotating LIDAR. Generates a 3D point cloud with extra information regarding instance and semantic segmentation.

That is a wrap on sensors and how do these retrieve simulation data.

Thus concludes the introduction to CARLA. However there is yet a lot to learn.

- **Gain some practise.** It may be a good idea to try some of the code recipes provided in this documentation. Combine them with the example scripts, test new ideas.

- **Continue learning.** There are some advanced features in CARLA: rendering options, traffic manager, the recorder, and some more. This is a great moment to learn on them.
- **Experiment freely.** Take a look at the **References** section of this documentation. It contains detailed information on the classes in the Python API, sensors, and much more.
- **Give your two cents.** Any doubts, suggestions and ideas are welcome in the forum.

4 Advanced steps

4.1 OpenDRIVE standalone mode

This feature allows users to ingest any OpenDRIVE file as a CARLA map out-of-the-box. In order to do so, the simulator will automatically generate a road mesh for actors to navigate through.

- **Overview**
- **Run a standalone map**
- **Mesh generation**

Overview

This mode runs a full simulation using only an OpenDRIVE file, without the need of any additional geometries or assets. To this end, the simulator takes an OpenDRIVE file and procedurally creates a temporal 3D mesh to run the simulation with.

The resulting mesh describes the road definition in a minimalistic manner. All the elements will correspond with the OpenDRIVE file, but besides that, there will be only void. In order to prevent vehicles from falling off the road, two measures have been taken.

- Lanes are a bit wider at junctions, where the flow of vehicles is most complex.
- Visible walls are created at the boundaries of the road, to act as a last safety measure.

Traffic lights, stops and yields will be generated on the fly. Pedestrians will navigate over the sidewalks and crosswalks that appear in the map. All of these elements, and every detail on the road, are based on the OpenDRIVE file. As the standalone mode uses the `.xodr` directly, any issues in it will propagate to the simulation. This could be an issue especially at junctions, where many lanes are mixed.



It is especially important to double check the OpenDRIVE file. Any issues in it will propagate when running the simulation.

Run a standalone map

Open an OpenDRIVE file is just a matter of calling `client.generate.opendrive_world()` through the API. This will generate the new map, and block the simulation until it is ready. The method needs for two parameters.

- **opendrive** is the content of the OpenDRIVE file parsed as a string.
- **parameters** is a `carla.OpendriveGenerationParameters` containing settings for the generation of the mesh. **This argument is optional.**
 - **vertex_distance** (*default 2.0 meters*) — Distance between the vertices of the mesh. The bigger, the distance, the more inaccurate the mesh will be. However, if the distance is too small, the resulting mesh will be too heavy to work with.
 - **max_road_length** (*default 50.0 meters*) — Maximum length of a portion of the mesh. The mesh is divided in portions to reduce rendering overhead. If a portion is not visible, UE will not render it. The smaller the portions, the more probably they are discarded. However, if the portions are too small, UE will have too many objects to manage, and performance will be affected too.
 - **wall_height** (*default 1.0 meters*) — Height of the additional walls created on the boundaries of the road. These prevent vehicles from falling to the void.
 - **additional_width** (*default 0.6 meters, 0.3 on each side*) — Small width increment applied to the junction lanes. This is a safety measure to prevent vehicles from falling.
 - **smooth_junctions** (*default True*) — If **True**, some information of the OpenDRIVE will be reinterpreted to smooth the final mesh at junctions. This is done to prevent some inaccuracies that may occur when various lanes meet. If set to **False**, the mesh will be generated exactly as described in the OpenDRIVE.

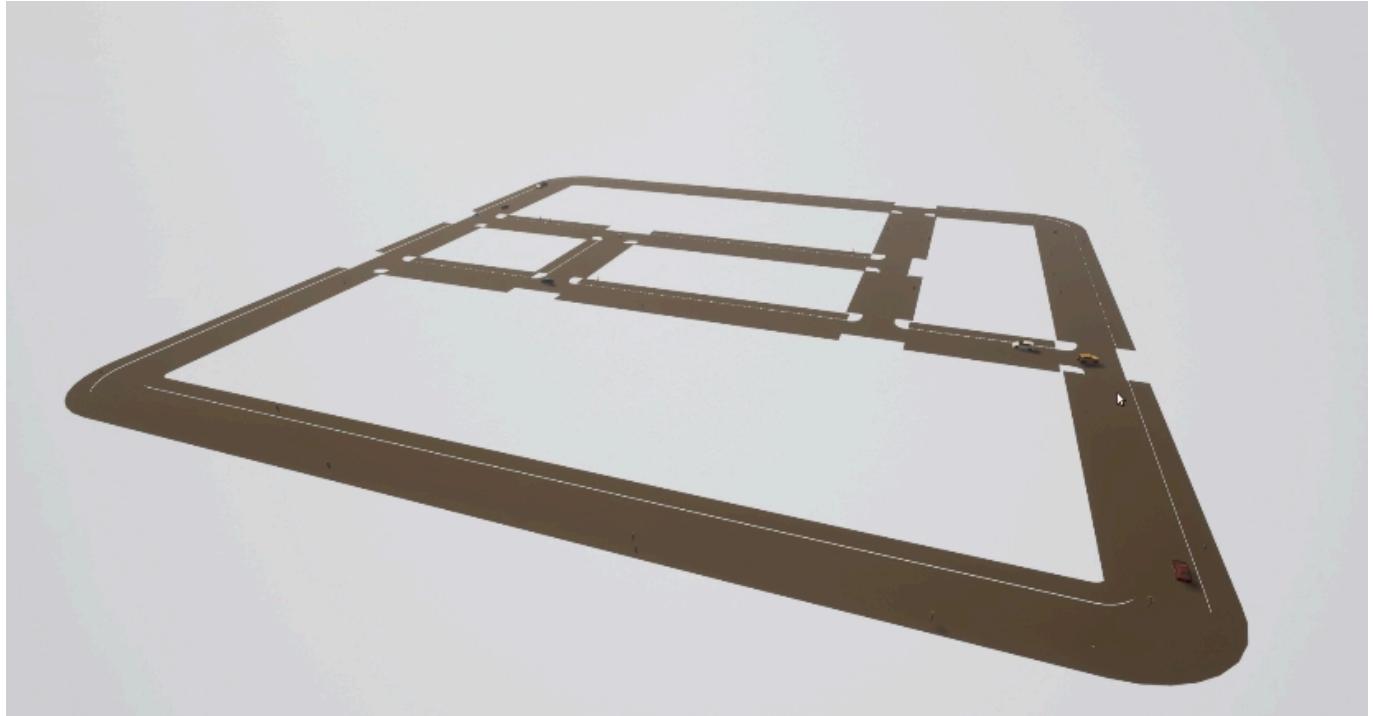


Figure 5: opendrive_standalone

- **enable_mesh_visibility** (*default True*) — If **False**, the mesh will not be rendered, which could save a lot of rendering work to the simulator.

In order to easily test this feature, the `config.py` script in `PythonAPI/util/` has a new argument, `-x` or `--xodr-path`. This argument expects a string with the path to the `.xodr` file, such as `path/example.xodr`. If the mesh is generated with this script, the parameters used will always be the default ones.

This feature can be tested with the new **TownBig** provided by CARLA.

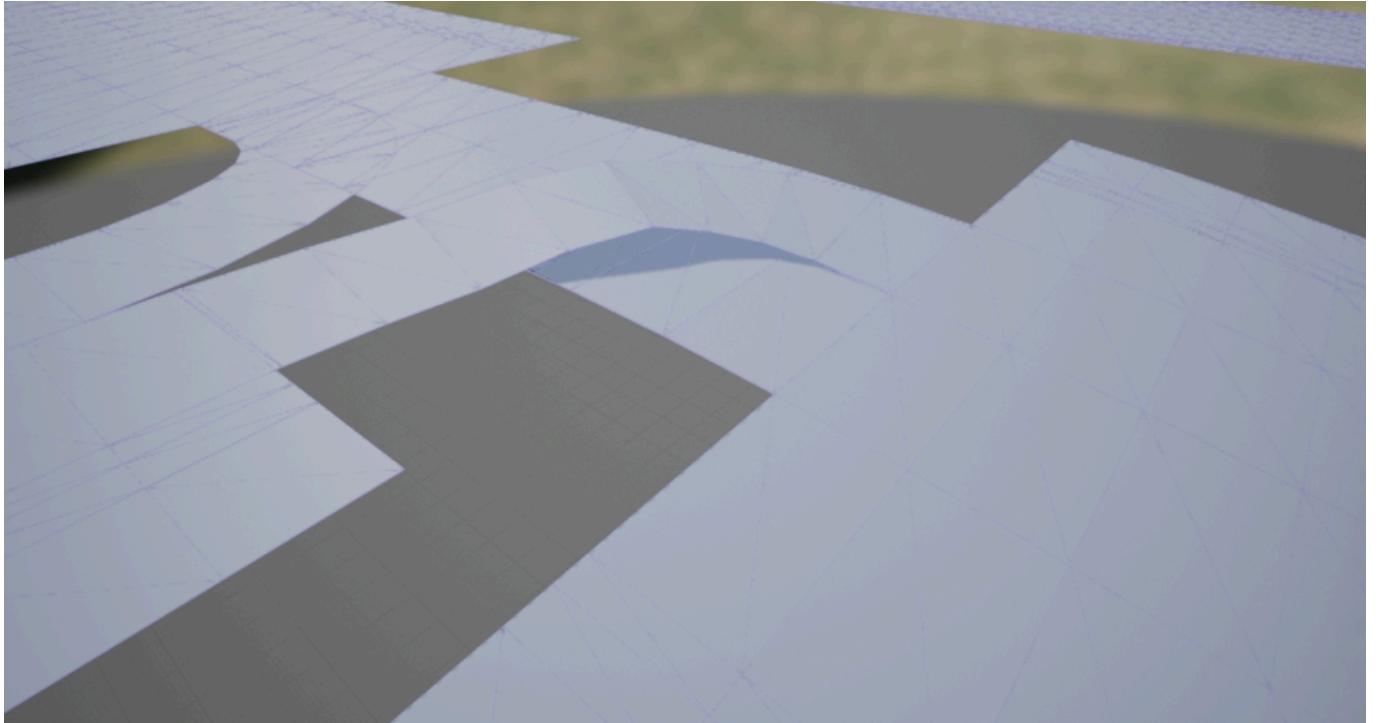
```
1 python3 config.py -x opendrive/TownBig.xodr
```



[`client.generate_opendrive_world()`] (`python api.md carla.Client.generate_opendrive_world`) uses content of the OpenDRIVE file parsed as string . On the contrary, ‘`config.py`‘ script needs the path to the ‘`.xodr`‘ file .

Mesh generation

The generation of the mesh is the key element of this mode. The feature can only be successful if the resulting mesh is smooth and fits its definition perfectly. For that reason, this step is constantly being improved. In the last iterations, junctions have been polished to avoid inaccuracies that occur, especially where uneven lanes joined.



When generating junction meshes, higher lanes tend to block the ones below them. The parameter `smooth_junctions` prevents this kind of issue.

Besides that, instead of creating the whole map as a unique mesh, different portions are created. By dividing the mesh, the simulator can avoid rendering portions that are not visible, and save up costs. Working smaller also allows to generate huge maps and contain issues that may occur on a small portion of the mesh.

Regarding the current state of the mesh generation, some considerations should be taken into account.

- **Junction smoothing.** The default smoothing prevents the issues at tilted junctions described above. However, it will modify the original mesh in the process. Set `smooth_junctions` to `False` to disable the smoothing if preferred.
- **Lateral slope.** This feature is currently not integrated in CARLA.
- **Sidewalk height.** This is currently hard-coded for all the sidewalks the same. Sidewalks must be higher than the road level for collisions to be detected, but RoadRunner does not export this value to the OpenDRIVE file. The height is hard-coded to guarantee collisions.

That covers all there is to know so far, regarding the OpenDRIVE standalone mode. Take the chance and use any OpenDRIVE map to test it in CARLA.

Doubts and suggestions in the forum.

4.2 PTV-Vissim co-simulation

CARLA has developed a co-simulation feature with PTV-Vissim. This allows to distribute the tasks at will, and exploit the capabilities of each simulation in favour of the user.

- **Requisites**
- **Run a co-simulation**
 - Create a new network

Requisites

In order to run the co-simulation, two things are necessary.

- Buy a license for **PTV-Vissim simulator**. It is necessary to acquire the Driving Simulator Interface add-on.
- In the PTV-Vissim installation folder, look for a `DrivingSimulatorProxy.dll`. Move it to `C:\Windows\System32`.

Run a co-simulation

Everything related with this feature can be found in `Co-Simulation/PTV-Vissim`. CARLA provides some examples that contain networks for `Town01`, and `Town03`.

To run a co-simulation, use the script `PTV-Vissim/run_synchronization.py`. This has one mandatory argument containing the PTV-Vissim network, and some other optional arguments.

- `vissim_network`— The vissim network file. This can be an example or a self-created PTV-Vissim network.
- `--carla-host (default: 127.0.0.1)`— IP of the carla host server.
- `--carla-port (default: 2000)` TCP port to listen to.
- `--vissim-version (default: 2020)`— PTV-Vissim version.
- `--step-length (default: 0.05s)`— set fixed delta seconds for the simulation time-step.
- `--simulator-vehicles (default: 1)`— number of vehicles that will be spawned in CARLA and passed to PTV-Vissim.

```
1 python3 run_synchronization.py examples/Town03/Town03.inpx
```

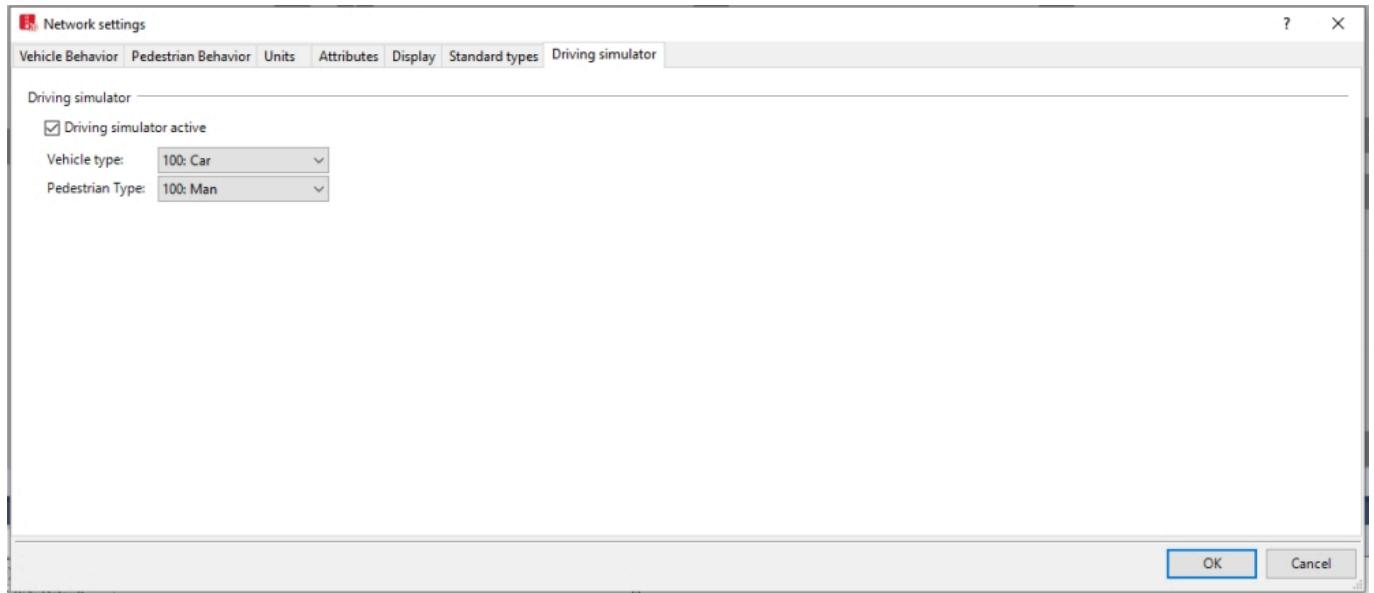


To stop the co-simulation, press ‘Ctrl+C’ in the terminal that run the script.

Both simulations will run in synchrony. The actions or events happening in one simulator will propagate to the other. So far, the feature only includes vehicle movement, and spawning. The spawning is limited due to PTV-Vissim types. * If a vehicle is spawned in CARLA, and the *Vehicle Type* in PTV-Vissim is set to `car`, it will spawn a car. No matter if it is a motorbike in CARLA. In the examples provided, the vehicle type is set to `car`. * If a vehicle is spawned in PTV-Vissim, CARLA will use a vehicle of the same type. The dimensions and characteristics will be similar, but not exactly the same.

Create a new network In order for a new PTV-Vissim network to run with CARLA, there are a few settings to be done.

- **Activate the driving simulator.** Go to `Base Data/Network settings/Driving simulator` and enable the option.
- **Specify the vehicle and pedestrian types.** These are the types that will be used in PTV-Vissim to sync with the spawnings done in CARLA. By default are empty.
- **Export the network as .inpx.** Create the network, export it, and run the co-simulation with `run_synchronization.py`.



Any vehicle that is spawned in CARLA, will be spawned in PTV-Vissim using these types.



PTV-Vissim will crash if the pedestrian and vehicle types are left empty.

That is all there is so far, regarding for the PTV-Vissim co-simulation with CARLA.

Open CARLA and mess around for a while. If there are any doubts, feel free to post these in the forum.

4.3 Recorder

This feature allows to record and reenact a previous simulation. All the events happened are registered in therecorder file. There are some high-level queries to trace and study those events.

- **Recording**
- **Simulation playback**
 - Setting a time factor
- **Recorded file**
- **Queries**
 - Collisions
 - Blocked actors
- **Sample Python scripts**

Recording

All the data is written in a binary file on the server side only. However, the recorder is managed using the carla.Client.

Actors are updated on every frame according to the data contained in the recorded file. Actors in the current simulation that appear in the recording will be either moved or re-spawned to emulate it. Those that do not appear in the recording will continue their way as if nothing happened.



By the end of the playback, vehicles will be set to autopilot, but pedestrians will stop .

The recorder file includes information regarding many different elements.

- **Actors** — creation and destruction, bounding and trigger boxes.
- **Traffic lights** — state changes and time settings.
- **Vehicles** — position and orientation, linear and angular velocity, light state, and physics control.
- **Pedestrians** — position and orientation, and linear and angular velocity.
- **Lights** — Light states from buildings, streets, and vehicles.

To start recording there is only need for a file name. Using \, / or : characters in the file name will define it as an absolute path. If no path is detailed, the file will be saved in `CarlaUE4/Saved`.

```
1 client.start_recorder("/home/carla/recording01.log")
```

By default, the recorder is set to store only the necessary information to play the simulation back. In order to save all the information previously mentioned, the argument `additional_data` has to be configured when starting the recording.

```
1 client.start_recorder("/home/carla/recording01.log", True)
```



Additional data includes: linear and angular velocity of vehicles and pedestrians, traffic light time settings, execution time, actors' trigger and bounding boxes, and physics controls for vehicles.

To stop the recording, the call is also straightforward.

```
1 client.stop_recorder()
```



As an estimate, 1h recording with 50 traffic lights and 100 vehicles takes around 200MB in size.

Simulation playback

A playback can be started at any point during a simulation. Besides the path to the log file, this method needs some parameters.

```
1 client.replay_file("recording01.log", start, duration, camera)
```

Parameter	
Description	
Notes	
start	Recording time in seconds to start the simulation at.
duration	If positive, time will be considered from the beginning of the recording. If negative, it will be considered from the end.
playback_time	Seconds to playback. 0 is all the recording.
camera	By the end of the playback, vehicles will be set to autopilot and pedestrians will stop.
target_id	ID of the actor that the camera will focus on.
Set it to 0 to let the spectator move freely.	

Setting a time factor The time factor will determine the playback speed. It can be changed any moment without stopping the playback.

```
1 client.set_replayer_time_factor(2.0)
```

Parameter

Default

Fast motion

Slow motion

time_factor

1.0

>1.0

<1.0



If ‘time factor>2.0‘, the actors’ position interpolation is disabled and just updated. Pedestrians’ animations are not affected by the time factor.

When the time factor is around **20x** traffic flow is easily appreciated.

Recorded file

The details of a recording can be retrieved using a simple API call. By default, it only retrieves those frames where an event was registered. Setting the parameter `show_all` would return all the information for every frame. The specifics on how the data is stored are detailed in therecorder’s reference.

```
1 # Show info for relevant frames
2 print(client.show_recorder_file_info("recording01.log"))
```

- **Opening information.** Map, date and time when the simulation was recorded.
- **Frame information.** Any event that could happen such as actors spawning or collisions. It contains the actors’ ID and some additional information.
- **Closing information.** Number of frames and total time recorded.

```
1 Version: 1
2 Map: Town05
3 Date: 02/21/19 10:46:20
4
5 Frame 1 at 0 seconds
```



Figure 6: flow

```

6 Create 2190: spectator (0) at (-260, -200, 382.001)
7 Create 2191: traffic.traffic_light (3) at (4255, 10020, 0)
8 Create 2192: traffic.traffic_light (3) at (4025, 7860, 0)
9 ...
10 Create 2258: traffic.speed_limit.90 (0) at (21651.7, -1347.59, 15)
11 Create 2259: traffic.speed_limit.90 (0) at (5357, 21457.1, 15)
12
13 Frame 2 at 0.0254253 seconds
14 Create 2276: vehicle.mini.cooperst (1) at (4347.63, -8409.51, 120)
15 number_of_wheels = 4
16 object_type =
17 color = 255,241,0
18 role_name = autopilot
19 ...
20 Frame 2350 at 60.2805 seconds
21 Destroy 2276
22
23 Frame 2351 at 60.3057 seconds
24 Destroy 2277
25 ...
26
27 Frames: 2354
28 Duration: 60.3753 seconds

```

Queries

Collisions Vehicles must have a collision detector attached to record collisions. These can be queried, using arguments to filter the type of the actors involved in the collisions. For example, `h` identifies actors whose `role_name` = `hero`, usually assigned to vehicles managed by the user. There is a specific set of actor types available for the query.

- `h` = Hero
- `v` = Vehicle
- `w` = Walker
- `t` = Traffic light
- `o` = Other
- `a` = Any



The ‘manual control.py’ script assigns ‘role name = hero’ for the ego vehicle.

The collision query requires two flags to filter the collisions. The following example would show collisions between vehicles, and any other object.

```
1 print(client.show_recorder_collisions("recording01.log", "v", "a"))
```

The output summarizes time of the collision, and type, ID and description of the actors involved.

```

1 Version: 1
2 Map: Town05
3 Date: 02/19/19 15:36:08
4
5     Time    Types      Id Actor 1                      Id Actor 2
6         16    v v      122 vehicle.yamaha.yzf          118 vehicle.dodge_charger.police
7         27    v o      122 vehicle.yamaha.yzf          0
8
9 Frames: 790
10 Duration: 46 seconds

```



| As it is the ‘hero’ or ‘ego’ vehicle who registers the collision, this will always be ‘Actor 1’.

The collision can be reenacted by using the recorder and setting it seconds before the event.

```
1 client.replay_file("col2.log", 13, 0, 122)
```

In this case, the playback showed this.



Figure 7: collision

Blocked actors Detects vehicles that were stuck during the recording. An actor is considered blocked if it does not move a minimum distance in a certain time. This definition is made by the user during the query.

```
1 print(client.show_recorder_actors_blocked("recording01.log", min_time, min_distance))
```

Parameter

Description

Default

min_time

Minimum seconds to move min_distance.

30secs.

min_distance

Minimum centimeters to move to not be considered blocked.

10cm.



| Sometimes vehicles are stopped at traffic lights for longer than expected.

The following example considers that vehicles are blocked when moving less than 1 meter during 60 seconds.

```
1 client.show_recorder_actors_blocked("col3.log", 60, 100)
```

The output is sorted by **duration**, which states how long it took to stop being “blocked” and move the **min_distance**.

```
1 Version: 1
2 Map: Town05
3 Date: 02/19/19 15:45:01
4
5     Time      Id Actor
6       36      173 vehicle.nissan.patrol
7       75      214 vehicle.chevrolet.impala
8      302      143 vehicle.bmw.grandtourer
9
10 Frames: 6985
11 Duration: 374 seconds
```

The vehicle 173 was stopped for 336 seconds at time 36 seconds. Reenact the simulation a few seconds before the second 36 to check it out.

```
1 client.replay_file("col3.log", 34, 0, 173)
```



Figure 8: accident

Sample python scripts

Some of the provided scripts in `PythonAPI/examples` facilitate the use of the recorder.

- `start_recording.py` starts the recording. The duration of the recording can be set, and actors can be spawned at the beginning of it.

Parameter

Description

-f

Filename.

-n (optional)

Vehicles to spawn. Default is 10.

-t (optional)

Duration of the recording.

- **start_replaying.py** starts the playback of a recording. Starting time, duration, and actor to follow can be set.

Parameter

Description

-f

Filename.

-s (optional)

Starting time. Default is 10.

-d (optional)

Duration. Default is all.

-c (optional)

ID of the actor to follow.

- **show_recorder_file_info.py** shows all the information in the recording file. By default, it only shows frames where an event is recorded. However, all of them can be shown.

Parameter

Description

-f

Filename.

-s (optional)

Flag to show all details.

- **show_recorder_collisions.py** shows recorded collisions between two flags of actors of types **A** and **B**. **-t = vv** would show all collisions between vehicles.

Parameter

Description

-f

Filename.

-t

Flags of the actors involved. h = hero v = vehicle w = walker t = traffic light o = other a = any

- **show_recorder_actors_blocked.py** lists vehicles considered blocked. Actors are considered blocked when not moving a minimum distance in a certain time.

Parameter

Description

-f

Filename.

-t (optional)

Time to move -d before being considered blocked.

-d (optional)

Distance to move to not be considered blocked.

Now it is time to experiment for a while. Use the recorder to playback a simulation, trace back events, make changes to see new outcomes. Feel free to say your word in the CARLA forum about this matter.

4.4 Rendering options

There are few details to take into account at the time of configuring a simulation. This page covers the more important ones.

- **Graphics quality**
 - Vulkan vs OpenGL
 - Quality levels
- **No-rendering mode**
- **Off-screen mode**
 - Off-screen Vs no-rendering
- **Running off-screen using a preferred GPU**
 - Docker - recommended approach
 - Deprecated - emulate the virtual display



Some of the command options below are not equivalent in the CARLA packaged releases. Read the [Command line options](#) section to learn more about this.

Graphics quality

Vulkan vs OpenGL Vulkan is the default graphics API used by Unreal Engine, and CARLA. It consumes more memory, but performs faster and makes for a better frame rate. However, it is quite experimental, especially in Linux, and it may lead to some issues.

There is the option to change to OpenGL. Use a flag when running CARLA.

```
1 cd carla && ./CarlaUE4.sh -opengl
```

When working with the build version of CARLA, Unreal Engine needs to be set to use OpenGL. [Here][UEdoc] is a documentation regarding different command line options for Unreal Engine. [UEdoc]: <https://docs.unrealengine.com/en-US/Programming/Basics/CommandLineArguments/index.html>

Quality levels CARLA also allows for two different graphic quality levels. **Epic**, the default is the most detailed. **Low** disables all post-processing and shadows, the drawing distance is set to 50m instead of infinite.

The simulation runs significantly faster in **Low** mode. This is not only used when there are technical limitations or precision is nonessential. It may be useful to train agents under conditions with simpler data or regarding only close elements.

The images below compare both modes. The flag used is the same for Windows and Linux. There is no equivalent option when working with the build, but the UE editor has its own quality settings. Go to [Settings/Engine Scalability Settings](#) for a greater customization of the desired quality.

Epic mode

```
./CarlaUE4.sh -quality-level=Epic
```

Epic mode screenshot

Low mode

```
./CarlaUE4.sh -quality-level=Low
```

Low mode screenshot



The issue that made Epic mode show an abnormal whiteness has been fixed. If the problem persists, delete ‘GameUserSettings.ini’. It is saving previous settings, and will be generated again in the next run. Ubuntu path: ‘./.config/Epic/CarlaUE4/Saved/Config/LinuxNoEditor/’ Windows path: ‘<Package folder>

WindowsNoEditor

CarlaUE4

Saved

Config

WindowsNoEditor

No-rendering mode

This mode disables rendering. Unreal Engine will skip everything regarding graphics. This mode prevents rendering overheads. It facilitates a lot traffic simulation and road behaviours at very high frequencies. To enable or disable no-rendering mode, change the world settings, or use the provided script in `/PythonAPI/util/config.py`.

Here is an example on how to enable and then disable it via script.

```
1 settings = world.get_settings()
2 settings.no_rendering_mode = True
3 world.apply_settings(settings)
4 ...
5 settings.no_rendering_mode = False
6 world.apply_settings(settings)
```

And here is an example on how to disable and then enable rendering using the `config.py`.

```
1 cd PythonAPI/util && python3 config.py --no-rendering
1 cd PythonAPI/util && python3 config.py --rendering
```

The script `PythonAPI/examples/no_rendering_mode.py` will enable no-rendering mode, and use **Pygame** to create an aerial view using simple graphics.

```
1 cd PythonAPI/examples && python3 no_rendering_mode.py
```



In no-rendering mode, cameras and GPU sensors will return empty data. The GPU is not used. Unreal Engine is not drawing any scene.

Off-screen mode

Unreal Engine needs for a screen in order to run. However, there is a workaround for remote servers with no display, or desktop users with a GPU not connected to any screen.

The simulator launches but there is no available window. It runs in the same way as normal mode. This mode tricks Unreal Engine into running in a “fake screen”.

Off-screen vs no-rendering It is important to understand the distinction between them to prevent misunderstandings.

- In **no-rendering**, Unreal Engine does not render anything. Graphics are not computed.
- In **off-screen**, Unreal Engine is working as usual, rendering is computed. Simply, there is no display available. GPU sensors return data when off-screen, and no-rendering mode can be enabled at will.

Setting off-screen mode This is **only possible in Linux while using OpenGL**. Unreal Engine crushes when Vulkan is running off-screen, and this issue is yet to be fixed by Epic.

To force the simulator run off-screen set the environment variable DISPLAY to empty and run CARLA using OpenGL.

```
1 # Linux  
2 DISPLAY= ./CarlaUE4.sh -opengl
```

Running off-screen using a preferred GPU

Docker - recommended approach The best way to run a headless CARLA and select the GPU is to run CARLA in a Docker.

This section contains an alternative tutorial, but this method is deprecated and performance is much worse. It is here only for those who Docker is not an option.

Deprecated - emulate the virtual display Show deprecated tutorial on how to emulate the virtual display



This tutorial is deprecated. To run headless CARLA, please run CARLA in a Docker .

- **Requirements:**

This tutorial only works in Linux and makes it possible for a remote server using several graphical cards to use CARLA on all GPUs. This is also translatable to a desktop user trying to use CARLA with a GPU that is not plugged to any screen. To achieve that, the steps can be summarized as:

1. Configure the server to have Nvidia working with no display.
2. Use VNC and VGL to simulate a display connected to any GPU.
3. Run CARLA.

This tutorial was tested in Ubuntu 16.04 using NVIDIA 384.11 drivers.

- **Latest Nvidia drivers**
- **OpenGL**: needed to use Virtual GL (VGL). OpenGL can be installed via apt:

```
1 sudo apt-get install freeglut3-dev mesa-utils
```

- **VGL**: redirects 3D rendering commands from Unix and Linux OpenGL to the hardware in a dedicated server.
- **TurboVNC 2.11**: graphical desktop-sharing system to connect remotely to the server.
- Extra packages : necessary to make Unreal work. sh sudo apt install x11-xserver-utils libxrandr-dev



Make sure that VNC version is compatible with Unreal. The one above worked properly during the making of this tutorial.

- **Configure the X**

Generate a X compatible with the Nvidia installed and able to run without display:

```
sudo nvidia-xconfig -a --use-display-device=None --virtual=1280x1024
```

- **Emulate the virtual display**

Run a Xorg. Here number 7 is used, but it could be labeled with any free number:

```
sudo nohup Xorg :7 &
```

Run an auxiliary remote VNC-Xserver. This will create a virtual display “8”:

```
/opt/TurboVNC/bin/vncserver :8
```

If everything is working fine the following command will run glxinfo on Xserver 7 selecting the GPU labeled as 0:

```
DISPLAY=:8 vglrun -d :7.0 glxinfo
```



To run on other GPU, change the ‘7.X’ pattern in the previous command. To set it to GPU 1: ‘DISPLAY=:8 vglrun -d :7.1 glxinfo’

- **Extra**

To disable the need of sudo when creating the `nohup Xorg` go to `/etc/X11/Xwrapper.config` and change `allowed_users=console` to `allowed_users=anybody`.

It may be needed to stop all `Xorg` servers before running `nohup Xorg`. The command for that could change depending on your system. Generally for Ubuntu 16.04 use:

```
sudo service lightdm stop
```

- **Running CARLA**

To run CARLA on a certain `<gpu_number>` in a certain `$CARLA_PATH` use the following command:

```
DISPLAY=:8 vglrun -d :7. $CARLA_PATH/CarlaUE4/Binaries/Linux/CarlaUE4
```



The ‘8’ and ‘7.X’ variables in the previous command depend on which were used while emulating the virtual display.

That is all there is to know about the different rendering options in CARLA.

Open CARLA and mess around for a while. If there are any doubts, feel free to post these in the forum.

4.5 RSS

CARLA integrates the C++ Library for Responsibility Sensitive Safety in the client library. This feature allows users to investigate behaviours of RSS without having to implement anything. CARLA will take care of providing the input, and applying the output to the AD systems on the fly.

- **Overview**

- **Compilation**

- Dependencies
- Build ***Current state**
- RssSensor
- RssRestrictor



This feature is a work in progress. Right now, it is only available for the Linux build.

Overview

The RSS library implements a mathematical model for safety assurance. It receives sensor information, and provides restrictions to the controllers of a vehicle. To sum up, the RSS module uses the sensor data to define **situations**. A situation describes the state of the ego vehicle with an element of the environment. For each situation, safety checks are made, and a proper response is calculated. The overall response is the result of all of the combined. For specific information on the library, read the documentation, especially the Background section.

This is implemented in CARLA using two elements.

- **RssSensor** is in charge of the situation analysis, and response generation using the *ad-rss-lib*.
- **RssRestrictor** applies the response by restricting the commands of the vehicle.

The following image sketches the integration of **RSS** into the CARLA architecture.

1. **The server.** - Sends a camera image to the client. (Only if the client needs visualization). - Provides the RssSensor with world data. - Sends a physics model of the vehicle to the RssRestrictor. (Only if the default values are overwritten).
2. **The client.** - Provides the *RssSensor* with some parameters to be considered. - Sends to the *RssRestrictor* an initial carla.VehicleControl.
3. **The RssSensor.** - Uses the *ad-rss-lib* to extract situations, do safety checks, and generate a response. - Sends the *RssRestrictor* a response containing the proper response and acceleration restrictions to be applied.
4. **The RssRestrictor** - If the client asks for it, applies the response to the carla.VehicleControl, and returns the resulting one.

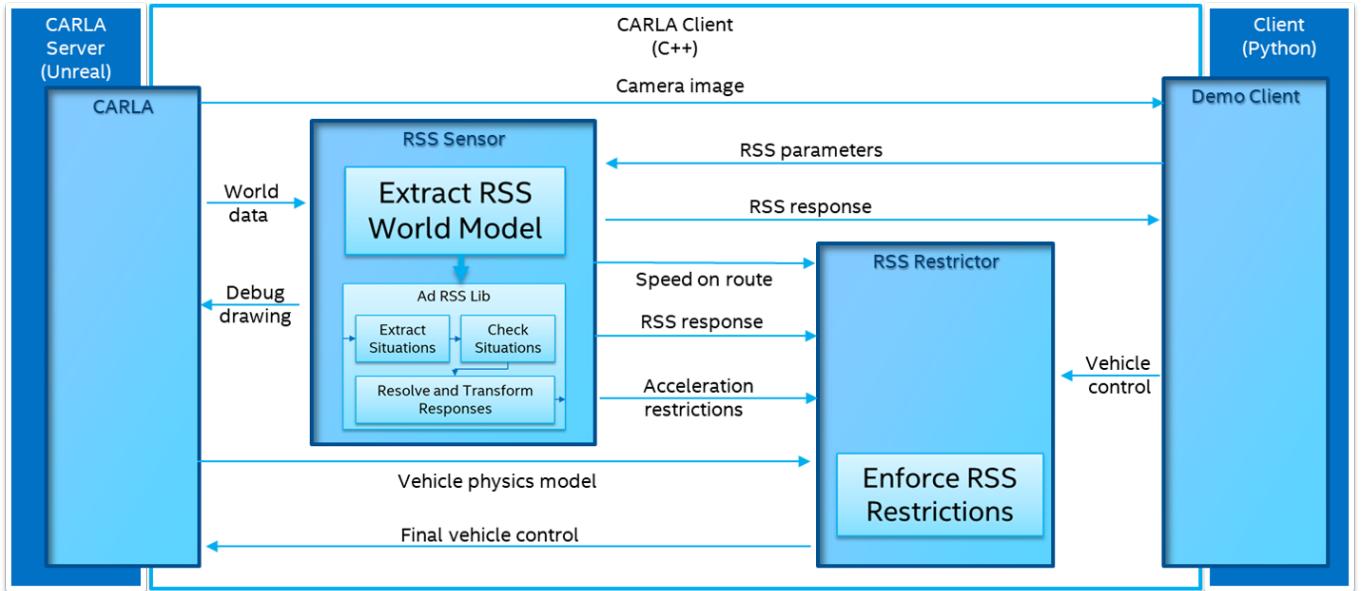
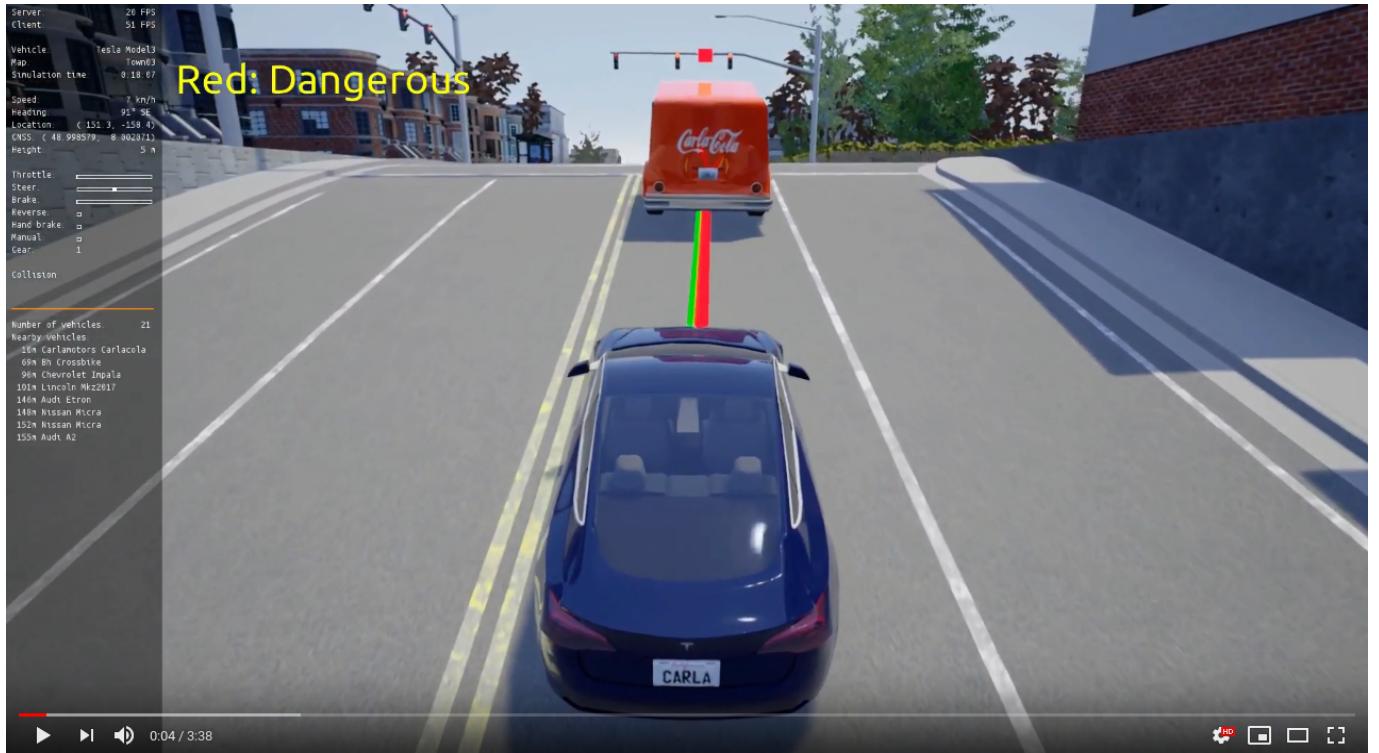


Figure 9: Interate RSS into CARLA



Visualization of the RssSensor results.

Compilation

The RSS integration has to be built aside from the rest of CARLA. The **ad-rss-lib** comes with an LGPL-2.1 open-source license that creates conflict. It has to be linked statically into *libCarla*.

As a reminder, the feature is only available for the Linux build so far.

Dependencies There are additional prerequisites required for building RSS and its dependencies. Take a look at the official documentation) to know more about this.

Dependencies provided by Ubuntu (≥ 16.04).

```
1 sudo apt-get install libgtest-dev libpython-dev libpugixml-dev libtbb-dev
```

The dependencies are built using colcon, so it has to be installed.

```
1 pip3 install --user -U colcon-common-extensions
```

There are some additional dependencies for the Python bindings.

```
1 sudo apt-get install castxml  
2 pip3 install pygccxml pyplusplus
```

Build Once this is done, the full set of dependencies and RSS components can be built.

- Compile LibCarla to work with RSS.

```
1 make LibCarla.client.rss
```

- Compile the PythonAPI to include the RSS feature.

```
1 make PythonAPI.rss
```

- As an alternative, a package can be built directly.

```
1 make package.rss
```

Current state

RssSensor `carla.RssSensor` supports ad-rss-lib v4.0.0 feature set completely, including intersections, stay on road support and unstructured scenes (e.g. with pedestrians).

So far, the server provides the sensor with ground truth data of the surroundings that includes the state of other traffic participants and traffic lights.

RssRestrictor When the client calls for it, the `carla.RssRestrictor` will modify the vehicle controller to best reach the desired accelerations or decelerations by a given response.

Due to the structure of `carla.VehicleControl` objects, the restrictions applied have certain limitations. These controllers include `throttle`, `brake` and `steering` values. However, due to car physics and the simple control options these might not be met. The restriction intervenes in lateral direction simply by counter steering towards the parallel lane direction. The brake will be activated if deceleration requested by RSS. This depends on vehicle mass and brake torques provided by the `carla.Vehicle`.



In an automated vehicle controller it might be possible to adapt the planned trajectory to the restrictions. A fast control loop (>1KHz) can be used to ensure these are followed.

That sets the basics regarding the RSS sensor in CARLA. Find out more about the specific attributes and parameters in the sensor reference.

Open CARLA and mess around for a while. If there are any doubts, feel free to post these in the forum.

4.6 SUMO co-simulation

CARLA has developed a co-simulation feature with SUMO. This allows to distribute the tasks at will, and exploit the capabilities of each simulation in favour of the user.

- **Requisites**
- **Run a custom co-simulation**
 - Create CARLA vtypes
 - Create the SUMO net
 - Run the synchronization
- **Spawn NPCs controlled by SUMO**

Requisites

First and foremost, it is necessary to **install SUMO** to run the co-simulation. Building from source is recommended over a simple installation, as there are new features and fixes that will improve the co-simulation.

Once that is done, set the SUMO environment variable.

```
1 echo "export SUMO_HOME=/usr/share/sumo" >> ~/.bashrc && source ~/.bashrc
```

SUMO is ready to run the co-simulations. There are some examples in `Co-Simulation/Sumo/examples` for **Town01**, **Town04**, and **Town05**. These `.sumocfg` files describe the configuration of the simulation (e.g., net, routes, vehicle types...). Use one of these to test the co-simulation. The script has different options that are detailed below. For the time being, let's run a simple example for **Town04**.

Run a CARLA simulation with **Town04**.

```
1 cd ~/carla
2 ./CarlaUE4.sh
3 cd PythonAPI/util
4 python3 config.py --map Town04
```

Then, run the SUMO co-simulation example. `sh cd ~/carla/Co-Simulation/Sumo python3 run synchronization.py examples/Town04.sumocfg --sumo-gui`



Run a custom co-simulation

Create carla vtypes With the script `Co-Simulation/Sumo/util/create_sumo_vtypes.py` the user can create sumo *vtypes*, the equivalent to CARLA blueprints, based on the CARLA blueprint library.

- `--carla-host` (*default: 127.0.0.1*) — IP of the carla host server.
- `--carla-port` (*default: 2000*) — TCP port to listen to.
- `--output-file` (*default: carlavtypes.rou.xml*) — The generated file containing the *vtypes*.

This script uses the information stored in `data/vtypes.json` to create the SUMO *vtypes*. These can be modified by editing said file.



A CARLA simulation must be running to execute the script.

Create the SUMO net The recommended way to create a SUMO net that synchronizes with CARLA is using the script `Co-Simulation/Sumo/util/netconvert_carla.py`. This will draw on the netconvert tool provided by SUMO. In order to run the script, some arguments are needed.

- `xodr_file` — OpenDRIVE file `.xodr`.
- `--output' (default:net.net.xml)` — output file `.net.xml`.
- `--guess-tls (default:false)` — SUMO can set traffic lights only for specific lanes in a road, but CARLA can't. If set to `True`, SUMO will not differentiate traffic lights for specific lanes, and these will be in sync with CARLA.

The output of the script will be a `.net.xml` that can be edited using **NETEDIT**. Use it to edit the routes, add demand, and eventually, prepare a simulation that can be saved as `.sumocfg`.

The examples provided may be helpful during this process. Take a look at `Co-Simulation/Sumo/examples`. For every `example.sumocfg` there are several related files under the same name. All of them comprise a co-simulation example.

Run the synchronization Once a simulation is ready and saved as a `.sumocfg`, it is ready to run. There are some optional parameters to change the settings of the co-simulation.

- `sumo_cfg_file` — The SUMO configuration file.
- `--carla-host (default: 127.0.0.1)` — IP of the carla host server
- `--carla-port (default: 2000)` — TCP port to listen to
- `--sumo-host (default: 127.0.0.1)` — IP of the SUMO host server.

- `--sumo-port` (*default: 8813*) — TCP port to listen to.
- `--sumo-gui` — Open a window to visualize the gui version of SUMO.
- `--step-length` (*default: 0.05s*) — Set fixed delta seconds for the simulation time-step.
- `--sync-vehicle-lights` (*default: False*) — Synchronize vehicle lights.
- `--sync-vehicle-color` (*default: False*) — Synchronize vehicle color.
- `--sync-vehicle-all` (*default: False*) — Synchronize all vehicle properties.
- `--tls-manager` (*default: none*) — Choose which simulator should manage the traffic lights. The other will update those accordingly. The options are `carla`, `sumo`, and `none`. If `none` is chosen, traffic lights will not be synchronized. Each vehicle would only obey the traffic lights in the simulator that spawn it.

```
1 python3 run_synchronization.py <SUMOCFG FILE> --tls-manager carla --sumo-gui
```



To stop the co-simulation, press ‘Ctrl+C’ in the terminal that run the script.

Spawn NPCs controlled by SUMO

The co-simulation with SUMO makes for an additional feature. Vehicles can be spawned in CARLA through SUMO, and managed by the later as the Traffic Manager would do.

The script `spawn_npc_sumo.py` is almost equivalent to the already-known `spawn_npc.py`. This script automatically generates a SUMO network in a temporal folder, based on the active town in CARLA. The script will create random routes and let the vehicles roam around.

As the script runs a synchronous simulation, and spawns vehicles in it, the arguments are the same that appear in `run_synchronization.py` and `spawn_npc.py`.

- `--host` (*default: 127.0.0.1*) — IP of the host server.
- `--port` (*default: 2000*) — TCP port to listen to.
- `-n,--number-of-vehicles` (*default: 10*) — Number of vehicles spawned.
- `--safe` — Avoid spawning vehicles prone to accidents.
- `--filterv` (*default: "vehicle."*)* — Filter the blueprint of the vehicles spawned.
- `--sumo-gui` — Open a window to visualize SUMO.
- `--step-length` (*default: 0.05s*) — Set fixed delta seconds for the simulation time-step.
- `--sync-vehicle-lights` (*default: False*) — Synchronize vehicle lights state.
- `--sync-vehicle-color` (*default: False*) — Synchronize vehicle color.
- `--sync-vehicle-all` (*default: False*) — Synchronize all vehicle properties.
- `--tls-manager` (*default: none*) — Choose which simulator will change the traffic lights’ state. The other will update them accordingly. If `none`, traffic lights will not be synchronized.

```
1 # Spawn 10 vehicles, that will be managed by SUMO instead of Traffic Manager.
```

```
2 # CARLA in charge of traffic lights.
```

```
3 # Open a window for SUMO visualization.
```

```
4 python3 spawn_sumo_npc.py -n 10 --tls-manager carla --sumo-gui
```

That is all there is so far, regarding for the SUMO co-simulation with CARLA.

Open CARLA and mess around for a while. If there are any doubts, feel free to post these in the forum.

4.7 Synchrony and time-step

This section deals with two fundamental concepts in CARLA. Their configuration defines how does time go by in the simulation, and how does the server make the simulation move forward.

- **Simulation time-step**
 - Variable time-step
 - Fixed time-step
 - Tips when recording the simulation
 - Time-step limitations
- **Client-server synchrony**
 - Setting synchronous mode
 - Using synchronous mode
- **Possible configurations**

Simulation time-step

There is a difference between real time, and simulation time. The simulated world has its own clock and time, conducted by the server. Computing two simulation steps takes some real time. However, there is also the time span that went by between those two simulation moments, the time-step.

To clarify, the server can take a few milliseconds to compute two steps of a simulation. However, the time-step between those two simulation moments can be configured to be, for instance, always a second.

Time-step can be fixed or variable depending on user preferences.



Time-step and synchrony are intertwined concepts. Make sure to read both sections to get a full understanding of how does CARLA work.

Variable time-step The default mode in CARLA. The simulation time that goes by between steps will be the time that the server takes to compute these.

```
1 settings = world.get_settings()
2 settings.fixed_delta_seconds = None # Set a variable time-step
3 world.apply_settings(settings)
```

PythonAPI/util/config.py sets the time-step using an argument. Zero equals variable time-step.

```
1 cd PythonAPI/util && python3 config.py --delta-seconds 0
```

Fixed time-step The elapsed time remains constant between steps. If it is set to 0.5 seconds, there will be two frames per simulated second. Using the same time increment on each step is the best way to gather data from the simulation. Physics and sensor data will correspond to an easy to comprehend moment of the simulation. Also, if the server is fast enough, it makes possible to simulate longer time periods in less real time.

Fixed delta seconds can be set in the world settings. To run the simulation at a fixed time-step of 0.05 seconds apply the following settings. In this case, the simulator will take twenty steps ($1/0.05$) to recreate one second of the simulated world.

```
1 settings = world.get_settings()
2 settings.fixed_delta_seconds = 0.05
3 world.apply_settings(settings)
```

This can also be set using the provided script PythonAPI/util/config.py.

```
1 cd PythonAPI/util && python3 config.py --delta-seconds 0.05
```

Tips when recording the simulation CARLA has a recorder feature that allows a simulation to be recorded and then reenacted. However, when looking for precision, some things need to be taken into account.

- With a **fixed time-step**, reenacting it will be easy. The server can be set to the same time-step used in the original simulation.
- With a **variable time-step**, things are a bit more complicated.
 - If the **server runs with a variable time-step**, the time-steps will be different from the original one, as logic cycles differ from time to time. The information will then be interpolated using the recorded data.
 - If the **server is forced to reproduce the exact same time-steps**, the steps simulated will be the same, but the real time between them changes. Time-steps should be passed one by one. Those original time-steps were the result of the original simulation running as fast as possible. As the time taken to represent these will mostly be different, the simulation is bound to be reproduced with weird time fluctuations.
 - There is also a **float-point arithmetic error** that variable time-step introduces. The simulation is running with a time-step equal to the real one. Real time is a continuous variable, represented in the simulation with a **float** value, which has decimal limitations. The time that is cropped for each step accumulates, and prevents the simulation from a precise repetition of what has happened.

Time-step limitations Physics must be computed within very low time steps to be precise. The more time goes by, the more variables and chaos come to place, and the more defective the simulation will be. CARLA uses up to 6 substeps to compute physics in every step, each with a maximum delta time of 0.016667s.

To know how many of these are needed, the time-step used gets divided by the maximum delta time a substep can use `number_of_substeps = time_step/0.016667`. Being these a maximum of 6, $6*0.016667 = 0.1$. If the time-step is greater than 0.1, there will not be enough physical substeps. Physics will not be in synchrony with the delta time.



Do not use a time-step greater than 0.1s. As explained above, the physics will not be representative for the simulation. The original issue can be found in ref. [695](<https://github.com/carla-simulator/carla/issues/695>)

Client-server synchrony

CARLA is built over a client-server architecture. The server runs the simulation. The client retrieves information, and demands for changes in the world. This section deals with communication between client and server.

By default, CARLA runs in **asynchronous mode**. The server runs the simulation as fast as possible, without waiting for the client. On **synchronous mode**, the server waits for a client tick, a “ready to go” message, before updating to the following simulation step.



In a multiclient architecture, only one client should tick. The server reacts to every tick received as if it came from the same client. Many client ticks will make the create inconsistencies between server and clients.

Setting synchronous mode Changing between synchronous and asynchronous mode is just a matter of a boolean state.

```
1 settings = world.get_settings()
2 settings.synchronous_mode = True # Enables synchronous mode
3 world.apply_settings(settings)
```

To disable synchronous mode just set the variable to false or use the script `PythonAPI/util/config.py`.

```
1 cd PythonAPI/util && python3 config.py --no-sync # Disables synchronous mode
```

Synchronous mode cannot be enabled using the script, only disabled. Enabling the synchronous mode makes the server wait for a client tick. Using this script, the user cannot send ticks when desired.

Using synchronous mode The synchronous mode becomes specially relevant with slow client applications, and when synchrony between different elements, such as sensors, is needed. If the client is too slow and the server does not wait, there will be an overflow of information. The client will not be able to manage everything, and it will be lost or mixed. On a similar tune, with many sensors and asynchrony, it would be impossible to know if all the sensors are using data from the same moment in the simulation.

The following fragment of code extends the previous one. The client creates a camera sensor, stores the image data of the current step in a queue, and ticks the server after retrieving it from the queue. A more complex example regarding several sensors can be found here.

```
1 settings = world.get_settings()
2 settings.synchronous_mode = True
3 world.apply_settings(settings)
4
5 camera = world.spawn_actor(blueprint, transform)
6 image_queue = queue.Queue()
7 camera.listen(image_queue.put)
8
9 while True:
10     world.tick()
11     image = image_queue.get()
```



Data coming from GPU-based sensors, mostly cameras, is usually generated with a delay of a couple of frames. Synchrony is essential here.

The world has asynchrony methods to make the client wait for a server tick, or do something when it is received.

```
1 # Wait for the next tick and retrieve the snapshot of the tick.  
2 world_snapshot = world.wait_for_tick()  
3  
4 # Register a callback to get called every time we receive a new snapshot.  
5 world.on_tick(lambda world_snapshot: do_something(world_snapshot))
```

Possible configurations

The configuration of time-step and synchrony, leads for different settings. Here is a brief summary on the possibilities.

Fixed time-step

Variable time-step

Synchronous mode

Client is in total control over the simulation and its information.

Risk of non reliable simulations.

Asynchronous mode

Good time references for information. Server runs as fast as possible.

Non easily repeatable simulations.

- **Synchronous mode + variable time-step.** This is almost for sure a non-desirable state. Physics cannot run properly when the time-step is bigger than 0.1s and. If the server has to wait for the client to compute the steps, this is likely to happen. Simulation time and physics will not be in synchrony. The simulation will not be reliable.
- **Asynchronous mode + variable time-step.** This is the default CARLA state. Client and server are asynchronous. The simulation time flows according to the real time. Reenacting the simulation needs to take into account float-arithmetic error, and possible differences in time steps between servers.
- **Asynchronous mode + fixed time-step.** The server will run as fast as possible. The information retrieved will be easily related with an exact moment in the simulation. This configuration makes possible to simulate long periods of time in much less real time, if the server is fast enough.
- **Synchronous mode + fixed time-step.** The client will rule the simulation. The time step will be fixed. The server will not compute the following step until the client sends a tick. This is the best mode when synchrony and precision is relevant. Especially when dealing with slow clients or different elements retrieving information.



In synchronous mode, always use a fixed time-step . If the server has to wait for the user, and it is using a variable time-step, time-steps will be too big. Physics will not be reliable. This issue is better explained in the time-step limitations section.

That is all there is to know about the roles of simulation time and client-server synchrony in CARLA.

Open CARLA and mess around for a while. Any suggestions or doubts are welcome in the forum.

4.8 Traffic Manager

- What is it?
- Architecture
 - ALSM
 - Command array
 - Control loop
 - In-Memory Map
 - PBVT
 - PID controller

- Simulation state
- Stages
- Vehicle registry
- **Using the Traffic Manager**
 - General considerations
 - Creating a Traffic Manager
 - Setting a Traffic Manager
 - Stopping a Traffic Manager
- **Hybrid physics mode**
- **Running multiple Traffic Managers**
 - Definitions
 - Multiclient
 - MultiTM
 - Multisimulation
- **Other considerations**
 - FPS limitations
 - Synchronous mode
- **Summary**

What is it?

The Traffic Manager, TM for short, is the module in charge of controlling vehicles inside the simulation. It is built on top of the CARLA API in C++. Its goal is to populate the simulation with realistic urban traffic conditions. Users can customize some behaviours, for example to set specific learning circumstances. Every TM controls vehicles registered to it by setting autopilot to true, and is accounting for the rest by considering them unregistered.

Structured design The TM is built on the client-side of the CARLA architecture. It replaces the server-side autopilot. The execution flow is divided in **stages** with independent operations and goals. This facilitates the development of phase-related functionalities and data structures, while also improving computational efficiency. Each stage runs on a different thread. Communication with the rest is managed through synchronous messaging between the stages. The information flows only in one direction.

User customization Users must have some control over the traffic flow by setting parameters that allow, force or encourage specific behaviours. Users can change the traffic behaviour as they prefer, both online and offline. For example they could allow a car to ignore the speed limits or force a lane change. Being able to play around with behaviours is a must when trying to simulate reality. It is necessary to train driving systems under specific and atypical circumstances.

Architecture

The previous diagram is a summary of the internal architecture of the Traffic Manager. The inner structure of the TM can be easily translated to code, and each relevant component has its equivalent in the C++ code (.cpp files) inside `LibCarla/source/carla/trafficmanager`. The functions and relations of these components are explained in the following sections.

Nevertheless, the logic of it can be simplified as follows.

1. Store and update the current state of the simulation. First of all, theALSM (Agent Lifecycle & State Management) scans the world to keep track of all the vehicles and walkers present in it, and clean up entries for those that no longer exist. All the data is retrieved from the server, and then passed to thestages. In such way, calls to the server are isolated in the ALSM, and these information can be easily accessible onwards. Thevehicle registry contains an array with the registered vehicles, and a list with the rest of vehicles and pedestrians. Thesimulation state stores in cache the position and velocity and some additional information of all the cars and walkers.

2. Calculate the movement of every registered vehicle. The main goal of the TM is to generate viable commands for all the vehicles in thevehicle registry, according to thesimulation state. The calculations for each vehicle are done separately. These calculations are divided in differentstages. Thecontrol loop makes sure that all the calculations are consistent by creating **synchronization barriers** in between stages. No one moves to the following stage before the calculations for all the vehicles are finished in the current one. Each vehicle has to go through the following stages.

2.1 -Localization Stage. TM vehicles do not have a predefined route, and path choices are taken randomly at junctions. Having this in mind, theIn-Memory Map simplifies the map as a grid of waypoints, and a near-future path to follow is created as a list of waypoints ahead. The path of every vehicle will be stored by

thePBVT component (Path Buffers & Vehicle Tracking), so that these can be easily accessible and modified in future stages.

2.2 -Collision Stage. During this stage, bounding boxes are extended over the path of each vehicle to identify potential collision hazards, which are then managed when necessary.

2.3 -Traffic Light Stage Similar to the Collision Stage, this stage identifies potential hazards that affect the path of the vehicle according to traffic light influence, stop signs, and junction priority.

2.4 -Motion Planner Stage. Once a path has been defined, this stage computes vehicle movement. APID controller is used to determine how to reach the target values. This movement is then translated into an actual CARLA command to be applied.

3. Apply the commands in the simulation. Finally, the TM has calculated the next command for every vehicle, and now it is only a matter of applying them. All the commands are gathered by thecommand array, and sent to the CARLA server in a batch so that they are applied in the same frame.

And thus the cycle is concluded. The TM follows this logic on every step of the simulation. For a better understanding of the role that each component plays, this section includes an in-depth description of each of them.

ALSM ALSM stands for **Agent Lifecycle and State Management**. First step in the logic cycle. Provides context over the current state of the simulation.

- Scans the world to keep track of all the vehicles and walkers in it, their positions and velocities. If physics are enabled, the velocity is retrieved byVehicle.get_velocity(). Instead, if physics are disabled, the velocity is computed using the history of position updates over time.
- Stores the position, velocity and additional information (traffic light influence, bounding boxes, etc) of every vehicle and walker in thesimulation state module.
- Updates the list of registered vehicles stored by thevehicle registry.
- Updates entries in thecontrol loop andPBVT modules to match the list of registered vehicles.

Related .cpp files: `ALSM.h`, `ALSM.cpp`.

Command array Last step in the TM logic cycle. Receives commands for all the registered vehicles and applies them.

- Receives a series ofcarla.VehicleControl from theMotion Planner Stage.
- Constructs a batch for all the commands to be applied during the same frame.
- Sends the batch to the CARLA server. Either `apply_batch()` or `apply_batch_sync()` incarla.Client will be called, depending if the simulation is running in asynchronous or synchronous mode, respectively.

Related .cpp files: `TrafficManagerLocal.cpp`.

Control loop Manages the process of calculating the next command for all the registered vehicles, so that these are done in synchrony.

- Receives from thevehicle registry an array of the vehicles registered to the TM.
- Loops over said array, performing calculations per vehicle separately.
- These calculations are divided in a series ofstages.
- Synchronization barriers are placed between stages so that consistency is guaranteed. Calculations for all vehicles must finish before any of them moves to the next stage, ensuring that all vehicles are updated in the same frame.
- Coordinates the transition betweenstages so that all the calculations are done in sync.
- When the last stage Motion Planner Stage) finishes, thecommand array is sent to the server. In this way, there are no frame delays between the calculations of the control loop, and the commands being applied.

Related .cpp files: `TrafficManagerLocal.cpp`.

In-Memory Map Helper module contained by thePBVT and used during theLocalization Stage.

- Discretizes the map into a grid of waypoints.
- Includes waypoints in a specific data structure with more information to connect waypoints and identify roads, junctions...
- Identifies these structures with an ID that is used to quickly spot vehicles in nearby areas.

Related .cpp files: `InMemoryMap.cpp` and `SimpleWaypoint.cpp`.

PBVT PBVT stands for **Path Buffer and Vehicle Tracking**. This data structure contains the expected path for every vehicle so that it can be easily accessible during thecontrol loop.

- Contains a map of deque objects with an entry per vehicle.

- For each vehicle, contains a set of waypoints describing its current location and near-future path.
- Contains theIn-Memory Map that will be used by theLocalization Stage to relate every vehicle to the nearest waypoint, and possible overlapping paths.

PID controller Helper module that performs calculations during theMotion Planner Stage.

- Using the information gathered by theMotion Planner Stage, estimates the throttle, brake and steering input needed to reach a target value.
- The adjustment is made depending on the specific parameterization of the controller, which can be modified if desired. Read more about PID controllers to learn how to do it.

Related .cpp files: PIDController.cpp.

Simulation state Stores information about the vehicles in the world so that it can be easily accessible during all the process.

- Receives the current state of all vehicles and walkers in the world from theALSM, including their position, velocity and some additional information (such as traffic light influence and state). It also stores some additional information such as whereas these vehicles are under the influence of a traffic light and what is the current state of said traffic light.
- Stores in cache all the information so that no additional calls to the server are needed during thecontrol loop.

Related .cpp files: SimulationState.cpp, SimulationState.h.

Stages Stage 1- Localization Stage

First stage in thecontrol loop. Defines a near-future path for registered vehicles.

- Obtains the position and velocity of all the vehicles fromsimulation state.
- Using theIn-Memory Map, relates every vehicle with a list of waypoints that describes its current location and near-future path, according to its trajectory. The faster the vehicle goes, the larger said list will be.
- The path is updated according to planning decisions such as lan changes, speed limit, distance to leading vehicle parameterization, etc.
- ThePBVT module stores the path for all the vehicles.
- These paths are compared with each other, in order to estimate possible collision situations. Results are passed to the following stage:Collision stage.

Related .cpp files: LocalizationStage.cpp and LocalizationUtils.cpp.

Stage 2- Collision Stage

Second stage in thecontrol loop. Triggers collision hazards.

- Receives a list of pairs of vehicles with possible overlapping paths from theLocalization Stage.
- For every pair, extends bounding boxes along the path ahead (geodesic boundaries), to check if they actually overlap and the risk of collision is real.
- Hazards for all the possible collisions will be sent to theMotion Planner Stage to modify the path accordingly.

Related .cpp files: CollisionStage.cpp.

Stage 3- Traffic Light Stage

Third stage in thecontrol loop. Triggers hazards to follow traffic regulations such as traffic lights, stop signs, and priority at junctions.

- If the vehicle is under the influence of a yellow or red traffic light, or a stop sign, sets a traffic hazard.
- If the vehicle is in a non-signalized junction, a bounding box is extended along its path. Vehicles with overlapping paths follow a FIFO order to move, and waits are set to a fixed time.

Related .cpp files: TrafficLightStage.cpp.

Stage 4- Motion Planner Stage

Fourth and last stage in thecontrol loop. Generates the CARLA command that will be applied to the vehicle.

- Gathers all the information so far: position and velocity of the vehicles simulation state), their path PBVT, hazards Collision Stage andTraffic Light State).
- Makes high-level decisins about how should the vehicle move, for example computing the brake needed to prevent a collision hazard. APID controller is used to estimate behaviors according to target values.

- Translates the desired movement to acarla.VehicleControl that can be applied to the vehicle.
- Sends the resulting CARLA commands to the command array.

Related .cpp files: MotionPlannerStage.cpp.

Vehicle registry Keeps track of all the vehicles and walkers in the simulation.

- The ALSM scans the world and passes an updated list of walkers and vehicles.
- Vehicles registered to the TM are stored in a separated array that will be iterated on during the control loop.

Related .cpp files: MotionPlannerStage.cpp.

Using the Traffic Manager

General considerations First of all there are some general behaviour patterns the TM will generate that should be understood beforehand. These statements are inherent to the way the TM is implemented:

- **Vehicles are not goal-oriented** they follow a trajectory and whenever approaching a junction, choose a path randomly. Their path is endless, and will never stop roaming around the city.
- **Vehicles' target speed is 70% their current speed limit:** unless any other value is set.
- **Junction priority does not follow traffic regulations:** the TM has a priority system to be used while junction complexity is solved. This may cause some issues such as a vehicle inside a roundabout yielding to a vehicle trying to get in.

The TM provides a set of possibilities so the user can establish specific behaviours. All the methods accessible from the Python API are listed in the documentation. However, here is a brief summary of what the current possibilities are.

General:

1. Use a carla.Client to create a TM instance connected to a port.
2. Retrieve the port where a TM is connected.

Safety conditions:

1. Set a minimum distance between stopped vehicles (for a vehicle or all of them). This will affect the minimum moving distance.
2. Set an intended speed regarding current speed limitation (for a vehicle or all of them).
3. Reset traffic lights.

Collision managing:

1. Enable/Disable collisions between a vehicle and a specific actor.
2. Make a vehicle ignore all the other vehicles.
3. Make a vehicle ignore all the walkers.
4. Make a vehicle ignore all the traffic lights.

Lane changes:

1. Force a lane change disregarding possible collisions.
2. Enable/Disable lane changes for a vehicle.

Hybrid physics mode:

1. Enable/Disable the hybrid physics mode.
2. Change the radius where physics are enabled.

Creating a Traffic Manager A TM instance can be created by any carla.Client specifying the port that will be used. The default port is 8000.

```
1 tm = client.get_trafficmanager(port)
```

Now the TM needs some vehicles to be in charge of. In order to do so, enable the autopilot mode for the set of vehicles to be managed. Retrieve the port of the TM object that has been created. If no port is provided, it will try to connect to a TM in the default port, 8000. If the TM does not exist, it will create it.

```
1 tm_port = tm.get_port()
2 for v in vehicles_list:
3     v.set_autopilot(True,tm_port)
```



In multiclient situations, creating or connecting to a TM is not that straightforward. Take a look into the other considerations section to learn more about this.

The script `spawn_npc.py` in `/PythonAPI/examples` creates a TM instance in the port passed as argument and registers every vehicle spawned to it by setting the autopilot to True on a batch.

```
1 traffic_manager = client.get_trafficmanager(args.tm_port)
2 tm_port = traffic_manager.get_port()
3 ...
4 batch.append(SpawnActor(blueprint, transform).then(SetAutopilot(FutureActor, True,tm_port)))
5 ...
6 traffic_manager.global_percentage_speed_difference(30.0)
```

Setting a Traffic Manager The following example creates an instance of the TM and sets a dangerous behaviour for a specific car that will ignore all traffic lights, leave no safety distance with the rest and drive at 120% its current speed limit.

```
1 tm = client.get_trafficmanager(port)
2 tm_port = tm.get_port()
3 for v in my_vehicles:
4     v.set_autopilot(True,tm_port)
5 danger_car = my_vehicles[0]
6 tm.ignore_lights_percentage(danger_car,100)
7 tm.distance_to_leading_vehicle(danger_car,0)
8 tm.vehicle_percentage_speed_difference(danger_car,-20)
```

Now, here is an example that registers that same list of vehicles but instead is set to conduct them with a moderate behaviour. The vehicles will drive at 80% their current speed limit, leaving at least 5 meters between them and never perform a lane change.

```
1 tm = client.get_trafficmanager(port)
2 tm_port = tm.get_port()
3 for v in my_vehicles:
4     v.set_autopilot(True,tm_port)
5 danger_car = my_vehicles[0]
6 tm.global_distance_to_leading_vehicle(5)
7 tm.global_percentage_speed_difference(80)
8 for v in my_vehicles:
9     tm.auto_lane_change(v,False)
```

Stopping a Traffic Manager The TM is not an actor that needs to be destroyed, it will stop when the corresponding client does so. This is automatically managed by the API so the user does not have to take care of it. However, it is important that when shutting down a TM, the vehicles registered to it are destroyed. Otherwise, they will stop at place, as no one will be conducting them. The script `spawn_npc.py` does this automatically.



Shutting down a TM-Server will shut down the TM-Clients connecting to it. To learn the difference between a TM-Server and a TM-Client read about `multiclient` and `multiTM`.

Hybrid physics mode

In hybrid mode, either all vehicle physics can be disabled, or enabled only in a radius around an ego vehicle with the tag `hero`. This feature removes the vehicle physics bottleneck from the simulator. Since vehicle physics are not active, all cars move by teleportation. This feature relies on `Actor.set_simulate_physics()`. However, not all the physics are disregarded. Basic calculations for a linear acceleration are maintained. By doing so, the position update, and vehicle speed still look realistic. That guarantees that when a vehicle enables or disables its physics, the transition is fluid.

The hybrid mode is disabled by default. There are two ways to enable it.

- `TrafficManager.set_hybrid_physics_mode(True)` — This method will enable the hybrid mode for the Traffic Manager object calling it.
- `Running spawn_npc.py with the flag --hybrid` — The vehicles spawned will be registered to a Traffic Manager stated inside the script, and this will run with the hybrid physics on.

There are two parameters ruling the hybrid mode. One is the `radius` that states the proximity area around any ego vehicle where physics are enabled. The other is the `vehicle` with , that will act as center of this radius.

- **Radius** (*default = 70 meters*) — States the proximity area around the ego vehicle where physics are enabled. The value can be changed with `traffic_manager.set_hybrid_mode_radius(r)`.
- **Ego vehicle** — A vehicle tagged with `role_name='hero'` that will act as the radius.
 - **If there is none**, all the vehicles will disable physics.
 - **If there are many**, the radius will be considered for all of them. That will create different areas of influence with physics enabled.

The following example shows how the physics are enabled and disabled when hybrid mode is on. The **ego vehicle** is tagged with a **red square**. Vehicles with **physics disabled** are tagged with a **blue square**. When inside the area of influence stated by the radius, **physics are enabled and the tag becomes green**.



Figure 10: Welcome to CARLA

Running multiple Traffic Managers

Definitions When working with different clients containing different TM, understanding inner implementation of the TM in the client-server architecture becomes specially relevant. There is one ruling these scenarios: **the port is the key**.

A client creates a TM by communicating with the server and passing the intended port to be used for said purpose. The port can either be stated or not, using the default as 8000.

- **TM-Server** — The port is free. This type of TM is in charge of its own logic, managed in `TrafficManagerLocal.cpp`. The following code creates two TM-Servers. Each one connects to a different port, not previously used.

```
1 tm01 = client01.get_trafficmanager() # tm01 --> tm01 (p=8000)
```

```
1 tm02 = client02.get_trafficmanager(5000) # tm02(p=5000) --> tm02 (p=5000)
```

- **TM-Client** — The port is occupied by another TM. This instances are not in charge of their own logic. Instead, they ask for changes in the parameters of the **TM-Server** they are connected to in `TrafficManagerRemote.cpp`. The following code creates two TM-Clients, that connect with the TM-Servers previously created.

```
1 tm03 = client03.get_trafficmanager() # tm03 --> tm01 (p=8000).
```

```
1 tm04 = client04.get_trafficmanager(5000) # tm04(p=5000) --> tm02 (p=5000)
```



Note how the default creation of a TM uses always ‘port=8000’, and so, only the first time a TM-Server is created. The rest will be TM-Clients connecting to it.

The CARLA server keeps register of all the TM instances internally by storing the port and also the client IP (hidden to the user) that link to them. Right now there is no way to check the TM instances that have been created so far. A connection will always be attempted when trying to create an instance and it will either create a new **TM-Server** or a **TM-Client**.



The class ‘TrafficManager.cpp’ acts as a central hub managing all the different TM instances.

Multiclient More than one TM instances created with the same port. The first will be a TM-Server. The rest will be TM-Clients connecting to it.

```
1 terminal 1: ./CarlaUE4.sh -carla-rpc-port=4000
2 terminal 2: python3 spawn_npc.py --port 4000 --tm-port 4050 # TM-Server
3 terminal 3: python3 spawn_npc.py --port 4000 --tm-port 4050 # TM-Client
```

MultiTM Different TM instances with different ports assigned.

```
1 terminal 1: ./CarlaUE4.sh -carla-rpc-port=4000
2 terminal 2: python3 spawn_npc.py --port 4000 --tm-port 4050 # TM-Server A
3 terminal 3: python3 spawn_npc.py --port 4000 --tm-port 4550 # TM-Server B
```

Multisimulation Multisimulation is when there are more than one CARLA server running at the same time. The TM declaration is not relevant. As long as the computational power allows for it, the TM can run multiple simulations at a time without any problems.

```
1 terminal 1: ./CarlaUE4.sh -carla-rpc-port=4000 # simulation A
2 terminal 2: ./CarlaUE4.sh -carla-rpc-port=5000 # simulation B
3 terminal 3: python3 spawn_npc.py --port 4000 --tm-port 4050 # TM-Server A connected to simulation A
4 terminal 4: python3 spawn_npc.py --port 5000 --tm-port 5050 # TM-Server B connected to simulation B
```

The concept of multisimulation is independent from the Traffic Manager itself. The example above runs two CARLA simulations in parallel, A and B. In each of them, a TM-Server is created independently from the other. Simulation A could run a Multiclient TM while simulation B is running a MultiTM, or no TM at all.

The only possible issue arising from this is a client trying to connect to an already existing TM which is not running on the selected simulation. In case this happens, an error message will appear and the connection will be aborted, to prevent interferences between simulations.

Other considerations

The TM is a module constantly evolving and trying to adapt the range of possibilities that it presents. For instance, in order to get more realistic behaviours we can have many clients with different TM in charge of sets of vehicles with specific and distinct behaviours. This range of possibilities also makes for a lot of different configurations that can get really complex and specific. For such reason, here are listed of considerations that should be taken into account when working with the TM as it is by the time of writing.

FPS limitations The TM stops working properly in asynchronous mode when the simulation is under 20fps. Below that rate, the server is going much faster than the client containing the TM and behaviours cannot be simulated properly. For said reason, under these circumstances it is recommended to work in **synchronous mode**.



The FPS limitations are specially relevant when working in the night mode.

Synchronous mode TM-Clients cannot tick the CARLA server in synchronous mode, **only a TM-Server can call for a tick**. If more than one TM-Server ticks, the synchrony will fail, as the server will move forward on every tick. This is specially relevant when working with the **ScenarioRunner**, which runs a TM. In this case, the TM will be subordinated to the ScenarioRunner and wait for it.



Disable the synchronous mode in the script doing the ticks before it finishes. Otherwise, the server will be blocked, waiting forever for a tick.

Summary

The Traffic Manager is one of the most complex features in CARLA and so, one that is prone to all kind of unexpected and really specific issues. The CARLA forum is open to everybody to post any doubts or suggestions, and it is the best way to keep track of issues and help the CARLA community to become greater. Feel free to login and join the community.

5 References

##Python API reference This reference contains all the details the Python API. To consult a previous reference for a specific CARLA release, change the documentation version using the panel in the bottom right corner. This will change the whole documentation to a previous state. Remember to go back to latest to get the details of the current state of CARLA.

carla.Actor

CARLA defines actors as anything that plays a role in the simulation or can be moved around. That includes: pedestrians, vehicles, sensors and traffic signs (considering traffic lights as part of these). Actors are spawned in the simulation by carla.World and they need for acarla.ActorBlueprint to be created. These blueprints belong into a library provided by CARLA, find more about themhere.

Instance Variables

- **attributes** (*dict*) A dictionary containing the attributes of the blueprint this actor was based on.
- **id** (*int*) Identifier for this actor. Unique during a given episode.
- **is_alive** (*bool*) Returns whether this object was destroyed using this actor handle.
- **parent** (*carla.Actor_*) Actors may be attached to a parent actor that they will follow around. This is said actor.
- **semantic_tags** (*list(int)*) A list of semantic tags provided by the blueprint listing components for this actor. E.g. a traffic light could be tagged with **Pole** and **TrafficLight**. These tags are used by the semantic segmentation sensor. Find more about this and other sensorshere.
- **type_id** (*str*) The identifier of the blueprint this actor was based on, e.g. **vehicle.ford.mustang**.

Methods

- **add_angular_impulse(self, angular_impulse)** Applies an angular impulse at the center of mass of the actor. This method should be used for instantaneous torques, usually applied once. Use **add_torque()** to apply rotation forces over a period of time.
 - **Parameters:**
 - * **angular_impulse** (*carla.Vector3D – degrees*s_*) – Angular impulse vector in global coordinates.
- **add_force(self, force)** Applies a force at the center of mass of the actor. This method should be used for forces that are applied over a certain period of time. Use **add_impulse()** to apply an impulse that only lasts an instant.
 - **Parameters:**
 - * **force** (*carla.Vector3D – N_*) – Force vector in global coordinates.
- **add_impulse(self, impulse)** Applies an impulse at the center of mass of the actor. This method should be used for instantaneous forces, usually applied once. Use **add_force()** to apply forces over a period of time.
 - **Parameters:**
 - * **impulse** (*carla.Vector3D – N*s_*) – Impulse vector in global coordinates.
- **add_torque(self, torque)** Applies a torque at the center of mass of the actor. This method should be used for torques that are applied over a certain period of time. Use **add_angular_impulse()** to apply a torque that only lasts an instant.
 - **Parameters:**
 - * **torque** (*carla.Vector3D – degrees_*) – Torque vector in global coordinates.
- **destroy(self)** Tells the simulator to destroy this actor and returns True if it was successful. It has no effect if it was already destroyed.
 - **Return:** *bool*
 - **Warning:** *This method blocks the script until the destruction is completed by the simulator.*
- **disable_constant_velocity(self)** Disables any constant velocity previously set for acarla.Vehicle actor.
- **enable_constant_velocity(self, velocity)** Sets a vehicle's velocity vector to a constant value over time. The resulting velocity will be approximately the **velocity** being set, as with **set_target_velocity()**.
 - **Parameters:**
 - * **velocity** (*carla.Vector3D – m/s_*) – Velocity vector in local space.

- **Warning:** Enabling a constant velocity for a vehicle managed by the Traffic Manager may cause conflicts. This method overrides any changes in velocity by the TM.

Getters

- **get_acceleration(self)** Returns the actor's 3D acceleration vector the client received during last tick. The method does not call the simulator.
 - **Return:** carla.Vector3D – m/s²
- **get_angular_velocity(self)** Returns the actor's angular velocity vector the client received during last tick. The method does not call the simulator.
 - **Return:** carla.Vector3D – rad/s
- **get_location(self)** Returns the actor's location the client received during last tick. The method does not call the simulator.
 - **Return:** carla.Location – meters
 - **Setter:** carla.Actor.set_location
- **get_transform(self)** Returns the actor's transform (location and rotation) the client received during last tick. The method does not call the simulator.
 - **Return:** carla.Transform
 - **Setter:** carla.Actor.set_transform
- **get_velocity(self)** Returns the actor's velocity vector the client received during last tick. The method does not call the simulator.
 - **Return:** carla.Vector3D – m/s
- **get_world(self)** Returns the world this actor belongs to.
 - **Return:** carla.World

Setters

- **set_target_angular_velocity(self, angular_velocity)** Sets the actor's angular velocity vector. The modification will be effective two frames after the setting. Also, this is applied before the physics step so the resulting angular velocity will be affected by external forces at this frame such as friction.
 - **Parameters:**
 - * angular_velocity (carla.Vector3D)
 - **Note:** The update will not be effective until two frames after it is set.
- **set_location(self, location)** Teleports the actor to a given location.
 - **Parameters:**
 - * location (carla.Location – meters)
 - **Getter:** carla.Actor.get_location
- **set_simulate_physics(self, enabled=True)** Enables or disables the simulation of physics on this actor.
 - **Parameters:**
 - * enabled (bool)
- **set_transform(self, transform)** Teleports the actor to a given transform (location and rotation).
 - **Parameters:**
 - * transform (carla.Transform)
 - **Getter:** carla.Actor.get_transform
- **set_target_velocity(self, velocity)** Sets the actor's velocity vector. The modification will be effective two frames after the setting. Also, this is applied before the physics step so the resulting velocity will be affected by external forces at this frame such as friction.
 - **Parameters:**
 - * velocity (carla.Vector3D)
 - **Note:** The update will not be effective until two frames after it is set.

Dunder methods

- ****__str__(self**)**

carla.ActorAttribute

CARLA provides a library of blueprints for actors that can be accessed as `carla.BlueprintLibrary`. Each of these blueprints has a series of attributes defined internally. Some of these are modifiable, others are not. A list of recommended values is provided for those that can be set.

Instance Variables

- **id (str)** The attribute's name and identifier in the library.
- **is_modifiable (bool)** It is True if the attribute's value can be modified.

- **recommended_values** (*list(str)*) A list of values suggested by those who designed the blueprint.
- **type** (*carla.ActorAttributeType*) The attribute's parameter type.

Methods

- **as_bool(self)** Reads the attribute as boolean value.
- **as_color(self)** Reads the attribute as *carla.Color*.
- **as_float(self)** Reads the attribute as float.
- **as_int(self)** Reads the attribute as int.
- **as_str(self)** Reads the attribute as string.

Dunder methods

- **__bool__(self**)**
- **__float__(self**)**
- **__int__(self**)**
- **__str__(self**)**
- **__eq__(self, other**=bool / int / float / str / carla.Color / carla.ActorAttribute)** Returns true if this actor's attribute and **other** are the same.
 - **Return:** *bool*
- **__ne__(self, other**=bool / int / float / str / carla.Color / carla.ActorAttribute)** Returns true if this actor's attribute and **other** are different.
 - **Return:** *bool*
- **__nonzero__(self**)** Returns true if this actor's attribute is not zero or null.
 - **Return:** *bool*

carla.ActorAttributeType

CARLA provides a library of blueprints for actors in *carla.BlueprintLibrary* with different attributes each. This class defines the types those *carla.ActorAttribute* can be as a series of enum. All this information is managed internally and listed here for a better comprehension of how CARLA works.

Instance Variables

- **Bool**
- **Int**
- **Float**
- **String**
- **RGBColor**

carla.ActorBlueprint

CARLA provides a blueprint library for actors that can be consulted through *carla.BlueprintLibrary*. Each of these consists of an identifier for the blueprint and a series of attributes that may be modifiable or not. This class is the intermediate step between the library and the actor creation. Actors need an actor blueprint to be spawned. These store the information for said blueprint in an object with its attributes and some tags to categorize them. The user can then customize some attributes and eventually spawn the actors through *carla.World*.

Instance Variables

- **id (str)** The identifier of said blueprint inside the library. E.g. `walker.pedestrian.0001`.
- **tags (list(str))** A list of tags each blueprint has that helps describing them. E.g. `['0001', 'pedestrian', 'walker']`.

Methods

- **has_attribute(self, id)** Returns True if the blueprint contains the attribute **id**.
 - **Parameters:**
 - * **id (str)** – e.g. `gender` would return **True** for pedestrians' blueprints.
 - **Return:** *bool*
- **has_tag(self, tag)** Returns True if the blueprint has the specified **tag** listed.
 - **Parameters:**
 - * **tag (str)** – e.g. `'walker'`.
 - **Return:** *bool*
- **match_tags(self, wildcard_pattern)** Returns True if any of the tags listed for this blueprint matches **wildcard_pattern**. Matching follows fnmatch standard.

- **Parameters:**
 - * `wildcard_pattern` (`str`)
- **Return:** `bool`

Getters

- `get_attribute(self, id)` Returns the actor's attribute with `id` as identifier if existing.
 - **Parameters:**
 - * `id` (`str`)
 - **Return:** `carla.ActorAttribute`
 - **Setter:** `carla.ActorBlueprint.set_attribute_`

Setters

- `set_attribute(self, id, value)` If the `id` attribute is modifiable, changes its value to `value`.
 - **Parameters:**
 - * `id` (`str`) – The identifier for the attribute that is intended to be changed.
 - * `value` (`str`) – The new value for said attribute.
 - **Getter:** `carla.ActorBlueprint.get_attribute_`

Dunder methods

- `__iter__(self**)` Allows iteration within this class' attributes.
- `__len__(self**)` Returns the amount of attributes for this blueprint.
- `__str__(self**)`

`carla.ActorList`

A class that contains every actor present on the scene and provides access to them. The list is automatically created and updated by the server and it can be returned using `carla.World`.

Methods

- `filter(self, wildcard_pattern)` Filters a list of Actors matching `wildcard_pattern` against their variable `type_id` (which identifies the blueprint used to spawn them). Matching follows fnmatch standard.
 - **Parameters:**
 - * `wildcard_pattern` (`str`)
 - **Return:** `list`
- `find(self, actor_id)` Finds an actor using its identifier and returns it or `None` if it is not present.
 - **Parameters:**
 - * `actor_id` (`int`)
 - **Return:** `carla.Actor`

Dunder methods

- `__getitem__(self, pos**=int)` Returns the actor corresponding to `pos` position in the list.
 - **Return:** `carla.Actor`
- `__iter__(self**)` Allows the iteration for the actors in this object.
- `__len__(self**)` Returns the amount of actors listed.
 - **Return:** `int`
- `__str__(self**)` Parses to the ID for every actor listed.
 - **Return:** `str`

`carla.ActorSnapshot`

A class that comprises all the information for an actor at a certain moment in time. These objects are contained in `carla.WorldSnapshot` and sent to the client once every tick.

Instance Variables

- `id` (`int`) An identifier for the snapshot itself.

Methods

Getters

- `get_acceleration(self)` Returns the acceleration vector registered for an actor in that tick.
 - **Return:** `carla.Vector3D` – m/s²
- `get_angular_velocity(self)` Returns the angular velocity vector registered for an actor in that tick.

- **Return:** carla.Vector3D – rad/s_
 - **get_transform(self)** Returns the actor's transform (location and rotation) for an actor in that tick.
 - **Return:** carla.Transform_
 - **get_velocity(self)** Returns the velocity vector registered for an actor in that tick.
 - **Return:** carla.Vector3D – m/s_

carla.AttachmentType

Class that defines attachment options between an actor and its parent. When spawning actors, these can be attached to another actor so their position changes accordingly. This is specially useful for sensors. Here is a brief recipe in which we can see how sensors can be attached to a car when spawned. Note that the attachment type is declared as an enum within the class.

Instance Variables

- **Rigid** With this fixed attachment the object follow its parent position strictly. This is the recommended attachment to retrieve precise data from the simulation.
- **SpringArm** An attachment that expands or retracts the position of the actor, depending on its parent. This attachment is only recommended to record videos from the simulation where a smooth movement is needed. SpringArms are an Unreal Engine component so check this out to learn some more about them. Warning: The SpringArm attachment presents weird behaviors when an actor is spawned with a relative translation in the Z-axis (e.g. child_location = Location(0,0,2)).

carla.BlueprintLibrary

A class that contains the blueprints provided for actor spawning. Its main application is to return carla.ActorBlueprint objects needed to spawn actors. Each blueprint has an identifier and attributes that may or may not be modifiable. The library is automatically created by the server and can be accessed through carla.World.

Here is a reference containing every available blueprint and its specifics.

Methods

- **filter(self, wildcard_pattern)** Filters a list of blueprints matching the `wildcard_pattern` against the id and tags of every blueprint contained in this library and returns the result as a new one. Matching follows fnmatch standard.
 - **Parameters:**
 - * `wildcard_pattern (str)`
 - **Return:** carla.BlueprintLibrary_
- **find(self, id)** Returns the blueprint corresponding to that identifier.
 - **Parameters:**
 - * `id (str)`
 - **Return:** carla.ActorBlueprint_

Dunder methods

- **__getitem__(self, pos**=int)** Returns the blueprint stored in `pos` position inside the data structure containing them.
 - **Return:** carla.ActorBlueprint_
- **__iter__(self***)** Method that allows iteration within the blueprints provided.
- **__len__(self***)** Returns the amount of blueprints comprising the library.
 - **Return:** int
- **__str__(self***)** Parses the identifiers for every blueprint to string.
 - **Return:** string

carla.BoundingBox

Bounding boxes contain the geometry of an actor or an element in the scene. They can be used by carla.DebugHelper or acarla.Client to draw their shapes for debugging. Check out this recipe where the user takes a snapshot of the world and then proceeds to draw bounding boxes for traffic lights.

Instance Variables

- **extent** (carla.Vector3D – meters_) Vector from the center of the box to one vertex. The value in each axis equals half the size of the box for that axis. `extent.x * 2` would return the size of the box in the X-axis.
- **location** (carla.Location – meters_) The center of the bounding box.

- **rotation** (carla.Rotation_) The orientation of the bounding box.

Methods

- **__init__(self, location, extent*)**
 - **Parameters:**
 - * **location** (carla.Location_) – Center of the box, relative to its parent.
 - * **extent** (carla.Vector3D – meters_) – Vector containing half the size of the box for every axis.
- **contains(self, world_point, transform)** Returns **True** if a point passed in world space is inside this bounding box.
 - **Parameters:**
 - * **world_point** (carla.Location – meters_) – The point in world space to be checked.
 - * **transform** (carla.Transform_) – Contains location and rotation needed to convert this object's local space to world space.
 - **Return:** *bool*

Getters

- **get_local_vertices(self)** Returns a list containing the locations of this object's vertices in local space.
 - **Return:** *listcarla.Location*
- **get_world_vertices(self, transform)** Returns a list containing the locations of this object's vertices in world space.
 - **Parameters:**
 - * **transform** (carla.Transform_) – Contains location and rotation needed to convert this object's local space to world space.
 - **Return:** *listcarla.Location*

Dunder methods

- **__eq__(self, other**carla.BoundingBox)** Returns true if both location and extent are equal for this and **other**.
 - **Return:** *bool*
- **__ne__(self, other**carla.BoundingBox)** Returns true if either location or extent are different for this and **other**.
 - **Return:** *bool*
- **__str__(self**)** Parses the location and extent of the bounding box to string.
 - **Return:** *str*

carla.CityObjectLabel

Enum declaration that contains the different tags available to filter the bounding boxes returned by `carla.World.get_level_bbs()`. These values correspond to the semantic tag that the elements in the scene have.

Instance Variables

- **None**
- **Buildings**
- **Fences**
- **Other**
- **Pedestrians**
- **Poles**
- **RoadLines**
- **Roads**
- **Sidewalks**
- **TrafficSigns**
- **Vegetation**
- **Vehicles**
- **Walls**
- **Sky**
- **Ground**
- **Bridge**
- **RailTrack**
- **GuardRail**
- **TrafficLight**
- **Static**

- Dynamic
- Water
- Terrain

carla.Client

The Client connects CARLA to the server which runs the simulation. Both server and client contain a CARLA library (libcarla) with some differences that allow communication between them. Many clients can be created and each of these will connect to the RPC server inside the simulation to send commands. The simulation runs server-side. Once the connection is established, the client will only receive data retrieved from the simulation. Walkers are the exception. The client is in charge of managing pedestrians so, if you are running a simulation with multiple clients, some issues may arise. For example, if you spawn walkers through different clients, collisions may happen, as each client is only aware of the ones it is in charge of.

The client also has a recording feature that saves all the information of a simulation while running it. This allows the server to replay it at will to obtain information and experiment with it. Here is some information about how to use this recorder.

Methods

- ****__init__(self, host=127.0.0.1, port=2000, worker_threads**=0)** Client constructor.
 - **Parameters:**
 - * `host (str)` – IP address where a CARLA Simulator instance is running. Default is localhost (127.0.0.1).
 - * `port (int)` – TCP port where the CARLA Simulator instance is running. Default are 2000 and the subsequent 2001.
 - * `worker_threads (int)` – Number of working threads used for background updates. If 0, use all available concurrency.
- **apply_batch(self, commands)** Executes a list of commands on a single simulation step and retrieves no information. If you need information about the response of each command, use the **apply_batch_sync()** method. Here is an example on how to delete the actors that appear in carla.ActorList all at once.
 - **Parameters:**
 - * `commands (list)` – A list of commands to execute in batch. Each command is different and has its own parameters. They appear listed at the bottom of this page.
- **apply_batch_sync(self, commands, due_tick_cue=False)** Executes a list of commands on a single simulation step, blocks until the commands are linked, and returns a list of command.Response that can be used to determine whether a single command succeeded or not. Here is an example of it being used to spawn actors.
 - **Parameters:**
 - * `commands (list)` – A list of commands to execute in batch. The commands available are listed right above, in the method **apply_batch()**.
 - * `due_tick_cue (bool)` – A boolean parameter to specify whether or not to perform carla.World.tick after applying the batch in *synchronous mode*. It is **False** by default.
 - **Return:** `list(command.Response)`
- **generate.opendrive_world(self, opendrive, parameters=(2.0, 50.0, 1.0, 0.6, true, true))** Loads a new world with a basic 3D topology generated from the content of an OpenDRIVE file. This content is passed as a `string` parameter. It is similar to `client.load_world(map_name)` but allows for custom OpenDRIVE maps in server side. Cars can drive around the map, but there are no graphics besides the road and sidewalks.
 - **Parameters:**
 - * `opendrive (str)` – Content of an OpenDRIVE file as `string`, **not the path to the .xodr**.
 - * `parameters (carla.OpendriveGenerationParameters_)` – Additional settings for the mesh generation. If none are provided, default values will be used.
- **load_world(self, map_name)** Creates a new world with default settings using `map_name` map. All actors in the current world will be destroyed.
 - **Parameters:**
 - * `map_name (str)` – Name of the map to be used in this world. Accepts both full paths and map names, e.g. '/Game/Carla/Maps/Town01' or 'Town01'. Remember that these paths are dynamic.
- **reload_world(self)** Reload the current world, note that a new world is created with default settings using the same map. All actors present in the world will be destroyed, **but** traffic manager instances will stay alive.
 - **Raises:** RuntimeError when corresponding.
- **replay_file(self, name, start, duration, follow_id)** Load a new world with default settings using `map_name` map. All actors present in the current world will be destroyed, **but** traffic manager instances will stay alive.
 - **Parameters:**
 - * `name (str)` – Name of the file containing the information of the simulation.

- * **start** (*float – seconds*) – Time where to start playing the simulation. Negative is read as beginning from the end, being -10 just 10 seconds before the recording finished.
- * **duration** (*float – seconds*) – Time that will be reenacted using the information **name** file. If the end is reached, the simulation will continue.
- * **follow_id** (*int*) – ID of the actor to follow. If this is 0 then camera is disabled.
- **stop_replayer(self, keep_actors)** Stop current replayer.
 - **Parameters:**
 - * **keep_actors** (*_bool_*) – True if you want autoremove all actors from the replayer, or False to keep them.
- **show_recorder_actors_blocked(self, filename, min_time, min_distance)** The terminal will show the information registered for actors considered blocked. An actor is considered blocked when it does not move a minimum distance in a period of time, being these **min_distance** and **min_time**.
 - **Parameters:**
 - * **filename** (*str*) – Name of the recorded file to load.
 - * **min_time** (*float – seconds*) – Minimum time the actor has to move a minimum distance before being considered blocked. Default is 60 seconds.
 - * **min_distance** (*float – centimeters*) – Minimum distance the actor has to move to not be considered blocked. Default is 100 centimeters.
 - **Return:** *string*
- **show_recorder_collisions(self, filename, category1, category2)** The terminal will show the collisions registered by the recorder. These can be filtered by specifying the type of actor involved. The categories will be specified in **category1** and **category2** as follows: ‘h’ = Hero, the one vehicle that can be controlled manually or managed by the user. ‘v’ = Vehicle ‘w’ = Walker ‘t’ = Traffic light ‘o’ = Other ‘a’ = Any If you want to see only collisions between a vehicles and a walkers, use for **category1** as ‘v’ and **category2** as ‘w’ or vice versa. If you want to see all the collisions (filter off) you can use ‘a’ for both parameters.
 - **Parameters:**
 - * **filename** (*str*) – Name or absolute path of the file recorded, depending on your previous choice.
 - * **category1** (*single char*) – Character variable specifying a first type of actor involved in the collision.
 - * **category2** (*single char*) – Character variable specifying the second type of actor involved in the collision.
 - **Return:** *string*
- **show_recorder_file_info(self, filename, show_all)** The information saved by the recorder will be parsed and shown in your terminal as text (frames, times, events, state, positions...). The information shown can be specified by using the **show_all** parameter. Here is some more information about how to read the recorder file.
 - **Parameters:**
 - * **filename** (*str*) – Name or absolute path of the file recorded, depending on your previous choice.
 - * **show_all** (*bool*) – If **True**, returns all the information stored for every frame (traffic light states, positions of all actors, orientation and animation data...). If **False**, returns a summary of key events and frames.
 - **Return:** *string*
- **start_recorder(self, filename, additional_data=False)** Enables the recording feature, which will start saving every information possible needed by the server to replay the simulation.
 - **Parameters:**
 - * **filename** (*str*) – Name of the file to write the recorded data. A simple name will save the recording in ‘CarlaUE4/Saved/recording.log’. Otherwise, if some folder appears in the name, it will be considered an absolute path.
 - * **additional_data** (*bool*) – Enables or disable recording non-essential data for reproducing the simulation (bounding box location, physics control parameters, etc).
 - **stop_recorder(self)** Stops the recording in progress. If you specified a path in **filename**, the recording will be there. If not, look inside **CarlaUE4/Saved/**.

Getters

- **get_available_maps(self)** Returns a list of strings containing the paths of the maps available on server. These paths are dynamic, they will be created during the simulation and so you will not find them when looking up in your files. One of the possible returns for this method would be: ['/Game/Carla/Maps/Town01', '/Game/Carla/Maps/Town02', '/Game/Carla/Maps/Town03', '/Game/Carla/Maps/Town04', '/Game/Carla/Maps/Town05', '/Game/Carla/Maps/Town06', '/Game/Carla/Maps/Town07'].
 - **Return:** *list(str)*
- **get_client_version(self)** Returns the client libcarla version by consulting it in the “Version.h” file. Both client and server can use different libcarla versions but some issues may arise regarding unexpected incompatibilities.
 - **Return:** *str*

- **get_server_version(self)** Returns the server libcarla version by consulting it in the “Version.h” file. Both client and server should use the same libcarla version.
 - **Return:** str
- **get_trafficmanager(self, client_connection=8000)** Returns an instance of the traffic manager related to the specified port. If it does not exist, this will be created.
 - **Parameters:**
 - * `client_connection (int)` – Port that will be used by the traffic manager. Default is 8000.
 - **Return:** carla.TrafficManager
- **get_world(self)** Returns the world object currently active in the simulation. This world will be later used for example to load maps.
 - **Return:** carla.World

Setters

- **set_replayer_time_factor(self, time_factor=1.0)** When used, the time speed of the reenacted simulation is modified at will. It can be used several times while a playback is in curse.
 - **Parameters:**
 - * `time_factor (float)` – 1.0 means normal time speed. Greater than 1.0 means fast motion (2.0 would be double speed) and lesser means slow motion (0.5 would be half speed).
- **set_timeout(self, seconds)** Sets the maximum time a network call is allowed before blocking it and raising a timeout exceeded error.
 - **Parameters:**
 - * `seconds (float - seconds)` – New timeout value. Default is 5 seconds.

carla.CollisionEvent

Inherited from carla.SensorData Class that defines a collision data for sensor.other.collision. The sensor creates one of this for every collision detected which may be many for one simulation step. Learn more about this here.

Instance Variables

- **actor** (carla.Actor) The actor the sensor is attached to, the one that measured the collision.
- **other_actor** (carla.Actor) The second actor involved in the collision.
- **normal impulse** (carla.Vector3D – N*s) Normal impulse resulting of the collision.

carla.Color

Class that defines a 32-bit RGBA color.

Instance Variables

- **r** (int) Red color (0-255).
- **g** (int) Green color (0-255).
- **b** (int) Blue color (0-255).
- **a** (int) Alpha channel (0-255).

Methods

- **__init__(self, r=0, g=0, b=0, a**=255)** Initializes a color, black by default.
 - **Parameters:**
 - * `r (int)`
 - * `g (int)`
 - * `b (int)`
 - * `a (int)`

Dunder methods

- **__eq__(self, other**carla.Color)**
- **__ne__(self, other**carla.Color)**
- **__str__(self**)**

carla.ColorConverter

Class that defines conversion patterns that can be applied to a carla.Image in order to show information provided by carla.Sensor. Depth conversions cause a loss of accuracy, as sensors detect depth as float that is then converted to a grayscale value between 0 and 255. Take a look at this recipe to see an example of how to create and save image data for sensor.camera.semantic_segmentation.

Instance Variables

- **CityScapesPalette** Converts the image to a segmented map using tags provided by the blueprint library. Used by the semantic segmentation camera.
- **Depth** Converts the image to a linear depth map. Used by the depth camera.
- **LogarithmicDepth** Converts the image to a depth map using a logarithmic scale, leading to better precision for small distances at the expense of losing it when further away.
- **Raw** No changes applied to the image. Used by the RGB camera.

carla.DVSEvent

Class that defines a DVS event. An event is a quadruple, so a tuple of 4 elements, with x, y pixel coordinate location, timestamp t and polarity pol of the event. Learn more about them [here](#).

Instance Variables

- **x (int)** X pixel coordinate.
- **y (int)** Y pixel coordinate.
- **t (int)** Timestamp of the moment the event happened.
- **pol (bool)** Polarity of the event. **True** for positive and **False** for negative.

Methods

Dunder methods

- **__str__(self*)**

carla.DVSEventArray

Class that defines a stream of events (carla.DVSEvent). Such stream is an array of arbitrary size depending on the number of events. This class also stores the field of view, the height and width of the image and the timestamp from convenience. Learn more about them [here](#).

Instance Variables

- **fov (float – degrees)** Horizontal field of view of the image.
- **height (int)** Image height in pixels.
- **width (int)** Image width in pixels.
- **raw_data (bytes)**

Methods

- **to_image(self)** Converts the image following this pattern: blue indicates positive events, red indicates negative events.
- **to_array(self)** Converts the stream of events to an array of int values in the following order [x, y, t, pol].
- **to_array_x(self)** Returns an array with X pixel coordinate of all the events in the stream.
- **to_array_y(self)** Returns an array with Y pixel coordinate of all the events in the stream.
- **to_array_t(self)** Returns an array with the timestamp of all the events in the stream.
- **to_array_pol(self)** Returns an array with the polarity of all the events in the stream.

Dunder methods

- **__getitem__(self, pos*=int)**
- **__iter__(self*)**
- **__len__(self*)**
- **__setitem__(self, pos=int, color**carla.Color)**
- **__str__(self*)**

carla.DebugHelper

Helper class part of carla.World that defines methods for creating debug shapes. By default, shapes last one second. They can be permanent, but take into account the resources needed to do so. Check out [this recipe](#) where the user takes a snapshot of the world and then proceeds to draw bounding boxes for traffic lights.

Methods

- **draw_arrow(self, begin, end, thickness=0.1, arrow_size=0.1, color=(255,0,0), life_time=-1.0)** Draws an arrow from begin to end pointing in that direction.
 - **Parameters:**

- * **begin** (carla.Location – meters_) – Point in the coordinate system where the arrow starts.
- * **end** (carla.Location – meters_) – Point in the coordinate system where the arrow ends and points towards to.
- * **thickness** (float – meters) – Density of the line.
- * **arrow_size** (float – meters) – Size of the tip of the arrow.
- * **color** (carla.Color_) – RGB code to color the object. Red by default.
- * **life_time** (float – seconds) – Shape's lifespan. By default it only lasts one frame. Set this to 0 for permanent shapes.
- **draw_box(self, box, rotation, thickness=0.1, color=(255,0,0), life_time=-1.0)** Draws a box, ussually to act for object colliders.
 - **Parameters:**
 - * **box** (carla.BoundingBox_) – Object containing a location and the length of a box for every axis.
 - * **rotation** (carla.Rotation – degrees (pitch,yaw,roll)_) – Orientation of the box according to Unreal Engine's axis system.
 - * **thickness** (float – meters) – Density of the lines that define the box.
 - * **color** (carla.Color_) – RGB code to color the object. Red by default.
 - * **life_time** (float – seconds) – Shape's lifespan. By default it only lasts one frame. Set this to 0 for permanent shapes.
- **draw_line(self, begin, end, thickness=0.1, color=(255,0,0), life_time=-1.0)** Draws a line in between **begin** and **end**.
 - **Parameters:**
 - * **begin** (carla.Location – meters_) – Point in the coordinate system where the line starts.
 - * **end** (carla.Location – meters_) – Spot in the coordinate system where the line ends.
 - * **thickness** (float – meters) – Density of the line.
 - * **color** (carla.Color_) – RGB code to color the object. Red by default.
 - * **life_time** (float – seconds) – Shape's lifespan. By default it only lasts one frame. Set this to 0 for permanent shapes.
- **draw_point(self, location, size=0.1, color=(255,0,0), life_time=-1.0)** Draws a point **location**.
 - **Parameters:**
 - * **location** (carla.Location – meters_) – Spot in the coordinate system to center the object.
 - * **size** (float – meters) – Density of the point.
 - * **color** (carla.Color_) – RGB code to color the object. Red by default.
 - * **life_time** (float – seconds) – Shape's lifespan. By default it only lasts one frame. Set this to 0 for permanent shapes.
- **draw_string(self, location, text, draw_shadow=False, color=(255,0,0), life_time=-1.0)** Draws a string in a given location of the simulation which can only be seen server-side.
 - **Parameters:**
 - * **location** (carla.Location – meters_) – Spot in the simulation where the text will be centered.
 - * **text** (str) – Text intended to be shown in the world.
 - * **draw_shadow** (_bool_) – Casts a shadow for the string that could help in visualization. It is disabled by default.
 - * **color** (carla.Color_) – RGB code to color the string. Red by default.
 - * **life_time** (float – seconds) – Shape's lifespan. By default it only lasts one frame. Set this to 0 for permanent shapes.

carla.GearPhysicsControl

Class that provides access to vehicle transmission details by defining a gear and when to run on it. This will be later used by carla.VehiclePhysicsControl to help simulate physics.

Instance Variables

- **ratio** (float) The transmission ratio of the gear.
- **down_ratio** (float) Quotient between current RPM and MaxRPM where the autonomous gear box should shift down.
- **up_ratio** (float) Quotient between current RPM and MaxRPM where the autonomous gear box should shift up.

Methods

- ****__init__(self, ratio=1.0, down_ratio=0.5, up_ratio**=0.65)**
 - **Parameters:**
 - * **ratio** (float)

```
* down_ratio (float)
* up_ratio (float)
```

Dunder methods

- **__eq__(self, other**carla.GearPhysicsControl)
- **__ne__(self, other**carla.GearPhysicsControl)
- **__str__(self**)

carla.GeoLocation

Class that contains geographical coordinates simulated data. The carla.Map can convert simulation locations by using the tag in the OpenDRIVE file.

Instance Variables

- **latitude** (float – degrees) North/South value of a point on the map.
- **longitude** (float – degrees) West/East value of a point on the map.
- **altitude** (float – meters) Height regarding ground level.

Methods

- **__init__(self, latitude=0.0, longitude=0.0, altitude**=0.0)
 - **Parameters:**
 - * **latitude** (float – degrees)
 - * **longitude** (float – degrees)
 - * **altitude** (float – meters)

Dunder methods

- **__eq__(self, other**carla.GeoLocation)
- **__ne__(self, other**carla.GeoLocation)
- **__str__(self**)

carla.GnssMeasurement

Inherited from carla.SensorData Class that defines the Gnss data registered by a sensor.other.gnss. It essentially reports its position with the position of the sensor and an OpenDRIVE geo-reference.

Instance Variables

- **altitude** (float – meters) Height regarding ground level.
- **latitude** (float – degrees) North/South value of a point on the map.
- **longitude** (float – degrees) West/East value of a point on the map.

Methods

Dunder methods

- **__str__(self**)

carla.IMUMeasurement

Inherited from carla.SensorData Class that defines the data registered by a sensor.other imu, regarding the sensor's transformation according to the current carla.World. It essentially acts as accelerometer, gyroscope and compass.

Instance Variables

- **accelerometer** (carla.Vector3D – m/s²) Linear acceleration.
- **compass** (float – radians) Orientation with regard to the North ([0.0, -1.0, 0.0] in Unreal Engine).
- **gyroscope** (carla.Vector3D – rad/s) Angular velocity.

Methods

Dunder methods

- **__str__(self**)

carla.Image

Inherited from carla.SensorData Class that defines an image of 32-bit BGRA colors that will be used as initial data retrieved by camera sensors. There are different camera sensors (currently three, RGB, depth and semantic segmentation) and each of these makes different use for the images. Learn more about themhere.

Instance Variables

- **fov** (*float – degrees*) Horizontal field of view of the image.
- **height** (*int*) Image height in pixels.
- **width** (*int*) Image width in pixels.
- **raw_data** (*bytes*)

Methods

- **convert(self, color_converter)** Converts the image following the `color_converter` pattern.
 - **Parameters:**
 - * `color_converter` (`carla.ColorConverter_`)
- **save_to_disk(self, path, color_converter=Raw)** Saves the image to disk using a converter pattern stated as `color_converter`. The default conversion pattern is Raw that will make no changes to the image.
 - **Parameters:**
 - * `path` (*str*) – Path that will contain the image.
 - * `color_converter` (`carla.ColorConverter_`) – Default Raw will make no changes.

Dunder methods

- **__getitem__(self, pos**=int)**
- **__iter__(self**)**
- **__len__(self**)**
- **__setitem__(self, pos=int, color**carla.Color)**
- **__str__(self**)**

carla.Junction

Class that embodies the intersections on the road described in the OpenDRIVE file according to OpenDRIVE 1.4 standards.

Instance Variables

- **id** (*int*) Identifier found in the OpenDRIVE file.
- **bounding_box** (`carla.BoundingBox_`) Bounding box encapsulating the junction lanes.

Methods

Getters

- **get_waypoints(self, lane_type)** Returns a list of pairs of waypoints. Every tuple on the list contains first an initial and then a final waypoint within the intersection boundaries that describe the beginning and the end of said lane along the junction. Lanes follow their OpenDRIVE definitions so there may be many different tuples with the same starting waypoint due to possible deviations, as this are considered different lanes.
 - **Parameters:**
 - * `lane_type` (`carla.LaneType_`) – Type of lanes to get the waypoints.
 - **Return:** `list(tuplecarla.Waypoint)`

carla.Landmark

Class that defines any type of traffic landmark or sign affecting a road. These class mediates between the OpenDRIVE 1.4 standard definition of the landmarks and their representation in the simulation. This class retrieves all the information defining a landmark in OpenDRIVE and facilitates information about which lanes does it affect and when. Landmarks will be accessed bycarla.Waypoint objects trying to retrieve the regulation of their lane. Therefore some attributes depend on the waypoint that is consulting the landmark and so, creating the object.

Instance Variables

- **road_id** (*int*) The OpenDRIVE ID of the road where this landmark is defined. Due to OpenDRIVE road definitions, this road may be different from the road the landmark is currently affecting. It is mostly the case in junctions where the road diverges in different routes. Example: a traffic light is defined in one of the divergent roads in a junction, but it affects all the possible routes.

- **distance** (*float – meters*) Distance between the landmark and the waypoint creating the object (querying `get_landmarks` or `get_landmarks_of_type`).
- **s** (*float – meters*) Distance where the landmark is positioned along the geometry of the road `road_id`.
- **t** (*float – meters*) Lateral distance where the landmark is positioned from the edge of the road `road_id`.
- **id** (*str*) Unique ID of the landmark in the OpenDRIVE file.
- **name** (*str*) Name of the landmark in the in the OpenDRIVE file.
- **is_dynamic** (*bool*) Indicates if the landmark has state changes over time such as traffic lights.
- **orientation** (`carla.LandmarkOrientation – degrees_`) Indicates which lanes the landmark is facing towards to.
- **z_offset** (*float – meters*) Height where the landmark is placed.
- **country** (*str*) Country code where the landmark is defined (default to OpenDRIVE is Germany 2017).
- **type** (*str*) Type identifier of the landmark according to the country code.
- **sub_type** (*str*) Subtype identifier of the landmark according to the country code.
- **value** (*float*) Value printed in the signal (e.g. speed limit, maximum weight, etc).
- **unit** (*str*) Units of measurement for the attribute `value`.
- **height** (*float – meters*) Total height of the signal.
- **width** (*float – meters*) Total width of the signal.
- **text** (*str*) Additional text in the signal.
- **h_offset** (*float – meters*) Orientation offset of the signal relative to the the definition of `road_id` at `s` in OpenDRIVE.
- **pitch** (*float – meters*) Pitch rotation of the signal (Y-axis in UE coordinates system).
- **roll** (*float*) Roll rotation of the signal (X-axis in UE coordinates system).
- **waypoint** (`carla.Waypoint_`) A waypoint placed in the lane of the one that made the query and at the `s` of the landmark. It is the first waypoint for which the landmark will be effective.
- **transform** (`carla.Transform_`) The location and orientation of the landmark in the simulation.

Methods

Getters

- **get_lane_validities(self)** Returns which lanes the landmark is affecting to. As there may be specific lanes where the landmark is not effective, the return is a list of pairs containing ranges of the `lane_id` affected:
Example: In a road with 5 lanes, being 3 not affected: [(from_lane1,to_lane2),(from_lane4,to_lane5)].
– **Return:** `list(tuple(int))`

`carla.LandmarkOrientation`

Helper class to define the orientation of a landmark in the road. The definition is not directly translated from OpenDRIVE but converted for the sake of understanding.

Instance Variables

- **Positive** The landmark faces towards vehicles going on the same direction as the road's geometry definition (lanes 0 and negative in OpenDRIVE).
- **Negative** The landmark faces towards vehicles going on the opposite direction to the road's geometry definition (positive lanes in OpenDRIVE).
- **Both** Affects vehicles going in both directions of the road.

`carla.LandmarkType`

Helper class containing a set of commonly used landmark types as defined by the default country code in the OpenDRIVE standard (Germany 2017). `_carla.Landmark` does not reference this class__. The landmark type is a string that varies greatly depending on the country code being used. This class only makes it easier to manage some of the most commonly used in the default set by describing them as an enum.

Instance Variables

- **Danger** Type 101.
- **LanesMerging** Type 121.
- **CautionPedestrian** Type 133.
- **CautionBicycle** Type 138.
- **LevelCrossing** Type 150.
- **StopSign** Type 206.
- **YieldSign** Type 205.
- **MandatoryTurnDirection** Type 209.
- **MandatoryLeftRightDirection** Type 211.

- **TwoChoiceTurnDirection** Type 214.
- **Roundabout** Type 215.
- **PassRightLeft** Type 222.
- **AccessForbidden** Type 250.
- **AccessForbiddenMotorvehicles** Type 251.
- **AccessForbiddenTrucks** Type 253.
- **AccessForbiddenBicycle** Type 254.
- **AccessForbiddenWeight** Type 263.
- **AccessForbiddenWidth** Type 264.
- **AccessForbiddenHeight** Type 265.
- **AccessForbiddenWrongDirection** Type 267.
- **ForbiddenUTurn** Type 272.
- **MaximumSpeed** Type 274.
- **ForbiddenOvertakingMotorvehicles** Type 276.
- **ForbiddenOvertakingTrucks** Type 277.
- **AbsoluteNoStop** Type 283.
- **RestrictedStop** Type 286.
- **HasWayNextIntersection** Type 301.
- **PriorityWay** Type 306.
- **PriorityWayEnd** Type 307.
- **CityBegin** Type 310.
- **CityEnd** Type 311.
- **Highway** Type 330.
- **RecomendedSpeed** Type 380.
- **RecomendedSpeedEnd** Type 381.

carla.LaneChange

Class that defines the permission to turn either left, right, both or none (meaning only going straight is allowed). This information is stored for every carla.Waypoint according to the OpenDRIVE file. In this recipe the user creates a waypoint for a current vehicle position and learns which turns are permitted.

Instance Variables

- **NONE** Traffic rules do not allow turning right or left, only going straight.
- **Right** Traffic rules allow turning right.
- **Left** Traffic rules allow turning left.
- **Both** Traffic rules allow turning either right or left.

carla.LaneInvasionEvent

Inherited from carla.SensorData Class that defines lanes invasion for sensor.other.lane_invasion. It works only client-side and is dependant on OpenDRIVE to provide reliable information. The sensor creates one of this every time there is a lane invasion, which may be more than once per simulation step. Learn more about this [here](#).

Instance Variables

- **actor** (carla.Actor) Gets the actor the sensor is attached to, the one that invaded another lane.
- **crossed_lane_markings** (*list* carla.LaneMarking) List of lane markings that have been crossed and detected by the sensor.

Methods

Dunder methods

- **__str__(self)**

carla.LaneMarking

Class that gathers all the information regarding a lane marking according to OpenDRIVE 1.4 standard standard.

Instance Variables

- **color** (carla.LaneMarkingColor) Actual color of the marking.
- **lane_change** (carla.LaneChange) Permissions for said lane marking to be crossed.
- **type** (carla.LaneMarkingType) Lane marking type.

- **width** (*float*) Horizontal lane marking thickness.

carla.LaneMarkingColor

Class that defines the lane marking colors according to OpenDRIVE 1.4.

Instance Variables

- **Standard** White by default.
- **Blue**
- **Green**
- **Red**
- **White**
- **Yellow**
- **Other**

carla.LaneMarkingType

Class that defines the lane marking types accepted by OpenDRIVE 1.4. Take a look at this recipe where the user creates a carla.Waypoint for a vehicle location and retrieves from it the information about adjacent lane markings.

Note on double types: Lane markings are defined under the OpenDRIVE standard that determines whereas a line will be considered “BrokenSolid” or “SolidBroken”. For each road there is a center lane marking, defined from left to right regarding the lane’s directions. The rest of the lane markings are defined in order from the center lane to the closest outside of the road.

Instance Variables

- **NONE**
- **Other**
- **Broken**
- **Solid**
- **SolidSolid**
- **SolidBroken**
- **BrokenSolid**
- **BrokenBroken**
- **BottsDots**
- **Grass**
- **Curb**

carla.LaneType

Class that defines the possible lane types accepted by OpenDRIVE 1.4. This standards define the road information. For instance in this recipe the user creates a carla.Waypoint for the current location of a vehicle and uses it to get the current and adjacent lane types.

Instance Variables

- **NONE**
- **Driving**
- **Stop**
- **Shoulder**
- **Biking**
- **Sidewalk**
- **Border**
- **Restricted**
- **Parking**
- **Bidirectional**
- **Median**
- **Special1**
- **Special2**
- **Special3**
- **RoadWorks**
- **Tram**
- **Rail**
- **Entry**

- **Exit**
- **OffRamp**
- **OnRamp**
- **Any** Every type except for NONE.

carla.LidarDetection

Data contained inside acarla.LidarMeasurement. Each of these represents one of the points in the cloud with its location and its associated intensity.

Instance Variables

- **point** (carla.Location – meters_) Point in xyz coordinates.
- **intensity** (float) Computed intensity for this point as a scalar value between [0.0 , 1.0].

Methods

Dunder methods

- **__str__(self**)

carla.LidarMeasurement

Inherited from carla.SensorData Class that defines the LIDAR data retrieved by a sensor.lidar.ray_cast. This essentially simulates a rotating LIDAR using ray-casting. Learn more about this [here](#).

Instance Variables

- **channels** (int) Number of lasers shot.
- **horizontal_angle** (float – radians) Horizontal angle the LIDAR is rotated at the time of the measurement.
- **raw_data** (bytes) Received list of 4D points. Each point consists of [x,y,z] coordinates plus the intensity computed for that point.

Methods

- **save_to_disk(self, path)** Saves the point cloud to disk as a .ply file describing data from 3D scanners. The files generated are ready to be used within MeshLab, an open source system for processing said files. Just take into account that axis may differ from Unreal Engine and so, need to be reallocated.
 - **Parameters:**
 - * **path** (str)

Getters

- **get_point_count(self, channel)** Retrieves the number of points sorted by channel that are generated by this measure. Sorting by channel allows to identify the original channel for every point.
 - **Parameters:**
 - * **channel** (int)

Dunder methods

- **__getitem__(self, pos**=int)
- **__iter__(self**)
- **__len__(self**)
- **__setitem__(self, pos=int, detection**carla.LidarDetection)
- **__str__(self**)

carla.Light

This class exposes the lights that exist in the scene, except for vehicle lights. The properties of a light can be queried and changed at will. Lights are automatically turned on when the simulator enters night mode (sun altitude is below zero).

Instance Variables

- **color** (carla.Color_) Color of the light.
- **id** (int) Identifier of the light.
- **intensity** (float – lumens) Intensity of the light.
- **is_on** (bool) Switch of the light. It is **True** when the light is on. When the night mode starts, this is set to **True**.

- **location** (carla.Location – meters_) Position of the light.
- **light_group** (carla.LightGroup_) Group the light belongs to.
- **light_state** (carla.LightState_) State of the light. Summarizes its attributes, group, and if it is on/off.

Methods

- **turn_off(self)** Switches off the light.
- **turn_on(self)** Switches on the light.

Setters

- **set_color(self, color)** Changes the color of the light to **color**.
 - **Parameters:**
 - * **color** (carla.Color_)
- **set_intensity(self, intensity)** Changes the intensity of the light to **intensity**.
 - **Parameters:**
 - * **intensity** (*float – lumens*)
- **set_light_group(self, light_group)** Changes the light to the group **light_group**.
 - **Parameters:**
 - * **light_group** (carla.LightGroup_)
- **set_light_state(self, light_state)** Changes the state of the light to **light_state**. This may change attributes, group and turn the light on/off all at once.
 - **Parameters:**
 - * **light_state** (carla.LightState_)

carla.LightGroup

This class categorizes the lights on scene into different groups. These groups available are provided as a enum values that can be used as flags.

Note. So far, though there is a **vehicle** group, vehicle lights are not available as carla.Light objects. These have to be managed using carla.Vehicle and carla.VehicleLightState.

Instance Variables

- **None** All lights.
- **Vehicle**
- **Street**
- **Building**
- **Other**

carla.LightManager

This class handles the lights in the scene. Its main use is to get and set the state of groups or lists of lights in one call. An instance of this class can be retrieved by the carla.World.get_light_manager().

Note. So far, though there is a **vehicle** group, vehicle lights are not available as carla.Light objects. These have to be managed using carla.Vehicle and carla.VehicleLightState.

Methods

- **is_active(self, lights)** Returns a list with booleans stating if the elements in **lights** are switched on/off.
 - **Parameters:**
 - * **lights** (*list*) – List of lights to be queried.
 - **Return:** *list(bool)*
- **turn_off(self, lights)** Switches off all the lights in **lights**.
 - **Parameters:**
 - * **lights** (*list*) – List of lights to be switched off.
- **turn_on(self, lights)** Switches on all the lights in **lights**.
 - **Parameters:**
 - * **lights** (*list*) – List of lights to be switched on.

Getters

- **get_all_lights(self, light_group=carla.LightGroup.None)** Returns a list containing the lights in a certain group. By default, the group is None.
 - **Parameters:**

- * `light_group` (`carla.LightGroup_`) – Group to filter the lights returned. Default is `None`.
 - **Return:** `listcarla.Light`
- **get_color(self, lights)** Returns a list with the colors of every element in `lights`.
 - **Parameters:**
 - * `lights` (`listcarla.Light`) – List of lights to be queried.
 - **Return:** `listcarla.Color`
 - **Setter:** `carla.LightManager.set_color_`
- **get_intensity(self, lights)** Returns a list with the intensity of every element in `lights`.
 - **Parameters:**
 - * `lights` (`listcarla.Light`) – List of lights to be queried.
 - **Return:** `list(float) – lumens`
 - **Setter:** `carla.LightManager.set_intensity_`
- **get_light_group(self, lights)** Returns a list with the group of every element in `lights`.
 - **Parameters:**
 - * `lights` (`listcarla.Light`) – List of lights to be queried.
 - **Return:** `listcarla.LightGroup`
 - **Setter:** `carla.LightManager.set_light_group_`
- **get_light_state(self, lights)** Returns a list with the state of all the attributes of every element in `lights`.
 - **Parameters:**
 - * `lights` (`listcarla.Light`) – List of lights to be queried.
 - **Return:** `listcarla.LightState`
 - **Setter:** `carla.LightManager.set_light_state_`
- **get_turned_off_lights(self, light_group)** Returns a list containing lights switched off in the scene, filtered by group.
 - **Parameters:**
 - * `light_group` (`carla.LightGroup_`) – List of lights to be queried.
 - **Return:** `listcarla.Light`
- **get_turned_on_lights(self, light_group)** Returns a list containing lights switched on in the scene, filtered by group.
 - **Parameters:**
 - * `light_group` (`carla.LightGroup_`) – List of lights to be queried.
 - **Return:** `listcarla.Light`

Setters

- **set_active(self, lights, active)** Switches on/off the elements in `lights`.
 - **Parameters:**
 - * `lights` (`listcarla.Light`) – List of lights to be switched on/off.
 - * `active` (`list(bool)`) – List of booleans to be applied.
- **set_color(self, lights, color)** Changes the color of the elements in `lights` to `color`.
 - **Parameters:**
 - * `lights` (`listcarla.Light`) – List of lights to be changed.
 - * `color` (`carla.Color_`) – Color to be applied.
 - **Getter:** `carla.LightManager.get_color_`
- **set_colors(self, lights, colors)** Changes the color of each element in `lights` to the corresponding in `colors`.
 - **Parameters:**
 - * `lights` (`listcarla.Light`) – List of lights to be changed.
 - * `colors` (`listcarla.Color`) – List of colors to be applied.
- **set_intensity(self, lights, intensity)** Changes the intensity of every element in `lights` to `intensity`.
 - **Parameters:**
 - * `lights` (`listcarla.Light`) – List of lights to be changed.
 - * `intensity` (`float – lumens`) – Intensity to be applied.
 - **Getter:** `carla.LightManager.get_intensity_`
- **set_intensities(self, lights, intensities)** Changes the intensity of each element in `lights` to the corresponding in `intensities`.
 - **Parameters:**
 - * `lights` (`listcarla.Light`) – List of lights to be changed.
 - * `intensities` (`_list(float) – lumens_`) – List of intensities to be applied.
- **set_light_group(self, lights, light_group)** Changes the group of every element in `lights` to `light_group`.
 - **Parameters:**
 - * `lights` (`listcarla.Light`) – List of lights to be changed.
 - * `light_group` (`carla.LightGroup_`) – Group to be applied.

- **Getter:** carla.LightManager.get_light_group_
- **set_light_groups(self, lights, light_groups)** Changes the group of each element in `lights` to the corresponding in `light_groups`.
 - **Parameters:**
 - * `lights` (`listcarla.Light`) – List of lights to be changed.
 - * `light_groups` (`listcarla.LightGroup`) – List of groups to be applied.
- **set_light_state(self, lights, light_state)** Changes the state of the attributes of every element in `lights` to `light_state`.
 - **Parameters:**
 - * `lights` (`listcarla.Light`) – List of lights to be changed.
 - * `light_state` (`carla.LightState_`) – State of the attributes to be applied.
 - **Getter:** carla.LightManager.get_light_state_
- **set_light_states(self, lights, light_states)** Changes the state of the attributes of each element in `lights` to the corresponding in `light_states`.
 - **Parameters:**
 - * `lights` (`listcarla.Light`) – List of lights to be changed.
 - * `light_states` (`listcarla.LightState`) – List of state of the attributes to be applied.

carla.LightState

This class represents all the light variables except the identifier and the location, which are should to be static. Using this class allows to manage all the parametrization of the light in one call.

Instance Variables

- **intensity** (*float – lumens*) Intensity of a light.
- **color** (`carla.Color_`) Color of a light.
- **group** (`carla.LightGroup_`) Group a light belongs to.
- **active** (*bool*) Switch of a light. It is **True** when the light is on.

Methods

- ****__init__(self, intensity=0.0, colorcarla.Color(), groupcarla.LightGroup.None, active**=False)**
 - **Parameters:**
 - * `intensity` (*float – lumens*) – Intensity of the light. Default is 0.0.
 - * `color` (`carla.Color_`) – Color of the light. Default is black.
 - * `group` (`carla.LightGroup_`) – Group the light belongs to. Default is the generic group `None`.
 - * `active` (*bool*) – Swith of the light. Default is `False`, light is off.

carla.Location

Inherited from `carla.Vector3D_` Represents a spot in the world.

Instance Variables

- **x** (*float – meters*) Distance from origin to spot on X axis.
- **y** (*float – meters*) Distance from origin to spot on Y axis.
- **z** (*float – meters*) Distance from origin to spot on Z axis.

Methods

- ****__init__(self, x=0.0, y=0.0, z**=0.0)**
 - **Parameters:**
 - * `x` (*float*)
 - * `y` (*float*)
 - * `z` (*float*)
- **distance(self, location)** Returns Euclidean distance from this location to another one.
 - **Parameters:**
 - * `location` (`carla.Location_`) – The other point to compute the distance with.
 - **Return:** *float – meters*

Dunder methods

- ****__eq__(self, other**carla.Location)** Returns **True** if both locations are the same point in space.
 - **Return:** *bool*
- ****__ne__(self, other**carla.Location)** Returns **True** if both locations are different points in space.

- **Return:** *bool*
- ****str(self**)** Parses the axis' values to string.
– **Return:** *str*

carla.Map

Class containing the road information and waypoint managing. Data is retrieved from an OpenDRIVE file that describes the road. A query system is defined which works hand in hand with carla.Waypoint to translate geometrical information from the .xodr to natural world points. CARLA is currently working with OpenDRIVE 1.4 standard.

Instance Variables

- **name (str)** The name of the map. It corresponds to the .umap from Unreal Engine that is loaded from a CARLA server, which then references to the .xodr road description.

Methods

- ****__init__(self, name, xodr_content**)** Constructor for this class. Though a map is automatically generated when initializing the world, using this method in no-rendering mode facilitates working with an .xodr without any CARLA server running.
 - **Parameters:**
 - * **name (str)** – Name of the current map.
 - * **xodr_content (str)** – .xodr content in string format.
 - **Return:** *listcarla.Transform*
- **generate_waypoints(self, distance)** Returns a list of waypoints with a certain distance between them for every lane and centered inside of it. Waypoints are not listed in any particular order. Remember that waypoints closer than 2cm within the same road, section and lane will have the same identifier.
 - **Parameters:**
 - * **distance (float – meters)** – Approximate distance between waypoints.
 - **Return:** *listcarla.Waypoint*
- **save_to_disk(self, path)** Saves the .xodr OpenDRIVE file of the current map to disk.
 - **Parameters:**
 - * **path** – Path where the file will be saved.
- **to_opendrive(self)** Returns the .xodr OpenDRIVe file of the current map as string.
 - **Return:** *str*
- **transform_to_geolocation(self, location)** Converts a given **location**, a point in the simulation, to acarla.GeoLocation, which represents world coordinates. The geographical location of the map is defined inside OpenDRIVE within the tag .
 - **Parameters:**
 - * **location (carla.Location_)**
 - **Return:** *carla.GeoLocation_*

Getters

- **get_all_landmarks(self)** Returns all the landmarks in the map. Landmarks retrieved using this method have a **null** waypoint.
 - **Return:** *listcarla.Landmark*
- **get_all_landmarks_from_id(self, opendrive_id)** Returns the landmarks with a certain OpenDRIVE ID. Landmarks retrieved using this method have a **null** waypoint.
 - **Parameters:**
 - * **opendrive_id (string)** – The OpenDRIVE ID of the landmarks.
 - **Return:** *listcarla.Landmark*
- **get_all_landmarks_of_type(self, type)** Returns the landmarks of a specific type. Landmarks retrieved using this method have a **null** waypoint.
 - **Parameters:**
 - * **type (string)** – The type of the landmarks.
 - **Return:** *listcarla.Landmark*
- **get_landmark_group(self, landmark)** Returns the landmarks in the same group as the specified landmark (including itself). Returns an empty list if the landmark does not belong to any group.
 - **Parameters:**
 - * **landmark (carla.Landmark_)** – A landmark that belongs to the group.
 - **Return:** *listcarla.Landmark*
- **get_spawn_points(self)** Returns a list of recommendations made by the creators of the map to be used as spawning points for the vehicles. The list includescarla.Transform objects with certain location and orientation.

Said locations are slightly on-air in order to avoid Z-collisions, so vehicles fall for a bit before starting their way.

– **Return:** `list(carla.Transform)`

- **get_topology(self)** Returns a list of tuples describing a minimal graph of the topology of the OpenDRIVE file. The tuples contain pairs of waypoints located either at the point a road begins or ends. The first one is the origin and the second one represents another road end that can be reached. This graph can be loaded into NetworkX to work with. Output could look like this: `[(w0, w1), (w0, w2), (w1, w3), (w2, w3), (w0, w4)]`.

– **Return:** `list(tuple(carla.Waypoint, carla.Waypoint))`

- **get_waypoint(self, location, project_to_road=True, lane_type=carla.LaneType.Driving)** Returns a waypoint that can be located in an exact location or translated to the center of the nearest lane. Said lane type can be defined using flags such as `LaneType.Driving` & `LaneType.Shoulder`. The method will return `None` if the waypoint is not found, which may happen only when trying to retrieve a waypoint for an exact location. That eases checking if a point is inside a certain road, as otherwise, it will return the corresponding waypoint.

– **Parameters:**

- * `location` (`carla.Location` – meters_) – Location used as reference for the `carla.Waypoint`.
- * `project_to_road` (`bool`) – If `True`, the waypoint will be at the center of the closest lane. This is the default setting. If `False`, the waypoint will be exactly in `location`. `None` means said location does not belong to a road.
- * `lane_type` (`carla.LaneType`) – Limits the search for nearest lane to one or various lane types that can be flagged.

– **Return:** `carla.Waypoint`

- **get_waypoint_xodr(self, road_id, lane_id, s)** Returns a waypoint if all the parameters passed are correct. Otherwise, returns `None`.

– **Parameters:**

- * `road_id` (`int`) – ID of the road to get the waypoint.
- * `lane_id` (`int`) – ID of the lane to get the waypoint.
- * `s` (`float – meters`) – Specify the length from the road start.

– **Return:** `carla.Waypoint`

Dunder methods

- `__str__(self)`

carla.ObstacleDetectionEvent

Inherited from `carla.SensorData` Class that defines the obstacle data for sensor.other.obstacle. Learn more about this here.

Instance Variables

- `actor` (`carla.Actor`) The actor the sensor is attached to.
- `other_actor` (`carla.Actor`) The actor or object considered to be an obstacle.
- `distance` (`float – meters`) Distance between `actor` and `other`.

Methods

Dunder methods

- `__str__(self)`

carla.OpendriveGenerationParameters

This class defines the parameters used when generating a world using an OpenDRIVE file.

Instance Variables

- `vertex_distance` (`float`) Distance between vertices of the mesh generated. **Default is 2.0**.
- `max_road_length` (`float`) Max road length for a single mesh portion. The mesh of the map is divided into portions, in order to avoid propagating issues. **Default is 50.0**.
- `wall_height` (`float`) Height of walls created on the boundaries of the road. These prevent vehicles from falling off the road. **Default is 1.0**.
- `additional_width` (`float`) Additional width applied to junction lanes. Complex situations tend to occur at junctions, and a little increase can prevent vehicles from falling off the road. **Default is 0.6**.
- `smooth_junctions` (`bool`) If `True`, the mesh at junctions will be smoothed to prevent issues where roads block other roads. **Default is True**.
- `enable_mesh_visibility` (`bool`) If `True`, the road mesh will be rendered. Setting this to `False` should reduce the rendering overhead. **Default is True**.

- **enable_pedestrian_navigation** (*bool*) If **True**, Pedestrian navigation will be enabled using Recast tool. For very large maps it is recommended to disable this option. **Default is True.**

carla.Osm2Odr

Class that converts an OpenStreetMap map to OpenDRIVE format, so that it can be loaded in CARLA. Find out more about this feature in the docs.

Methods

- **convert(osm_file, settings)** Takes the content of an .osm file (OpenStreetMap format) and returns the content of the .xodr (OpenDRIVE format) describing said map. Some parameterization is passed to do the conversion.
 - **Parameters:**
 - * **osm_file** (*str*) – The content of the input OpenStreetMap file parsed as string.
 - * **settings** (`carla.OSM2ODRSettings_`) – Parameterization for the conversion.
 - **Return:** *str*

carla.Osm2OdrSettings

Helper class that contains the parameterization that will be used by `carla.Osm2Odr` to convert an OpenStreetMap map to OpenDRIVE format. Find out more about this feature in the docs.

Instance Variables

- **use_offsets** (*bool*) Enables the use of offset for the conversion. The offset will move the origin position of the map. Default value is **False**.
- **offset_x** (*float – meters*) Offset in the X axis. Default value is **0.0**.
- **offset_y** (*float – meters*) Offset in the Y axis. Default value is **0.0**.
- **default_lane_width** (*float – meters*) Width of the lanes described in the resulting XODR map. Default value is **4.0**.
- **elevation_layer_height** (*float – meters*) Defines the height separating two different OpenStreetMap layers. Default value is **0.0**.

carla.RadarDetection

Data contained inside `acarla.RadarMeasurement`. Each of these represents one of the points in the cloud that a `sensor.other.radar` registers and contains the distance, angle and velocity in relation to the radar.

Instance Variables

- **altitude** (*float – radians*) Altitude angle of the detection.
- **azimuth** (*float – radians*) Azimuth angle of the detection.
- **depth** (*float – meters*) Distance from the sensor to the detection position.
- **velocity** (*float – m/s*) The velocity of the detected object towards the sensor.

Methods

Dunder methods

- **__str__(self)**

carla.RadarMeasurement

Inherited from carla.SensorData Class that defines and gathers the measures registered by a `sensor.other.radar`, representing a wall of points in front of the sensor with a distance, angle and velocity in relation to it. The data consists of `acarla.RadarDetection` array. Learn more about this here.

Instance Variables

- **raw_data** (*bytes*) The complete information of the `carla.RadarDetection` the radar has registered.

Methods

Getters

- **get_detection_count(self)** Retrieves the number of entries generated, same as `__str__()`.

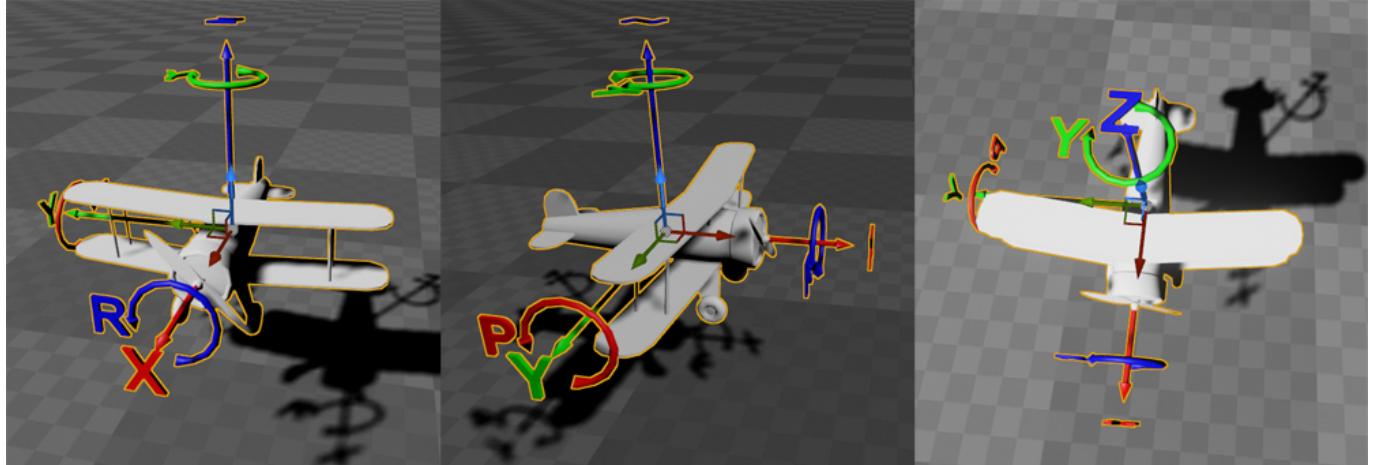
Dunder methods

- **__getitem__(self, pos=int)**

- `__iter__(self**)`
- `__len__(self**)`
- `__setitem__(self, pos=int, detection**carla.RadarDetection)`
- `__str__(self**)`

carla.Rotation

Class that represents a 3D rotation and therefore, an orientation in space. CARLA uses the Unreal Engine coordinates system. This is a Z-up left-handed system. The constructor method follows a specific order of declaration: (pitch, yaw, roll), which corresponds to (Y-rotation,Z-rotation,X-rotation).



Unreal Engine's coordinates system.

Instance Variables

- `pitch (float – degrees)` Y-axis rotation angle.
- `yaw (float – degrees)` Z-axis rotation angle.
- `roll (float – degrees)` X-axis rotation angle.

Methods

- `__init__(self, pitch=0.0, yaw=0.0, roll**=0.0)`
 - **Parameters:**
 - * `pitch (float – degrees)` – Y-axis rotation angle.
 - * `yaw (float – degrees)` – Z-axis rotation angle.
 - * `roll (float – degrees)` – X-axis rotation angle.
 - **Warning:** The declaration order is different in CARLA (pitch,yaw,roll), and in the Unreal Engine Editor (roll,pitch,yaw). When working in a build from source, don't mix up the axes' rotations.

Getters

- `get_forward_vector(self)` Computes the vector pointing forward according to the rotation of the object.
 - **Return:** carla.Vector3D_
- `get_right_vector(self)` Computes the vector pointing to the right according to the rotation of the object.
 - **Return:** carla.Vector3D_
- `get_up_vector(self)` Computes the vector pointing upwards according to the rotation of the object.
 - **Return:** carla.Vector3D_

Dunder methods

- `__eq__(self, other**carla.Rotation)` Returns **True** if both rotations represent the same orientation for every axis.
 - **Return:** bool
- `__ne__(self, other**carla.Rotation)` Returns **True** if both rotations represent the same orientation for every axis.
 - **Return:** bool
- `__str__(self**)` Parses the axis' orientations to string.

carla.RssActorConstellationData

Data structure that is provided within the callback registered by `RssSensor.register_actor_constellation_callback()`.

Instance Variables

- **ego_match_object** (____) The ego map matched information.
- **ego_route** (____) The ego route.
- **ego_dynamics_on_route** (carla.RssEgoDynamicsOnRoute_) Current ego vehicle dynamics regarding the route.
- **other_match_object** (____) The other object's map matched information. This is only valid if 'other_actor' is not 'None'.
- **other_actor** (carla.Actor_) The other actor. This is 'None' in case of query of default parameters or articial objects of kind with no dedicated carla.Actor' (as e.g. for theroad boundaries at the moment).

Methods

Dunder methods

- ****__str__(self**)**

carla.RssActorConstellationResult

Data structure that should be returned by the callback registered by RssSensor.register_actor_constellation_callback().

Instance Variables

- **rss_calculation_mode** (____) The calculation mode to be applied with the actor.
- **restrict_speed_limit_mode** (____) The mode for restricting speed limit.
- **ego_vehicle_dynamics** (____) The RSS dynamics to be applied for the ego vehicle.
- **actor_object_type** (____) The RSS object type to be used for the actor.
- **actor_dynamics** (____) The RSS dynamics to be applied for the actor.

Methods

Dunder methods

- ****__str__(self**)**

carla.RssEgoDynamicsOnRoute

Part of the data contained inside acarla.RssResponse describing the state of the vehicle. The parameters include its current dynamics, and how it is heading regarding the target route.

Instance Variables

- **ego_speed** (____) The ego vehicle's speed.
- **min_stopping_distance** (____) The current minimum stopping distance.
- **ego_center** (____) The considered enu position of the ego vehicle.
- **ego_heading** (____) The considered heading of the ego vehicle.
- **ego_center_within_route** (bool) States if the ego vehicle's center is within the route.
- **crossing_border** (bool) States if the vehicle is already crossing one of the lane borders.
- **route_heading** (____) The considered heading of the route.
- **route_nominal_center** (____) The considered nominal center of the current route.
- **heading_diff** (____) The considered heading diff towards the route.
- **route_speed_lat** (____) The ego vehicle's speed component *lat* regarding the route.
- **route_speed_lon** (____) The ego vehicle's speed component *lon* regarding the route.
- **route_accel_lat** (____) The ego vehicle's acceleration component *lat* regarding the route.
- **route_accel_lon** (____) The ego vehicle's acceleration component *lon* regarding the route.
- **avg_route_accel_lat** (____) The ego vehicle's acceleration component *lat* regarding the route smoothed by an average filter.
- **avg_route_accel_lon** (____) The ego acceleration component *lon* regarding the route smoothed by an average filter.

Methods

Dunder methods

- ****__str__(self**)**

carla.RssLogLevel

Enum declaration used in carla.RssSensor to set the log level.

Instance Variables

- **trace**
- **debug**
- **info**
- **warn**
- **err**
- **critical**
- **off**

carla.RssResponse

Inherited from **carla.SensorData** Class that contains the output of acarla.RssSensor. This is the result of the RSS calculations performed for the parent vehicle of the sensor.

Acarla.RssRestrictor will use the data to modify thecarla.VehicleControl of the vehicle.

Instance Variables

- **response_valid** (*bool*) States if the response is valid. It is **False** if calculations failed or an exception occurred.
- **proper_response** (*__*) The proper response that the RSS calculated for the vehicle.
- **rss_state_snapshot** (*__*) Detailed RSS states at the current moment in time.
- **ego_dynamics_on_route** (*carla.RssEgoDynamicsOnRoute*) Current ego vehicle dynamics regarding the route.
- **world_model** (*__*) World model used for calculations.
- **situation_snapshot** (*__*) Detailed RSS situations extracted from the world model.

Methods

Dunder methods

- **** __str__(self**)**

carla.RssRestrictor

These objects apply restrictions to acarla.VehicleControl. It is part of the CARLA implementation of the C++ Library for Responsibility Sensitive Safety. This class works hand in hand with arss sensor, which provides the data of the restrictions to be applied.

Methods

- **restrict_vehicle_control(self, vehicle_control, restriction, ego_dynamics_on_route, vehicle_physics)** Applies the safety restrictions given by acarla.RssSensor to acarla.VehicleControl.
 - **Parameters:**
 - * **vehicle_control** (*carla.VehicleControl*) – The input vehicle control to be restricted.
 - * **restriction** (*__*) – Part of the response generated by the sensor. Contains restrictions to be applied to the acceleration of the vehicle.
 - * **ego_dynamics_on_route** (*carla.RssEgoDynamicsOnRoute*) – Part of the response generated by the sensor. Contains dynamics and heading of the vehicle regarding its route.
 - * **vehicle_physics** (*carla.RssEgoDynamicsOnRoute*) – The current physics of the vehicle. Used to apply the restrictions properly.
 - **Return:** *carla.VehicleControl*

carla.RssRoadBoundariesMode

Enum declaration used in carla.RssSensor to enable or disable the stay on road feature. In summary, this feature considers the road boundaries as virtual objects. The minimum safety distance check is applied to these virtual walls, in order to make sure the vehicle does not drive off the road.

Instance Variables

- **On** Enables the *stay on road* feature.
- **Off** Disables the *stay on road* feature.

carla.RssSensor

Inherited from `carla.Sensor` This sensor works a bit differently than the rest. Take look at the specific documentation, and the RSS sensor reference to gain full understanding of it.

The RSS sensor uses world information, and a RSS library to make safety checks on a vehicle. The output retrieved by the sensor is `carla.RssResponse`. This will be used by `carla.RssRestrictor` to modify `carla.VehicleControl` before applying it to a vehicle.

Instance Variables

- **ego_vehicle_dynamics** (`libad_rss_python.RssDynamics`) States the RSS parameters that the sensor will consider for the ego vehicle if no actor constellation callback is registered.
- **other_vehicle_dynamics** (`libad_rss_python.RssDynamics`) States the RSS parameters that the sensor will consider for the rest of vehicles if no actor constellation callback is registered.
- **pedestrian_dynamics** (`libad_rss_python.RssDynamics`) States the RSS parameters that the sensor will consider for pedestrians if no actor constellation callback is registered.
- **road_boundaries_mode** (`carla.RssRoadBoundariesMode`) Switches the stay on road feature. By default is **On**.
- **routing_targets** (`vector<carla.Transform>`) The current list of targets considered to route the vehicle. If no routing targets are defined, a route is generated at random.

Methods

- **append_routing_target(self, routing_target)** Appends a new target position to the current route of the vehicle.
 - **Parameters:**
 - * `routing_target` (`carla.Transform`) – New target point for the route. Choose these after the intersections to force the route to take the desired turn.
- **reset_routing_targets(self)** Erases the targets that have been appended to the route.
- **drop_route(self)** Discards the current route. If there are targets remaining in `routing_targets`, creates a new route using those. Otherwise, a new route is created at random.
- **register_actor_constellation_callback(self, callback)** Register a callback to customize `carla.RssActorConstellation`. By this callback the settings of RSS parameters are done per actor constellation and the settings (`ego_vehicle_dynamics`, `other_vehicle_dynamics` and `pedestrian_dynamics`) have no effect.
 - **Parameters:**
 - * `callback` – The function to be called whenever a RSS situation is about to be calculated.

Setters

- **set_log_level(self, log_level)** Sets the log level.
 - **Parameters:**
 - * `log_level` (`carla.RssLogLevel`) – New log level.

Dunder methods

- **__str__(self)**

carla.SemanticLidarDetection

Data contained inside `carla.SemanticLidarMeasurement`. Each of these represents one of the points in the cloud with its location, the cosine of the incident angle, index of the object hit, and its semantic tag.

Instance Variables

- **point** (`carla.Location` – meters) [x,y,z] coordinates of the point.
- **cos_inc_angle** (`float`) Cosine of the incident angle between the ray, and the normal of the hit object.
- **object_idx** (`uint`) ID of the actor hit by the ray.
- **object_tag** (`uint`) Semantic tag of the component hit by the ray.

Methods

Dunder methods

- **__str__(self)**

carla.SemanticLidarMeasurement

Inherited from carla.SensorData Class that defines the semantic LIDAR data retrieved by a sensor.lidar.ray_cast_semantic. This essentially simulates a rotating LIDAR using ray-casting. Learn more about thishere.

Instance Variables

- **channels** (*int*) Number of lasers shot.
- **horizontal_angle** (*float – radians*) Horizontal angle the LIDAR is rotated at the time of the measurement.
- **raw_data** (*bytes*) Received list of raw detection points. Each point consists of [x,y,z] coordinates plus the cosine of the incident angle, the index of the hit actor, and its semantic tag.

Methods

- **save_to_disk(self, path)** Saves the point cloud to disk as a .ply file describing data from 3D scanners. The files generated are ready to be used within MeshLab, an open-source system for processing said files. Just take into account that axis may differ from Unreal Engine and so, need to be reallocated.
 - **Parameters:**
 - * *path* (*str*)

Getters

- **get_point_count(self, channel)** Retrieves the number of points sorted by channel that are generated by this measure. Sorting by channel allows to identify the original channel for every point.
 - **Parameters:**
 - * *channel* (*int*)

Dunder methods

- **__getitem__(self, pos**=int)**
- **__iter__(self**)**
- **__len__(self**)**
- **__setitem__(self, pos=int, detection**carla.SemanticLidarDetection)**
- **__str__(self**)**

carla.Sensor

Inherited from carla.Actor Sensors compound a specific family of actors quite diverse and unique. They are normally spawned as attachment/sons of a vehicle (take a look atcarla.World to learn about actor spawning). Sensors are thoroughly designed to retrieve different types of data that they are listening to. The data they receive is shaped as different subclasses inherited fromcarla.SensorData (depending on the sensor).

Most sensors can be divided in two groups: those receiving data on every tick (cameras, point clouds and some specific sensors) and those who only receive under certain circumstances (trigger detectors). CARLA provides a specific set of sensors and their blueprint can be found incarla.BlueprintLibrary. All the information on their preferences and settlement can be foundhere, but the list of those available in CARLA so far goes as follow. Receive data on every tick. -Depth camera. -Gnss sensor. -IMU sensor. -Lidar raycast. -SemanticLidar raycast. -Radar. -RGB camera. -RSS sensor. -Semantic Segmentation camera. Only receive data when triggered. -Collision detector. -Lane invasion detector. -Obstacle detector.

Instance Variables

- **is_listening** (*boolean*) When True the sensor will be waiting for data.

Methods

- **listen(self, callback)** The function the sensor will be calling to every time a new measurement is received. This function needs for an argument containing an object typecarla.SensorData to work with.
 - **Parameters:**
 - * *callback* (*function*) – The called function with one argument containing the sensor data.
- **stop(self)** Commands the sensor to stop listening for data.

Dunder methods

- **__str__(self**)**

carla.SensorData

Base class for all the objects containing data generated by acarla.Sensor. This objects should be the argument of the function said sensor is listening to, in order to work with them. Each of these sensors needs for a specific type of sensor data. Hereunder is a list of the sensors and their corresponding data.

- Cameras (RGB, depth and semantic segmentation):carla.Image.
- Collision detector:carla.CollisionEvent.
- GNSS sensor:carla.GnssMeasurement.
- IMU sensor:carla.IMUMeasurement.
- Lane invasion detector:carla.LaneInvasionEvent.
- LIDAR sensor:carla.LidarMeasurement.
- Obstacle detector:carla.ObstacleDetectionEvent.
- Radar sensor:carla.RadarMeasurement.
- RSS sensor:carla.RssResponse.
- Semantic LIDAR sensor:carla.SemanticLidarMeasurement.

Instance Variables

- **frame** (*int*) Frame count when the data was generated.
- **timestamp** (*float – seconds*) Simulation-time when the data was generated.
- **transform** (*carla.Transform*) Sensor's transform when the data was generated.

carla.Timestamp

Class that contains time information for simulated data. This information is automatically retrieved as part of thecarla.WorldSnapshot the client gets on every frame, but might also be used in many other situations such as acarla.Sensor retrieveing data.

Instance Variables

- **frame** (*int*) The number of frames elapsed since the simulator was launched.
- **elapsed_seconds** (*float – seconds*) Simulated seconds elapsed since the beginning of the current episode.
- **delta_seconds** (*float – seconds*) Simulated seconds elapsed since the previous frame.
- **platform_timestamp** (*float – seconds*) Time register of the frame at which this measurement was taken given by the OS in seconds.

Methods

- **__init__(self, frame, elapsed_seconds, delta_seconds, platform_timestamp)**
 - **Parameters:**
 - * **frame** (*int*)
 - * **elapsed_seconds** (*float – seconds*)
 - * **delta_seconds** (*float – seconds*)
 - * **platform_timestamp** (*float – seconds*)

Dunder methods

- **__eq__(self, other)**carla.Timestamp
- **__ne__(self, other)**carla.Timestamp
- **__str__(self)**

carla.TrafficLight

Inherited from carla.TrafficSign A traffic light actor, considered a specific type of traffic sign. As traffic lights will mostly appear at junctions, they belong to a group which contains the different traffic lights in it. Inside the group, traffic lights are differentiated by their pole index.

Within a group the state of traffic lights is changed in a cyclic pattern: one index is chosen and it spends a few seconds in green, yellow and eventually red. The rest of the traffic lights remain frozen in red this whole time, meaning that there is a gap in the last seconds of the cycle where all the traffic lights are red. However, the state of a traffic light can be changed manually. Take a look at thisrecipe to learn how to do so.

Instance Variables

- **state** (*carla.TrafficLightState*) Current state of the traffic light.

Methods

- **freeze(self, freeze)** Stops all the traffic lights in the scene at their current state.
 - **Parameters:**
 - * **freeze** (*bool*)
- **is_frozen(self)** The client returns True if a traffic light is frozen according to last tick. The method does not call the simulator.
 - **Return:** *bool*

- **reset_group(self)** Resets the state of the traffic lights of the group to the initial state at the start of the simulation.
 - **Note:** This method calls the simulator.

Getters

- **get_elapsed_time(self)** The client returns the time in seconds since current light state started according to last tick. The method does not call the simulator.
 - **Return:** float – seconds
- **get_group_traffic_lights(self)** Returns all traffic lights in the group this one belongs to.
 - **Return:** list`carla.TrafficLight`
 - **Note:** This method calls the simulator.
- **get_pole_index(self)** Returns the index of the pole that identifies it as part of the traffic light group of a junction.
 - **Return:** int
- **get_state(self)** The client returns the state of the traffic light according to last tick. The method does not call the simulator.
 - **Return:** carla.TrafficLightState
 - **Setter:** carla.TrafficLight.set_state
- **get_green_time(self)** The client returns the time set for the traffic light to be green, according to last tick. The method does not call the simulator.
 - **Return:** float – seconds
 - **Setter:** carla.TrafficLight.set_green_time
- **get_red_time(self)** The client returns the time set for the traffic light to be red, according to last tick. The method does not call the simulator.
 - **Return:** float – seconds
 - **Setter:** carla.TrafficLight.set_red_time
- **get_yellow_time(self)** The client returns the time set for the traffic light to be yellow, according to last tick. The method does not call the simulator.
 - **Return:** float – seconds
 - **Setter:** carla.TrafficLight.set_yellow_time

Setters

- **set_state(self, state)** Sets a given state to a traffic light actor.
 - **Parameters:**
 - * `state` (carla.TrafficLightState)
 - **Getter:** carla.TrafficLight.get_state
- **set_green_time(self, green_time)**
 - **Parameters:**
 - * `green_time` (float – seconds) – Sets a given time for the green light to be active.
 - **Getter:** carla.TrafficLight.get_green_time
- **set_red_time(self, red_time)** Sets a given time for the red state to be active.
 - **Parameters:**
 - * `red_time` (float – seconds)
 - **Getter:** carla.TrafficLight.get_red_time
- **set_yellow_time(self, yellow_time)** Sets a given time for the yellow light to be active.
 - **Parameters:**
 - * `yellow_time` (float – seconds)
 - **Getter:** carla.TrafficLight.get_yellow_time

Dunder methods

- **__str__(self)**

`carla.TrafficLightState`

All possible states for traffic lights. These can either change at a specific time step or be changed manually. Take a look at `thisrecipe` to see an example.

Instance Variables

- **Red**
- **Yellow**
- **Green**

- Off
- Unknown

carla.TrafficManager

The traffic manager is a module built on top of the CARLA API in C++. It handles any group of vehicles set to autopilot mode to populate the simulation with realistic urban traffic conditions and give the chance to user to customize some behaviours. The architecture of the traffic manager is divided in five different goal-oriented stages and a PID controller where the information flows until eventually, acarla.VehicleControl is applied to every vehicle registered in a traffic manager. In order to learn more, visit the documentation regarding this module.

Methods

- **auto_lane_change(self, actor, enable)** Turns on or off lane changing behaviour for a vehicle.
 - **Parameters:**
 - * **actor** (carla.Actor_) – The vehicle whose settings are changed.
 - * **enable** (*bool*) – **True** is default and enables lane changes. **False** will disable them.
- **collision_detection(self, reference_actor, other_actor, detect_collision)** Tunes on/off collisions between a vehicle and another specific actor. In order to ignore all other vehicles, traffic lights or walkers, use the specific **ignore** methods described in this same section.
 - **Parameters:**
 - * **reference_actor** (carla.Actor_) – Vehicle that is going to ignore collisions.
 - * **other_actor** (carla.Actor_) – The actor that **reference_actor** is going to ignore collisions with.
 - * **detect_collision** (*bool*) – **True** is default and enables collisions. **False** will disable them.
- **distance_to_leading_vehicle(self, actor, distance)** Sets the minimum distance in meters that a vehicle has to keep with the others. The distance is in meters and will affect the minimum moving distance. It is computed from front to back of the vehicle objects.
 - **Parameters:**
 - * **actor** (carla.Actor_) – Vehicle whose minimum distance is being changed.
 - * **distance** (*float – meters*) – Meters between both vehicles.
- **force_lane_change(self, actor, direction)** Forces a vehicle to change either to the lane on its left or right, if existing, as indicated in **direction**. This method applies the lane change no matter what, disregarding possible collisions.
 - **Parameters:**
 - * **actor** (carla.Actor_) – Vehicle being forced to change lanes.
 - * **direction** (*bool*) – Destination lane. **True** is the one on the right and **False** is the left one.
- **global_distance_to_leading_vehicle(self, distance)** Sets the minimum distance in meters that vehicles have to keep with the rest. The distance is in meters and will affect the minimum moving distance. It is computed from center to center of the vehicle objects.
 - **Parameters:**
 - * **distance** (*float – meters*) – Meters between vehicles.
- **global_percentage_speed_difference(self, percentage)** Sets the difference the vehicle's intended speed and its current speed limit. Speed limits can be exceeded by setting the **perc** to a negative value. Default is 30. Exceeding a speed limit can be done using negative percentages.
 - **Parameters:**
 - * **percentage** (*float*) – Percentage difference between intended speed and the current limit.
- **ignore_lights_percentage(self, actor, perc)** During the traffic light stage, which runs every frame, this method sets the percent chance that traffic lights will be ignored for a vehicle.
 - **Parameters:**
 - * **actor** (carla.Actor_) – The actor that is going to ignore traffic lights.
 - * **perc** (*float*) – Between 0 and 100. Amount of times traffic lights will be ignored.
- **ignore_vehicles_percentage(self, actor, perc)** During the collision detection stage, which runs every frame, this method sets a percent chance that collisions with another vehicle will be ignored for a vehicle.
 - **Parameters:**
 - * **actor** (carla.Actor_) – The vehicle that is going to ignore other vehicles.
 - * **perc** (*float*) – Between 0 and 100. Amount of times collisions will be ignored.
- **ignore_walkers_percentage(self, actor, perc)** During the collision detection stage, which runs every frame, this method sets a percent chance that collisions with walkers will be ignored for a vehicle.
 - **Parameters:**
 - * **actor** (carla.Actor_) – The vehicle that is going to ignore walkers on scene.
 - * **perc** (*float*) – Between 0 and 100. Amount of times collisions will be ignored.
- **reset_traffic_lights(self)** Resets every traffic light in the map to its initial state.

- **vehicle_percentage_speed_difference(self, actor, percentage)** Sets the difference the vehicle's intended speed and its current speed limit. Speed limits can be exceeded by setting the `perc` to a negative value. Default is 30. Exceeding a speed limit can be done using negative percentages.

– **Parameters:**

- * `actor` (`carla.Actor_`) – Vehicle whose speed behaviour is being changed.
- * `percentage` (`float`) – Percentage difference between intended speed and the current limit.

Getters

- **get_port(self)** Returns the port where the Traffic Manager is connected. If the object is a TM-Client, it will return the port of its TM-Server. Read the documentation to learn the difference.

– **Return:** `uint16`

Setters

- **set_hybrid_physics_mode(self, enabled=False)** Enables or disables the hybrid physics mode. In this mode, vehicle's farther than a certain radius from the ego vehicle will have their physics disabled. Computation cost will be reduced by not calculating vehicle dynamics. Vehicles will be teleported.

– **Parameters:**

- * `enabled` (`bool`) – If **True**, enables the hybrid physics.

- **set_hybrid_mode_radius(self, r=70.0)** With hybrid physics on, changes the radius of the area of influence where physics are enabled.

– **Parameters:**

- * `r` (`float – meters`) – New radius where physics are enabled.

- **set_osm_mode(self, mode_switch=True)** Enables or disables the OSM mode. This mode allows the user to run TM in a map created with the OSM feature. These maps allow having dead-end streets. Normally, if vehicles cannot find the next waypoint, TM crashes. If OSM mode is enabled, it will show a warning, and destroy vehicles when necessary.

– **Parameters:**

- * `mode_switch` (`_bool_`) – If **True**, the OSM mode is enabled.

carla.TrafficSign

Inherited from carla.Actor_ Traffic signs appearing in the simulation except for traffic lights. These have their own class inherited from this `incarla.TrafficLight`. Right now, speed signs, stops and yields are mainly the ones implemented, but many others are borne in mind.

Instance Variables

- **trigger_volume** A `carla.BoundingBox` situated near a traffic sign where the `carla.Actor` who is inside can know about it.

carla.Transform

Class that defines a transformation, a combination of location and rotation, without scaling.

Instance Variables

- **location** (`carla.Location_`) Describes a point in the coordinate system.
- **rotation** (`carla.Rotation – degrees (pitch, yaw, roll)_`) Describes a rotation for an object according to Unreal Engine's axis system.

Methods

- ****__init__(self, location, rotation)****

– **Parameters:**

 - * `location` (`carla.Location_`)
 - * `rotation` (`carla.Rotation – degrees (pitch, yaw, roll)_`)
- **transform(self, in_point)** Translates a 3D point from local to global coordinates using the current transformation as frame of reference.

– **Parameters:**

 - * `in_point` (`carla.Location_`) – Location in the space to which the transformation will be applied.

Getters

- **get_forward_vector(self)** Computes a forward vector using the rotation of the object.

– **Return:** `carla.Vector3D_`

- **get_right_vector(self)** Computes a right vector using the rotation of the object.
– **Return:** carla.Vector3D
- **get_up_vector(self)** Computes an up vector using the rotation of the object.
– **Return:** carla.Vector3D
- **get_matrix(self)** Computes the 4-matrix representation of the transformation.
– **Return:** list(list(float))
- **get_inverse_matrix(self)** Computes the 4-matrix representation of the inverse transformation.
– **Return:** list(list(float))

Dunder methods

- **__eq__(self, other**carla.Transform)** Returns **True** if both location and rotation are equal for this and **other**.
– **Return:** bool
- **__ne__(self, other**carla.Transform)** Returns **True** if any location and rotation are not equal for this and **other**.
– **Return:** bool
- **__str__(self**)** Parses both location and rotation to string.
– **Return:** str

carla.Vector2D

Helper class to perform 2D operations.

Instance Variables

- **x (float)** X-axis value.
- **y (float)** Y-axis value.

Methods

- **__init__(self, x=0.0, y=0.0)**
– **Parameters:**
 * **x (float)**
 * **y (float)**

Dunder methods

- **__add__(self, other**carla.Vector2D)**
- **__sub__(self, other**carla.Vector2D)**
- **__mul__(self, other**carla.Vector2D)**
- **__truediv__(self, other**carla.Vector2D)**
- **__eq__(self, other**carla.Vector2D)** Returns **True** if values for every axis are equal.
– **Return:** bool
- **__ne__(self, other**carla.Vector2D)** Returns **True** if the value for any axis is different.
– **Return:** bool
- **__str__(self**)** Returns the axis values for the vector parsed as string.
– **Return:** str

carla.Vector3D

Helper class to perform 3D operations.

Instance Variables

- **x (float)** X-axis value.
- **y (float)** Y-axis value.
- **z (float)** Z-axis value.

Methods

- **__init__(self, x=0.0, y=0.0, z=0.0)**
– **Parameters:**
 * **x (float)**
 * **y (float)**
 * **z (float)**

Dunder methods

- `**__add__(self, other**carla.Vector3D)`
- `**__sub__(self, other**carla.Vector3D)`
- `**__mul__(self, other**carla.Vector3D)`
- `**__truediv__(self, other**carla.Vector3D)`
- `**__eq__(self, other**carla.Vector3D)` Returns **True** if values for every axis are equal.
– **Return:** `bool`
- `**__ne__(self, other**carla.Vector3D)` Returns **True** if the value for any axis is different.
– **Return:** `bool`
- `**__str__(self**)` Returns the axis values for the vector parsed as string.
– **Return:** `str`

carla.Vehicle

Inherited from carla.Actor One of the most important group of actors in CARLA. These include any type of vehicle from cars to trucks, motorbikes, vans, bycicles and also official vehicles such as police cars. A wide set of these actors is provided in `carla.BlueprintLibrary` to facilitate differente requirements. Vehicles can be either manually controlled or set to an autopilot mode that will be conducted client-side by the traffic manager.

Instance Variables

- **bounding_box** (`carla.BoundingBox`) Bounding box containing the geometry of the vehicle. Its location and rotation are relative to the vehicle it is attached to.

Methods

- **apply_control(self, control)** Applies a control object on the next tick, containing driving parameters such as throttle, steering or gear shifting.
– **Parameters:**
 * `control` (`carla.VehicleControl`)
- **apply_physics_control(self, physics_control)** Applies a physics control object in the next tick containing the parameters that define the vehicle as a corporeal body. E.g.: moment of inertia, mass, drag coefficient and many more.
– **Parameters:**
 * `physics_control` (`carla.VehiclePhysicsControl`)
- **is_at_traffic_light(self)** Vehicles will be affected by a traffic light when the light is red and the vehicle is inside its bounding box. The client returns whether a traffic light is affecting this vehicle according to last tick (it does not call the simulator).
– **Return:** `bool`

Getters

- **get_control(self)** The client returns the control applied in the last tick. The method does not call the simulator.
– **Return:** `carla.VehicleControl`
- **get_light_state(self)** Returns a flag representing the vehicle light state, this represents which lights are active or not.
– **Return:** `carla.VehicleLightState`
– **Setter:** `carla.Vehicle.set_light_state`
- **get_physics_control(self)** The simulator returns the last physics control applied to this vehicle.
– **Return:** `carla.VehiclePhysicsControl`
– **Warning:** *This method does call the simulator to retrieve the value.*
- **get_speed_limit(self)** The client returns the speed limit affecting this vehicle according to last tick (it does not call the simulator). The speed limit is updated when passing by a speed limit signal, so a vehicle might have none right after spawning.
– **Return:** `float - m/s`
- **get_traffic_light(self)** Retrieves the traffic light actor affecting this vehicle (if any) according to last tick. The method does not call the simulator.
– **Return:** `carla.TrafficLight`
- **get_traffic_light_state(self)** The client returns the state of the traffic light affecting this vehicle according to last tick. The method does not call the simulator. If no traffic light is currently affecting the vehicle, returns green.
– **Return:** `carla.TrafficLightState`

Setters

- **set_autopilot(self, enabled=True, port=8000)** Registers or deletes the vehicle from a Traffic Manager's list. When **True**, the Traffic Manager passed as parameter will move the vehicle around. The autopilot takes place client-side.
 - **Parameters:**
 - * **enabled (bool)**
 - * **port (uint16)** – The port of the TM-Server where the vehicle is to be registered or unlisted. If **None** is passed, it will consider a TM at default port 8000.
- **set_light_state(self, light_state)** Sets the light state of a vehicle using a flag that represents the lights that are on and off.
 - **Parameters:**
 - * **light_state (carla.VehicleLightState_)**
 - **Getter:** carla.Vehicle.get_light_state_

Dunder methods

- **__str__(self**)

carla.VehicleControl

Manages the basic movement of a vehicle using typical driving controls.

Instance Variables

- **throttle (float)** A scalar value to control the vehicle throttle [0.0, 1.0]. Default is 0.0.
- **steer (float)** A scalar value to control the vehicle steering [-1.0, 1.0]. Default is 0.0.
- **brake (float)** A scalar value to control the vehicle brake [0.0, 1.0]. Default is 0.0.
- **hand_brake (bool)** Determines whether hand brake will be used. Default is False.
- **reverse (bool)** Determines whether the vehicle will move backwards. Default is False.
- **manual_gear_shift (bool)** Determines whether the vehicle will be controlled by changing gears manually. Default is False.
- **gear (int)** States which gear is the vehicle running on.

Methods

- **__init__(self, throttle=**0.0**, steer=**0.0**, brake=**0.0**, hand_brake=**False**, reverse=**False**, manual_gear_shift=**False**, gear**=**0**)
 - **Parameters:**
 - * **throttle (float)** – Scalar value between [0.0,1.0].
 - * **steer (float)** – Scalar value between [0.0,1.0].
 - * **brake (float)** – Scalar value between [0.0,1.0].
 - * **hand_brake (bool)**
 - * **reverse (bool)**
 - * **manual_gear_shift (bool)**
 - * **gear (int)**

Dunder methods

- **__eq__(self, other**carla.VehicleControl)
- **__ne__(self, other**carla.VehicleControl)
- **__str__(self**)

carla.VehicleLightState

Class that recaps the state of the lights of a vehicle, these can be used as a flags. E.g: `VehicleLightState.HighBeam & VehicleLightState.Blinker` will return `True` when both are active. Lights are off by default in any situation and should be managed by the user via script. The blinkers blink automatically. *Warning: Right now, not all vehicles have been prepared to work with this functionality, this will be added to all of them in later updates.*

Instance Variables

- **NONE** All lights off.
- **Position**
- **LowBeam**
- **HighBeam**
- **Brake**
- **RightBlinker**

- **LeftBlinker**
- **Reverse**
- **Fog**
- **Interior**
- **Special1** This is reserved for certain vehicles that can have special lights, like a siren.
- **Special2** This is reserved for certain vehicles that can have special lights, like a siren.
- **All** All lights on.

carla.VehiclePhysicsControl

Summarizes the parameters that will be used to simulate a carla.Vehicle as a physical object. The specific settings for the wheels though are stipulated using carla.WheelPhysicsControl.

Instance Variables

- **torque_curve** (*listcarla.Vector2D*) Curve that indicates the torque measured in Nm for a specific RPM of the vehicle's engine.
- **max_rpm** (*float*) The maximum RPM of the vehicle's engine.
- **moi** (*_float – kg*m2_*) The moment of inertia of the vehicle's engine.
- **damping_rate_full_throttle** (*float*) Damping ratio when the throttle is maximum.
- **damping_rate_zero_throttle_clutch_engaged** (*float*) Damping ratio when the throttle is zero with clutch engaged.
- **damping_rate_zero_throttle_clutch_disengaged** (*float*) Damping ratio when the throttle is zero with clutch disengaged.
- **use_gear_autobox** (*bool*) If True, the vehicle will have an automatic transmission.
- **gear_switch_time** (*float – seconds*) Switching time between gears.
- **clutch_strength** (*_float – kg*m2/s_*) Clutch strength of the vehicle.
- **final_ratio** (*float*) Fixed ratio from transmission to wheels.
- **forward_gears** (*listcarla.GearPhysicsControl*) List of objects defining the vehicle's gears.
- **mass** (*float – kilograms*) Mass of the vehicle.
- **drag_coefficient** (*float*) Drag coefficient of the vehicle's chassis.
- **center_of_mass** (*carla.Vector3D – meters_*) Center of mass of the vehicle.
- **steering_curve** (*listcarla.Vector2D*) Curve that indicates the maximum steering for a specific forward speed.
- **wheels** (*listcarla.WheelPhysicsControl*) List of wheel physics objects. This list should have 4 elements, where index 0 corresponds to the front left wheel, index 1 corresponds to the front right wheel, index 2 corresponds to the back left wheel and index 3 corresponds to the back right wheel. For 2 wheeled vehicles, set the same values for both front and back wheels.

Methods

- **__init__(self, torque_curve=[[0.0, 500.0], [5000.0, 500.0]], max_rpm=5000.0, moi=1.0, damping_rate_full_throttle=0.15, damping_rate_zero_throttle_clutch_engaged=2.0, damping_rate_zero_throttle_clutch_disengaged=0.5, use_gear_autobox=True, gear_switch_time=0.5, clutch_strength=10.0, final_ratio=4.0, forward_gears=list(), mass=1000.0, drag_coefficient=0.3, center_of_mass=[0.0, 0.0, 0.0], steering_curve=[[0.0, 1.0], [10.0, 0.5]], wheels=list())** VehiclePhysicsControl constructor.
 - **Parameters:**
 - * **torque_curve** (*listcarla.Vector2D*)
 - * **max_rpm** (*float*)
 - * **moi** (*_float – kg*m2_*)
 - * **damping_rate_full_throttle** (*float*)
 - * **damping_rate_zero_throttle_clutch_engaged** (*float*)
 - * **damping_rate_zero_throttle_clutch_disengaged** (*float*)
 - * **use_gear_autobox** (*bool*)
 - * **gear_switch_time** (*float – seconds*)
 - * **clutch_strength** (*_float – kg*m2/s_*)
 - * **final_ratio** (*_float_*)
 - * **forward_gears** (*listcarla.GearPhysicsControl*)
 - * **drag_coefficient** (*float*)
 - * **center_of_mass** (*carla.Vector3D_*)
 - * **steering_curve** (*carla.Vector2D_*)
 - * **wheels** (*listcarla.WheelPhysicsControl*)

Dunder methods

- `__eq__(self, other**carla.VehiclePhysicsControl)`
- `__ne__(self, other**carla.VehiclePhysicsControl)`
- `__str__(self**)`

carla.Walker

Inherited from carla.Actor This class inherits from thecarla.Actor and defines pedestrians in the simulation. Walkers are a special type of actor that can be controlled either by an AI (`carla.WalkerAIController`) or manually via script, using a series of `carla.WalkerControl` to move these and their skeletons.

Instance Variables

- **bounding_box** (`carla.BoundingBox`) Bounding box containing the geometry of the walker. Its location and rotation are relative to the walker it is attached to.

Methods

- **apply_control(self, control)** On the next tick, the control will move the walker in a certain direction with a certain speed. Jumps can be commanded too.
 - **Parameters:**
 - * `control` (`carla.WalkerControl`)
- **apply_control(self, control)** On the next tick, the control defines a list of bone transformations that will be applied to the walker’s skeleton.
 - **Parameters:**
 - * `control` (`carla.WalkerBoneControl`)

Getters

- **get_control(self)** The client returns the control applied to this walker during last tick. The method does not call the simulator.
 - **Return:** `carla.WalkerControl`

Dunder methods

- `__str__(self**)`

carla.WalkerAIController

Inherited from carla.Actor Class that conducts AI control for a walker. The controllers are defined as actors, but they are quite different from the rest. They need to be attached to a parent actor during their creation, which is the walker they will be controlling (take a look at `carla.World` if you are yet to learn on how to spawn actors). They also need for a special blueprint (already defined in `carla.BlueprintLibrary` as “controller.ai.walker”). This is an empty blueprint, as the AI controller will be invisible in the simulation but will follow its parent around to dictate every step of the way.

Methods

- **go_to_location(self, destination)** Sets the destination that the pedestrian will reach.
 - **Parameters:**
 - * `destination` (`carla.Location` – meters_)
- **start(self)** Enables AI control for its parent walker.
- **stop(self)** Disables AI control for its parent walker.

Setters

- **set_max_speed(self, speed=1.4)** Sets a speed for the walker in meters per second.
 - **Parameters:**
 - * `speed` (`float` – m/s) – An easy walking speed is set by default.

Dunder methods

- `__str__(self**)`

carla.WalkerBoneControl

This class grants bone specific manipulation for walker. The skeletons of walkers have been unified for clarity and the transform applied to each bone are always relative to its parent. Take a look here to learn more on how to create a walker and define its movement.

Instance Variables

- **bone_transforms** (*list([name, transform])*) List of tuples where the first value is the bone's name and the second value stores the transformation (changes in location and rotation) that will be applied to it.

Methods

- **__init__(self, list(name,transform)**)** Initializes an object containing moves to be applied on tick. These are listed with the name of the bone and the transform that will be applied to it.

- Parameters:

* *list(name, transform)* (*tuple*)

Dunder methods

- **__str__(self**)**

carla.WalkerControl

This class defines specific directions that can be commanded to acarla.Walker to control it via script. The walker's animations will blend automatically with the parameters defined in this class when applied, though specific skeleton moves can be obtained through class.WalkerBoneControl.

AI control can be settled for walkers, but the control used to do so is carla.WalkerAIController.

Instance Variables

- **direction** (*carla.Vector3D_*) Vector using global coordinates that will correspond to the direction of the walker.
- **speed** (*float – m/s*) A scalar value to control the walker's speed.
- **jump** (*bool*) If True, the walker will perform a jump.

Methods

- **__init__(self, direction=[1.0, 0.0, 0.0], speed=0.0, jump=False)**

- Parameters:

* *direction* (*carla.Vector3D_*)
* *speed* (*float – m/s*)
* *jump* (*bool*)

Dunder methods

- **__eq__(self, other**carla.WalkerControl)** Compares every variable with *other* and returns True if these are all the same.
- **__ne__(self, other**carla.WalkerControl)** Compares every variable with *other* and returns True if any of these differ.
- **__str__(self**)**

carla.Waypoint

Waypoints in CARLA are described as 3D directed points. They have acarla.Transform which locates the waypoint in a road and orientates it according to the lane. They also store the road information belonging to said point regarding its lane and lane markings. All the information regarding waypoints and the waypoint API is retrieved as provided by the OpenDRIVE file. Once the client asks for the map object to the server, no longer communication will be needed.

Instance Variables

- **id** (*int*) The identifier is generated using a hash combination of the road, section, lane and s values that correspond to said point in the OpenDRIVE geometry. The s precision is set to 2 centimeters, so 2 waypoints closer than 2 centimeters in the same road, section and lane, will have the same identifier.
- **transform** (*carla.Transform_*) Position and orientation of the waypoint according to the current lane information. This data is computed the first time it is accessed. It is not created right away in order to ease computing costs when lots of waypoints are created but their specific transform is not needed.
- **road_id** (*int*) OpenDRIVE road's id.
- **section_id** (*int*) OpenDRIVE section's id, based on the order that they are originally defined.
- **lane_id** (*int*) OpenDRIVE lane's id, this value can be positive or negative which represents the direction of the current lane with respect to the road. For more information refer to OpenDRIVE documentation.
- **s** (*float*) OpenDRIVE s value of the current position.
- **is_junction** (*bool*) True if the current Waypoint is on a junction as defined by OpenDRIVE.
- **lane_width** (*float*) Horizontal size of the road at current s.

- **lane_change** (carla.LaneChange_) Lane change definition of the current Waypoint's location, based on the traffic rules defined in the OpenDRIVE file. It states if a lane change can be done and in which direction.
- **lane_type** (carla.LaneType_) The lane type of the current Waypoint, based on OpenDRIVE 1.4 standard.
- **right_lane_marking** (carla.LaneMarking_) The right lane marking information based on the direction of the Waypoint.
- **left_lane_marking** (carla.LaneMarking_) The left lane marking information based on the direction of the Waypoint.

Methods

- **next(self, distance)** Returns a list of waypoints at a certain approximate **distance** from the current one. It takes into account the road and its possible deviations without performing any lane change and returns one waypoint per option. The list may be empty if the lane is not connected to any other at the specified distance.
 - **Parameters:**
 - * **distance** (*float – meters*) – The approximate distance where to get the next waypoints.
 - **Return:** *listcarla.Waypoint*
- **next_until_lane_end(self, distance)** Returns a list of waypoints from this to the end of the lane separated by a certain **distance**.
 - **Parameters:**
 - * **distance** (*float – meters*) – The approximate distance between waypoints.
 - **Return:** *listcarla.Waypoint*
- **previous(self, distance)** This method does not return the waypoint previously visited by an actor, but a list of waypoints at an approximate **distance** but in the opposite direction of the lane. Similarly to **next()**, it takes into account the road and its possible deviations without performing any lane change and returns one waypoint per option. The list may be empty if the lane is not connected to any other at the specified distance.
 - **Parameters:**
 - * **distance** (*float – meters*) – The approximate distance where to get the previous waypoints.
 - **Return:** *listcarla.Waypoint*
- **previous_until_lane_start(self, distance)** Returns a list of waypoints from this to the start of the lane separated by a certain **distance**.
 - **Parameters:**
 - * **distance** (*float – meters*) – The approximate distance between waypoints.
 - **Return:** *listcarla.Waypoint*

Getters

- **get_junction(self)** If the waypoint belongs to a junction this method returns the associated junction object. Otherwise returns null.
 - **Return:** *carla.Junction*
- **get_landmarks(self, distance, stop_at_junction=False)** Returns a list of landmarks in the road from the current waypoint until the specified distance.
 - **Parameters:**
 - * **distance** (*float – meters*) – The maximum distance to search for landmarks from the current waypoint.
 - * **stop_at_junction** (*bool*) – Enables or disables the landmark search through junctions.
 - **Return:** *listcarla.Landmark*
- **get_landmarks_of_type(self, distance, type, stop_at_junction=False)** Returns a list of landmarks in the road of a specified type from the current waypoint until the specified distance.
 - **Parameters:**
 - * **distance** (*float – meters*) – The maximum distance to search for landmarks from the current waypoint.
 - * **type** (*str*) – The type of landmarks to search.
 - * **stop_at_junction** (*bool*) – Enables or disables the landmark search through junctions.
 - **Return:** *listcarla.Landmark*
- **get_left_lane(self)** Generates a Waypoint at the center of the left lane based on the direction of the current Waypoint, taking into account if the lane change is allowed in this location. Will return None if the lane does not exist.
 - **Return:** *carla.Waypoint*
- **get_right_lane(self)** Generates a waypoint at the center of the right lane based on the direction of the current waypoint, taking into account if the lane change is allowed in this location. Will return None if the lane does not exist.
 - **Return:** *carla.Waypoint*

Dunder methods

- ****__str__(self**)**

carla.WeatherParameters

This class defines objects containing lighting and weather specifications that can later be applied in carla.World. So far, these conditions only intervene with sensor.camera.rgb. They neither affect the actor's physics nor other sensors. Each of these parameters acts independently from the rest. Increasing the rainfall will not automatically create puddles nor change the road's humidity. That makes for a better customization but means that realistic conditions need to be scripted. However an example of dynamic weather conditions working realistically can be found here.

Instance Variables

- **cloudiness** (*float*) Values range from 0 to 100, being 0 a clear sky and 100 one completely covered with clouds.
- **precipitation** (*float*) Rain intensity values range from 0 to 100, being 0 none at all and 100 a heavy rain.
- **precipitation_deposits** (*float*) Determines the creation of puddles. Values range from 0 to 100, being 0 none at all and 100 a road completely capped with water. Puddles are created with static noise, meaning that they will always appear at the same locations.
- **wind_intensity** (*float*) Controls the strength of the wind with values from 0, no wind at all, to 100, a strong wind. The wind does affect rain direction and leaves from trees, so this value is restricted to avoid animation issues.
- **sun_azimuth_angle** (*float – degrees*) The azimuth angle of the sun. Values range from 0 to 360. Zero is an origin point in a sphere determined by Unreal Engine.
- **sun_altitude_angle** (*float – degrees*) Altitude angle of the sun. Values range from -90 to 90 corresponding to midnight and midday each.
- **fog_density** (*float*) Fog concentration or thickness. It only affects the RGB camera sensor. Values range from 0 to 100.
- **fog_distance** (*float – meters*) Fog start distance. Values range from 0 to infinite.
- **wetness** (*float*) Wetness intensity. It only affects the RGB camera sensor. Values range from 0 to 100.
- **fog_falloff** (*float*) Density of the fog (as in specific mass) from 0 to infinity. The bigger the value, the more dense and heavy it will be, and the fog will reach smaller heights. Corresponds to in the UE docs. If the value is 0, the fog will be lighter than air, and will cover the whole scene. A value of 1 is approximately as dense as the air, and reaches normal-sized buildings. For values greater than 5, the air will be so dense that it will be compressed on ground level.

Methods

- **__init__(self, cloudiness=0.0, precipitation=0.0, precipitation_deposits=0.0, wind_intensity=0.0, sun_azimuth_angle=0.0, sun_altitude_angle=0.0, fog_density=0.0, fog_distance=0.0, wetness=0.0, fog_falloff**=0.2)** Method to initialize an object defining weather conditions. This class has some presets for different noon and sunset conditions listed in a note below.
 - **Parameters:**
 - * **cloudiness** (*float*) – 0 is a clear sky, 100 complete overcast.
 - * **precipitation** (*float*) – 0 is no rain at all, 100 a heavy rain.
 - * **precipitation_deposits** (*float*) – 0 means no puddles on the road, 100 means roads completely capped by rain.
 - * **wind_intensity** (*float*) – 0 is calm, 100 a strong wind.
 - * **sun_azimuth_angle** (*float – degrees*) – 0 is an arbitrary North, 180 its corresponding South.
 - * **sun_altitude_angle** (*float – degrees*) – 90 is midday, -90 is midnight.
 - * **fog_density** (*float*) – Concentration or thickness of the fog, from 0 to 100.
 - * **fog_distance** (*float – meters*) – Distance where the fog starts in meters.
 - * **wetness** (*float*) – Humidity percentages of the road, from 0 to 100.
 - * **fog_falloff** (*float*) – Density (specific mass) of the fog, from 0 to infinity.
 - **Note:** *ClearNoon, CloudyNoon, WetNoon, WetCloudyNoon, SoftRainNoon, MidRainyNoon, HardRainNoon, ClearSunset, CloudySunset, WetSunset, WetCloudySunset, SoftRainSunset, MidRainSunset, HardRainSunset.*

Dunder methods

- **__eq__(self, other**)** Returns True if both objects' variables are the same.
 - **Return:** *bool*
- **__ne__(self, other**)** Returns True if both objects' variables are different.
 - **Return:** *bool*
- **__str__(self**)**

carla.WheelPhysicsControl

Class that defines specific physical parameters for wheel objects that will be part of acarla.VehiclePhysicsControl to simulate vehicle it as a material object.

Instance Variables

- **tire_friction** (*float*) A scalar value that indicates the friction of the wheel.
- **damping_rate** (*float*) Damping rate of the wheel.
- **max_steer_angle** (*float – degrees*) Maximum angle that the wheel can steer.
- **radius** (*float – centimeters*) Radius of the wheel.
- **max_brake_torque** (*float – N*m*) Maximum brake torque.
- **max_handbrake_torque** (*float – N*m*) Maximum handbrake torque.
- **position** (*carla.Vector3D*) World position of the wheel. This is a read-only parameter.

Methods

- **__init__(self, tire_friction=2.0, damping_rate=0.25, max_steer_angle=70.0, radius=30.0, max_brake_torque=max_handbrake_torque=3000.0, position=(0.0,0.0,0.0))**

– Parameters:

- * **tire_friction** (*float*)
- * **damping_rate** (*float*)
- * **max_steer_angle** (*float – degrees*)
- * **radius** (*float – centimeters*)
- * **max_brake_torque** (*float – N*m*)
- * **max_handbrake_torque** (*float – N*m*)
- * **position** (*carla.Vector3D – meters*)

Dunder methods

- **__eq__(self, other)** (*carla.WheelPhysicsControl*)
- **__ne__(self, other)** (*carla.WheelPhysicsControl*)
- **__str__(self)**

carla.World

World objects are created by the client to have a place for the simulation to happen. The world contains the map we can see, meaning the asset, not the navigation map. Navigation maps are part of thecarla.Map class. It also manages the weather and actors present in it. There can only be one world per simulation, but it can be changed anytime.

Instance Variables

- **id** (*int*) The ID of the episode associated with this world. Episodes are different sessions of a simulation. These change everytime a world is disabled or reloaded. Keeping track is useful to avoid possible issues.
- **debug** (*carla.DebugHelper*) Responsible for creating different shapes for debugging. Take a look at its class to learn more about it.

Methods

- **apply_settings(self, world_settings)** This method applies settings contained in an object to the simulation running and returns the ID of the frame they were implemented.
 - **Parameters:**
 - * **world_settings** (*carla.WorldSettings*)
 - **Return:** *int*
- **on_tick(self, callback)** The method will start callbacks for a defined function **callback**. It will return the ID for this callback so it can be removed with **remove_on_tick()**.
 - **Parameters:**
 - * **callback** (*carla.WorldSnapshot*) – A defined function with a snapshot as compulsory parameter that will be called every tick.
 - **Return:** *int*
- **remove_on_tick(self, callback_id)** Stops the callback for **callback_id** started with **on_tick()**.
 - **Parameters:**
 - * **callback_id** (*callback*) – The callback to be removed.
- **tick(self, seconds=10.0)** This method only has effect on synchronous mode, when both client and server move together. The client tells the server when to step to the next frame and returns the id of the newly started frame.

- **Parameters:**
 - * `seconds` (`float - seconds`) – Maximum time the server should wait for a tick. It is set to 10.0 by default.
 - **Return:** `int`
- `wait_for_tick(self, seconds=10.0)` The client tells the server to pause the simulation until a `World.tick()` is received.
 - **Parameters:**
 - * `seconds` (`float - seconds`) – Maximum time the server should wait for a tick. It is set to 10.0 by default.
 - **Return:** `carla.WorldSnapshot`
- `spawn_actor(self, blueprint, transform, attach_to=None, attachment=Rigid)` The method will create, return and spawn an actor into the world. The actor will need an available blueprint to be created and a transform (location and rotation). It can also be attached to a parent with a certain attachment type.
 - **Parameters:**
 - * `blueprint` (`carla.ActorBlueprint`) – The reference from which the actor will be created.
 - * `transform` (`carla.Transform`) – Contains the location and orientation the actor will be spawned with.
 - * `attach_to` (`carla.Actor`) – The parent object that the spawned actor will follow around.
 - * `attachment` (`carla.AttachmentType`) – Determines how fixed and rigorous should be the changes in position according to its parent object.
 - **Return:** `carla.Actor`
- `try_spawn_actor(self, blueprint, transform, attach_to=None, attachment=Rigid)` Same as `spawn_actor()` but returns None on failure instead of throwing an exception.
 - **Parameters:**
 - * `blueprint` (`carla.ActorBlueprint`) – The reference from which the actor will be created.
 - * `transform` (`carla.Transform`) – Contains the location and orientation the actor will be spawned with.
 - * `attach_to` (`carla.Actor`) – The parent object that the spawned actor will follow around.
 - * `attachment` (`carla.AttachmentType`) – Determines how fixed and rigorous should be the changes in position according to its parent object.
 - **Return:** `carla.Actor`
- `freeze_all_traffic_lights(self, frozen)` Freezes or unfreezes all traffic lights in the scene. Frozen traffic lights can be modified by the user but the time will not update them until unfrozen.
 - **Parameters:**
 - * `frozen` (`bool`)
- `reset_all_traffic_lights(self)` Resets the cycle of all traffic lights in the map to the initial state.

Getters

- `get_actor(self, actor_id)` Looks up for an actor by ID and returns None if not found.
 - **Parameters:**
 - * `actor_id` (`int`)
 - **Return:** `carla.Actor`
- `get_actors(self, actor_ids=None)` Retrieves a list of `carla.Actor` elements, either using a list of IDs provided or just listing everyone on stage. If an ID does not correspond with any actor, it will be excluded from the list returned, meaning that both the list of IDs and the list of actors may have different lengths.
 - **Parameters:**
 - * `actor_ids` (`list`) – The IDs of the actors being searched. By default it is set to None and returns every actor on scene.
 - **Return:** `carla.ActorList`
- `get_blueprint_library(self)` Returns a list of actor blueprints available to ease the spawn of these into the world.
 - **Return:** `carla.BlueprintLibrary`
- `get_vehicles_light_states(self)` Returns a dict where the keys are `carla.Actor` IDs and the values are `carla.VehicleLightState` of that vehicle.
 - **Return:** `dict`
- `get_level_bbs(self, actor_type=None)` Returns an array of bounding boxes with location and rotation in world space. The method returns all the bounding boxes in the level by default, but the query can be filtered by semantic tags with the argument `actor_type`.
 - **Parameters:**
 - * `actor_type` (`carla.CityObjectLabel`) – Semantic tag of the elements contained in the bounding boxes that are returned.
 - **Return:** `array<carla.BoundingBox>`
- `get_lightmanager(self)` Returns an instance of `carla.LightManager` that can be used to handle the lights in the scene.
 - **Return:** `carla.LightManager`

- **get_map(self)** Asks the server for the XODR containing the map file, and returns this parsed as acarla.Map.
 - **Return:** carla.Map
 - **Warning:** *This method does call the simulation. It is expensive, and should only be called once.*
- **get_traffic_light(self, landmark)** Provided a landmark, returns the traffic light object it describes.
 - **Parameters:**
 - * `landmark` (carla.Landmark) – The landmark object describing a traffic light.
 - **Return:** carla.TrafficLight
- **get_traffic_sign(self, landmark)** Provided a landmark, returns the traffic sign object it describes.
 - **Parameters:**
 - * `landmark` (carla.Landmark) – The landmark object describing a traffic sign.
 - **Return:** carla.TrafficSign
- **get_random_location_from_navigation(self)** This can only be used with walkers. It retrieves a random location to be used as a destination using the `go_to_location()` method in carla.WalkerAIController. This location will be part of a sidewalk. Roads, crosswalks and grass zones are excluded. The method does not take into consideration locations of existing actors so if a collision happens when trying to spawn an actor, it will return an error. Take a look at `spawn_npc.py` for an example.
 - **Return:** carla.Location
- **get_settings(self)** Returns an object containing some data about the simulation such as synchrony between client and server or rendering mode.
 - **Return:** carla.WorldSettings
- **get_snapshot(self)** Returns a snapshot of the world at a certain moment comprising all the information about the actors.
 - **Return:** carla.WorldSnapshot
- **get_spectator(self)** Returns the spectator actor. The spectator is a special type of actor created by Unreal Engine, usually with ID=0, that acts as a camera and controls the view in the simulator window.
 - **Return:** carla.Actor
- **get_weather(self)** Retrieves an object containing weather parameters currently active in the simulation, mainly cloudiness, precipitation, wind and sun position.
 - **Return:** carla.WeatherParameters
 - **Setter:** carla.World.set_weather

Setters

- **set_weather(self, weather)** Changes the weather parameteres ruling the simulation to another ones defined in an object.
 - **Parameters:**
 - * `weather` (carla.WeatherParameters) – New conditions to be applied.
 - **Getter:** carla.World.get_weather

Dunder methods

- ****__str__(self)**** The content of the world is parsed and printed as a brief report of its current state.
 - **Return:** string

carla.WorldSettings

The simulation has some advanced configuration options that are contained in this class and can be managed using carla.World and its methods. These allow the user to choose between client-server synchrony/asynchrony, activation of “no rendering mode” and either if the simulation should run with a fixed or variable time-step. Check this out if you want to learn about it.

Instance Variables

- **synchronous_mode (bool)** States the synchrony between client and server. When set to true, the server will wait for a client tick in order to move forward. It is false by default.
- **no_rendering_mode (bool)** When enabled, the simulation will run no rendering at all. This is mainly used to avoid overhead during heavy traffic simulations. It is false by default.
- **fixed_delta_seconds (float)** Ensures that the time elapsed between two steps of the simulation is fixed. Set this to 0.0 to work with a variable time-step, as happens by default.

Methods

- ****__init__(self, synchronous_mode=False, no_rendering_mode=False, fixed_delta_seconds=0.0)** Creates an object containing desired settings that could later be applied through carla.World and its method `apply_settings()`.

– **Parameters:**

- * `synchronous_mode` (`bool`) – Set this to true to enable client-server synchrony.
- * `no_rendering_mode` (`bool`) – Set this to true to completely disable rendering in the simulation.
- * `fixed_delta_seconds` (`float – seconds`) – Set a fixed time-step in between frames. 0.0 means variable time-step and it is the default mode.

Dunder methods

- `__eq__(self, other**carla.WorldSettings)` Returns True if both objects' variables are the same.
 - **Return:** `bool`
- `__ne__(self, other**carla.WorldSettings)` Returns True if both objects' variables are different.
 - **Return:** `bool`
- `__str__(self**)` Parses the established settings to a string and shows them in command line.
 - **Return:** `str`

carla.WorldSnapshot

This snapshot comprises all the information for every actor on scene at a certain moment of time. It creates and gives access to a data structure containing a series of carla.ActorSnapshot. The client receives a new snapshot on every tick that cannot be stored.

Instance Variables

- `id` (`int`) A value unique for every snapshot to differentiate them.
- `frame` (`int`) Simulation frame in which the snapshot was taken.
- `timestamp` (`carla.Timestamp – seconds_`) Precise moment in time when snapshot was taken. This class works in seconds as given by the operative system.

Methods

- `find(self, actor_id)` Given a certain actor ID, returns its corresponding snapshot or None if it is not found.
 - **Parameters:**
 - * `actor_id` (`int`)
 - **Return:** `carla.ActorSnapshot_`
- `has_actor(self, actor_id)` Given a certain actor ID, checks if there is a snapshot corresponding to it and so, if the actor was present at that moment.
 - **Parameters:**
 - * `actor_id` (`int`)
 - **Return:** `bool`

Dunder methods

- `__iter__(self)` Method that enables iteration for this class using timestamp** as reference value.
- `__len__(self**)` Returns the amount of carla.ActorSnapshot present in this snapshot.
 - **Return:** `int`
- `__eq__(self, othercarla.WorldSnapshot)` Returns True if both timestamp** are the same.
 - **Return:** `bool`
- `__ne__(self, othercarla.WorldSnapshot)` Returns True if both timestamp** are different.
 - **Return:** `bool`

command.ApplyAngularImpulse

Command adaptation of `add_angular_impulse()` in carla.Actor. Applies an angular impulse to an actor.

Instance Variables

- `actor_id` (`int`) Actor affected by the command.
- `impulse` (`carla.Vector3D – degrees*s_`) Angular impulse applied to the actor.

Methods

- `__init__(self, actor, impulse**)`
 - **Parameters:**
 - * `actor` (`carla.Actor` or `int_`) – Actor or its ID to whom the command will be applied to.
 - * `impulse` (`carla.Vector3D – degrees*s_`)

command.ApplyForce

Command adaptation of **add_force()** in carla.Actor. Applies a force to an actor.

Instance Variables

- **actor_id** (*int*) Actor affected by the command.
- **force** (carla.Vector3D – N_–) Force applied to the actor over time.

Methods

- ****__init__(self, actor, force**)**
 - **Parameters:**
 - * **actor** (carla.Actor or int_–) – Actor or its ID to whom the command will be applied to.
 - * **force** (carla.Vector3D – N_–)

command.ApplyImpulse

Command adaptation of **add_impulse()** in carla.Actor. Applies an impulse to an actor.

Instance Variables

- **actor_id** (*int*) Actor affected by the command.
- **impulse** (carla.Vector3D – N*s_–) Impulse applied to the actor.

Methods

- ****__init__(self, actor, impulse**)**
 - **Parameters:**
 - * **actor** (carla.Actor or int_–) – Actor or its ID to whom the command will be applied to.
 - * **impulse** (carla.Vector3D – N*s_–)

command.ApplyTargetAngularVelocity

Command adaptation of **set_target_angular_velocity()** in carla.Actor. Sets the actor's angular velocity vector.

Instance Variables

- **actor_id** (*int*) Actor affected by the command.
- **angular_velocity** (carla.Vector3D – deg/s_–) The 3D angular velocity that will be applied to the actor.

Methods

- ****__init__(self, actor, angular_velocity**)**
 - **Parameters:**
 - * **actor** (carla.Actor or int_–) – Actor or its ID to whom the command will be applied to.
 - * **angular_velocity** (carla.Vector3D – deg/s_–) – Angular velocity vector applied to the actor.

command.ApplyTargetVelocity

Command adaptation of **set_target_velocity()** in carla.Actor.

Instance Variables

- **actor_id** (*int*) Actor affected by the command.
- **velocity** (carla.Vector3D – m/s_–) The 3D velocity applied to the actor.

Methods

- ****__init__(self, actor, velocity**)**
 - **Parameters:**
 - * **actor** (carla.Actor or int_–) – Actor or its ID to whom the command will be applied to.
 - * **velocity** (carla.Vector3D – m/s_–) – Velocity vector applied to the actor.

command.ApplyTorque

Command adaptation of **add_torque()** in carla.Actor. Applies a torque to an actor.

Instance Variables

- **actor_id** (*int*) Actor affected by the command.

- **torque** (carla.Vector3D – degrees_) Torque applied to the actor over time.

Methods

- **__init__(self, actor, torque**)
 - **Parameters:**
 - * **actor** (carla.Actor or int_) – Actor or its ID to whom the command will be applied to.
 - * **torque** (carla.Vector3D – degrees_)

command.ApplyTransform

Command adaptation of **set_transform()** in carla.Actor. Sets a new transform to an actor.

Instance Variables

- **actor_id** (*int*) Actor affected by the command.
- **transform** (carla.Transform_) Transformation to be applied.

Methods

- **__init__(self, actor, transform**)
 - **Parameters:**
 - * **actor** (carla.Actor or int_) – Actor or its ID to whom the command will be applied to.
 - * **transform** (carla.Transform_)

command.ApplyVehicleControl

Command adaptation of **apply_control()** in carla.Vehicle. Applies a certain control to a vehicle.

Instance Variables

- **actor_id** (*int*) Vehicle actor affected by the command.
- **control** (carla.VehicleControl_) Vehicle control to be applied.

Methods

- **__init__(self, actor, control**)
 - **Parameters:**
 - * **actor** (carla.Actor or int_) – Actor or its ID to whom the command will be applied to.
 - * **control** (carla.VehicleControl_)

command.ApplyWalkerControl

Command adaptation of **apply_control()** in carla.Walker. Applies a control to a walker.

Instance Variables

- **actor_id** (*int*) Walker actor affected by the command.
- **control** (carla.WalkerControl_) Walker control to be applied.

Methods

- **__init__(self, actor, control**)
 - **Parameters:**
 - * **actor** (carla.Actor or int_) – Actor or its ID to whom the command will be applied to.
 - * **control** (carla.WalkerControl_)

command.ApplyWalkerState

Apply a state to the walker actor. Specially useful to initialize an actor them with a specific location, orientation and speed.

Instance Variables

- **actor_id** (*int*) Walker actor affected by the command.
- **transform** (carla.Transform_) Transform to be applied.
- **speed** (*float – m/s*) Speed to be applied.

Methods

- **__init__(self, actor, transform, speed**)

– **Parameters:**

- * **actor** (carla.Actor or int_) – Actor or its ID to whom the command will be applied to.
- * **transform** (carla.Transform_)
- * **speed** (float – m/s)

command.DestroyActor

Command adaptation of **destroy()** incarla.Actor that tells the simulator to destroy this actor. It has no effect if the actor was already destroyed. When executed with **apply_batch_sync()** incarla.Client there will be a command.Response that will return a boolean stating whether the actor was successfully destroyed.

Instance Variables

- **actor_id** (int) Actor affected by the command.

Methods

- **__init__(self, actor**)

– **Parameters:**

- * **actor** (carla.Actor or int_) – Actor or its ID to whom the command will be applied to.

command.Response

States the result of executing a command as either the ID of the actor to whom the command was applied to (when succeeded) or an error string (when failed). actor ID, depending on whether or not the command succeeded. The method **apply_batch_sync()** incarla.Client returns a list of these to summarize the execution of a batch.

Instance Variables

- **actor_id** (int) Actor to whom the command was applied to. States that the command was successful.
- **error** (str) A string stating the command has failed.

Methods

- **has_error(self)** Returns True if the command execution fails, and False if it was successful.
– **Return:** bool

command.SetAutopilot

Command adaptation of **set_autopilot()** incarla.Vehicle. Turns on/off the vehicle's autopilot mode.

Instance Variables

- **actor_id** (int) Actor that is affected by the command.
- **enabled** (bool) If autopilot should be activated or not.
- **port** (uint16) Port of the Traffic Manager where the vehicle is to be registered or unlisted.

Methods

- **__init__(self, actor, enabled, port**=8000)

– **Parameters:**

- * **actor** (carla.Actor or int_) – Actor or its ID to whom the command will be applied to.
- * **enabled** (bool)
- * **port** (uint16) – The Traffic Manager port where the vehicle is to be registered or unlisted. If **None** is passed, it will consider a TM at default port 8000.

command.SetSimulatePhysics

Command adaptation of **set_simulate_physics()** incarla.Actor. Determines whether an actor will be affected by physics or not.

Instance Variables

- **actor_id** (int) Actor affected by the command.
- **enabled** (bool) If physics should be activated or not.

Methods

- **__init__(self, actor, enabled**)

– **Parameters:**

- * **actor** (carla.Actor or int_) – Actor or its ID to whom the command will be applied to.
- * **enabled** (bool)

command.SetVehicleLightState

Command adaptation of `set_light_state()` in carla.Vehicle. Sets the light state of a vehicle.

Instance Variables

- **actor_id** (int) Actor that is affected by the command.
- **light_state** (carla.VehicleLightState_) Defines the light state of a vehicle.

Methods

- **`__init__(self, actor, light_state***)`
 - **Parameters:**
 - * **actor** (carla.Actor or int_) – Actor or its ID to whom the command will be applied to.
 - * **light_state** (carla.VehicleLightState_) – Recaps the state of the lights of a vehicle, these can be used as a flags.

command.SpawnActor

Command adaptation of `spawn_actor()` in carla.World. Spawns an actor into the world based on the blueprint provided and the transform. If a parent is provided, the actor is attached to it.

Instance Variables

- **transform** (carla.Transform_) Transform to be applied.
- **parent_id** (int) Identifier of the parent actor.

Methods

- **`__init__(self***)`
- **`__init__(self, blueprint, transform***)`
 - **Parameters:**
 - * **blueprint** (carla.ActorBlueprint_)
 - * **transform** (carla.Transform_)
- **`__init__(self, blueprint, transform, parent***)`
 - **Parameters:**
 - * **blueprint** (carla.ActorBlueprint_)
 - * **transform** (carla.Transform_)
 - * **parent** (carla.Actor or int_)
- **then(self, command)** Links another command to be executed right after. It allows to ease very common flows such as spawning a set of vehicles by command and then using this method to set them to autopilot automatically.
 - **Parameters:**
 - * **command** (*any carla Command*) – a Carla command.

5.1 Code recipes

This section contains a list of recipes that complement the first steps section and are used to illustrate the use of Python API methods.

Each recipe has a list of python API classes, which is divided into those in which the recipe is centered, and those that need to be used.

There are more recipes to come!

- **Actor Spectator Recipe**
- **Attach Sensors Recipe**
- **Actor Attribute Recipe**
- **Converted Image Recipe**
- **Lanes Recipe**
- **Debug Bounding Box Recipe**
- **Debug Vehicle Trail Recipe**
- **Parsing Client Arguments Recipe**
- **Traffic Light Recipe**
- **Walker Batch Recipe**

Actor Spectator Recipe

This recipe spawns an actor and the spectator camera at the actor's location.

Focused on: `carla.World carla.Actor`

Used: `carla.WorldSnapshot carla.ActorSnapshot`

```
1 # ...
2 world = client.get_world()
3 spectator = world.get_spectator()
4
5 vehicle_bp = random.choice(world.get_blueprint_library().filter('vehicle.bmw.*'))
6 transform = random.choice(world.get_map().get_spawn_points())
7 vehicle = world.try_spawn_actor(vehicle_bp, transform)
8
9 # Wait for world to get the vehicle actor
10 world.tick()
11
12 world_snapshot = world.wait_for_tick()
13 actor_snapshot = world_snapshot.find(vehicle.id)
14
15 # Set spectator at given transform (vehicle transform)
16 spectator.set_transform(actor_snapshot.get_transform())
17 # ...
```

Attach Sensors Recipe

This recipe attaches different camera / sensors to a vehicle with different attachments.

Focused on: `carla.Sensor carla.AttachmentType`

Used: `carla.World`

```
1 # ...
2 camera = world.spawn_actor(rgb_camera_bp, transform, attach_to=vehicle,
    attachment_type=Attachment.Rigid)
3 # Default attachment: Attachment.Rigid
4 gnss_sensor = world.spawn_actor(sensor_gnss_bp, transform, attach_to=vehicle)
5 collision_sensor = world.spawn_actor(sensor_collision_bp, transform, attach_to=vehicle)
6 lane_invasion_sensor = world.spawn_actor(sensor_lane_invasion_bp, transform, attach_to=vehicle)
7 # ...
```

Actor Attribute Recipe

This recipe changes attributes of different type of blueprint actors.

Focused on: `carla.ActorAttribute carla.ActorBlueprint`

Used: `carla.World carla.BlueprintLibrary`

```
1 # ...
2 walker_bp = world.get_blueprint_library().filter('walker.pedestrian.0002')
3 walker_bp.set_attribute('is_invincible', True)
4
5 # ...
6 # Changes attribute randomly by the recommended value
7 vehicle_bp = world.get_blueprint_library().filter('vehicle.bmw.*')
8 color = random.choice(vehicle_bp.get_attribute('color').recommended_values)
9 vehicle_bp.set_attribute('color', color)
10
11 # ...
12
13 camera_bp = world.get_blueprint_library().filter('sensor.camera.rgb')
14 camera_bp.set_attribute('image_size_x', 600)
```

```
15 camera_bp.set_attribute('image_size_y', 600)
16 # ...
```

Converted Image Recipe

This recipe applies a color conversion to the image taken by a camera sensor, so it is converted to a semantic segmentation image.

Focused on: `carla.ColorConverter carla.Sensor`

```
1 # ...
2 camera_bp = world.get_blueprint_library().filter('sensor.camera.semantic_segmentation')
3 # ...
4 cc = carla.ColorConverter.CityScapesPalette
5 camera.listen(lambda image: image.save_to_disk('output/%06d.png' % image.frame, cc))
6 # ...
```

Lanes Recipe

This recipe shows the current traffic rules affecting the vehicle. Shows the current lane type and if a lane change can be done in the actual lane or the surrounding ones.

Focused on: `carla.LaneMarking carla.LaneMarkingType carla.LaneChange carla.LaneType`

Used: `carla.Waypoint carla.World`

```
1 # ...
2 waypoint = world.get_map().get_waypoint(vehicle.get_location(), project_to_road=True,
    lane_type=(carla.LaneType.Driving | carla.LaneType.Shoulder | carla.LaneType.Sidewalk))
3 print("Current lane type: " + str(waypoint.lane_type))
4 # Check current lane change allowed
5 print("Current Lane change: " + str(waypoint.lane_change))
6 # Left and Right lane markings
7 print("L lane marking type: " + str(waypoint.left_lane_marking.type))
8 print("L lane marking change: " + str(waypoint.left_lane_marking.lane_change))
9 print("R lane marking type: " + str(waypoint.right_lane_marking.type))
10 print("R lane marking change: " + str(waypoint.right_lane_marking.lane_change))
11 # ...
```

Debug Bounding Box Recipe

This recipe shows how to draw traffic light actor bounding boxes from a world snapshot.

Focused on: `carla.DebugHelper carla.BoundingBox`

Used: `carla.ActorSnapshot carla.Actor carla.Vector3D carla.Color`

```
1 # ....
2 debug = world.debug
3 world_snapshot = world.get_snapshot()
4
5 for actor_snapshot in world_snapshot:
6     actual_actor = world.get_actor(actor_snapshot.id)
7     if actual_actor.type_id == 'traffic.traffic_light':
8         debug.draw_box(carla.BoundingBox(actor_snapshot.get_transform().location,carla.Vector3D(0.5,0.5,2)),act
9 # ...
```

Debug Vehicle Trail Recipe

This recipe is a modification of `lane_explorer.py` example. It draws the path of an actor through the world, printing information at each waypoint.

Focused on: `carla.DebugHelper carla.Waypoint carla.Actor`

Used: `carla.ActorSnapshot carla.Vector3D carla.LaneType carla.Color carla.Map`



Figure 11: lane_marking_recipe



Figure 12: debug_bb_recipe

```

1 # ...
2 current_w = map.get_waypoint(vehicle.get_location())
3 while True:
4
5     next_w = map.get_waypoint(vehicle.get_location(), lane_type=carla.LaneType.Driving |
6         carla.LaneType.Shoulder | carla.LaneType.Sidewalk )
7     # Check if the vehicle is moving
8     if next_w.id != current_w.id:
9         vector = vehicle.get_velocity()
10        # Check if the vehicle is on a sidewalk
11        if current_w.lane_type == carla.LaneType.Sidewalk:
12            draw_waypoint_union(debug, current_w, next_w, cyan if current_w.is_junction else red, 60)
13        else:
14            draw_waypoint_union(debug, current_w, next_w, cyan if current_w.is_junction else green,
15                60)
15        debug.draw_string(current_w.transform.location, str('%15.0f km/h' % (3.6 *
16            math.sqrt(vector.x**2 + vector.y**2 + vector.z**2))), False, orange, 60)
17        draw_transform(debug, current_w.transform, white, 60)
18
19    # Update the current waypoint and sleep for some time
20    current_w = next_w
21    time.sleep(args.tick_time)
22 # ...

```

The image below shows how a vehicle loses control and drives on a sidewalk. The trail shows the path it was following and the speed at each waypoint.



Figure 13: debug_trail_recipe

Parsing Client Arguments Recipe

This recipe shows in every script provided in `PythonAPI/Examples` and it is used to parse the client creation arguments when running the script.

Focused on: `carla.Client`

Used: carla.Client

```
1  argparser = argparse.ArgumentParser(
2      description=_doc__)
3  argparser.add_argument(
4      '--host',
5      metavar='H',
6      default='127.0.0.1',
7      help='IP of the host server (default: 127.0.0.1)')
8  argparser.add_argument(
9      '-p', '--port',
10     metavar='P',
11     default=2000,
12     type=int,
13     help='TCP port to listen to (default: 2000)')
14 argparser.add_argument(
15     '-s', '--speed',
16     metavar='FACTOR',
17     default=1.0,
18     type=float,
19     help='rate at which the weather changes (default: 1.0)')
20 args = argparser.parse_args()
21
22 speed_factor = args.speed
23 update_freq = 0.1 / speed_factor
24
25 client = carla.Client(args.host, args.port)
```

Traffic Light Recipe

This recipe changes from red to green the traffic light that affects the vehicle. This is done by detecting if the vehicle actor is at a traffic light.

Focused on: carla.TrafficLight carla.TrafficLightState

Used: carla.Vehicle

```
1 # ...
2 if vehicle_actor.is_at_traffic_light():
3     traffic_light = vehicle_actor.get_traffic_light()
4     if traffic_light.get_state() == carla.TrafficLightState.Red:
5         # world.hud.notification("Traffic light changed! Good to go!")
6         traffic_light.set_state(carla.TrafficLightState.Green)
7 # ...
```

Walker Batch Recipe

```
1 # 0. Choose a blueprint fo the walkers
2 world = client.get_world()
3 blueprintsWalkers = world.get_blueprint_library().filter("walker.pedestrian.*")
4 walker_bp = random.choice(blueprintsWalkers)
5
6 # 1. Take all the random locations to spawn
7 spawn_points = []
8 for i in range(50):
9     spawn_point = carla.Transform()
10    spawn_point.location = world.get_random_location_from_navigation()
11    if (spawn_point.location != None):
12        spawn_points.append(spawn_point)
13
14 # 2. Build the batch of commands to spawn the pedestrians
15 batch = []
16 for spawn_point in spawn_points:
```



Figure 14: tl_recipe

```

17 walker_bp = random.choice(blueprintsWalkers)
18 batch.append(carla.command.SpawnActor(walker_bp, spawn_point))
19
20 # 2.1 apply the batch
21 results = client.apply_batch_sync(batch, True)
22 for i in range(len(results)):
23     if results[i].error:
24         logging.error(results[i].error)
25     else:
26         walkers_list.append({"id": results[i].actor_id})
27
28 # 3. Spawn walker AI controllers for each walker
29 batch = []
30 walker_controller_bp = world.get_blueprint_library().find('controller.ai.walker')
31 for i in range(len(walkers_list)):
32     batch.append(carla.command.SpawnActor(walker_controller_bp, carla.Transform(),
33                                         walkers_list[i]["id"]))
34
35 # 3.1 apply the batch
36 results = client.apply_batch_sync(batch, True)
37 for i in range(len(results)):
38     if results[i].error:
39         logging.error(results[i].error)
40     else:
41         walkers_list[i]["con"] = results[i].actor_id
42
43 # 4. Put altogether the walker and controller ids
44 for i in range(len(walkers_list)):
45     all_id.append(walkers_list[i]["con"])
46     all_id.append(walkers_list[i]["id"])
47 all_actors = world.get_actors(all_id)
48
49 # wait for a tick to ensure client receives the last transform of the walkers we have just created

```

```

49 world.wait_for_tick()
50
51 # 5. initialize each controller and set target to walk to (list is [controller, actor, controller,
   actor ...])
52 for i in range(0, len(all_actors), 2):
53     # start walker
54     all_actors[i].start()
55     # set walk to random point
56     all_actors[i].go_to_location(world.get_random_location_from_navigation())
57     # random max speed
58     all_actors[i].set_max_speed(1 + random.random())      # max speed between 1 and 2 (default is 1.4
                                                               m/s)

```

To **destroy the pedestrians**, stop them from the navigation, and then destroy the objects (actor and controller):

```

1 # stop pedestrians (list is [controller, actor, controller, actor ...])
2 for i in range(0, len(all_id), 2):
3     all_actors[i].stop()
4
5 # destroy pedestrian (actor and controller)
6 client.apply_batch([carla.command.DestroyActor(x) for x in all_id])

```

##Blueprint Library The Blueprint Library `carla.BlueprintLibrary`) is a summary of all`carla.ActorBlueprint` and its attributes `carla.ActorAttribute`) available to the user in CARLA.

Here is an example code for printing all actor blueprints and their attributes:

```

1 blueprints = [bp for bp in world.get_blueprint_library().filter('*')]
2 for blueprint in blueprints:
3     print(blueprint.id)
4     for attr in blueprint:
5         print(' - {}'.format(attr))

```

Check out the introduction to blueprints.

controller

- `controller.ai.walker`
 - **Attributes:**
 - * `role_name (String)` – Modifiable

sensor

- `sensor.camera.depth`
 - **Attributes:**
 - * `fov (Float)` – Modifiable
 - * `image_size_x (Int)` – Modifiable
 - * `image_size_y (Int)` – Modifiable
 - * `lens_circle_falloff (Float)` – Modifiable
 - * `lens_circle_multiplier (Float)` – Modifiable
 - * `lens_k (Float)` – Modifiable
 - * `lens_kcube (Float)` – Modifiable
 - * `lens_x_size (_Float)` – Modifiable
 - * `lens_y_size (_Float)` – Modifiable
 - * `role_name (String)` – Modifiable
 - * `sensor_tick (_Float)` – Modifiable
- `sensor.camera.dvs`
 - **Attributes:**
 - * `black_clip (Float)` – Modifiable
 - * `blade_count (_Int)` – Modifiable
 - * `bloom_intensity (Float)` – Modifiable
 - * `blur_amount (_Float)` – Modifiable
 - * `blur_radius (_Float)` – Modifiable
 - * `calibration_constant (Float)` – Modifiable

- * `chromatic_aberration_intensity (Float)` – Modifiable
- * `chromatic_aberration_offset (Float)` – Modifiable
- * `enable_postprocess_effects (Bool)` – Modifiable
- * `exposure_compensation (Float)` – Modifiable
- * `exposure_max_bright (Float)` – Modifiable
- * `exposure_min_bright (Float)` – Modifiable
- * `exposure_mode (String)` – Modifiable
- * `exposure_speed_down (Float)` – Modifiable
- * `exposure_speed_up (Float)` – Modifiable
- * `focal_distance (Float)` – Modifiable
- * `fov (Float)` – Modifiable
- * `fstop (Float)` – Modifiable
- * `gamma (Float)` – Modifiable
- * `image_size_x (Int)` – Modifiable
- * `image_size_y (Int)` – Modifiable
- * `iso (Float)` – Modifiable
- * `lens_circle_falloff (Float)` – Modifiable
- * `lens_circle_multiplier (Float)` – Modifiable
- * `lens_flare_intensity (Float)` – Modifiable
- * `lens_k (Float)` – Modifiable
- * `lens_kcube (Float)` – Modifiable
- * `lens_x_size (_Float_)` – Modifiable
- * `lens_y_size (_Float_)` – Modifiable
- * `log_eps (Float)` – Modifiable
- * `min_fstop (Float)` – Modifiable
- * `motion_blur_intensity (Float)` – Modifiable
- * `motion_blur_max_distortion (Float)` – Modifiable
- * `motion_blur_min_object_screen_size (Float)` – Modifiable
- * `negative_threshold (Float)` – Modifiable
- * `positive_threshold (Float)` – Modifiable
- * `refractory_period_ns (Int)` – Modifiable
- * `role_name (String)` – Modifiable
- * `sensor_tick (_Float_)` – Modifiable
- * `shoulder (Float)` – Modifiable
- * `shutter_speed (Float)` – Modifiable
- * `sigma_negative_threshold (Float)` – Modifiable
- * `sigma_positive_threshold (Float)` – Modifiable
- * `slope (Float)` – Modifiable
- * `temp (Float)` – Modifiable
- * `tint (Float)` – Modifiable
- * `toe (Float)` – Modifiable
- * `use_log (Bool)` – Modifiable
- * `white_clip (Float)` – Modifiable

- sensor.camera.rgb

– Attributes:

- * `black_clip` (*Float*) – Modifiable
- * `blade_count` (*_Int*) – Modifiable
- * `bloom_intensity` (*Float*) – Modifiable
- * `blur_amount` (*_Float*) – Modifiable
- * `blur_radius` (*_Float*) – Modifiable
- * `calibration_constant` (*Float*) – Modifiable
- * `chromatic_aberration_intensity` (*Float*) – Modifiable
- * `chromatic_aberration_offset` (*Float*) – Modifiable
- * `enable_postprocess_effects` (*Bool*) – Modifiable
- * `exposure_compensation` (*Float*) – Modifiable
- * `exposure_max_bright` (*Float*) – Modifiable
- * `exposure_min_bright` (*Float*) – Modifiable
- * `exposure_mode` (*String*) – Modifiable
- * `exposure_speed_down` (*Float*) – Modifiable
- * `exposure_speed_up` (*Float*) – Modifiable
- * `focal_distance` (*Float*) – Modifiable

- * `fov (Float)` – Modifiable
- * `fstop (Float)` – Modifiable
- * `gamma (Float)` – Modifiable
- * `image_size_x (Int)` – Modifiable
- * `image_size_y (Int)` – Modifiable
- * `iso (Float)` – Modifiable
- * `lens_circle_falloff (Float)` – Modifiable
- * `lens_circle_multiplier (Float)` – Modifiable
- * `lens_flare_intensity (Float)` – Modifiable
- * `lens_k (Float)` – Modifiable
- * `lens_kcube (Float)` – Modifiable
- * `lens_x_size (_Float_)` – Modifiable
- * `lens_y_size (_Float_)` – Modifiable
- * `min_fstop (Float)` – Modifiable
- * `motion.blur_intensity (Float)` – Modifiable
- * `motion.blur_max_distortion (Float)` – Modifiable
- * `motion.blur_min_object_screen_size (Float)` – Modifiable
- * `role_name (String)` – Modifiable
- * `sensor_tick (_Float_)` – Modifiable
- * `shoulder (Float)` – Modifiable
- * `shutter_speed (Float)` – Modifiable
- * `slope (Float)` – Modifiable
- * `temp (Float)` – Modifiable
- * `tint (Float)` – Modifiable
- * `toe (Float)` – Modifiable
- * `white_clip (Float)` – Modifiable

- **sensor.camera.semantic_segmentation**

- **Attributes:**

- * `fov (Float)` – Modifiable
- * `image_size_x (Int)` – Modifiable
- * `image_size_y (Int)` – Modifiable
- * `lens_circle_falloff (Float)` – Modifiable
- * `lens_circle_multiplier (Float)` – Modifiable
- * `lens_k (Float)` – Modifiable
- * `lens_kcube (Float)` – Modifiable
- * `lens_x_size (_Float_)` – Modifiable
- * `lens_y_size (_Float_)` – Modifiable
- * `role_name (String)` – Modifiable
- * `sensor_tick (_Float_)` – Modifiable

- **sensor.lidar.ray_cast**

- **Attributes:**

- * `atmosphere_attenuation_rate (Float)` – Modifiable
- * `channels (Int)` – Modifiable
- * `dropoff_general_rate (Float)` – Modifiable
- * `dropoff_intensity_limit (Float)` – Modifiable
- * `dropoff_zero_intensity (Float)` – Modifiable
- * `lower_fov (Float)` – Modifiable
- * `noise_stddev (Float)` – Modifiable
- * `points_per_second (Int)` – Modifiable
- * `range (Float)` – Modifiable
- * `role_name (String)` – Modifiable
- * `rotation_frequency (Float)` – Modifiable
- * `sensor_tick (_Float_)` – Modifiable
- * `upper_fov (Float)` – Modifiable

- **sensor.lidar.ray_cast_semantic**

- **Attributes:**

- * `channels (Int)` – Modifiable
- * `lower_fov (Float)` – Modifiable
- * `points_per_second (Int)` – Modifiable
- * `range (Float)` – Modifiable
- * `role_name (String)` – Modifiable

- * `rotation_frequency` (`Float`) – Modifiable_
 - * `sensor_tick` (`_Float`) – Modifiable_
 - * `upper_fov` (`Float`) – Modifiable_
- `sensor.other.collision`
 - Attributes:
 - * `role_name` (`String`) – Modifiable_
- `sensor.other.gnss`
 - Attributes:
 - * `noise_alt_bias` (`Float`) – Modifiable_
 - * `noise_alt_stddev` (`Float`) – Modifiable_
 - * `noise_lat_bias` (`Float`) – Modifiable_
 - * `noise_lat_stddev` (`Float`) – Modifiable_
 - * `noise_lon_bias` (`Float`) – Modifiable_
 - * `noise_lon_stddev` (`Float`) – Modifiable_
 - * `noise_seed` (`Int`) – Modifiable_
 - * `role_name` (`String`) – Modifiable_
 - * `sensor_tick` (`_Float`) – Modifiable_
 - `sensor.other.imu`
 - Attributes:
 - * `noise_accel_stddev_x` (`Float`) – Modifiable_
 - * `noise_accel_stddev_y` (`Float`) – Modifiable_
 - * `noise_accel_stddev_z` (`Float`) – Modifiable_
 - * `noise_gyro_bias_x` (`Float`) – Modifiable_
 - * `noise_gyro_bias_y` (`Float`) – Modifiable_
 - * `noise_gyro_bias_z` (`Float`) – Modifiable_
 - * `noise_gyro_stddev_x` (`Float`) – Modifiable_
 - * `noise_gyro_stddev_y` (`Float`) – Modifiable_
 - * `noise_gyro_stddev_z` (`Float`) – Modifiable_
 - * `noise_seed` (`Int`) – Modifiable_
 - * `role_name` (`String`) – Modifiable_
 - * `sensor_tick` (`_Float`) – Modifiable_
 - `sensor.other.lane_invasion`
 - Attributes:
 - * `role_name` (`String`) – Modifiable_
 - `sensor.other.obstacle`
 - Attributes:
 - * `debug_linetrace` (`Bool`) – Modifiable_
 - * `distance` (`Float`) – Modifiable_
 - * `hit_radius` (`Float`) – Modifiable_
 - * `only_dynamics` (`Bool`) – Modifiable_
 - * `role_name` (`String`) – Modifiable_
 - * `sensor_tick` (`_Float`) – Modifiable_
 - `sensor.other.radar`
 - Attributes:
 - * `horizontal_fov` (`Float`) – Modifiable_
 - * `points_per_second` (`Int`) – Modifiable_
 - * `range` (`Float`) – Modifiable_
 - * `role_name` (`String`) – Modifiable_
 - * `sensor_tick` (`_Float`) – Modifiable_
 - * `vertical_fov` (`Float`) – Modifiable_
 - `sensor.other.rss`
 - Attributes:
 - * `role_name` (`String`) – Modifiable_

static

- `static.prop.advertisement`
 - Attributes:
 - * `role_name` (`String`) – Modifiable_
 - * `size` (`String`)
- `static.prop.atm`

- **Attributes:**
 - * `role_name` (*String*) – Modifiable
 - * `size` (*String*)
- **static.prop.barbeque**
 - **Attributes:**
 - * `role_name` (*String*) – Modifiable
 - * `size` (*String*)
- **static.prop.barrel**
 - **Attributes:**
 - * `role_name` (*String*) – Modifiable
 - * `size` (*String*)
- **static.prop.bench01**
 - **Attributes:**
 - * `role_name` (*String*) – Modifiable
 - * `size` (*String*)
- **static.prop.bench02**
 - **Attributes:**
 - * `role_name` (*String*) – Modifiable
 - * `size` (*String*)
- **static.prop.bench03**
 - **Attributes:**
 - * `role_name` (*String*) – Modifiable
 - * `size` (*String*)
- **static.prop.bike helmet**
 - **Attributes:**
 - * `role_name` (*String*) – Modifiable
 - * `size` (*String*)
- **static.prop.bikeparking**
 - **Attributes:**
 - * `role_name` (*String*) – Modifiable
 - * `size` (*String*)
- **static.prop.bin**
 - **Attributes:**
 - * `role_name` (*String*) – Modifiable
 - * `size` (*String*)
- **static.prop.box01**
 - **Attributes:**
 - * `role_name` (*String*) – Modifiable
 - * `size` (*String*)
- **static.prop.box02**
 - **Attributes:**
 - * `role_name` (*String*) – Modifiable
 - * `size` (*String*)
- **static.prop.box03**
 - **Attributes:**
 - * `role_name` (*String*) – Modifiable
 - * `size` (*String*)
- **static.prop.briefcase**
 - **Attributes:**
 - * `role_name` (*String*) – Modifiable
 - * `size` (*String*)
- **static.prop.brokentile01**
 - **Attributes:**
 - * `role_name` (*String*) – Modifiable
 - * `size` (*String*)
- **static.prop.brokentile02**
 - **Attributes:**
 - * `role_name` (*String*) – Modifiable
 - * `size` (*String*)
- **static.prop.brokentile03**
 - **Attributes:**

- * `role_name` (`String`) – Modifiable_
 - * `size` (`String`)
- `static.prop.brokentile04`
 - Attributes:
 - * `role_name` (`String`) – Modifiable_
 - * `size` (`String`)
- `static.prop.busstop`
 - Attributes:
 - * `role_name` (`String`) – Modifiable_
 - * `size` (`String`)
- `static.prop.chainbarrier`
 - Attributes:
 - * `role_name` (`String`) – Modifiable_
 - * `size` (`String`)
- `static.prop.chainbarrierend`
 - Attributes:
 - * `role_name` (`String`) – Modifiable_
 - * `size` (`String`)
- `static.prop.clothcontainer`
 - Attributes:
 - * `role_name` (`String`) – Modifiable_
 - * `size` (`String`)
- `static.prop.clothesline`
 - Attributes:
 - * `role_name` (`String`) – Modifiable_
 - * `size` (`String`)
- `static.prop.colacan`
 - Attributes:
 - * `role_name` (`String`) – Modifiable_
 - * `size` (`String`)
- `static.prop.constructioncone`
 - Attributes:
 - * `role_name` (`String`) – Modifiable_
 - * `size` (`String`)
- `static.prop.container`
 - Attributes:
 - * `role_name` (`String`) – Modifiable_
 - * `size` (`String`)
- `static.prop.creasedbox01`
 - Attributes:
 - * `role_name` (`String`) – Modifiable_
 - * `size` (`String`)
- `static.prop.creasedbox02`
 - Attributes:
 - * `role_name` (`String`) – Modifiable_
 - * `size` (`String`)
- `static.prop.creasedbox03`
 - Attributes:
 - * `role_name` (`String`) – Modifiable_
 - * `size` (`String`)
- `static.prop.dirtdebris01`
 - Attributes:
 - * `role_name` (`String`) – Modifiable_
 - * `size` (`String`)
- `static.prop.dirtdebris02`
 - Attributes:
 - * `role_name` (`String`) – Modifiable_
 - * `size` (`String`)
- `static.prop.dirtdebris03`
 - Attributes:
 - * `role_name` (`String`) – Modifiable_
 - * `size` (`String`)

- * `size (String)`
- **static.prop.doghouse**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.fountain**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.garbage01**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.garbage02**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.garbage03**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.garbage04**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.garbage05**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.garbage06**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.gardenlamp**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.glasscontainer**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.gnome**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.guitarcase**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.ironplank**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.kiosk_01**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.mailbox**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`

- **static.prop.maptable**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.mobile**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.motorhelmet**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.pergola**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.plantpot01**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.plantpot02**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.plantpot03**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.plantpot04**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.plantpot05**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.plantpot06**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.plantpot07**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.plantpot08**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.plasticbag**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.plasticchair**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.plastictable**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`
- **static.prop.platformgarbage01**
 - Attributes:
 - * `role_name (String)` – Modifiable
 - * `size (String)`

- **Attributes:**
 - * `role_name` (`String`) – Modifiable
 - * `size` (`String`)
- **static.prop.purse**
 - **Attributes:**
 - * `role_name` (`String`) – Modifiable
 - * `size` (`String`)
- **static.prop.shop01**
 - **Attributes:**
 - * `role_name` (`String`) – Modifiable
 - * `size` (`String`)
- **static.prop.shoppingbag**
 - **Attributes:**
 - * `role_name` (`String`) – Modifiable
 - * `size` (`String`)
- **static.prop.shoppingcart**
 - **Attributes:**
 - * `role_name` (`String`) – Modifiable
 - * `size` (`String`)
- **static.prop.shoppingtrolley**
 - **Attributes:**
 - * `role_name` (`String`) – Modifiable
 - * `size` (`String`)
- **static.prop.slide**
 - **Attributes:**
 - * `role_name` (`String`) – Modifiable
 - * `size` (`String`)
- **static.prop.streetbarrier**
 - **Attributes:**
 - * `role_name` (`String`) – Modifiable
 - * `size` (`String`)
- **static.prop.streetfountain**
 - **Attributes:**
 - * `role_name` (`String`) – Modifiable
 - * `size` (`String`)
- **static.prop.streetsign**
 - **Attributes:**
 - * `role_name` (`String`) – Modifiable
 - * `size` (`String`)
- **static.prop.streetsign01**
 - **Attributes:**
 - * `role_name` (`String`) – Modifiable
 - * `size` (`String`)
- **static.prop.streetsign04**
 - **Attributes:**
 - * `role_name` (`String`) – Modifiable
 - * `size` (`String`)
- **static.prop.swing**
 - **Attributes:**
 - * `role_name` (`String`) – Modifiable
 - * `size` (`String`)
- **static.prop.swingcouch**
 - **Attributes:**
 - * `role_name` (`String`) – Modifiable
 - * `size` (`String`)
- **static.prop.table**
 - **Attributes:**
 - * `role_name` (`String`) – Modifiable
 - * `size` (`String`)
- **static.prop.trafficcone01**
 - **Attributes:**

- * `role_name` (*String*) – Modifiable_
 - * `size` (*String*)
- **static.prop.trafficcone02**
 - Attributes:
 - * `role_name` (*String*) – Modifiable_
 - * `size` (*String*)
- **static.prop.trafficwarning**
 - Attributes:
 - * `role_name` (*String*) – Modifiable_
 - * `size` (*String*)
- **static.prop.trampoline**
 - Attributes:
 - * `role_name` (*String*) – Modifiable_
 - * `size` (*String*)
- **static.prop.trashbag**
 - Attributes:
 - * `role_name` (*String*) – Modifiable_
 - * `size` (*String*)
- **static.prop.trashcan01**
 - Attributes:
 - * `role_name` (*String*) – Modifiable_
 - * `size` (*String*)
- **static.prop.trashcan02**
 - Attributes:
 - * `role_name` (*String*) – Modifiable_
 - * `size` (*String*)
- **static.prop.trashcan03**
 - Attributes:
 - * `role_name` (*String*) – Modifiable_
 - * `size` (*String*)
- **static.prop.trashcan04**
 - Attributes:
 - * `role_name` (*String*) – Modifiable_
 - * `size` (*String*)
- **static.prop.trashcan05**
 - Attributes:
 - * `role_name` (*String*) – Modifiable_
 - * `size` (*String*)
- **static.prop.travelcase**
 - Attributes:
 - * `role_name` (*String*) – Modifiable_
 - * `size` (*String*)
- **static.prop.vendingmachine**
 - Attributes:
 - * `role_name` (*String*) – Modifiable_
 - * `size` (*String*)
- **static.prop.wateringcan**
 - Attributes:
 - * `role_name` (*String*) – Modifiable_
 - * `size` (*String*)
- **static.trigger.friction**
 - Attributes:
 - * `extent_x` (*Float*) – Modifiable_
 - * `extent_y` (*Float*) – Modifiable_
 - * `extent_z` (*Float*) – Modifiable_
 - * `friction` (*Float*) – Modifiable_
 - * `role_name` (*String*) – Modifiable_

vehicle

- **vehicle.audi.a2**

- **Attributes:**
 - * `color (RGBColor)` – Modifiable
 - * `number_of_wheels (Int)`
 - * `object_type (_String_)`
 - * `role_name (String)` – Modifiable
 - * `sticky_control (Bool)` – Modifiable
- `vehicle.audi.etron`
 - **Attributes:**
 - * `color (RGBColor)` – Modifiable
 - * `number_of_wheels (Int)`
 - * `object_type (_String_)`
 - * `role_name (String)` – Modifiable
 - * `sticky_control (Bool)` – Modifiable
- `vehicle.audi.tt`
 - **Attributes:**
 - * `color (RGBColor)` – Modifiable
 - * `number_of_wheels (Int)`
 - * `object_type (_String_)`
 - * `role_name (String)` – Modifiable
 - * `sticky_control (Bool)` – Modifiable
- `vehicle.bh.crossbike`
 - **Attributes:**
 - * `color (RGBColor)` – Modifiable
 - * `driver_id (Int)` – Modifiable
 - * `number_of_wheels (Int)`
 - * `object_type (_String_)`
 - * `role_name (String)` – Modifiable
 - * `sticky_control (Bool)` – Modifiable
- `vehicle.bmw.grandtourer`
 - **Attributes:**
 - * `color (RGBColor)` – Modifiable
 - * `number_of_wheels (Int)`
 - * `object_type (_String_)`
 - * `role_name (String)` – Modifiable
 - * `sticky_control (Bool)` – Modifiable
- `vehicle.bmw.isetta`
 - **Attributes:**
 - * `color (RGBColor)` – Modifiable
 - * `number_of_wheels (Int)`
 - * `object_type (_String_)`
 - * `role_name (String)` – Modifiable
 - * `sticky_control (Bool)` – Modifiable
- `vehicle.carlamotors.carlacola`
 - **Attributes:**
 - * `color (RGBColor)` – Modifiable
 - * `number_of_wheels (Int)`
 - * `object_type (_String_)`
 - * `role_name (String)` – Modifiable
 - * `sticky_control (Bool)` – Modifiable
- `vehicle.chevrolet.impala`
 - **Attributes:**
 - * `color (RGBColor)` – Modifiable
 - * `number_of_wheels (Int)`
 - * `object_type (_String_)`
 - * `role_name (String)` – Modifiable
 - * `sticky_control (Bool)` – Modifiable
- `vehicle.citroen.c3`
 - **Attributes:**
 - * `color (RGBColor)` – Modifiable
 - * `number_of_wheels (Int)`
 - * `object_type (_String_)`

- * `role_name` (`String`) – Modifiable_
 - * `sticky_control` (`Bool`) – Modifiable_
- **vehicle.diamondback.century**
 - **Attributes:**
 - * `color` (`RGBColor`) – Modifiable_
 - * `driver_id` (`Int`) – Modifiable_
 - * `number_of_wheels` (`Int`)
 - * `object_type` (`_String`)
 - * `role_name` (`String`) – Modifiable_
 - * `sticky_control` (`Bool`) – Modifiable_
 - **vehicle.dodge_charger.police**
 - **Attributes:**
 - * `color` (`RGBColor`) – Modifiable_
 - * `number_of_wheels` (`Int`)
 - * `object_type` (`_String`)
 - * `role_name` (`String`) – Modifiable_
 - * `sticky_control` (`Bool`) – Modifiable_
 - **vehicle.gazelle.omafiets**
 - **Attributes:**
 - * `color` (`RGBColor`) – Modifiable_
 - * `driver_id` (`Int`) – Modifiable_
 - * `number_of_wheels` (`Int`)
 - * `object_type` (`_String`)
 - * `role_name` (`String`) – Modifiable_
 - * `sticky_control` (`Bool`) – Modifiable_
 - **vehicle.harley-davidson.low_rider**
 - **Attributes:**
 - * `color` (`RGBColor`) – Modifiable_
 - * `driver_id` (`Int`) – Modifiable_
 - * `number_of_wheels` (`Int`)
 - * `object_type` (`_String`)
 - * `role_name` (`String`) – Modifiable_
 - * `sticky_control` (`Bool`) – Modifiable_
 - **vehicle.jeep.wrangler_rubicon**
 - **Attributes:**
 - * `color` (`RGBColor`) – Modifiable_
 - * `number_of_wheels` (`Int`)
 - * `object_type` (`_String`)
 - * `role_name` (`String`) – Modifiable_
 - * `sticky_control` (`Bool`) – Modifiable_
 - **vehicle.kawasaki.ninja**
 - **Attributes:**
 - * `color` (`RGBColor`) – Modifiable_
 - * `driver_id` (`Int`) – Modifiable_
 - * `number_of_wheels` (`Int`)
 - * `object_type` (`_String`)
 - * `role_name` (`String`) – Modifiable_
 - * `sticky_control` (`Bool`) – Modifiable_
 - **vehicle.lincoln.mkz2017**
 - **Attributes:**
 - * `color` (`RGBColor`) – Modifiable_
 - * `number_of_wheels` (`Int`)
 - * `object_type` (`_String`)
 - * `role_name` (`String`) – Modifiable_
 - * `sticky_control` (`Bool`) – Modifiable_
 - **vehicle.mercedes-benz.coupe**
 - **Attributes:**
 - * `color` (`RGBColor`) – Modifiable_
 - * `number_of_wheels` (`Int`)
 - * `object_type` (`_String`)
 - * `role_name` (`String`) – Modifiable_
 - * `sticky_control` (`Bool`) – Modifiable_

- * `sticky_control (Bool)` – Modifiable
- **vehicle.mini.cooperst**
 - **Attributes:**
 - * `color (RGBColor)` – Modifiable
 - * `number_of_wheels (Int)`
 - * `object_type (_String_)`
 - * `role_name (String)` – Modifiable
 - * `sticky_control (Bool)` – Modifiable
- **vehicle.mustang.mustang**
 - **Attributes:**
 - * `color (RGBColor)` – Modifiable
 - * `number_of_wheels (Int)`
 - * `object_type (_String_)`
 - * `role_name (String)` – Modifiable
 - * `sticky_control (Bool)` – Modifiable
- **vehicle.nissan.micra**
 - **Attributes:**
 - * `color (RGBColor)` – Modifiable
 - * `number_of_wheels (Int)`
 - * `object_type (_String_)`
 - * `role_name (String)` – Modifiable
 - * `sticky_control (Bool)` – Modifiable
- **vehicle.nissan.patrol**
 - **Attributes:**
 - * `color (RGBColor)` – Modifiable
 - * `number_of_wheels (Int)`
 - * `object_type (_String_)`
 - * `role_name (String)` – Modifiable
 - * `sticky_control (Bool)` – Modifiable
- **vehicle.seat.leon**
 - **Attributes:**
 - * `color (RGBColor)` – Modifiable
 - * `number_of_wheels (Int)`
 - * `object_type (_String_)`
 - * `role_name (String)` – Modifiable
 - * `sticky_control (Bool)` – Modifiable
- **vehicle.tesla.cybertruck**
 - **Attributes:**
 - * `number_of_wheels (Int)`
 - * `object_type (_String_)`
 - * `role_name (String)` – Modifiable
 - * `sticky_control (Bool)` – Modifiable
- **vehicle.tesla.model3**
 - **Attributes:**
 - * `color (RGBColor)` – Modifiable
 - * `number_of_wheels (Int)`
 - * `object_type (_String_)`
 - * `role_name (String)` – Modifiable
 - * `sticky_control (Bool)` – Modifiable
- **vehicle.toyota.prius**
 - **Attributes:**
 - * `color (RGBColor)` – Modifiable
 - * `number_of_wheels (Int)`
 - * `object_type (_String_)`
 - * `role_name (String)` – Modifiable
 - * `sticky_control (Bool)` – Modifiable
- **vehicle.volkswagen.t2**
 - **Attributes:**
 - * `color (RGBColor)` – Modifiable
 - * `number_of_wheels (Int)`
 - * `object_type (_String_)`

- * `role_name` (`String`) – Modifiable_
 - * `sticky_control` (`Bool`) – Modifiable_
- **vehicle.yamaha.yzf**
 - **Attributes:**
 - * `color` (`RGBColor`) – Modifiable_
 - * `driver_id` (`Int`) – Modifiable_
 - * `number_of_wheels` (`Int`)
 - * `object_type` (`_String`)
 - * `role_name` (`String`) – Modifiable_
 - * `sticky_control` (`Bool`) – Modifiable_

walker

- **walker.pedestrian.0001**
 - **Attributes:**
 - * `age` (`String`)
 - * `gender` (`String`)
 - * `is_invincible` (`Bool`) – Modifiable_
 - * `role_name` (`String`) – Modifiable_
 - * `speed` (`Float`) – Modifiable_
- **walker.pedestrian.0002**
 - **Attributes:**
 - * `age` (`String`)
 - * `gender` (`String`)
 - * `is_invincible` (`Bool`) – Modifiable_
 - * `role_name` (`String`) – Modifiable_
 - * `speed` (`Float`) – Modifiable_
- **walker.pedestrian.0003**
 - **Attributes:**
 - * `age` (`String`)
 - * `gender` (`String`)
 - * `is_invincible` (`Bool`) – Modifiable_
 - * `role_name` (`String`) – Modifiable_
 - * `speed` (`Float`) – Modifiable_
- **walker.pedestrian.0004**
 - **Attributes:**
 - * `age` (`String`)
 - * `gender` (`String`)
 - * `is_invincible` (`Bool`) – Modifiable_
 - * `role_name` (`String`) – Modifiable_
 - * `speed` (`Float`) – Modifiable_
- **walker.pedestrian.0005**
 - **Attributes:**
 - * `age` (`String`)
 - * `gender` (`String`)
 - * `is_invincible` (`Bool`) – Modifiable_
 - * `role_name` (`String`) – Modifiable_
 - * `speed` (`Float`) – Modifiable_
- **walker.pedestrian.0006**
 - **Attributes:**
 - * `age` (`String`)
 - * `gender` (`String`)
 - * `is_invincible` (`Bool`) – Modifiable_
 - * `role_name` (`String`) – Modifiable_
 - * `speed` (`Float`) – Modifiable_
- **walker.pedestrian.0007**
 - **Attributes:**
 - * `age` (`String`)
 - * `gender` (`String`)
 - * `is_invincible` (`Bool`) – Modifiable_
 - * `role_name` (`String`) – Modifiable_

- * speed (*Float*) – Modifiable
- **walker.pedestrian.0008**
 - Attributes:
 - * age (*String*)
 - * gender (*String*)
 - * is_invincible (*Bool*) – Modifiable
 - * role_name (*String*) – Modifiable
 - * speed (*Float*) – Modifiable
- **walker.pedestrian.0009**
 - Attributes:
 - * age (*String*)
 - * gender (*String*)
 - * is_invincible (*Bool*) – Modifiable
 - * role_name (*String*) – Modifiable
 - * speed (*Float*) – Modifiable
- **walker.pedestrian.0010**
 - Attributes:
 - * age (*String*)
 - * gender (*String*)
 - * is_invincible (*Bool*) – Modifiable
 - * role_name (*String*) – Modifiable
 - * speed (*Float*) – Modifiable
- **walker.pedestrian.0011**
 - Attributes:
 - * age (*String*)
 - * gender (*String*)
 - * is_invincible (*Bool*) – Modifiable
 - * role_name (*String*) – Modifiable
 - * speed (*Float*) – Modifiable
- **walker.pedestrian.0012**
 - Attributes:
 - * age (*String*)
 - * gender (*String*)
 - * is_invincible (*Bool*) – Modifiable
 - * role_name (*String*) – Modifiable
 - * speed (*Float*) – Modifiable
- **walker.pedestrian.0013**
 - Attributes:
 - * age (*String*)
 - * gender (*String*)
 - * is_invincible (*Bool*) – Modifiable
 - * role_name (*String*) – Modifiable
 - * speed (*Float*) – Modifiable
- **walker.pedestrian.0014**
 - Attributes:
 - * age (*String*)
 - * gender (*String*)
 - * is_invincible (*Bool*) – Modifiable
 - * role_name (*String*) – Modifiable
 - * speed (*Float*) – Modifiable
- **walker.pedestrian.0015**
 - Attributes:
 - * age (*String*)
 - * gender (*String*)
 - * is_invincible (*Bool*) – Modifiable
 - * role_name (*String*) – Modifiable
 - * speed (*Float*) – Modifiable

5.2 C++ Reference

We use Doxygen to generate the documentation of our C++ code:



Document updates are done automatically by GitHub.

Create doxygen documentation



[Doxygen](<http://www.doxygen.nl/index.html>) is required to generate [Graphviz](<https://www.graphviz.org/>) for the graph drawing toolkit.

1- Install doxygen and graphviz with the following command:

```
1 # linux  
2 > sudo apt-get install doxygen graphviz
```

2- Once installed, go to the project root folder where the *Doxyfile* file is situated and run the following command:

```
1 > doxygen
```

It will start to build the documentation webpage. The resulting webpage can be found at [Doxygen/html/](http://carla.org/Doxygen/html/)

3- Open *index.html* in a browser. Now you have your local cpp documentation!

5.3 Recorder Binary File Format

The recorder system saves all the info needed to replay the simulation in a binary file, using little endian byte order for the multibyte values.

- **1- Strings in binary**
- **2- Info header**
- **3- Packets**
 - Packet 0 - Frame Start
 - Packet 1 - Frame End
 - Packet 2 - Event Add
 - Packet 3 - Event Del
 - Packet 4 - Event Parent
 - Packet 5 - Event Collision
 - Packet 6 - Position
 - Packet 7 - TrafficLight
 - Packet 8 - Vehicle Animation
 - Packet 9 - Walker Animation
- **4- Frame Layout**
- **5- File Layout**

In the next image representing the file format, we can get a quick view of all the detailed information. Each part that is visualized in the image will be explained in the following sections:

In summary, the file format has a small header with general info (version, magic string, date and the map used) and a collection of packets of different types (currently we use 10 types, but that will continue growing up in the future).

1- Strings in binary

Strings are encoded first with the length of it, followed by its characters without null character ending. For example, the string ‘Town06’ will be saved as hex values: 06 00 54 6f 77 6e 30 36

2- Info header

The info header has general information about the recorded file. Basically, it contains the version and a magic string to identify the file as a recorder file. If the header changes then the version will change also. Furthermore, it contains a date timestamp, with the number of seconds from the Epoch 1900, and also it contains a string with the name of the map that has been used for recording.

A sample info header is:

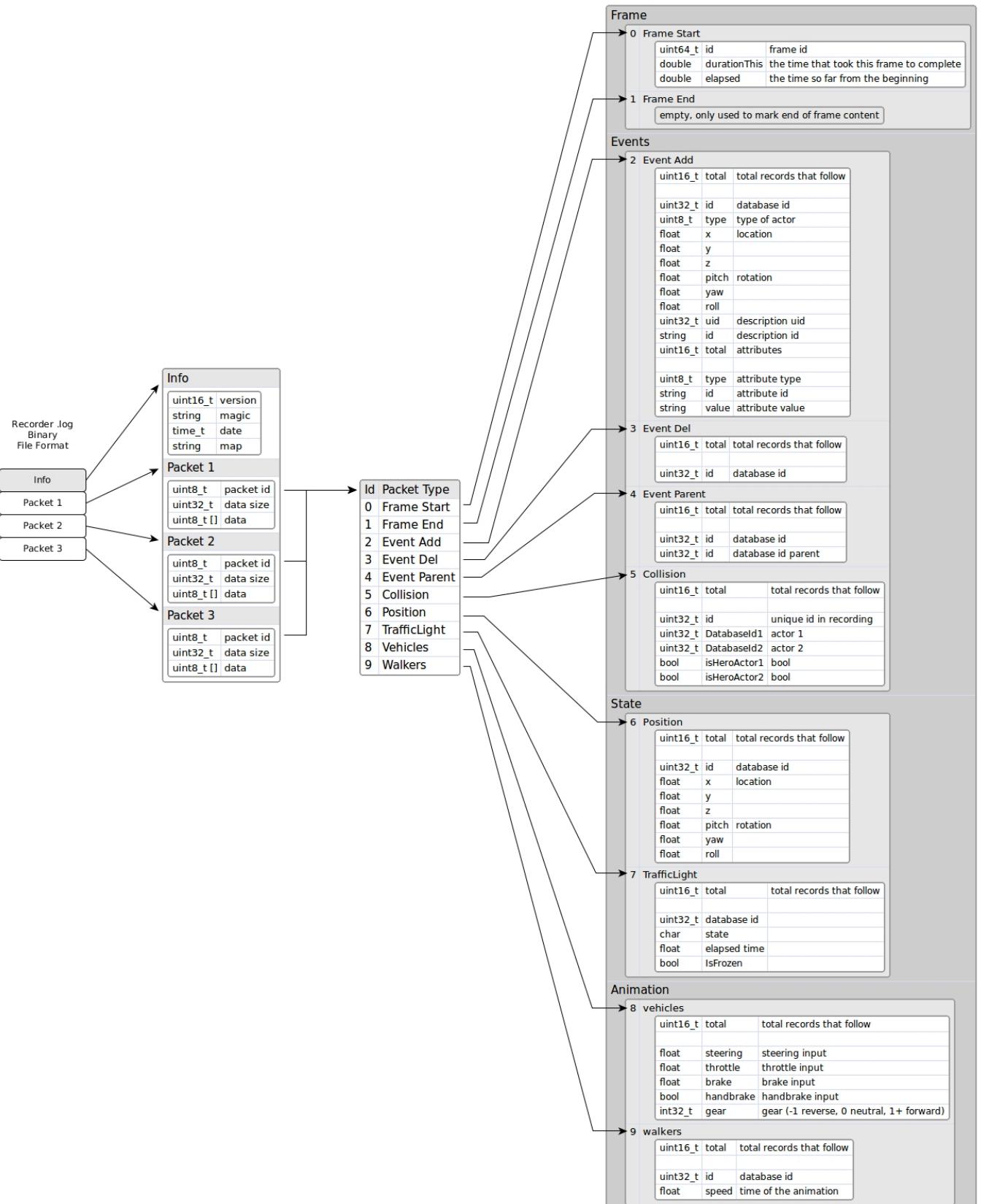


Figure 15: file format 1

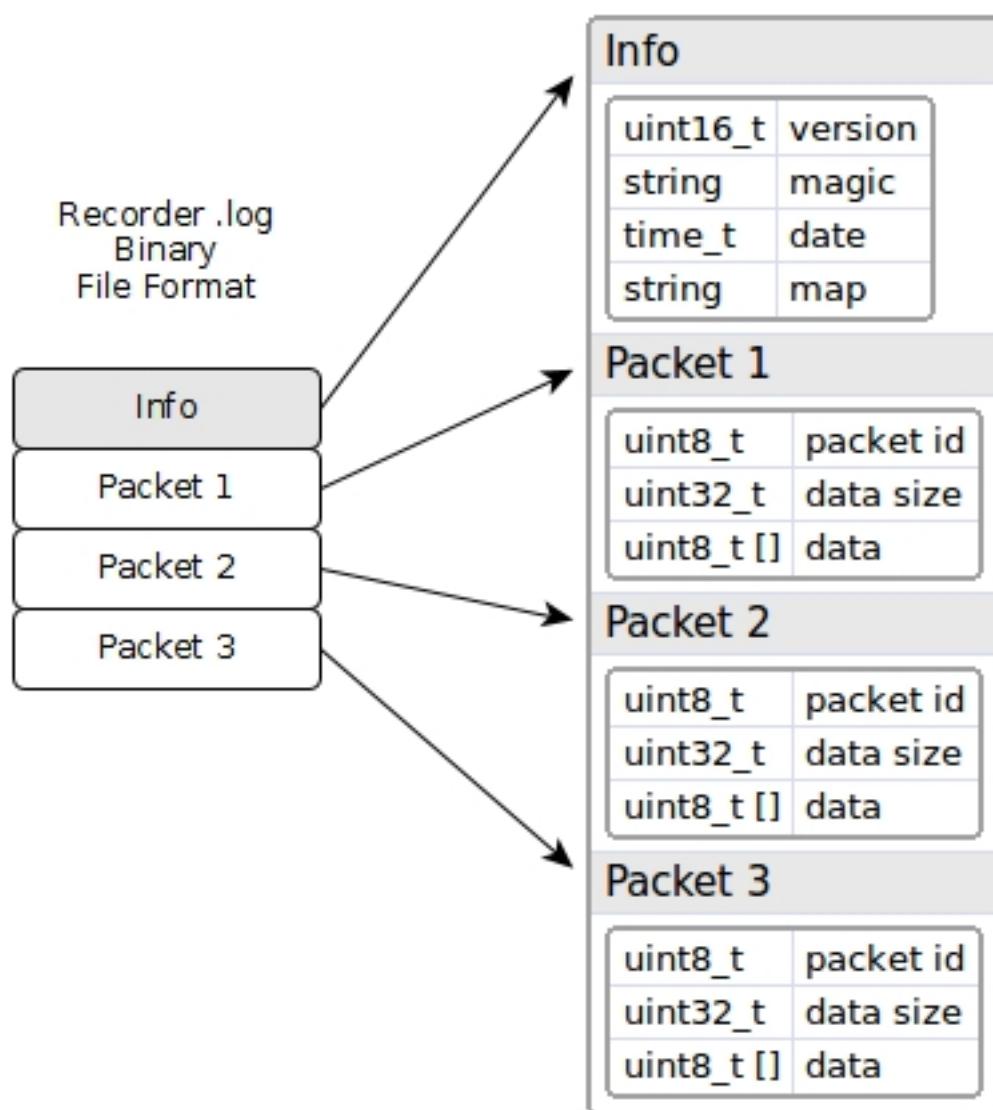


Figure 16: global file format

| | | | | | | | |
|----|----|-----|-----|-----|-----|-----|-----|
| 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
| 06 | 00 | 54 | 6f | 77 | 6e | 30 | 36 |
| 06 | 00 | 'T' | 'o' | 'w' | 'n' | '0' | '6' |

Figure 17: binary dynamic string

| Info | |
|----------|--------------------------|
| uint16_t | version |
| string | magic = 'CARLA_RECORDER' |
| time_t | date |
| string | map |

Figure 18: info header

| raw hex values | | | | | | | | | | | | | | | | |
|----------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|--|
| 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0a | 0b | 0c | 0d | 0e | 0f | |
| 01 | 00 | 0E | 00 | 43 | 41 | 52 | 4C | 41 | 5F | 52 | 45 | 43 | 4F | 52 | 44 | |
| 45 | 52 | 1F | 6D | AC | 5C | 00 | 00 | 00 | 00 | 06 | 00 | 54 | 6F | 77 | 6E | |
| 30 | 34 | | | | | | | | | | | | | | | |

| hex values grouped by fields | | | | | | | | | | | | | | | | |
|------------------------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|-------------------|
| version | 01 | 00 | | | | | | | | | | | | | | 1.0 |
| magic | 0E | 00 | 43 | 41 | 52 | 4C | 41 | 5F | 52 | 45 | 43 | 4F | 52 | 44 | 45 | 52 |
| date | 1F | 6D | AC | 5C | 00 | 00 | 00 | 00 | | | | | | | | 04/09/19 11:59:59 |
| map | 06 | 00 | 54 | 6F | 77 | 6E | 30 | 34 | | | | | | | | 'Town04' |

Figure 19: info header sample

3- Packets

Each packet starts with a little header of two fields (5 bytes):

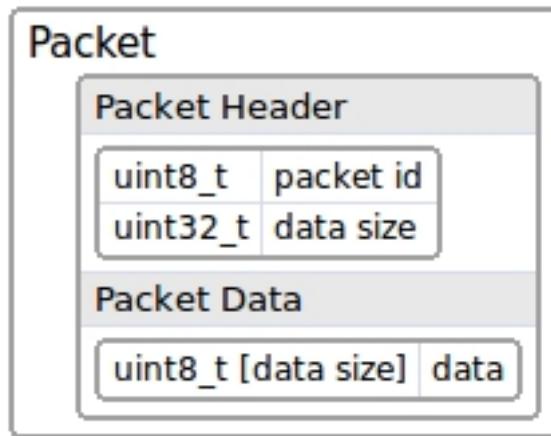


Figure 20: packet header

- **id:** The packet type
- **size:** Size of packet data

Header information is then followed by the **data**. The **data** is optional, a **size** of 0 means there is no **data** in the packet. If the **size** is greater than 0 it means that the packet has **data** bytes. Therefore, the **data** needs to be reinterpreted depending on the type of the packet.

The header of the packet is useful because we can just ignore those packets we are not interested in when doing playback. We only need to read the header (first 5 bytes) of the packet and jump to the next packet just skipping the data of the packet:

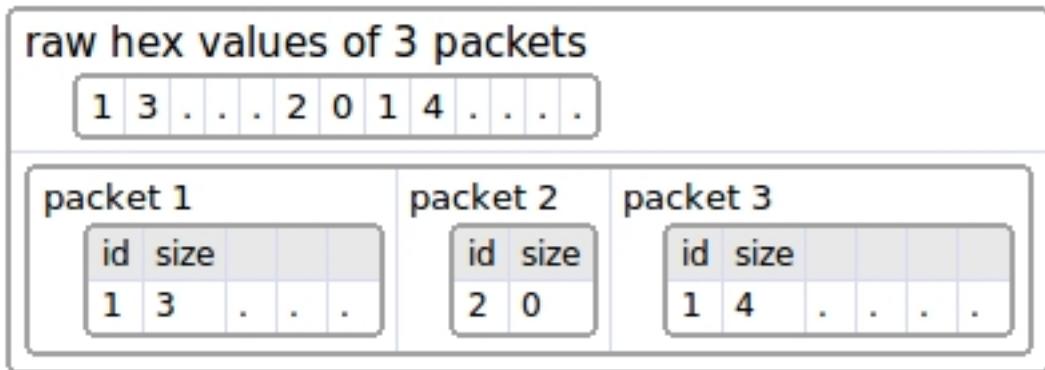


Figure 21: packets size

The types of packets are:

We suggest to use **id** over 100 for user custom packets, because this list will keep growing in the future.

Packet 0 - Frame Start This packet marks the start of a new frame, and it will be the first one to start each frame. All packets need to be placed between a **Frame Start** and a **Frame End**.

So, elapsed + durationThis = elapsed time for next frame

Packet 1 - Frame End This frame has no data and it only marks the end of the current frame. That helps the replayer to know the end of each frame just before the new one starts. Usually, the next frame should be a Frame Start packet to start a new frame.

Packet 2 - Event Add This packet says how many actors we need to create at current frame.

| Frame | | |
|-----------|--------------|-------------------------------------------------|
| id | | |
| 0 | Frame Start | mark the start of a new frame |
| 1 | Frame End | mark the end of current frame |
| Events | | |
| id | | |
| 2 | Event Add | say which actor needs to be created |
| 3 | Event Del | say which actor needs to be removed |
| 4 | Event Parent | say which actor needs to be parented with which |
| 5 | Collision | say which collision happened between two actors |
| State | | |
| id | | |
| 6 | Position | say the position of all actors |
| 7 | TrafficLight | say the state of all traffic lights |
| Animation | | |
| id | | |
| 8 | Vehicles | animate the wheels, bikes and cycles |
| 9 | Walkers | animate the walkers |

Figure 22: packets type list

| Frame Start | | |
|-------------|--------------|-------------------------------------------|
| uint64_t | id | frame id |
| double | durationThis | the time that took this frame to complete |
| double | elapsed | the time so far from the beginning |

Figure 23: frame start

Frame End
empty, only used to mark end of frame content

Figure 24: frame end

Event Add

| | | |
|----------|-------|---------------------------|
| uint16_t | total | total records that follow |
| uint32_t | id | database id |
| uint8_t | type | type of actor |
| float | x | location |
| float | y | |
| float | z | |
| float | pitch | rotation |
| float | yaw | |
| float | roll | |
| uint32_t | uid | description uid |
| string | id | description id |
| uint16_t | total | attributes |
| | | |
| uint8_t | type | attribute type |
| string | id | attribute id |
| string | value | attribute value |

Figure 25: event add

The field **total** says how many records follow. Each record starts with the **id** field, that is the id the actor has when it was recorded (on playback that id could change internally, but we need to use this id). The **type** of actor can have these possible values:

- 0 = Other
- 1 = Vehicle
- 2 = Walker
- 3 = TrafficLight
- 4 = INVALID

After that, the **location** and the **rotation** where we want to create the actor is proceeded.

Right after we have the **description** of the actor. The description **uid** is the numeric id of the description and the **id** is the textual id, like ‘vehicle.seat.leon’.

Then comes a collection of its **attributes** like color, number of wheels, role, etc. The number of attributes is variable and should look similar to this:

- number_of_wheels = 4
- sticky_control = true
- color = 79,33,85
- role_name = autopilot

Packet 3 - Event Del This packet says how many actors need to be destroyed this frame.

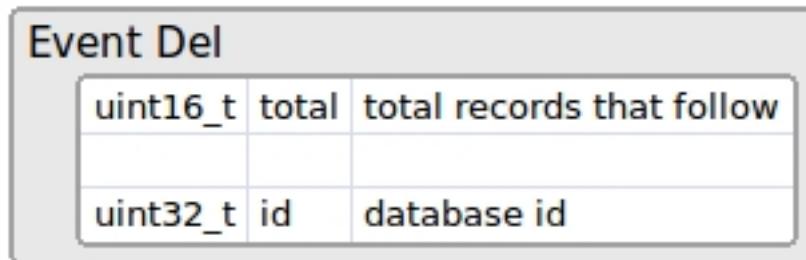


Figure 26: event del

It has the **total** of records, and each record has the **id** of the actor to remove.

For example, this packet could be like this:

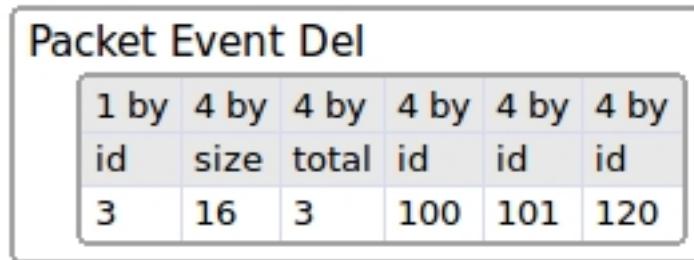


Figure 27: event del

The number 3 identifies the packet as (Event Del). The number 16 is the size of the data of the packet (4 fields of 4 bytes each). So if we don't want to process this packet, we could skip the next 16 bytes and will be directly to the start of the next packet. The next 3 says the total records that follows, and each record is the id of the actor to remove. So, we need to remove at this frame the actors 100, 101 and 120.

Packet 4 - Event Parent This packet says which actor is the child of another (the parent).

The first id is the child actor, and the second one will be the parent actor.

Event Parent

| <code>uint16_t</code> | <code>total</code> | <code>total records that follow</code> |
|-----------------------|--------------------|----------------------------------------|
| | | |
| <code>uint32_t</code> | <code>id</code> | <code>database id</code> |
| <code>uint32_t</code> | <code>id</code> | <code>database id parent</code> |

Figure 28: event parent

Packet 5 - Event Collision If a collision happens between two actors, it will be registered in this packet. Currently only actors with a collision sensor will report collisions, so currently only hero vehicles have that sensor attached automatically.

Collision

| <code>uint16_t</code> | <code>total</code> | <code>total records that follow</code> |
|-----------------------|---------------------------|----------------------------------------|
| | | |
| <code>uint32_t</code> | <code>id</code> | <code>unique id in recording</code> |
| <code>uint32_t</code> | <code>DatabaseId1</code> | <code>actor 1</code> |
| <code>uint32_t</code> | <code>DatabaseId2</code> | <code>actor 2</code> |
| <code>bool</code> | <code>isHeroActor1</code> | <code>bool</code> |
| <code>bool</code> | <code>isHeroActor2</code> | <code>bool</code> |

Figure 29: event collision

The **id** is just a sequence to identify each collision internally. Several collisions between the same pair of actors can happen in the same frame, because physics frame rate is fixed and usually there are several physics substeps in the same rendered frame.

Packet 6 - Position This packet records the position and orientation of all actors of type **vehicle** and **walker** that exist in the scene.

Packet 7 - TrafficLight This packet records the state of all **traffic lights** in the scene. Which means that it stores the state (red, orange or green) and the time it is waiting to change to a new state.

Packet 8 - Vehicle animation This packet records the animation of the vehicles, bikes and cycles. This packet stores the **throttle**, **sterring**, **brake**, **handbrake** and **gear** inputs, and then set them at playback.

Packet 9 - Walker animation This packet records the animation of the walker. It just saves the **speed** of the walker that is used in the animation.

4- Frame Layout

A frame consists of several packets, where all of them are optional, except the ones that have the **start** and **end** in that frame, that must be there always.

Event packets exist only in the frame where they happen.

Position

| uint16_t | total | total records that follow |
|----------|-------|---------------------------|
| | | |
| uint32_t | id | database id |
| float | x | location |
| float | y | |
| float | z | |
| float | pitch | rotation |
| float | yaw | |
| float | roll | |

Figure 30: position

TrafficLight

| uint16_t | total | total records that follow |
|----------|--------------|---------------------------|
| | | |
| uint32_t | database id | |
| char | state | |
| float | elapsed time | |
| bool | IsFrozen | |

Figure 31: state

vehicle

| uint16_t | total | total records that follow |
|----------|-----------|------------------------------------------|
| | | |
| uint32_t | id | database id |
| float | steering | steering input |
| float | throttle | throttle input |
| float | brake | brake input |
| bool | handbrake | handbrake input |
| int32_t | gear | gear (-1 reverse, 0 neutral, 1+ forward) |

Figure 32: state

| walker | | |
|----------|-------|---------------------------|
| uint16_t | total | total records that follow |
| uint32_t | id | database id |
| float | speed | time of the animation |

Figure 33: state



Figure 34: layout

Position and **traffic light** packets should exist in all frames, because they are required to move all actors and set the traffic lights to its state. They are optional but if they are not present then the replayer will not be able to move or set the state of traffic lights.

The **animation** packets are also optional, but by default they are recorded. That way the walkers are animated and also the vehicle wheels follow the direction of the vehicles.

5- File Layout

The layout of the file starts with the **info header** and then follows a collection of packets in groups. The first in each group is the **Frame Start** packet, and the last in the group is the **Frame End** packet. In between, we can find the rest of packets as well.

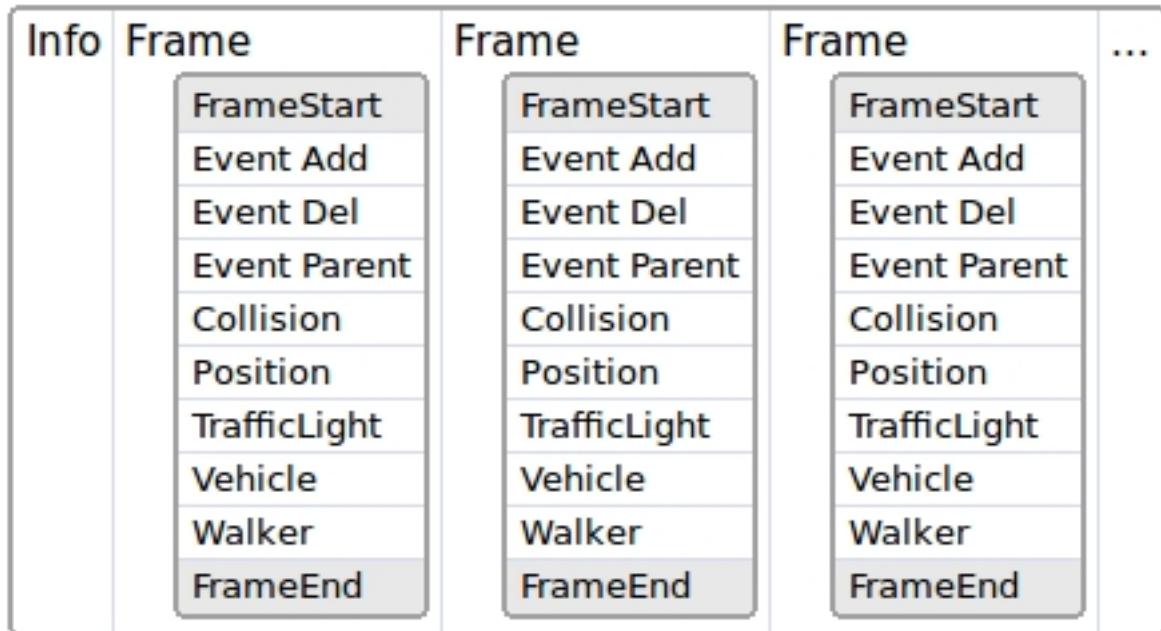


Figure 35: layout

Usually, it is a good idea to have all packets regarding events first, and then the packets regarding position and state later.

The event packets are optional, since they appear when they happen, so we could have a layout like this one:

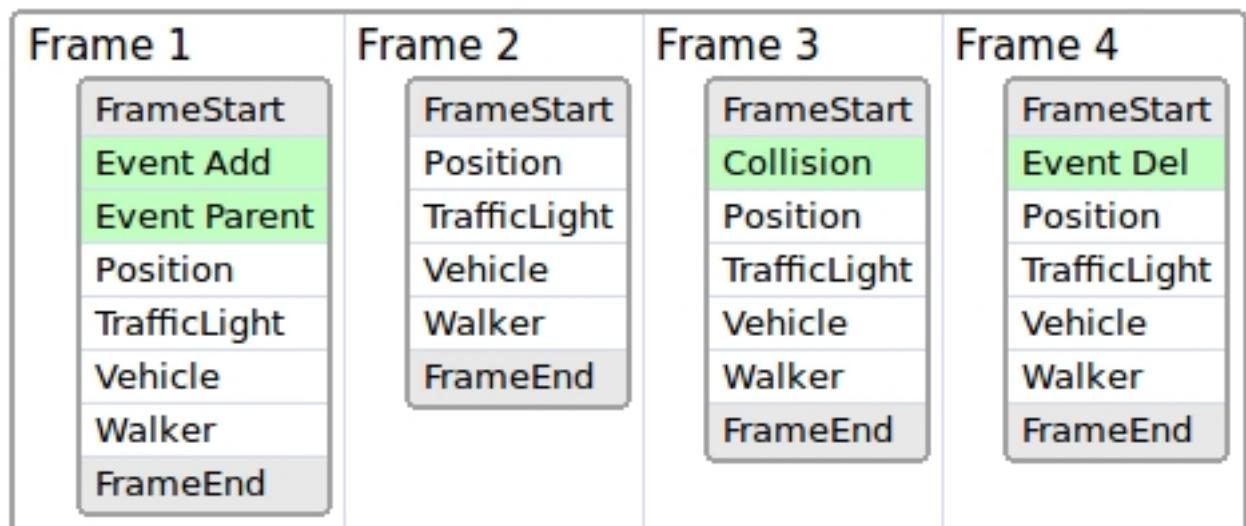


Figure 36: layout

In **frame 1** some actors are created and reparented, so we can observe its events in the image. In **frame 2** there are no events. In **frame 3** some actors have collided so the collision event appears with that info. In **frame 4** the actors are destroyed.

5.4 Sensors reference

- Collision detector
- Depth camera
- GNSS sensor
- IMU sensor
- Lane invasion detector
- LIDAR sensor
- Obstacle detector
- Radar sensor
- RGB camera
- RSS sensor
- Semantic LIDAR sensor
- Semantic segmentation camera
- DVS camera

Collision detector

- **Blueprint:** sensor.other.collision
- **Output:** carla.CollisionEvent per collision.

This sensor registers an event each time its parent actor collides against something in the world. Several collisions may be detected during a single simulation step. To ensure that collisions with any kind of object are detected, the server creates “fake” actors for elements such as buildings or bushes so the semantic tag can be retrieved to identify it.

Collision detectors do not have any configurable attribute.

Output attributes Sensor data attribute

Type

Description

frame

int

Frame number when the measurement took place.

timestamp

double

Simulation time of the measurement in seconds since the beginning of the episode.

transform

Location and rotation in world coordinates of the sensor at the time of the measurement.

actor

Actor that measured the collision (sensor’s parent).

other_actor

Actor against whom the parent collided.

normal_impulse

Normal impulse result of the collision.

Depth camera

- **Blueprint:** sensor.camera.depth
- **Output:** carla.Image per step (unless `sensor_tick` says otherwise).

The camera provides a raw data of the scene codifying the distance of each pixel to the camera (also known as **depth buffer** or **z-buffer**) to create a depth map of the elements.

The image codifies depth value per pixel using 3 channels of the RGB color space, from less to more significant bytes: $R \rightarrow G \rightarrow B$. The actual distance in meters can be decoded with:

```
1 normalized = (R + G * 256 + B * 256 * 256) / (256 * 256 * 256 - 1)
2 in_meters = 1000 * normalized
```

The output carla.Image should then be saved to disk using a carla.colorConverter that will turn the distance stored in RGB channels into a $[0,1]$ float containing the distance and then translate this to grayscale. There are two options in carla.colorConverter to get a depth view: **Depth** and **Logarithmic depth**. The precision is milimetric in both, but the logarithmic approach provides better results for closer objects.

```
1 ...
2 raw_image.save_to_disk("path/to/save/converted/image",carla.Depth)
```

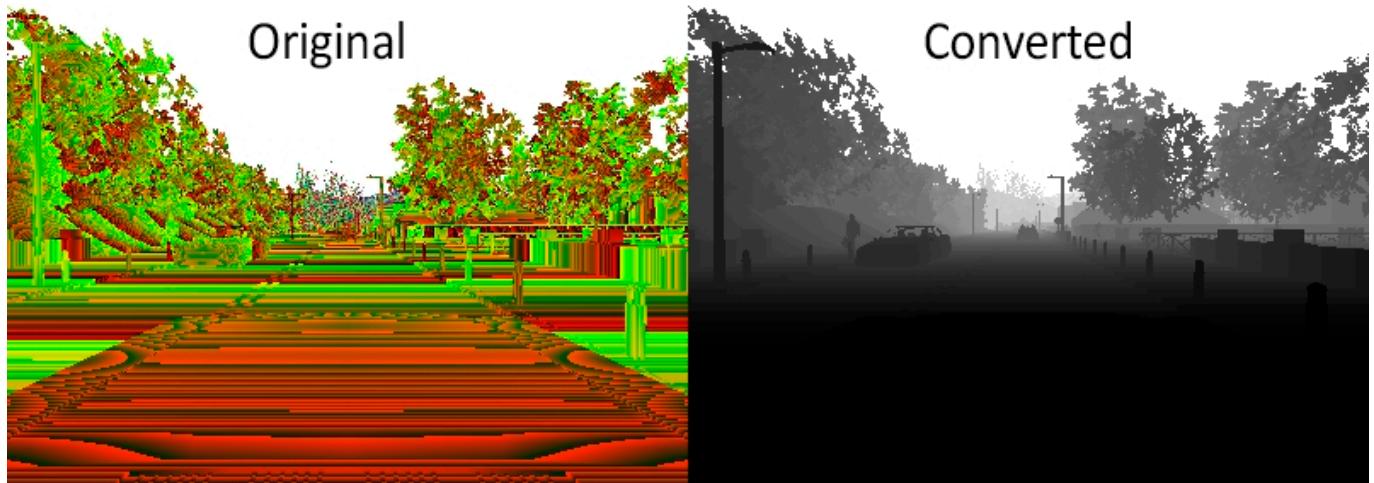


Figure 37: ImageDepth

Basic camera attributes Blueprint attribute

Type

Default

Description

image_size_x

int

800

Image width in pixels.

image_size_y

int

600

Image height in pixels.

fov

float

90.0

Horizontal field of view in degrees.

sensor_tick

float

0.0

Simulation seconds between sensor captures (ticks).

Camera lens distortion attributes Blueprint attribute

Type

Default

Description

lens_circle_falloff

float

5.0

Range: [0.0, 10.0]

lens_circle_multiplier

float

0.0

Range: [0.0, 10.0]

lens_k

float

-1.0

Range: [-inf, inf]

lens_kcube

float

0.0

Range: [-inf, inf]

lens_x_size

float

0.08

Range: [0.0, 1.0]

lens_y_size

float

0.08

Range: [0.0, 1.0]

Output attributes Sensor data attribute

Type

Description

frame

int

Frame number when the measurement took place.

timestamp
double
Simulation time of the measurement in seconds since the beginning of the episode.

transform
Location and rotation in world coordinates of the sensor at the time of the measurement.

width
int
Image width in pixels.

height
int
Image height in pixels.

fov
float
Horizontal field of view in degrees.

raw_data
bytes
Array of BGRA 32-bit pixels.

GNSS sensor

- **Blueprint:** sensor.other.gnss
- **Output:** carla.GNSSMeasurement per step (unless `sensor_tick` says otherwise).

Reports current gnss position of its parent object. This is calculated by adding the metric position to an initial geo reference location defined within the OpenDRIVE map definition.

GNSS attributes Blueprint attribute

Type

Default

Description

noise_alt_bias

float

0.0

Mean parameter in the noise model for altitude.

noise_alt_stddev

float

0.0

Standard deviation parameter in the noise model for altitude.

noise_lat_bias

float

0.0

Mean parameter in the noise model for latitude.

noise_lat_stddev

float

0.0
Standard deviation parameter in the noise model for latitude.
noise_lon_bias
float
0.0
Mean parameter in the noise model for longitude.
noise_lon_stddev
float
0.0
Standard deviation parameter in the noise model for longitude.
noise_seed
int
0
Initializer for a pseudorandom number generator.
sensor_tick
float
0.0
Simulation seconds between sensor captures (ticks).

Output attributes Sensor data attribute

Type
Description
frame
int
Frame number when the measurement took place.
timestamp
double
Simulation time of the measurement in seconds since the beginning of the episode.
transform
Location and rotation in world coordinates of the sensor at the time of the measurement.
latitude
double
Latitude of the actor.
longitude
double
Longitude of the actor.
altitude
double
Altitude of the actor.

IMU sensor

- **Blueprint:** sensor.other imu
- **Output:** carla.IMUMeasurement per step (unless `sensor_tick` says otherwise).

Provides measures that accelerometer, gyroscope and compass would retrieve for the parent object. The data is collected from the object's current state.

IMU attributes Blueprint attribute

Type

Default

Description

`noise_accel_stddev_x`

float

0.0

Standard deviation parameter in the noise model for acceleration (X axis).

`noise_accel_stddev_y`

float

0.0

Standard deviation parameter in the noise model for acceleration (Y axis).

`noise_accel_stddev_z`

float

0.0

Standard deviation parameter in the noise model for acceleration (Z axis).

`noise_gyro_bias_x`

float

0.0

Mean parameter in the noise model for the gyroscope (X axis).

`noise_gyro_bias_y`

float

0.0

Mean parameter in the noise model for the gyroscope (Y axis).

`noise_gyro_bias_z`

float

0.0

Mean parameter in the noise model for the gyroscope (Z axis).

`noise_gyro_stddev_x`

float

0.0

Standard deviation parameter in the noise model for the gyroscope (X axis).

`noise_gyro_stddev_y`

float

0.0

Standard deviation parameter in the noise model for the gyroscope (Y axis).

```

noise_gyro_stddev_z
float
0.0
Standard deviation parameter in the noise model for the gyroscope (Z axis).

noise_seed
int
0
Initializer for a pseudorandom number generator.

sensor_tick
float
0.0
Simulation seconds between sensor captures (ticks).

```

Output attributes Sensor data attribute

Type
 Description
 frame
 int
 Frame number when the measurement took place.

timestamp
 double
 Simulation time of the measurement in seconds since the beginning of the episode.

transform
 Location and rotation in world coordinates of the sensor at the time of the measurement.

accelerometer
 Measures linear acceleration in m/s².

gyroscope
 Measures angular velocity in rad/sec.
 compass
 float
 Orientation in radians. North is (0.0, -1.0, 0.0) in UE.

Lane invasion detector

- **Blueprint:** sensor.other.lane_invasion
- **Output:** carla.LaneInvasionEvent per crossing.

Registers an event each time its parent crosses a lane marking. The sensor uses road data provided by the OpenDRIVE description of the map to determine whether the parent vehicle is invading another lane by considering the space between wheels. However there are some things to be taken into consideration:

- Discrepancies between the OpenDRIVE file and the map will create irregularities such as crossing lanes that are not visible in the map.
- The output retrieves a list of crossed lane markings: the computation is done in OpenDRIVE and considering the whole space between the four wheels as a whole. Thus, there may be more than one lane being crossed at the same time.

This sensor does not have any configurable attribute.



| This sensor works fully on the client-side.

Output attributes Sensor data attribute

Type

Description

frame

int

Frame number when the measurement took place.

timestamp

double

Simulation time of the measurement in seconds since the beginning of the episode.

transform

Location and rotation in world coordinates of the sensor at the time of the measurement.

actor

Vehicle that invaded another lane (parent actor).

crossed_lane_markings

list()

List of lane markings that have been crossed.

LIDAR sensor

- **Blueprint:** sensor.lidar.ray_cast
- **Output:** carla.LidarMeasurement per step (unless `sensor_tick` says otherwise).

This sensor simulates a rotating LIDAR implemented using ray-casting. The points are computed by adding a laser for each channel distributed in the vertical FOV. The rotation is simulated computing the horizontal angle that the Lidar rotated in a frame. The point cloud is calculated by doing a ray-cast for each laser in every step.
`points_per_channel_each_step = points_per_second / (FPS * channels)`

A LIDAR measurement contains a package with all the points generated during a 1/FPS interval. During this interval the physics are not updated so all the points in a measurement reflect the same “static picture” of the scene.

This output contains a cloud of simulation points and thus, it can be iterated to retrieve a list of their `carla.Location`:

```
1 for location in lidar_measurement:  
2     print(location)
```

The information of the LIDAR measurement is encoded 4D points. Being the first three, the space points in xyz coordinates and the last one intensity loss during the travel. This intensity is computed by the following formula.

$$I/I_0 = e^{-a \cdot d}$$

a — Attenuation coefficient. This may depend on the sensor’s wavelength, and the conditions of the atmosphere. It can be modified with the LIDAR attribute `atmosphere_attenuation_rate`. **d** — Distance from the hit point to the sensor.

For a better realism, points in the cloud can be dropped off. This is an easy way to simulate loss due to external perturbations. This can be done combining two different.

- **General drop-off** — Proportion of points that are dropped off randomly. This is done before the tracking, meaning the points being dropped are not calculated, and therefore improves the performance. If `dropoff_general_rate = 0.5`, half of the points will be dropped.

- **Intensity-based drop-off** — For each point detected, an extra drop-off is performed with a probability based in the computed intensity. This probability is determined by two parameters. `dropoff_zero_intensity` is the probability of points with zero intensity to be dropped. `dropoff_intensity_limit` is a threshold intensity above which no points will be dropped. The probability of a point within the range to be dropped is a linear proportion based on these two parameters.

Additionally, the `noise_stddev` attribute makes for a noise model to simulate unexpected deviations that appear in real-life sensors. For positive values, each point is randomly perturbed along the vector of the laser ray. The result is a LIDAR sensor with perfect angular positioning, but noisy distance measurement.

The rotation of the LIDAR can be tuned to cover a specific angle on every simulation step (using a fixed time-step). For example, to rotate once per step (full circle output, as in the picture below), the rotation frequency and the simulated FPS should be equal. 1. Set the sensor's frequency `sensors_bp['lidar'][0].set_attribute('rotation_frequency', '10')`. 2. Run the simulation using `python3 config.py --fps=10`.

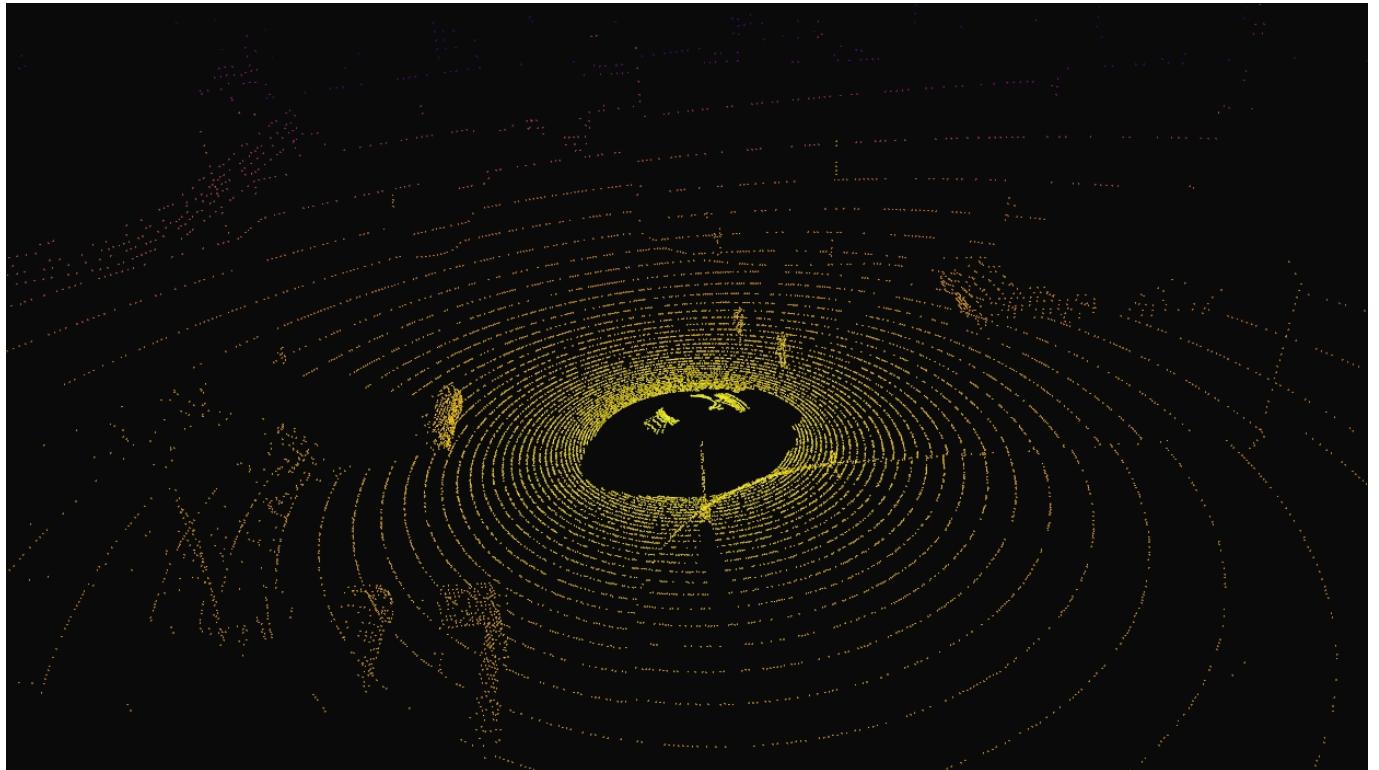


Figure 38: LidarPointCloud

Lidar attributes Blueprint attribute

Type

Default

Description

channels

int

32

Number of lasers.

range

float

10.0

Maximum distance to measure/raycast in meters (centimeters for CARLA 0.9.6 or previous).

points_per_second

int
56000
Points generated by all lasers per second.

rotation_frequency
float
10.0
LIDAR rotation frequency.

upper_fov
float
10.0
Angle in degrees of the highest laser.

lower_fov
float
-30.0
Angle in degrees of the lowest laser.

atmosphere_attenuation_rate
float
0.004
Coefficient that measures the LIDAR intensity loss per meter. Check the intensity computation above.

dropoff_general_rate
float
0.45
General proportion of points that are randomly dropped.

dropoff_intensity_limit
float
0.8
For the intensity based drop-off, the threshold intensity value above which no points are dropped.

dropoff_zero_intensity
float
0.4
For the intensity based drop-off, the probability of each point with zero intensity being dropped.

sensor_tick
float
0.0
Simulation seconds between sensor captures (ticks).

noise_stddev
float
0.0
Standard deviation of the noise model to disturb each point along the vector of its raycast.

Output attributes Sensor data attribute

Type

Description

frame

int

Frame number when the measurement took place.

timestamp

double

Simulation time of the measurement in seconds since the beginning of the episode.

transform

Location and rotation in world coordinates of the sensor at the time of the measurement.

horizontal_angle

float

Angle (radians) in the XY plane of the LIDAR in the current frame.

channels

int

Number of channels (lasers) of the LIDAR.

get_point_count(channel)

int

Number of points per channel captured this frame.

raw_data

bytes

Array of 32-bits floats (XYZI of each point).

Obstacle detector

- **Blueprint:** sensor.other.obstacle

- **Output:** carla.ObstacleDetectionEvent per obstacle (unless **sensor_tick** says otherwise).

Registers an event every time the parent actor has an obstacle ahead. In order to anticipate obstacles, the sensor creates a capsular shape ahead of the parent vehicle and uses it to check for collisions. To ensure that collisions with any kind of object are detected, the server creates “fake” actors for elements such as buildings or bushes so the semantic tag can be retrieved to identify it.

Blueprint attribute

Type

Default

Description

distance

float

5

Distance to trace.

hit_radius

float

0.5

Radius of the trace.
only_dynamics
bool
False
If true, the trace will only consider dynamic objects.
debug_linetrace
bool
False
If true, the trace will be visible.
sensor_tick
float
0.0
Simulation seconds between sensor captures (ticks).

Output attributes Sensor data attribute

Type
Description
frame
int
Frame number when the measurement took place.
timestamp
double
Simulation time of the measurement in seconds since the beginning of the episode.
transform
Location and rotation in world coordinates of the sensor at the time of the measurement.
actor
Actor that detected the obstacle (parent actor).
other_actor
Actor detected as an obstacle.
distance
float
Distance from actor to other_actor.

Radar sensor

- **Blueprint:** sensor.other.radar
- **Output:** carla.RadarMeasurement per step (unless `sensor_tick` says otherwise).

The sensor creates a conic view that is translated to a 2D point map of the elements in sight and their speed regarding the sensor. This can be used to shape elements and evaluate their movement and direction. Due to the use of polar coordinates, the points will concentrate around the center of the view.

Points measured are contained in `carla.RadarMeasurement` as an array of `carla.RadarDetection`, which specifies their polar coordinates, distance and velocity. This raw data provided by the radar sensor can be easily converted to a format manageable by **numpy**:

```

1 # To get a numpy [[vel, altitude, azimuth, depth],...[,,]]:
2 points = np.frombuffer(radar_data.raw_data, dtype=np.dtype('f4'))
3 points = np.reshape(points, (len(radar_data), 4))

```

The provided script `manual_control.py` uses this sensor to show the points being detected and paint them white when static, red when moving towards the object and blue when moving away:



Figure 39: ImageRadar

Blueprint attribute

Type

Default

Description

horizontal_fov

float

30.0

Horizontal field of view in degrees.

points_per_second

int

1500

Points generated by all lasers per second.

range

float

100

Maximum distance to measure/raycast in meters.

sensor_tick

float

0.0

Simulation seconds between sensor captures (ticks).

vertical_fov

float

30.0

Vertical field of view in degrees.

Output attributes Sensor data attribute

Type

Description

raw_data

The list of points detected.

RadarDetection attributes

Type

Description

altitude

float

Altitude angle in radians.

azimuth

float

Azimuth angle in radians.

depth

float

Distance in meters.

velocity

float

Velocity towards the sensor.

RGB camera

- **Blueprint:** sensor.camera.rgb
- **Output:** carla.Image per step (unless `sensor_tick` says otherwise)..

The “RGB” camera acts as a regular camera capturing images from the scene. carla.colorConverter

If `enable_postprocess_effects` is enabled, a set of post-process effects is applied to the image for the sake of realism:

- **Vignette:** darkens the border of the screen.
- **Grain jitter:** adds some noise to the render.
- **Bloom:** intense lights burn the area around them.
- **Auto exposure:** modifies the image gamma to simulate the eye adaptation to darker or brighter areas.
- **Lens flares:** simulates the reflection of bright objects on the lens.
- **Depth of field:** blurs objects near or very far away of the camera.

The `sensor_tick` tells how fast we want the sensor to capture the data. A value of 1.5 means that we want the sensor to capture data each second and a half. By default a value of 0.0 means as fast as possible.

Basic camera attributes Blueprint attribute

Type

Default

Description

bloom_intensity

float

0.675

Intensity for the bloom post-process effect, 0.0 for disabling it.

fov



Figure 40: ImageRGB

```
float  
90.0  
Horizontal field of view in degrees.  
fstop  
float  
1.4  
Opening of the camera lens. Aperture is 1/fstop with typical lens going down to f/1.2 (larger opening). Larger numbers will reduce the Depth of Field effect.  
image_size_x  
int  
800  
Image width in pixels.  
image_size_y  
int  
600  
Image height in pixels.  
iso  
float  
100.0  
The camera sensor sensitivity.  
gamma  
float
```

2.2

Target gamma value of the camera.

lens_flare_intensity

float

0.1

Intensity for the lens flare post-process effect, 0.0 for disabling it.

sensor_tick

float

0.0

Simulation seconds between sensor captures (ticks).

shutter_speed

float

200.0

The camera shutter speed in seconds (1.0/s).

Camera lens distortion attributes Blueprint attribute

Type

Default

Description

lens_circle_falloff

float

5.0

Range: [0.0, 10.0]

lens_circle_multiplier

float

0.0

Range: [0.0, 10.0]

lens_k

float

-1.0

Range: [-inf, inf]

lens_kcube

float

0.0

Range: [-inf, inf]

lens_x_size

float

0.08

Range: [0.0, 1.0]

lens_y_size

float

0.08

Range: [0.0, 1.0]

Advanced camera attributes Since these effects are provided by UE, please make sure to check their documentation:

- Automatic Exposure
- Cinematic Depth of Field Method
- Color Grading and Filmic Tonemapper

Blueprint attribute

Type

Default

Description

min_fstop

float

1.2

Maximum aperture.

blade_count

int

5

Number of blades that make up the diaphragm mechanism.

exposure_mode

str

histogram

Can be manual or histogram. More in .

exposure_compensation

float

Linux: -1.5 Windows: 0.0

Logarithmic adjustment for the exposure. 0: no adjustment, -1:2x darker, -2:4 darker, 1:2x brighter, 2:4x brighter.

exposure_min_bright

float

7.0

In `exposure_mode: "histogram"`. Minimum brightness for auto exposure. The lowest the eye can adapt within. Must be greater than 0 and less than or equal to `exposure_max_bright`.

exposure_max_bright

float

9.0

In `exposure_mode: "histogram"`. Maximum brightness for auto exposure. The highest the eye can adapt within. Must be greater than 0 and greater than or equal to `exposure_min_bright`.

exposure_speed_up

float

3.0

In `exposure_mode: "histogram"`. Speed at which the adaptation occurs from dark to bright environment.

exposure_speed_down

float
1.0
In exposure_mode: "histogram". Speed at which the adaptation occurs from bright to dark environment.

calibration_constant
float
16.0
Calibration constant for 18% albedo.

focal_distance
float
1000.0
Distance at which the depth of field effect should be sharp. Measured in cm (UE units).

blur_amount
float
1.0
Strength/intensity of motion blur.

blur_radius
float
0.0
Radius in pixels at 1080p resolution to emulate atmospheric scattering according to distance from camera.

motion_blur_intensity
float
0.45
Strength of motion blur [0,1].

motion_blur_max_distortion
float
0.35
Max distortion caused by motion blur. Percentage of screen width.

motion_blur_min_object_screen_size
float
0.1
Percentage of screen width objects must have for motion blur, lower value means less draw calls.

slope
float
0.88
Steepness of the S-curve for the tonemapper. Larger values make the slope steeper (darker) [0.0, 1.0].

toe
float
0.55
Adjusts dark color in the tonemapper [0.0, 1.0].

shoulder
float

0.26

Adjusts bright color in the tonemapper [0.0, 1.0].

black_clip

float

0.0

This should NOT be adjusted. Sets where the crossover happens and black tones start to cut off their value [0.0, 1.0].

white_clip

float

0.04

Set where the crossover happens and white tones start to cut off their value. Subtle change in most cases [0.0, 1.0].

temp

float

6500.0

White balance in relation to the temperature of the light in the scene. White light: when this matches light temperature. Warm light: When higher than the light in the scene, it is a yellowish color. Cool light: When lower than the light. Blueish color.

tint

float

0.0

White balance temperature tint. Adjusts cyan and magenta color ranges. This should be used along with the white balance Temp property to get accurate colors. Under some light temperatures, the colors may appear to be more yellow or blue. This can be used to balance the resulting color to look more natural.

chromatic_aberration_intensity

float

0.0

Scaling factor to control color shifting, more noticeable on the screen borders.

chromatic_aberration_offset

float

0.0

Normalized distance to the center of the image where the effect takes place.

enable_postprocess_effects

bool

True

Post-process effects activation.

Output attributes Sensor data attribute

Type

Description

frame

int

Frame number when the measurement took place.

timestamp

```

double
Simulation time of the measurement in seconds since the beginning of the episode.

transform
Location and rotation in world coordinates of the sensor at the time of the measurement.

width
int
Image width in pixels.

height
int
Image height in pixels.

fov
float
Horizontal field of view in degrees.

raw_data
bytes
Array of BGRA 32-bit pixels.

```

RSS sensor

- **Blueprint:** sensor.other.rss
- **Output:** carla.RssResponse per step (unless `sensor_tick` says otherwise).



| It is highly recommended to read the specific RSS documentation before reading this.

This sensor integrates the C++ Library for Responsibility Sensitive Safety in CARLA. It is disabled by default in CARLA, and it has to be explicitly built in order to be used.

The RSS sensor calculates the RSS state of a vehicle and retrieves the current RSS Response as sensor data. The carla.RssRestrictor will use this data to adapt a carla.VehicleControl before applying it to a vehicle.

These controllers can be generated by an *Automated Driving* stack or user input. For instance, hereunder there is a fragment of code from `PythonAPI/examples/manual_control_rss.py`, where the user input is modified using RSS when necessary.

1. Checks if the **RssSensor** generates a valid response containing restrictions. **2.** Gathers the current dynamics of the vehicle and the vehicle physics. **3.** Applies restrictions to the vehicle control using the response from the RssSensor, and the current dynamics and physics of the vehicle.

```

1 rss_restriction = self._world.rss_sensor.acceleration_restriction if self._world.rss_sensor and
   self._world.rss_sensor.response_valid else None
2 if rss_restriction:
3     rss_ego_dynamics_on_route = self._world.rss_sensor.ego_dynamics_on_route
4     vehicle_physics = world.player.get_physics_control()
5 ...
6     vehicle_control = self._restrictor.restrict_vehicle_control(
7         vehicle_control, rss_restriction, rss_ego_dynamics_on_route, vehicle_physics)

```

The carla.RssSensor class The blueprint for this sensor has no modifiable attributes. However, the carla.RssSensor object that it instantiates has attributes and methods that are detailed in the Python API reference. Here is a summary of them.

Type

Description

ego_vehicle_dynamics
RSS parameters to be applied for the ego vehicle

other_vehicle_dynamics
RSS parameters to be applied for the other vehicles

road_boundaries_mode
Enables/Disables the feature. Default is Off.

visualization_mode
States the visualization of the RSS calculations. Default is All.

```

1 # Fragment of manual_control_rss.py
2 # The carla.RssSensor is updated when listening for a new carla.RssResponse
3 def _on_rss_response(weak_self, response):
4 ...
5     self.timestamp = response.timestamp
6     self.response_valid = response.response_valid
7     self.proper_response = response.proper_response
8     self.acceleration_restriction = response.acceleration_restriction
9     self.ego_dynamics_on_route = response.ego_dynamics_on_route

```



This sensor works fully on the client side. There is no blueprint in the server. Changes on the attributes will have effect after the `*listen()*` has been called.

The methods available in this class are related to the routing of the vehicle. RSS calculations are always based on a route of the ego vehicle through the road network.

The sensor allows to control the considered route by providing some key points, which could be the carla.Transform in a carla.Waypoint. These points are best selected after the intersections to force the route to take the desired turn.

Description

routing_targets
Get the current list of routing targets used for route.

append_routing_target
Append an additional position to the current routing targets.

reset_routing_targets
Deletes the appended routing targets.

drop_route
Discards the current route and creates a new one.

register_actor_constellation_callback
Register a callback to customize the calculations.

set_log_level
Sets the log level.

```

1 # Update the current route
2 self.sensor.reset_routing_targets()
3 if routing_targets:
4     for target in routing_targets:
5         self.sensor.append_routing_target(target)

```



If no routing targets are defined, a random route is created.

Output attributes Type

Description

response_valid

bool

Validity of the response data.

proper_response

Proper response that the RSS calculated for the vehicle.

acceleration_restriction

Acceleration restrictions of the RSS calculation.

rss_state_snapshot

RSS states at the current point in time.

ego_dynamics_on_route

Current ego vehicle dynamics regarding the route.

situation_snapshot

Current situation snapshot extracted from the world model.

In case a actor_constellation_callback is registered, a call is triggered for:

1. default calculation (`actor_constellation_data.other_actor=None`)
2. per-actor calculation

```
1 # Fragment of manual_control_rss.py
2 # The function is registered as actor_constellation_callback
3 def _on_actor_constellation_request(self, actor_constellation_data):
4     actor_constellation_result = carla.RssActorConstellationResult()
5     actor_constellation_result.rss_calculation_mode = rssmap.RssMode.NotRelevant
6     actor_constellation_result.restrict_speed_limit_mode =
7         rssmap.RssSceneCreation.RestrictSpeedLimitMode.IncreasedSpeedLimit10
8     actor_constellation_result.ego_vehicle_dynamics = self.current_vehicle_parameters
9     actor_constellation_result.actor_object_type = rss.ObjectType.Invalid
10    actor_constellation_result.actor_dynamics = self.current_vehicle_parameters
11
12    actor_id = -1
13    actor_type_id = "none"
14    if actor_constellation_data.other_actor != None:
15        # customize actor_constellation_result for specific actor
16        ...
17    else:
18        # default
19        ...
20
21    return actor_constellation_result
```

Semantic LIDAR sensor

- **Blueprint:** `sensor.lidar.ray_cast_semantic`
- **Output:** `carla.SemanticLidarMeasurement` per step (unless `sensor_tick` says otherwise).

This sensor simulates a rotating LIDAR implemented using ray-casting that exposes all the information about the raycast hit. Its behaviour is quite similar to the LIDAR sensor, but there are two main differences between them.

- The raw data retrieved by the semantic LIDAR includes more data per point.
 - Coordinates of the point (as the normal LIDAR does).

- The cosine between the angle of incidence and the normal of the surface hit.
- Instance and semantic ground-truth. Basically the index of the CARLA object hit, and its semantic tag.
- The semantic LIDAR does not include neither intensity, drop-off nor noise model attributes.

The points are computed by adding a laser for each channel distributed in the vertical FOV. The rotation is simulated computing the horizontal angle that the LIDAR rotated in a frame. The point cloud is calculated by doing a ray-cast for each laser in every step.

```
1 points_per_channel_each_step = points_per_second / (FPS * channels)
```

A LIDAR measurement contains a package with all the points generated during a $1/\text{FPS}$ interval. During this interval the physics are not updated so all the points in a measurement reflect the same “static picture” of the scene.

This output contains a cloud of lidar semantic detections and therefore, it can be iterated to retrieve a list of their `carla.SemanticLidarDetection`:

```
1 for detection in semantic_lidar_measurement:
2     print(detection)
```

The rotation of the LIDAR can be tuned to cover a specific angle on every simulation step (using a fixed time-step). For example, to rotate once per step (full circle output, as in the picture below), the rotation frequency and the simulated FPS should be equal. 1. Set the sensor’s frequency `sensors_bp['lidar'][0].set_attribute('rotation_frequency', '10')`. 2. Run the simulation using `python3 config.py --fps=10`.

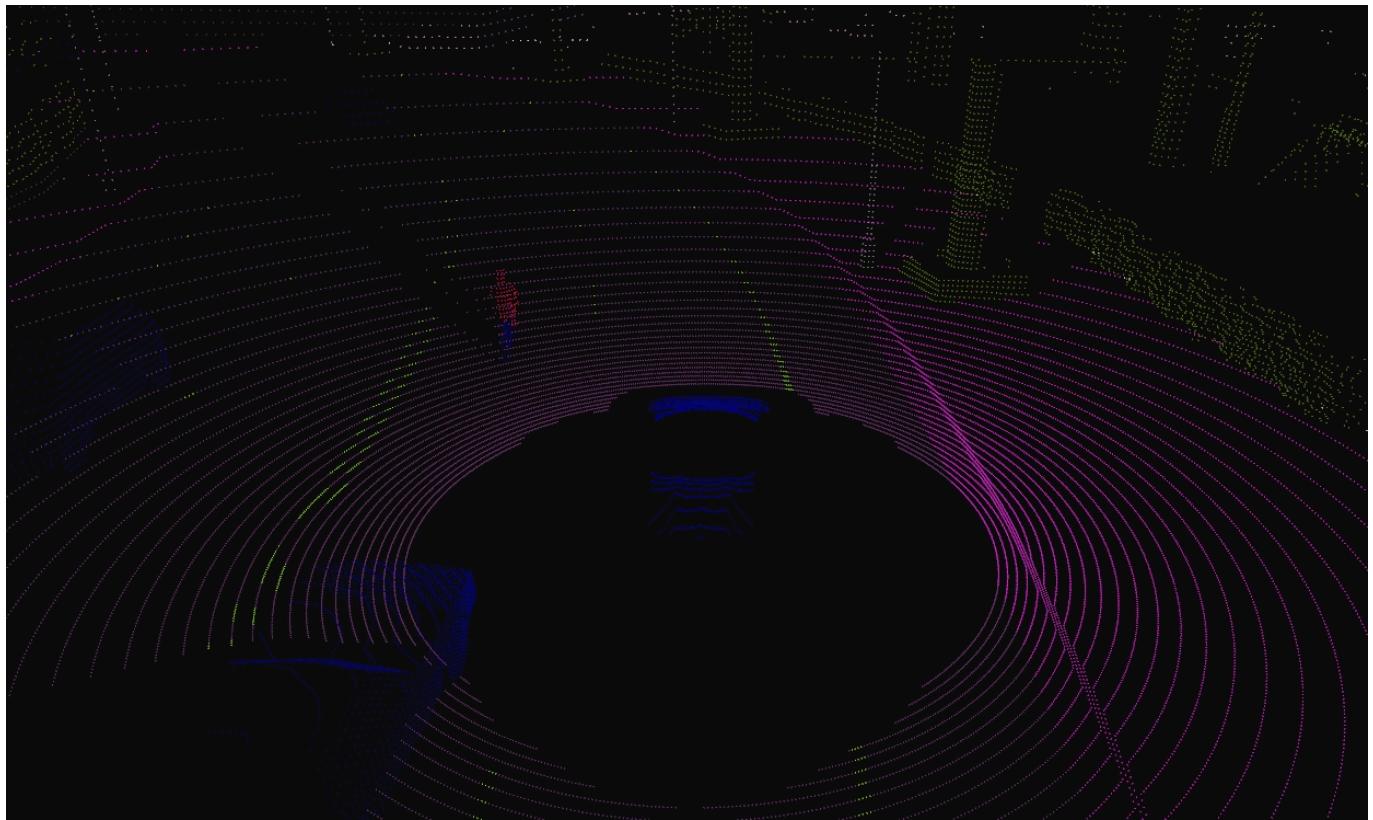


Figure 41: LidarPointCloud

SemanticLidar attributes Blueprint attribute

Type

Default

Description

channels

int

32

Number of lasers.

range
float
10.0

Maximum distance to measure/raycast in meters (centimeters for CARLA 0.9.6 or previous).

points_per_second
int
56000

Points generated by all lasers per second.

rotation_frequency
float
10.0

LIDAR rotation frequency.

upper_fov
float
10.0

Angle in degrees of the highest laser.

lower_fov
float
-30.0

Angle in degrees of the lowest laser.

sensor_tick
float
0.0

Simulation seconds between sensor captures (ticks).

Output attributes Sensor data attribute

Type

Description

frame
int

Frame number when the measurement took place.

timestamp
double

Simulation time of the measurement in seconds since the beginning of the episode.

transform

Location and rotation in world coordinates of the sensor at the time of the measurement.

horizontal_angle
float

Angle (radians) in the XY plane of the LIDAR in the current frame.

channels

```

int
Number of channels (lasers) of the LIDAR.
get_point_count(channel)
int
Number of points per channel captured in the current frame.
raw_data
bytes
Array containing the point cloud with instance and semantic information. For each point, four 32-bits floats are stored.
- XYZ coordinates. - cosine of the incident angle. - Unsigned int containing the index of the object hit. - Unsigned int containing the semantic tag of the object it.

```

Semantic segmentation camera

- **Blueprint:** sensor.camera.semantic_segmentation
- **Output:** carla.Image per step (unless **sensor_tick** says otherwise).

This camera classifies every object in sight by displaying it in a different color according to its tags (e.g., pedestrians in a different color than vehicles). When the simulation starts, every element in scene is created with a tag. So it happens when an actor is spawned. The objects are classified by their relative file path in the project. For example, meshes stored in `Unreal/CarlaUE4/Content/Static/Pedestrians` are tagged as `Pedestrian`.

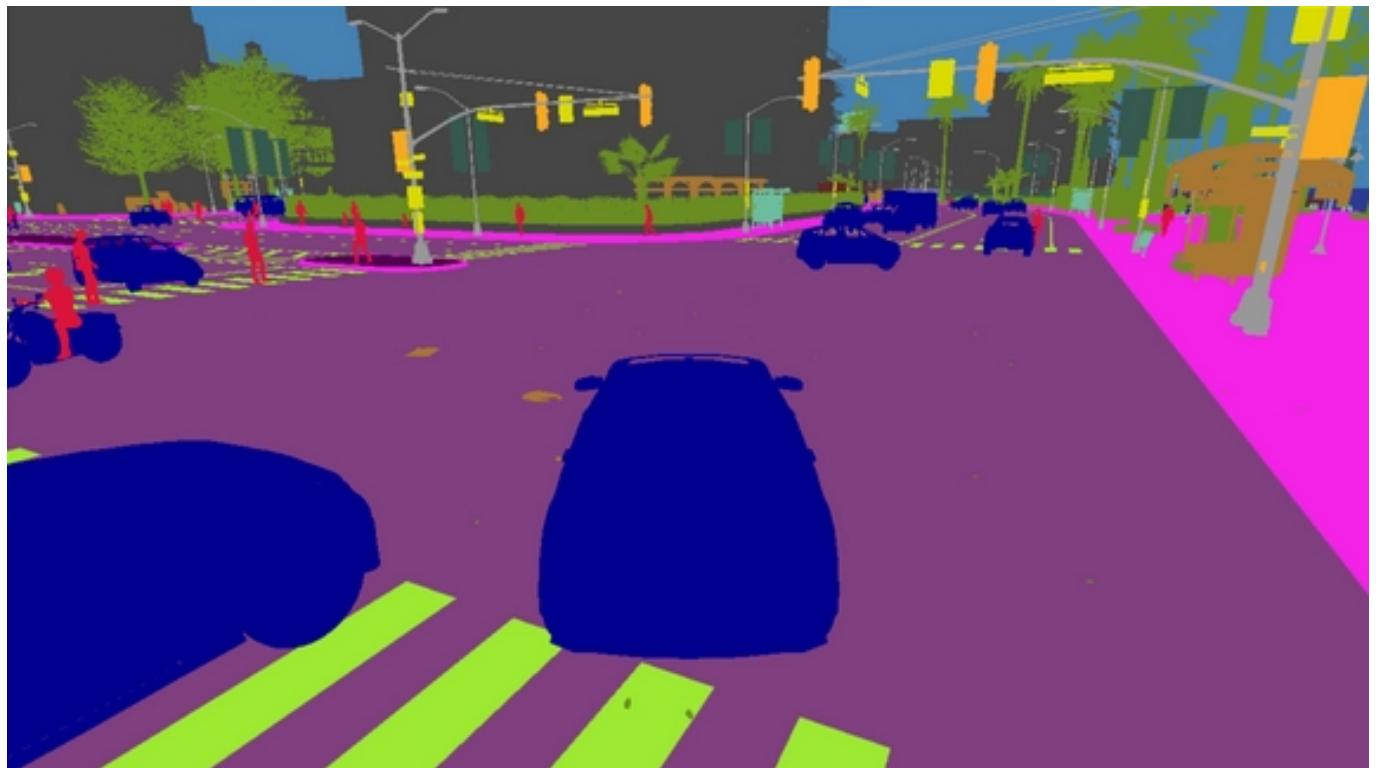


Figure 42: ImageSemanticSegmentation

The server provides an image with the tag information **encoded in the red channel**: A pixel with a red value of x belongs to an object with tag x . This raw carla.Image can be stored and converted it with the help of **CityScapes-Palette** in carla.ColorConverter to apply the tags information and show picture with the semantic segmentation.

```

1 ...
2 raw_image.save_to_disk("path/to/save/converted/image",carla.cityScapesPalette)

```

The following tags are currently available:

Value

Tag

Converted color

Description

0

Unlabeled

(0, 0, 0)

Elements that have not been categorized are considered Unlabeled. This category is meant to be empty or at least contain elements with no collisions.

1

Building

(70, 70, 70)

Buildings like houses, skyscrapers,... and the elements attached to them. E.g. air conditioners, scaffolding, awning or ladders and much more.

2

Fence

(100, 40, 40)

Barriers, railing, or other upright structures. Basically wood or wire assemblies that enclose an area of ground.

3

Other

(55, 90, 80)

Everything that does not belong to any other category.

4

Pedestrian

(220, 20, 60)

Humans that walk or ride/drive any kind of vehicle or mobility system. E.g. bicycles or scooters, skateboards, horses, roller-blades, wheel-chairs, etc.

5

Pole

(153, 153, 153)

Small mainly vertically oriented pole. If the pole has a horizontal part (often for traffic light poles) this is also considered pole. E.g. sign pole, traffic light poles.

6

RoadLine

(157, 234, 50)

The markings on the road.

7

Road

(128, 64, 128)

Part of ground on which cars usually drive. E.g. lanes in any directions, and streets.

8

SideWalk

(244, 35, 232)

Part of ground designated for pedestrians or cyclists. Delimited from the road by some obstacle (such as curbs or poles), not only by markings. This label includes a possibly delimiting curb, traffic islands (the walkable part), and pedestrian zones.

9

Vegetation

(107, 142, 35)

Trees, hedges, all kinds of vertical vegetation. Ground-level vegetation is considered Terrain.

10

Vehicles

(0, 0, 142)

Cars, vans, trucks, motorcycles, bikes, buses, trains.

11

Wall

(102, 102, 156)

Individual standing walls. Not part of a building.

12

TrafficSign

(220, 220, 0)

Signs installed by the state/city authority, usually for traffic regulation. This category does not include the poles where signs are attached to. E.g. traffic- signs, parking signs, direction signs...

13

Sky

(70, 130, 180)

Open sky. Includes clouds and the sun.

14

Ground

(81, 0, 81)

Any horizontal ground-level structures that does not match any other category. For example areas shared by vehicles and pedestrians, or flat roundabouts delimited from the road by a curb.

15

Bridge

(150, 100, 100)

Only the structure of the bridge. Fences, people, vehicles, an other elements on top of it are labeled separately.

16

RailTrack

(230, 150, 140)

All kind of rail tracks that are non-drivable by cars. E.g. subway and train rail tracks.

17

GuardRail

(180, 165, 180)

All types of guard rails/crash barriers.

18

TrafficLight

(250, 170, 30)

Traffic light boxes without their poles.

19

Static

(110, 190, 160)

Elements in the scene and props that are immovable. E.g. fire hydrants, fixed benches, fountains, bus stops, etc.

20

Dynamic

(170, 120, 50)

Elements whose position is susceptible to change over time. E.g. Movable trash bins, buggies, bags, wheelchairs, animals, etc.

21

Water

(45, 60, 150)

Horizontal water surfaces. E.g. Lakes, sea, rivers.

22

Terrain

(145, 170, 100)

Grass, ground-level vegetation, soil or sand. These areas are not meant to be driven on. This label includes a possibly delimiting curb.



Adding new tags : It requires some C++ coding. Add a new label to the 'ECityObjectLabel' enum in "Tagger.h", and its corresponding filepath check inside 'GetLabelByFolderName()' function in "Tagger.cpp".

Basic camera attributes Blueprint attribute

Type

Default

Description

fov

float

90.0

Horizontal field of view in degrees.

image_size_x

int

800

Image width in pixels.

image_size_y

int

600

Image height in pixels.

sensor_tick
float
0.0
Simulation seconds between sensor captures (ticks).

Camera lens distortion attributes Blueprint attribute

Type

Default

Description

lens_circle_falloff

float

5.0

Range: [0.0, 10.0]

lens_circle_multiplier

float

0.0

Range: [0.0, 10.0]

lens_k

float

-1.0

Range: [-inf, inf]

lens_kcube

float

0.0

Range: [-inf, inf]

lens_x_size

float

0.08

Range: [0.0, 1.0]

lens_y_size

float

0.08

Range: [0.0, 1.0]

Output attributes Sensor data attribute

Type

Description

fov

float

Horizontal field of view in degrees.

frame

int

Frame number when the measurement took place.

height

int

Image height in pixels.

raw_data

bytes

Array of BGRA 32-bit pixels.

timestamp

double

Simulation time of the measurement in seconds since the beginning of the episode.

transform

Location and rotation in world coordinates of the sensor at the time of the measurement.

width

int

Image width in pixels.

DVS camera

- **Blueprint:** sensor.camera.dvs
- **Output:** carla.DVSEventArray per step (unless `sensor_tick` says otherwise).

A Dynamic Vision Sensor (DVS) or Event camera is a sensor that works radically differently from a conventional camera. Instead of capturing intensity images at a fixed rate, event cameras measure changes of intensity asynchronously, in the form of a stream of events, which encode per-pixel brightness changes. Event cameras possess outstanding properties when compared to standard cameras. They have a very high dynamic range (140 dB versus 60 dB), no motion blur, and high temporal resolution (in the order of microseconds). Event cameras are thus sensors that can provide high-quality visual information even in challenging high-speed scenarios and high dynamic range environments, enabling new application domains for vision-based algorithms.

The DVS camera outputs a stream of events. An event $e=(x,y,t,pol)$ is triggered at a pixel x, y at a timestamp t when the change in logarithmic intensity L reaches a predefined constant threshold C (typically between 15% and 30%).

$$L(x,y,t) - L(x,y,t-\Delta t) = pol \ C$$

Δt is the time when the last event at that pixel was triggered and pol is the polarity of the event according to the sign of the brightness change. The polarity is positive +1 when there is increment in brightness and negative -1 when a decrement in brightness occurs. The working principles depicted in the following figure. The standard camera outputs frames at a fixed rate, thus sending redundant information when no motion is present in the scene. In contrast, event cameras are data-driven sensors that respond to brightness changes with microsecond latency. At the plot, a positive (resp. negative) event (blue dot, resp. red dot) is generated whenever the (signed) brightness change exceeds the contrast threshold C for one dimension x over time t . Observe how the event rate grows when the signal changes rapidly.

The current implementation of the DVS camera works in a uniform sampling manner between two consecutive synchronous frames. Therefore, in order to emulate the high temporal resolution (order of microseconds) of a real event camera, the sensor requires to execute at a high frequency (much higher frequency than a conventional camera). Effectively, the number of events increases as faster a CARLA car drives. Therefore, the sensor frequency should increase accordingly with the dynamic of the scene. The user should find their balance between time accuracy and computational cost.

The provided script `manual_control.py` uses the DVS camera in order to show how to configure the sensor, how to get the stream of events and how to depict such events in an image format, usually called event frame.

DVS is a camera and therefore has all the attributes available in the RGB camera. Nevertheless, there are few attributes exclusive to the working principle of an Event camera.

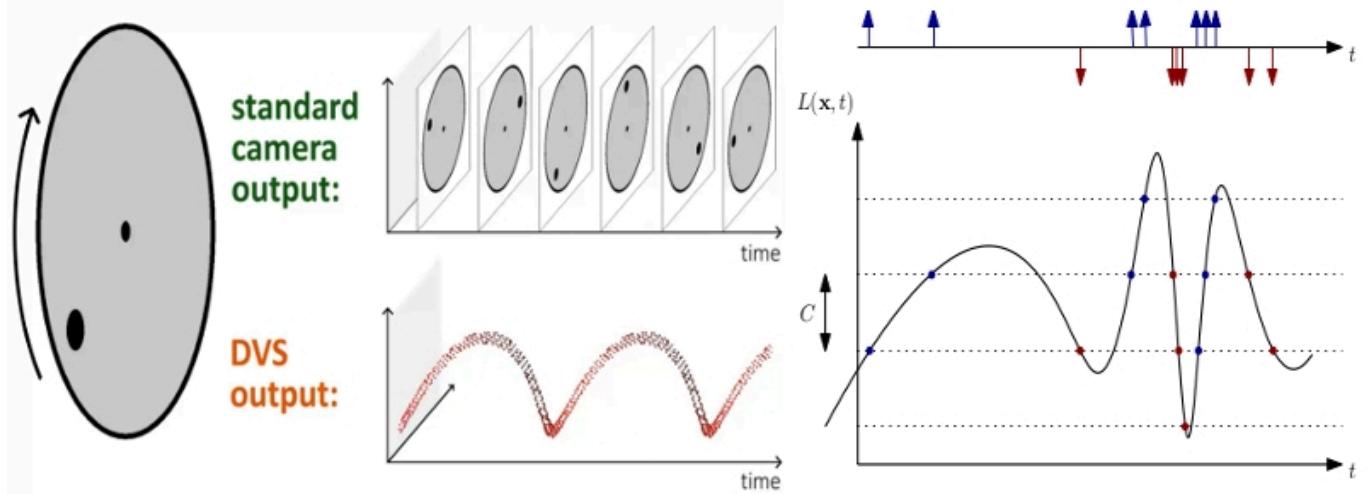


Figure 43: DVSCameraWorkingPrinciple

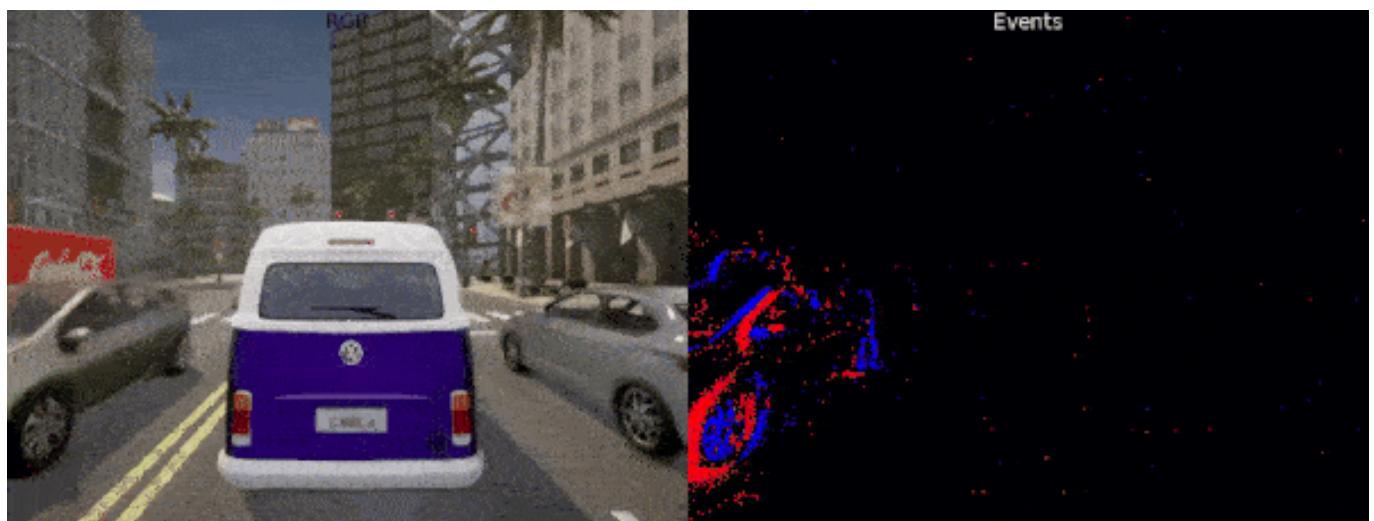


Figure 44: DVSCameraWorkingPrinciple

DVS camera attributes Blueprint attribute

Type

Default

Description

positive_threshold

float

0.3

Positive threshold C associated to a increment in brightness change (0-1).

negative_threshold

float

0.3

Negative threshold C associated to a decrement in brightness change (0-1).

sigma_positive_threshold

float

0

White noise standard deviation for positive events (0-1).

sigma_negative_threshold

float

0

White noise standard deviation for negative events (0-1).

refractory_period_ns

int

0.0

Refractory period (time during which a pixel cannot fire events just after it fired one), in nanoseconds. It limits the highest frequency of triggering events.

use_log

bool

true

Whether to work in the logarithmic intensity scale.

log_eps

float

0.001

Epsilon value used to convert images to log: $L = \log(\text{eps} + I / 255.0)$. Where I is the grayscale value of the RGB image: $I = 0.2989R + 0.5870G + 0.1140*B$.

6 Plugins

6.1 carlaviz

The carlaviz plugin is used to visualize the simulation in a web browser. A windows with some basic representation of the scene is created. Actors are updated on-the-fly, sensor data can be retrieved, and additional text, lines and polylines can be drawn in the scene.

- General information

- Support

- **Get carlaviz**
 - Prerequisites
 - Download the plugin
- **Utilities**

General information

- **Contributor** — Minjun Xu, also known as wx9698.
- **License** — MIT.

Support

- **Linux** — CARLA 0.9.6, 0.9.7, 0.9.8, 0.9.9, 0.9.10.
- **Windows** — CARLA 0.9.9, 0.9.10.
- **Build from source** — Latest updates.

Get carlaviz

Prerequisites

- **Docker** — Visit the docs and install Docker.
- **Operative system** — Any OS able to run CARLA should work.
- **WebSocket-client** — `pip3 install websocket_client`. Install pip if it is not already in the system.

Download the plugin

Open a terminal and pull the Docker image of carlaviz, based on the CARLA version to be run.

```

1 ## Pull only the image that matches the CARLA package being used
2 docker pull mjaxu96/carlaviz:0.9.6
3 docker pull mjaxu96/carlaviz:0.9.7
4 docker pull mjaxu96/carlaviz:0.9.8
5 docker pull mjaxu96/carlaviz:0.9.9
6 docker pull mjaxu96/carlaviz:0.9.10
7
8 ## Pull this image if working on a CARLA build from source
9 docker pull mjaxu96/carlaviz:latest

```



Currently in Windows there is only support for 0.9.9 and 0.9.10.

CARLA up to 0.9.9 (included) is set to be single-stream. For later versions, multi-streaming for sensors is implemented.

- In **single-stream**, a sensor can only be heard by one client. When a sensor is already being heard by another client, for example when running `manual_control.py`, carlaviz is forced to duplicate the sensor in order to retrieve the data, and performance may suffer.
- In **multi-stream**, a sensor can be heard by multiple clients. carlaviz has no need to duplicate these and performance does not suffer.



Alternatively on Linux, users can build carlaviz following the instructions [here](<https://github.com/carla-simulator/carlaviz/blob/master/docs/build.md>), but using a Docker image will make things much easier.

6.2 Run carlaviz

1. Run CARLA.

- a) **In a CARLA package** — Go to the CARLA folder and start the simulation with `CarlaUE4.exe` (Windows) or `./CarlaUE4.sh` (Linux).
- b) **In a build from source** — Go to the CARLA folder, run the UE editor with `make launch` and press Play.

2. Run carlaviz. In another terminal run the following command according to the Docker image that has been downloaded.

Change <name_of_Docker_image> for the name of the image previously downloaded, e.g. `mjxu96/carlaviz:latest` or `mjxu96/carlaviz:0.9.10`.

```
1 ## On Linux system
2 docker run -it --network="host" -e CARLAVIZ_HOST_IP=localhost -e CARLA_SERVER_IP=localhost -e
   CARLA_SERVER_PORT=2000 <name_of_Docker_image>
3
4 ## On Windows/MacOS system
5 docker run -it -e CARLAVIZ_HOST_IP=localhost -e CARLA_SERVER_IP=host.docker.internal -e
   CARLA_SERVER_PORT=2000 -p 8080-8081:8080-8081 -p 8089:8089 <name_of_Docker_image>
```

If the everything has been properly set, carlaviz will show a successful message similar to the following.

```
Built at: 07/15/2020 1:43:15 PM
          Asset      Size  Chunks      Chunk Names
573fe91c6c748074195655e1f0159864.png  7.61 KiB      [emitted]
          bundle.js  9.36 MiB     app  [emitted]      app
          bundle.js.map 11.5 MiB     app  [emitted] [dev]  app
Entrypoint app = bundle.js bundle.js.map
[0] multi (webpack)-dev-server/client?http://0.0.0.0:8082 (webpack)/hot/dev-server.js ./src/app.js 52 bytes {app} [built]
[./node_modules/@deck.gl/core/dist/esm/index.js] 2.78 KiB {app} [built]
[./node_modules/@deck.gl/layers/dist/esm/index.js] 1.01 KiB {app} [built]
[./node_modules/@streetscape.gl/monochrome/dist/esm/index.js] 560 bytes {app} [built]
[./node_modules/react-dom/index.js] 1.33 KiB {app} [built]
[./node_modules/react/index.js] 190 bytes {app} [built]
[./node_modules/streetscape.gl/dist/esm/index.js] 78 bytes {app} [built]
[./node_modules/strip-ansi/index.js] 161 bytes {app} [built]
[./node_modules/webpack-dev-server/client/index.js?http://0.0.0.0:8082] (webpack)-dev-server/client?http://0.0.0.0:8082 4.29 KiB {app} [built]
[./node_modules/webpack-dev-server/client/overlay.js] (webpack)-dev-server/client/overlay.js 3.51 KiB {app} [built]
[./node_modules/webpack-dev-server/client/socket.js] (webpack)-dev-server/client/socket.js 1.53 KiB {app} [built]
[./node_modules/webpack-dev-server/client/utils/createSocketUrl.js] (webpack)-dev-server/client/utils/createSocketUrl.js 2.91 KiB {app} [built]
[./node_modules/webpack-dev-server/client/utils/log.js] (webpack)-dev-server/client/utils/log.js 964 bytes {app} [built]
[./node_modules/webpack/hot/dev-server.js] (webpack)/hot/dev-server.js 1.59 KiB {app} [built]
[./src/app.js] 10.6 KiB {app} [built]
+ 1454 hidden modules
i 「wdm」: Compiled successfully.
```

Figure 45: carlaviz_run



Remember to edit the previous command to match the Docker image being used.

3. Open the localhost Open your web browser and go to `http://127.0.0.1:8080/`. carlaviz runs by default in port 8080. The output should be similar to the following.

6.3 Utilities

Once the plugin is operative, it can be used to visualize the simulation, the actors that live in it, and the data the sensors retrieve. The plugin shows a visualization window on the right, were the scene is updated in real-time, and a sidebar on the left with a list of items to be shown. Some of these items will appear in the visualization window, others (mainly sensor and game data) appear just above the items list. Here is a list of options available for visualization. Additional elements may show, such as

- **View Mode** — Change the point of view in the visualization window.
 - **Top Down** — Aerial point of view.
 - **Perspective** — Free point of view.
 - **Driver** — First person point of view.
- **/vehicle** — Show properties of the ego vehicle. Includes a speedometer and accelerometer in the visualization window, and the data retrieved by IMU, GNSS and collision detector sensors.
 - **/velocity** — Velocity of the ego vehicle.
 - **/acceleration** — Acceleration of the ego vehicle.
- **/drawing** — Show additional elements in the visualization window drawn with CarlaPainter.
 - **/texts** — Text elements.
 - **/points** — Point elements.
 - **/polylines** — Polyline elements.
- **/objects** — Show actors in the visualization window.
 - **/walkers** — Update walkers.

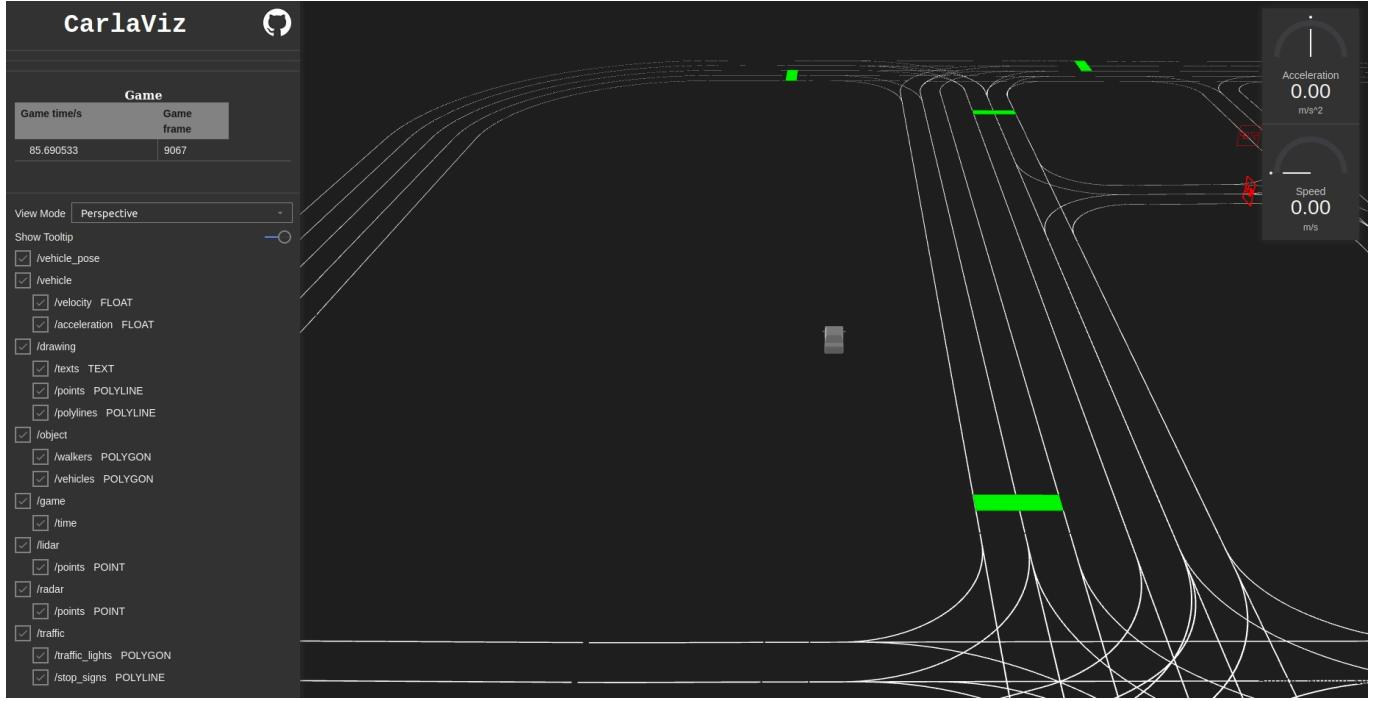


Figure 46: carlaviz_empty

- **/vehicles** — Update vehicles.
- **/game** — Show game data.
 - **/time** — Current simulation time and frame.
- **/lidar** — LIDAR sensor data.
 - **/points** — Cloud of points detected by a LIDAR sensor.
- **/radar** — RADAR sensor data.
 - **/points** — Cloud of points detected by a RADAR sensor.
- **/traffic** — Landmark data.
 - **/traffic_light** — Show the map’s traffic lights in the visualization window.
 - **/stop_sign** — Show the map’s stop signs in the visualization window.

Try to spawn some actors. These will be automatically updated in the visualization window.

```
1 cd PythonAPI/examples
2 python3 spawn_npc.py -n 10 -w 5
```

Spawn an ego vehicle with manual control and move around, to see how the plugin updates sensor data.

```
1 cd PythonAPI/examples
2 python3 manual_control.py
```

The contributor (wx9698), created an additional class, CarlaPainter, that allows the user to draw elements to be shown in the visualization window. These include text, points and polylines. Follow this example to spawn an ego vehicle with a LIDAR, and draw the LIDAR data, the trajectory and velocity of the vehicle.

That is all there is to know about the carlaviz plugin. If there are any doubts, feel free to post these in the forum.

7 ROS bridge

7.1 ROS bridge installation

The ROS bridge enables two-way communication between ROS and CARLA. The information from the CARLA server is translated to ROS topics. In the same way, the messages sent between nodes in ROS get translated to commands to be applied in CARLA.

- **Requirements**
 - Python2
- **Bridge installation**

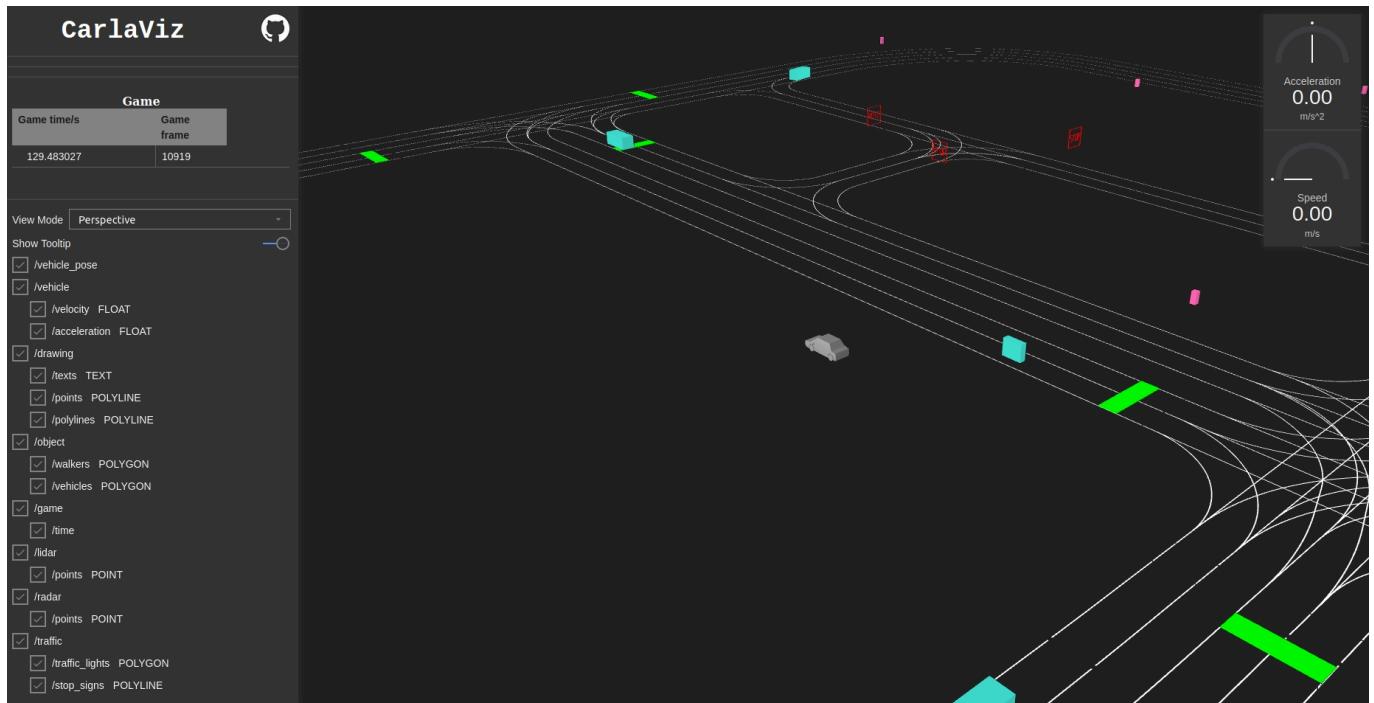


Figure 47: carlaviz_full



Figure 48: carlaviz_data

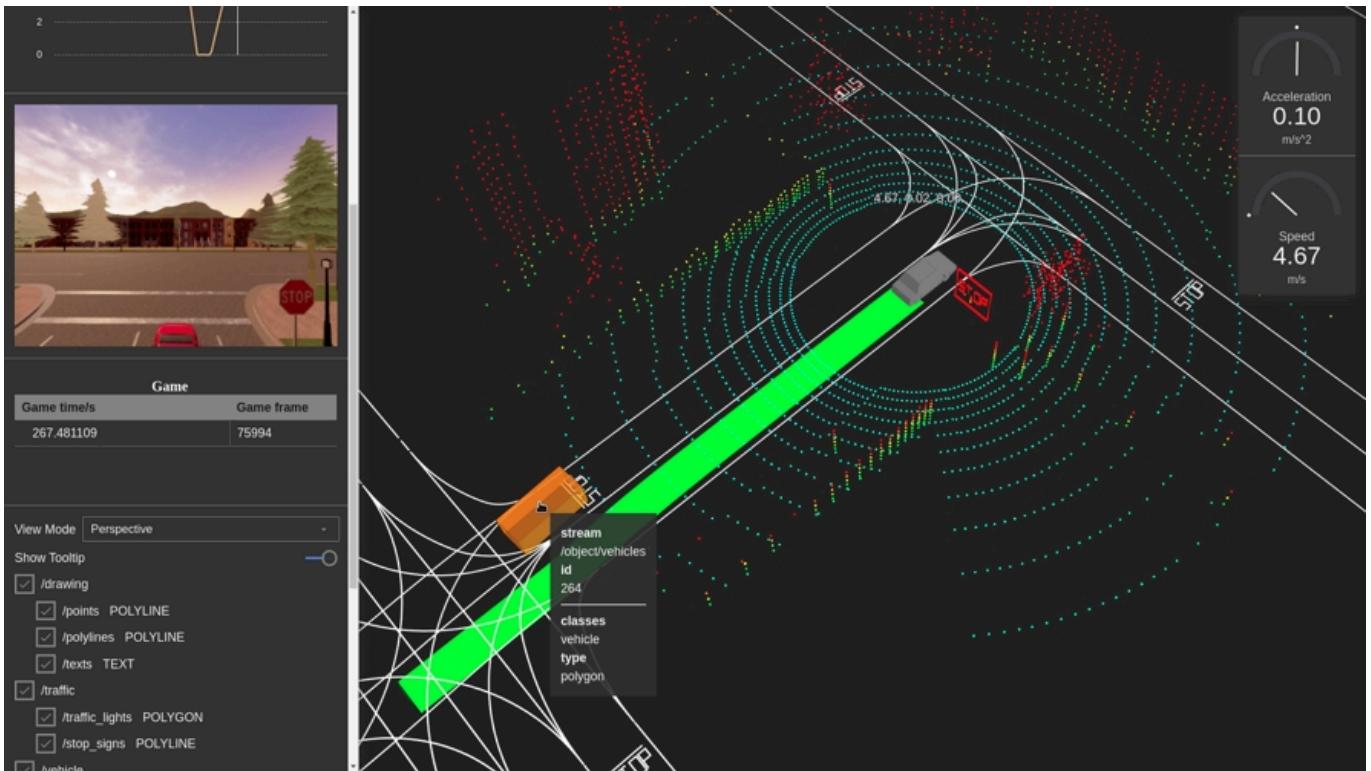


Figure 49: carlaviz_demo

- A. Using Debian repository
- B. Using source repository
- **Run the ROS bridge**
- **Setting CARLA**



ROS is still [experimental](<http://wiki.ros.org/noetic/Installation>) for Windows, so the ROS bridge has only been tested for Linux systems.

Requirements

Make sure that both requirements work properly before continuing with the installation.

- **ROS Kinetic/Melodic** — Install the ROS version corresponding to your system. Additional ROS packages may be required, depending on the user needs. rviz is highly recommended to visualize ROS data.
 - **ROS Kinetic** — For Ubuntu 16.04 (Xenial).
 - **ROS Melodic** — For Ubuntu 18.04 (Bionic).
 - **ROS Noetic** — For Ubuntu 20.04 (Focal).
- **CARLA 0.9.7 or later** — Previous versions are not compatible with the ROS bridge. Follow the quick start installation or make the build for Linux.

Python The version of Python needed to run the ROS bridge depends on the ROS version being used.

- **ROS Kinetic and ROS Melodic** — Python2.
- **ROS Noetic** — Python3.

All the CARLA releases provide support for Python3, so ROS Noetic users do not need more preparation. However, ROS Kinetic/Melodic users cannot use the latest CARLA release packages. Since 0.9.10 (included), CARLA does not provide support for Python2, so users will have to make the build from source, and compile the PythonAPI for Python2.

- Run the following command in the root CARLA directory of your Linux build to compile the PythonAPI for Python2.

```
1 make PythonAPI ARGS="--python-version=2"
```

Bridge installation



To install ROS bridge versions prior to 0.9.10, change to a previous version of the documentation using the pannel in the bottom right corner of the window, and follow the old instructions.

A. Using Debian repository

Set up the Debian repository in the system.

```
1 sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 1AF1527DE64CB8D9
2 sudo add-apt-repository "deb [arch=amd64] http://dist.carla.org/carla $(lsb_release -sc) main"
```

Install the ROS bridge, and check for the installation in the `/opt/` folder.

```
1 sudo apt-get update # Update the Debian package index
2 sudo apt-get install carla-ros-bridge # Install the latest ROS bridge version, or update the current
   installation
```

This repository contains features from CARLA 0.9.10 and later versions. To install a specific version add the version tag to the installation command.

```
1 sudo apt-get install carla-ros-bridge=0.9.10-1 # In this case, "0.9.10" refers to the ROS bridge
   version, and "1" to the Debian revision
```

B. Using source repository

A catkin workspace is needed to use the ROS bridge. It should be cloned and built in there. The following code creates a new workspace, and clones the repository in there.

```
1 # Setup folder structure
2 mkdir -p ~/carla-ros-bridge/catkin_ws/src
3 cd ~/carla-ros-bridge
4 git clone https://github.com/carla-simulator/ros-bridge.git
5 cd ros-bridge
6 git submodule update --init
7 cd ../catkin_ws/src
8 ln -s ../../ros-bridge
9 source /opt/ros/kinetic/setup.bash # Watch out, this sets ROS Kinetic
10 cd ..
11
12 # Install required ros-dependencies
13 rosdep update
14 rosdep install --from-paths src --ignore-src -r
15
16 # Build
17 catkin_make
```

Run the ROS bridge

1) Run CARLA.

The way to do so depends on the CARLA installation.

- Quick start/release package. `./CarlaUE4.sh` in `carla/`.
- Debian installation. `./CarlaUE4.sh` in `opt/carla-simulator/`.
- Build installation. `make launch` in `carla/`.

2) Add the source path.

The source path for the workspace has to be added, so that the ROS bridge can be used from a terminal.

- Source for apt ROS bridge.

```
1 source /opt/carla-ros-bridge/<kinetic or melodic>/setup.bash
```

- Source for ROS bridge repository download.

```
1 source ~/carla-ros-bridge/catkin_ws/devel/setup.bash
```



The source path can be set permanently, but it will cause conflict when working with another workspace.

3) Start the ROS bridge. Use any of the different launch files available to check the installation. Here are some suggestions.

```
1 # Option 1: start the ros bridge
2 rosrun carla_ros_bridge carla_ros_bridge.launch
3
4 # Option 2: start the ros bridge together with RVIZ
5 rosrun carla_ros_bridge carla_ros_bridge_with_rviz.launch
6
7 # Option 3: start the ros bridge together with an example ego vehicle
8 rosrun carla_ros_bridge carla_ros_bridge_with_example_ego_vehicle.launch
```

ImportError: no module named CARLA

The path to CARLA Python is missing. The apt installation does this automatically, but it may be missing for other installations. Execute the following command with the complete path to the .egg file (included). Use the one supported by the Python version installed. Note: .egg files may be either in /PythonAPI/ or /PythonAPI/dist/ depending on the CARLA installation.

```
1 export PYTHONPATH=$PYTHONPATH:<path/to/carla/>/PythonAPI/<your_egg_file>
```

Import CARLA from Python and wait for a sucess message to check the installation.

```
1 python3 -c 'import carla;print("Success")'
```

Setting CARLA

To modify the way CARLA works along with the ROS bridge, edit `share/carla_ros_bridge/config/settings.yaml`.

- **Host/port.** Network settings to connect to CARLA using a Python client.
- **Synchronous mode.**
 - **If false (default).** Data is published on every `world.on_tick()` and every `sensor.listen()` callbacks.
 - **If true** The bridge waits for all the sensor messages expected before the next tick. This might slow down the overall simulation but ensures reproducible results.
- **Wait for vehicle command.** In synchronous mode, pauses the tick until a vehicle control is completed.
- **Simulation time-step.** Simulation time (delta seconds) between simulation steps. **It must be lower than 0.1.** Take a look at the documentation to learn more about this.
- **Role names for the Ego vehicles.** Role names to identify ego vehicles. These will be controllable from ROS and thus, relevant topics will be created.



In synchronous mode only the ros-bridge is allowed to tick. Other clients must passively wait.

Synchronous mode To control the step update when in synchronous mode, use the following topic. The message contains a constant named `command` that allows to **Pause/Play** the simulation, and execute a **single step**.

Topic

Message type

/carla/control

The Control rqt plugin launches a new window with a simple interface. It is used to manage the steps and publish in the corresponding topic. Simply run the following when CARLA in synchronous mode.

```
1 rqt --standalone rqt_carla_control
```

7.2 CARLA messages reference

The following reference lists all the CARLA messages available in the ROS bridge.

Any doubts regarding these messages or the CARLA-ROS bridge can be solved in the forum.

CarlaActorInfo.msg

Information shared between ROS and CARLA regarding an actor.

Field

Type

Description

id

uint32

The ID of the actor.

parent_id

uint32

The ID of the parent actor. 0 if no parent available.

type

string

The identifier of the blueprint this actor was based on.

rolename

string

Role assigned to the actor when spawned.

CarlaActorList.msg

A list of messages with some basic information for CARLA actors.

Field

Type

Description

actors

List of messages with actors' information.

CarlaCollisionEvent.msg

Data retrieved on a collision event detected by the collision sensor of an actor.

Field

Type

Description

header

Time stamp and frame ID when the message is published.

other_actor_id

uint32

ID of the actor against whom the collision was detected.

normal_impulse

geometry_msgs/Vector3

Vector representing resulting impulse from the collision.

CarlaControl.msg

These messages control the simulation while in synchronous mode. The constant defined is translated as stepping commands.

Field

Type

Description

command

int8

PLAY=0 PAUSE=1 STEP_ONCE=2



In synchronous mode, only the ROS bridge client is allowed to tick.

CarlaEgoVehicleControl.msg

Messages sent to apply a control to a vehicle in both modes, autopilot and manual. These are published in a stack.

Field

Type

Description

header

Time stamp and frame ID when the message is published.

throttle

float32

Scalar value to control the vehicle throttle: [0.0, 1.0]

steer

float32

Scalar value to control the vehicle steering direction: [-1.0, 1.0] to control the vehicle steering

brake

float32

Scalar value to control the vehicle brakes: [0.0, 1.0]

hand_brake

bool

If True, the hand brake is enabled.

reverse

bool

If True, the vehicle will move reverse.

gear

int32

Changes between the available gears in a vehicle.

manual_gear_shift

bool

If True, the gears will be shifted using gear.

CarlaEgoVehicleInfo.msg

Static information regarding a vehicle, mostly the attributes used to define the vehicle's physics.

Field

Type

Description

id

uint32

ID of the vehicle actor.

type

string

The identifier of the blueprint this vehicle was based on.

type

string

The identifier of the blueprint this vehicle was based on.

rolename

string

Role assigned to the vehicle.

wheels

List of messages with information regarding wheels.

max_rpm

float32

Maximum RPM of the vehicle's engine.

moi

float32

Moment of inertia of the vehicle's engine.

damping_rate_full_throttle

float32

Damping rate when the throttle is at maximum.

damping_rate_zero_throttle_clutch_engaged

float32

Damping rate when the throttle is zero with clutch engaged.

damping_rate_zero_throttle_clutch_disengaged

float32

Damping rate when the throttle is zero with clutch disengaged.

use_gear_autobox

bool

If True, the vehicle will have an automatic transmission.

gear_switch_time

float32

Switching time between gears.

clutch_strength

float32

The clutch strength of the vehicle. Measured in Kgm²/s.

mass

float32

The mass of the vehicle measured in Kg.

drag_coefficient

float32

Drag coefficient of the vehicle's chassis.

center_of_mass

geometry_msgs/Vector3

The center of mass of the vehicle.

CarlaEgoVehicleInfoWheel.msg

Static information regarding a wheel that will be part of aCarlaEgoVehicleInfo.msg message.

Field

Type

Description

tire_friction

float32

A scalar value that indicates the friction of the wheel.

damping_rate

float32

The damping rate of the wheel.

max_steer_angle

float32

The maximum angle in degrees that the wheel can steer.

radius

float32

The radius of the wheel in centimeters.

max_brake_torque

float32

The maximum brake torque in Nm.

max_handbrake_torque

float32

The maximum handbrake torque in Nm.

position

geometry_msgs/Vector3

World position of the wheel.

CarlaEgoVehicleStatus.msg

Current status of the vehicle as an object in the world.

Field

Type

Description

header

Time stamp and frame ID when the message is published.

velocity

float32

Current speed of the vehicle.

acceleration

geometry_msgs/Accel

Current acceleration of the vehicle.

orientation

geometry_msgs/Quaternion

Current orientation of the vehicle.

control

Current control values as reported by CARLA.

CarlaLaneInvasionEvent.msg

These messages publish lane invasions detected by a lane-invasion sensor attached to a vehicle. The invasions detected in the last step are passed as a list with a constant definition to identify the lane crossed.

Field

Type

Description

header

Time stamp and frame ID when the message is published.

crossed_lane_markings

int32[]

LANE_MARKING_OTHER=0 LANE_MARKING_BROKEN=1 LANE_MARKING_SOLID=2

CarlaScenario.msg

Details for a test scenario.

Field

Type

Description

name

string

Name of the scenario.

scenario_file

string

Test file for the scenario.

destination
geometry_msgs/Pose
Goal location of the scenario.
target_speed
float64
Desired speed during the scenario.

CarlaScenarioList.msg

List of test scenarios to run in ScenarioRunner.

Field

Type

Description

scenarios

List of scenarios.

CarlaScenarioRunnerStatus.msg

Current state of the ScenarioRunner. It is managed using a constant.

Field

Type

Description

status

uint8

Current state of the scenario as an enum: STOPPED=0 STARTING=1 RUNNING=2 SHUTTINGDOWN=3 ER-ROR=4

CarlaStatus.msg

Current world settings of the simulation.

Field

Type

Description

frame

uint64

Current frame number.

fixed_delta_seconds

float32

Simulation time between last and current step.

synchronous_mode

bool

If True, synchronous mode is enabled.

synchronous_mode_running

bool

True when the simulation is running. False when it is paused.

CarlaTrafficLightStatus.msg

Constant definition regarding the state of a traffic light.

Field

Type

Description

id

uint32

ID of the traffic light actor.

state

uint8

RED=0 YELLOW=1 GREEN=2 OFF=3 UNKNOWN=4

CarlaTrafficLightStatusList.msg

List of traffic lights with their status.

Field

Type

Description

scenarios

A list of messages summarizing traffic light states.

CarlaWalkerControl.msg

Information needed to apply a movement controller to a walker.

Field

Type

Description

direction

geometry_msgs/Vector3

Vector that controls the direction of the walker.

speed

float32

A scalar value to control the walker's speed.

jump

bool

If True, the walker will jump.

CarlaWaypoint.msg

Data contained in a waypoint object.

Field

Type

Description

road_id

int32

OpenDRIVE road's id.

section_id

int32

OpenDRIVE section's id, based on the order that they are originally defined.

lane_id

int32

OpenDRIVE lane's id, this value can be positive or negative which represents the direction of the current lane with respect to the road.

is_junction

bool

True, if the current Waypoint is on a junction as defined by OpenDRIVE.

is_junction

True when the simulation is running. False when it is paused.

CarlaWorldInfo.msg

Information about the current CARLA map.

Field

Type

Description

map_name

string

Name of the CARLA map loaded in the current world.

opendrive

string

.xodr OpenDRIVE file of the current map as a string.

EgoVehicleControlCurrent.msg

Current time, speed and acceleration values of the vehicle. Used by the controller. It is part of a `Carla_Ackermann_Control.EgoVehicles` message.

Field

Type

Description

time_sec

float32

Current time when the controller is applied.

speed

float32

Current speed applied by the controller.

speed_abs

float32

Speed as an absolute value.

accel

float32

Current acceleration applied by the controller.

EgoVehicleControlInfo.msg

Current values within an Ackermann controller. These messages are useful for debugging.

Field

Type

Description

header

Time stamp and frame ID when the message is published.

restrictions

Limits to the controller values.

target

Limits to the controller values.

current

Limits to the controller values.

status

Limits to the controller values.

output

Limits to the controller values.

EgoVehicleControlMaxima.msg

Controller restrictions (limit values). It is part of a `Carla_Ackermann_Control.EgoVehicleControlInfo.msg` message.

Field

Type

Description

max_steering_angle

float32

Max. steering angle for a vehicle.

max_speed

float32

Max. speed for a vehicle.

max_accel

float32

Max. acceleration for a vehicle.

max_decel

float32

Max. deceleration for a vehicle. Default: 8m/s²

min_accel

float32

Min. acceleration for a vehicle. When the Ackermann target accel. exceeds this value, the input accel. is controlled.

```
max_pedal
<!- TBF> <->
float32
Min. pedal.
```

EgoVehicleControlStatus.msg

Current status of the ego vehicle controller. It is part of a `Carla_Ackermann_Control.EgoVehicleControlInfo.msg` message.

Field

Type

Description

status

```
<!- TBF> <->
```

string

Current control status.

speed_control_activation_count

```
<!- TBF> <->
```

uint8

Speed controller.

speed_control_accel_delta

```
<!- TBF> <->
```

float32

Speed controller.

speed_control_accel_target

```
<!- TBF> <->
```

float32

Speed controller.

accel_control_pedal_delta

```
<!- TBF> <->
```

float32

Acceleration controller.

accel_control_pedal_target

```
<!- TBF> <->
```

float32

Acceleration controller.

brake_upper_border

```
<!- TBF> <->
```

float32

Borders for lay off pedal.

throttle_lower_border

```
<!- TBF> <->
```

float32

Borders for lay off pedal.

EgoVehicleControlTarget.msg

Target values of the ego vehicle controller. It is part of a `Carla_Ackermann_Control.EgoVehicleControlInfo.msg` message.

Field

Type

Description

steering_angle

float32

Target steering angle for the controller.

speed

float32

Target speed for the controller.

speed_abs

float32

Speed as an absolute value.

accel

float32

Target acceleration for the controller.

jerk

float32

Target jerk for the controller.

7.3 Launchfiles reference

carla_ackermann_control.launch

Creates a node to manage a vehicle using Ackermann controls instead of the CARLA control messages. The node reads the vehicle info from CARLA and uses it to define the controller. A simple Python PID is used to adjust acceleration/velocity. This is a Python dependency that can be easily installed with `pip`.

```
1 pip install --user simple-pid
```

It is possible to modify the parameters in runtime via ROS dynamic reconfigure. Initial parameters can be set using a `settings.yaml`. The path to it depends on the bridge installation.

- **Deb repository installation**, `/opt/carla-ros-bridge/melodic/share/carla_ackermann_control/config/settings.yaml`
- **Source repository installation**, `/catkin_ws/src/ros-bridge/carla_ackermann_control/config/settings.yaml`.

<!NODE->

`/carla_ackermann_control_ego_vehicle (Node)`

Converts AckermannDrive messages toCarlaEgoVehicleControl.msg. Speed is in **m/s**, steering angle in **radians** and refers to driving angle, not wheel angle.

Subscribed to:

- `/carla/ego_vehicle/ackermann_cmd` — `ackermann_msgs.AckermannDrive`
- `/carla/ego_vehicle/vehicle_info` — `carla_msgs.CarlaEgoVehicleInfo`
- `/carla/ego_vehicle/vehicle_status` — `carla_msgs.CarlaEgoVehicleStatus`

Publishes in:

- /carla/ego_vehicle/ackermann_control/parameter_descriptions — dynamic_reconfigure/ConfigDescription
- /carla/ego_vehicle/ackermann_control/control_info — carla_ackermann_control.EgoVehicleControlInfo
- /carla/ego_vehicle/ackermann_control/parameter_updates — dynamic_reconfigure/Config
- /carla/ego_vehicle/vehicle_control_cmd — carla_msgs.CarlaEgoVehicleControl

carla_ego_vehicle.launch

Spawns an ego vehicle (`role-name="ego_vehicle"`). The argument `sensor_definition_file` describes the sensors attached to the vehicle. It is the location of a `.json` file. The format of this file is explained here.

To spawn the vehicle at a specific location, publish in `/carla/ego_vehicle/initialpose`, or use **RVIZ** and select a position with **2D Pose estimate**.

<!NODE->

carla_ego_vehicle_ego_vehicle (Node)

Spawns an ego vehicle with sensors attached, and waits for world information.

Subscribed to:

- /carla/ego_vehicle/initialpose — geometry_msgs/PoseWithCovarianceStamped
- /carla/world_info — carla_msgs.CarlaWorldInfo

carla_example_ego_vehicle.launch

Based on `carla_ego_vehicle.launch`, spawns an ego vehicle (`role-name="ego_vehicle"`). The file `sensors.json` describes the sensors attached. The path to it depends on the bridge installation.

- **Deb repository installation**, `/opt/carla-ros-bridge/melodic/share/carla_ego_vehicle/config/sensors.json`.
- **Source repository installation**, `/catkin_ws/src/ros-bridge/carla_ego_vehicle/config/sensors.json`.

<!NODE->

carla_ego_vehicle_ego_vehicle (Node)

Spawns an ego vehicle with sensors attached and waits for world information.

Subscribed to:

- /carla/ego_vehicle/initialpose — geometry_msgs/PoseWithCovarianceStamped
- /carla/world_info — carla_msgs.CarlaWorldInfo

carla_infrastructure.launch

Spawns infrastructure sensors and requires the argument `infrastructure_sensor_definition_file` with the location of a `.json` file describing these sensors.

<!NODE->

/carla_infrastructure (Node)

Spawns the infrastructure sensors passed as arguments.

Subscribed to:

- /carla/world_info — carla_msgs.CarlaWorldInfo

carla_ros_bridge.launch

Creates a node with some basic communications between CARLA and ROS.

<!NODE->

carla_ros_bridge (Node)

Publishes the data regarding the current state of the simulation. Reads the debug shapes being drawn.

Subscribed to:

- /carla/debug_marker — visualization_msgs.MarkerArray

Publishes in:

- /carla/actor_list — carla_msgs.CarlaActorList
- /carla/objects — derived_object_msgs.ObjectArrayring
- /carla/status — carla_msgs.CarlaStatus
- /carla/traffic_lights — carla_msgs.CarlaTrafficLightStatusList
- /carla/world_info — carla_msgs.CarlaWorldInfo

carla_ros_bridge_with_ackermann_control.launch

Launches two basic nodes. One retrieves simulation data, the other controls a vehicle using AckermannDrive messages.

<!NODE->

carla_ros_bridge (Node)

Publishes data regarding the current state of the simulation. Reads the debug shapes being drawn.

Subscribed to:

- /carla/debug_marker — visualization_msgs.MarkerArray

Publishes in:

- /carla/actor_list — carla_msgs.CarlaActorList
- /carla/objects — derived_object_msgs.ObjectArrayring
- /carla/status — carla_msgs.CarlaStatus
- /carla/traffic_lights — carla_msgs.CarlaTrafficLightStatusList
- /carla/world_info — carla_msgs.CarlaWorldInfo

<!NODE->

/carla_ackermann_control_ego_vehicle (Node)

Converts AckermannDrive messages toCarlaEgoVehicleControl.msg. Speed is in **m/s**, steering angle is in **radians** and refers to driving angle, not wheel angle.

Subscribed to:

- /carla/ego_vehicle/ackermann_cmd — ackermann_msgs.AckermannDrive
- /carla/ego_vehicle/vehicle_info — carla_msgs.CarlaEgoVehicleInfo
- /carla/ego_vehicle/vehicle_status — carla_msgs.CarlaEgoVehicleStatus

Publishes in:

- /carla/ego_vehicle/ackermann_control/parameter_descriptions — dynamic_reconfigure/ConfigDescription
- /carla/ego_vehicle/ackermann_control/control_info — carla_ackermann_control.EgoVehicleControlInfo
- /carla/ego_vehicle/ackermann_control/parameter_updates — dynamic_reconfigure/Config
- /carla/ego_vehicle/vehicle_control_cmd — carla_msgs.CarlaEgoVehicleControl

carla_ros_bridge_with_example_ego_vehicle.launch

Spawns an ego vehicle with sensors attached, and starts communications between CARLA and ROS. Both share current simulation state, sensor and ego vehicle data. The ego vehicle is set ready to be used in manual control.

<!NODE->

carla_ros_bridge (Node)

In charge of the communications between CARLA and ROS. They share the current state of the simulation, traffic lights, vehicle controllers and sensor data.

Subscribed to:

- /carla/debug_marker — visualization_msgs.MarkerArray
- /carla/ego_vehicle/enable_autopilot — std_msgs.Bool
- /carla/ego_vehicle/twist — geometry_msgs.Twist
- /carla/ego_vehicle/vehicle_control_cmd — carla_msgs.CarlaEgoVehicleControl
- /carla/ego_vehicle/vehicle_control_cmd_manual — carla_msgs.CarlaEgoVehicleControl
- /carla/ego_vehicle/vehicle_control_manual_override — std_msgs.Bool

Publishes in:

- /carla/actor_list — carla_msgs.CarlaActorList

- /carla/ego_vehicle/camera/rgb/front/camera_info — sensor_msgs.CameraInfo
- /carla/ego_vehicle/camera/rgb/front/image_color — sensor_msgs.Image
- /carla/ego_vehicle/camera/rgb/view/camera_info — sensor_msgs.CameraInfo
- /carla/ego_vehicle/camera/rgb/view/image_color — sensor_msgs.Image
- /carla/ego_vehicle/gnss/gnss1/fix — sensor_msgs.NavSatFix
- /carla/ego_vehicle/imu — sensor_msgs.Imu
- /carla/ego_vehicle/lidar/lidar1/point_cloud — sensor_msgs.PointCloud2
- /carla/ego_vehicle/objects — derived_object_msgs.ObjectArray
- /carla/ego_vehicle/odometry — nav_msgs.Odometry
- /carla/ego_vehicle/radar/front/radar — ainstein_radar_msgs.RadarTargetArray
- /carla/ego_vehicle/vehicle_info — carla_msgs.CarlaEgoVehicleInfo
- /carla/ego_vehicle/vehicle_status — carla_msgs.CarlaEgoVehicleStatus
- /carla/objects — derived_object_msgs.ObjectArrayring
- /carla/status — carla_msgs.CarlaStatus
- /carla/traffic_lights — carla_msgs.CarlaTrafficLightStatusList
- /carla/world_info — carla_msgs.CarlaWorldInfo

<!NODE->

/carla_manual_control_ego_vehicle (Node)

Retrieves information from CARLA regarding the ego vehicle. Uses keyboard input to publish controller messages for the ego vehicle. The information retrieved includes static data, current state, sensor data, and simulation settings.

Subscribed to:

- /carla/ego_vehicle/camera/rgb/view/image_color — sensor_msgs.Image
- /carla/ego_vehicle/collision — carla_msgs.CarlaCollisionEvent
- /carla/ego_vehicle/gnss/gnss1/fix — sensor_msgs.NavSatFix
- /carla/ego_vehicle/lane_invasion — carla_msgs.CarlaLaneInvasionEvent
- /carla/ego_vehicle/vehicle_control_manual_override — std_msgs.Bool
- /carla/ego_vehicle/vehicle_info — carla_msgs.CarlaEgoVehicleInfo
- /carla/ego_vehicle/vehicle_status — carla_msgs.CarlaEgoVehicleStatus
- /carla/status — carla_msgs.CarlaStatus

Publishes in:

- /carla/ego_vehicle/enable_autopilot — std_msgs.Bool
- /carla/ego_vehicle/vehicle_control_cmd_manual — carla_msgs.CarlaEgoVehicleControl
- /carla/ego_vehicle/vehicle_control_manual_override — std_msgs.Bool

<!NODE->

/carla_ego_vehicle_ego_vehicle (Node)

Spawns an ego vehicle with sensors attached. Reads world information.

Subscribed to:

- /carla/ego_vehicle/initialpose — geometry_msgs/PoseWithCovarianceStamped

carla_ros_bridge_with_rviz.launch

Starts communications between CARLA and ROS, and launches RVIZ to retrieve Lidar data. <!NODE->

carla_ros_bridge (Node)

Shares information between CARLA and ROS regarding the current simulation state.

Subscribed to:

- /carla/debug_marker — visualization_msgs.MarkerArray

Publishes in:

- /carla/actor_list — carla_msgs.CarlaActorList
- /carla/objects — derived_object_msgs.ObjectArrayring
- /carla/status — carla_msgs.CarlaStatus
- /carla/traffic_lights — carla_msgs.CarlaTrafficLightStatusList
- /carla/world_info — carla_msgs.CarlaWorldInfo

```
<!NODE->
```

```
/rviz (Node)
```

Runs an instance of RVIZ, and waits for Lidar data.

Subscribed to:

- /carla/vehicle_marker — visualization_msgs/Marker
- /carla/vehicle_marker_array — visualization_msgs/MarkerArray
- /carla/ego_vehicle/lidar/front/point_cloud — sensor_msgs.PointCloud2

carla_manual_control.launch

A ROS version of the CARLA script `manual_control.py`. It has some prerequisites.

- **To display an image**, a camera with role-name `view` and resolution 800x600.
- **To display the position**, a gnss sensor with role-name `gnss1`.
- **To detect other sensor data**, the corresponding sensor.

```
<!NODE->
```

```
/carla_manual_control_ego_vehicle (Node)
```

Retrieves information from CARLA regarding the ego vehicle. Uses keyboard input to publish controller messages for the ego vehicle. The information retrieved includes static data, current state, sensor data, and simulation settings.

Subscribed to:

- /carla/ego_vehicle/camera/rgb/view/image_color — sensor_msgs.Image
- /carla/ego_vehicle/collision — carla_msgs.CarlaCollisionEvent
- /carla/ego_vehicle/gnss/gnss1/fix — sensor_msgs.NavSatFix
- /carla/ego_vehicle/lane_invasion — carla_msgs.CarlaLaneInvasionEvent
- /carla/ego_vehicle/vehicle_control_manual_override — std_msgs.Bool
- /carla/ego_vehicle/vehicle_info — carla_msgs.CarlaEgoVehicleInfo
- /carla/ego_vehicle/vehicle_status — carla_msgs.CarlaEgoVehicleStatus
- /carla/status — carla_msgs.CarlaStatus

Publishes in:

- /carla/ego_vehicle/enable_autopilot — std_msgs.Bool
- /carla/ego_vehicle/vehicle_control_cmd_manual — carla_msgs.CarlaEgoVehicleControl
- /carla/ego_vehicle/vehicle_control_manual_override — std_msgs.Bool

carla_pcl_recorder.launch

Creates a pointcloud map for the current CARLA level. An autopilot ego vehicle roams around the map with a LIDAR sensor attached.

The point clouds are saved in the `/tmp/pcl_capture` directory. Once the capture is done, the overall size can be reduced.

```
1 #create one point cloud file
2 pcl.concatenate_points_pcd /tmp/pcl_capture/*.pcd
3
4 #filter duplicates
5 pcl_voxel_grid -leaf 0.1,0.1,0.1 output.pcd map.pcd
6
7 #verify the result
8 pcl_viewer map.pcd
```

The launch file requires some functionality that is not part of the python egg-file. The PYTHONPATH has to be extended.

```
1 export
  PYTHONPATH=<path-to-carla>/PythonAPI/carla/dist/carla-<version_and_arch>.egg:<path-to-carla>/PythonAPI/carla
```

```
<!NODE->
```

carla_ros_bridge (Node)

In charge of most of the communications between CARLA and ROS. Both share the current state of the simulation, traffic lights, vehicle controllers and sensor data.

Subscribed to:

- /carla/debug_marker — visualization_msgs.MarkerArray
- /carla/ego_vehicle/enable_autopilot — std_msgs.Bool
- /carla/ego_vehicle/twist — geometry_msgs.Twist
- /carla/ego_vehicle/vehicle_control_cmd — carla_msgs.CarlaEgoVehicleControl
- /carla/ego_vehicle/vehicle_control_cmd_manual — carla_msgs.CarlaEgoVehicleControl
- /carla/ego_vehicle/vehicle_control_manual_override — std_msgs.Bool

Publishes in:

- /carla/actor_list — carla_msgs.CarlaActorList
- /carla/ego_vehicle/camera/rgb/front/camera_info — sensor_msgs.CameraInfo
- /carla/ego_vehicle/camera/rgb/front/image_color — sensor_msgs.Image
- /carla/ego_vehicle/camera/rgb/view/camera_info — sensor_msgs.CameraInfo
- /carla/ego_vehicle/camera/rgb/view/image_color — sensor_msgs.Image
- /carla/ego_vehicle/gnss/gnss1/fix — sensor_msgs.NavSatFix
- /carla/ego_vehicle/imu — sensor_msgs.Imu
- /carla/ego_vehicle/lidar/lidar1/point_cloud — sensor_msgs.PointCloud2
- /carla/ego_vehicle/objects — derived_object_msgs.ObjectArray
- /carla/ego_vehicle/odometry — nav_msgs.Odometry
- /carla/ego_vehicle/vehicle_status — carla_msgs.CarlaEgoVehicleStatus
- /carla/ego_vehicle/radar/front/radar — ainstein_radar_msgs.RadarTargetArray
- /carla/ego_vehicle/vehicle_info — carla_msgs.CarlaEgoVehicleInfo
- /carla/marker — visualization_msgs.Marker
- /carla/objects — derived_object_msgs.ObjectArrayring
- /carla/status — carla_msgs.CarlaStatus
- /carla/traffic_lights — carla_msgs.CarlaTrafficLightStatusList
- /carla/world_info — carla_msgs.CarlaWorldInfo

<!NODE->

/carla_manual_control_ego_vehicle (Node)

Retrieves information from CARLA regarding the ego vehicle. Uses keyboard input to publish controller messages for the ego vehicle. The information retrieved includes static data, current state, sensor data, and simulation settings.

Subscribed to:

- /carla/ego_vehicle/camera/rgb/view/image_color — sensor_msgs.Image
- /carla/ego_vehicle/collision — carla_msgs.CarlaCollisionEvent
- /carla/ego_vehicle/gnss/gnss1/fix — sensor_msgs.NavSatFix
- /carla/ego_vehicle/lane_invasion — carla_msgs.CarlaLaneInvasionEvent
- /carla/ego_vehicle/vehicle_control_manual_override — std_msgs.Bool
- /carla/ego_vehicle/vehicle_info — carla_msgs.CarlaEgoVehicleInfo
- /carla/ego_vehicle/vehicle_status — carla_msgs.CarlaEgoVehicleStatus
- /carla/status — carla_msgs.CarlaStatus

Publishes in:

- /carla/ego_vehicle/enable_autopilot — std_msgs.Bool
- /carla/ego_vehicle/vehicle_control_cmd_manual — carla_msgs.CarlaEgoVehicleControl
- /carla/ego_vehicle/vehicle_control_manual_override — std_msgs.Bool

<!NODE->

/carla_ego_vehicle_ego_vehicle (Node)

Spawns an ego vehicle with sensors attached. Waits for world information.

Subscribed to:

- /carla/ego_vehicle/initialpose — geometry_msgs/PoseWithCovarianceStamped

```
<!NODE->
```

```
/enable_autopilot_rostopic (Node)
```

Changes between autopilot and manual control modes.

Publishes in:

- /carla/ego_vehicle/enable_autopilot — std_msgs.Bool

```
<!NODE->
```

```
/pcl_recorder_node (Node)
```

Receives the cloud point data.

Subscribed to:

- /carla/ego_vehicle/lidar/lidar1/point_cloud — sensor_msgs.PointCloud2

carla_waypoint_publisher.launch

Calculates a waypoint route for an ego vehicle. The route is published in /carla/<ego vehicle name>/waypoints. The goal is either read from the ROS topic /carla/<ROLE NAME>/goal, or a fixed spawnpoint is used. The preferred way of setting a goal is to click **2D Nav Goal** in RVIZ.

The launch file requires some functionality that is not part of the python egg-file. The PYTHONPATH has to be extended.

```
1 export
```

```
PYTHONPATH=$PYTHONPATH:<path-to-carla>/PythonAPI/carla-<carla_version_and_arch>.egg:<path-to-carla>/PythonA
```

```
<!NODE->
```

```
/carla_waypoint_publisher (Node)
```

Uses the current pose of the ego vehicle as starting point. If the vehicle is respawned or moved, the route is calculated again.

Subscribed to:

- /carla/world_info — carla_msgs.CarlaWorldInfo

8 Tutorials (general)

8.1 How to add friction triggers

Friction Triggers are box triggers that can be added on runtime and let users define a different friction of the vehicles' wheels when being inside those type of triggers. For example, this could be useful for making slippery surfaces in certain regions of a map dynamically.

In order to spawn a friction trigger using PythonAPI, users must first get the `static.trigger.friction` blueprint definition, and then set the following necessary attributes to that blueprint definition:

- `friction`: The friction of the trigger box when vehicles are inside it.
- `extent_x`: The extent of the bounding box in the X coordinate in centimeters.
- `extent_y`: The extent of the bounding box in the Y coordinate in centimeters.
- `extent_z`: The extent of the bounding box in the Z coordinate in centimeters.

Once done that, define a transform to specify the location and rotation for the friction trigger and spawn it.

Example

```
1 import carla
2
3 def main():
4     # Connect to client
5     client = carla.Client('127.0.0.1', 2000)
6     client.set_timeout(2.0)
7
8     # Get World and Actors
```

```

9     world = client.get_world()
10    actors = world.get_actors()
11
12    # Find Trigger Friction Blueprint
13    friction_bp = world.get_blueprint_library().find('static.trigger.friction')
14
15    extent = carla.Location(700.0, 700.0, 700.0)
16
17    friction_bp.set_attribute('friction', str(0.0))
18    friction_bp.set_attribute('extent_x', str(extent.x))
19    friction_bp.set_attribute('extent_y', str(extent.y))
20    friction_bp.set_attribute('extent_z', str(extent.z))
21
22    # Spawn Trigger Friction
23    transform = carla.Transform()
24    transform.location = carla.Location(100.0, 0.0, 0.0)
25    world.spawn_actor(friction_bp, transform)
26
27    # Optional for visualizing trigger
28    world.debug.draw_box(box=carla.BoundingBox(transform.location, extent * 1e-2),
29                          rotation=transform.rotation, life_time=100, thickness=0.5, color=carla.Color(r=255,g=0,b=0))
30
31 if __name__ == '__main__':
32     main()

```

8.2 How to control vehicle physics

Physics properties can be tuned for vehicles and its wheels. These changes are applied **only** on runtime, and values are set back to default ones when the execution ends.

These properties are controlled through a `carla.VehiclePhysicsControl` object, which also provides the control of each wheel's physics through a `carla.WheelPhysicsControl` object.

Example

```

1 import carla
2 import random
3
4 def main():
5     # Connect to client
6     client = carla.Client('127.0.0.1', 2000)
7     client.set_timeout(2.0)
8
9     # Get World and Actors
10    world = client.get_world()
11    current_map = world.get_map()
12    actors = world.get_actors()
13
14    # Get a random vehicle from world (there should be one at least)
15    vehicle = random.choice([actor for actor in actors if 'vehicle' in actor.type_id])
16
17    # Create Wheels Physics Control
18    front_left_wheel = carla.WheelPhysicsControl(tire_friction=4.5, damping_rate=1.0,
19                                                   max_steer_angle=70.0, radius=30.0)
20    front_right_wheel = carla.WheelPhysicsControl(tire_friction=2.5, damping_rate=1.5,
21                                                   max_steer_angle=70.0, radius=25.0)
22    rear_left_wheel = carla.WheelPhysicsControl(tire_friction=1.0, damping_rate=0.2,
23                                                 max_steer_angle=0.0, radius=15.0)
24    rear_right_wheel = carla.WheelPhysicsControl(tire_friction=1.5, damping_rate=1.3,
25                                                 max_steer_angle=0.0, radius=20.0)

```

```

23     wheels = [front_left_wheel, front_right_wheel, rear_left_wheel, rear_right_wheel]
24
25     # Change Vehicle Physics Control parameters of the vehicle
26     physics_control = vehicle.get_physics_control()
27
28     physics_control.torque_curve = [carla.Vector2D(x=0, y=400), carla.Vector2D(x=1300, y=600)]
29     physics_control.max_rpm = 10000
30     physics_control.moi = 1.0
31     physics_control.damping_rate_full_throttle = 0.0
32     physics_control.use_gear_autobox = True
33     physics_control.gear_switch_time = 0.5
34     physics_control.clutch_strength = 10
35     physics_control.mass = 10000
36     physics_control.drag_coefficient = 0.25
37     physics_control.steering_curve = [carla.Vector2D(x=0, y=1), carla.Vector2D(x=100, y=1),
38                                         carla.Vector2D(x=300, y=1)]
39     physics_control.wheels = wheels
40
41     # Apply Vehicle Physics Control for the vehicle
42     vehicle.apply_physics_control(physics_control)
43
44 if __name__ == '__main__':
45     main()

```

8.3 Walker Bone Control

In this tutorial we describe how to manually control and animate the skeletons of walkers from the CARLA Python API. The reference of all classes and methods available can be found at [Python API reference](#).

- **Walker skeleton structure**
- **Manually control walker bones**
 - Connect to the simulator
 - Spawn a walker
 - Control walker skeletons



This document assumes the user is familiar with the Python API . The user should read the first steps tutorial before reading this document. Core concepts.

Walker skeleton structure

All walkers have the same skeleton hierarchy and bone names. Below is an image of the skeleton hierarchy.

```

1 crl_root
2   crl_hips__C
3     crl_spine__C
4       crl_spine01__C
5         ctrl_shoulder__L
6           crl_arm__L
7             crl_foreArm__L
8               crl_hand__L
9                 crl_handThumb__L
10                crl_handThumb01__L
11                  crl_handThumb02__L
12                    crl_handThumbEnd__L
13                      crl_handIndex__L
14                        crl_handIndex01__L
15                          crl_handIndex02__L
16                            crl_handIndexEnd__L
17                              crl_handMiddle__L
18                                crl_handMiddle01__L
19                                  crl_handMiddle02__L

```

```

20             crl_handMiddleEnd__L
21             crl_handRing_L
22                 crl_handRing01__L
23                     crl_handRing02__L
24                         crl_handRingEnd__L
25             crl_handPinky_L
26                 crl_handPinky01__L
27                     crl_handPinky02__L
28                         crl_handPinkyEnd__L
29             crl_neck__C
30             crl_Head__C
31                 crl_eye__L
32                     crl_eye__R
33             crl_shoulder__R
34             crl_arm__R
35                 crl_foreArm__R
36             crl_hand__R
37                 crl_handThumb__R
38                     crl_handThumb01__R
39                         crl_handThumb02__R
40                         crl_handThumbEnd__R
41             crl_handIndex__R
42                 crl_handIndex01__R
43                     crl_handIndex02__R
44                         crl_handIndexEnd__R
45             crl_handMiddle__R
46                 crl_handMiddle01__R
47                     crl_handMiddle02__R
48                         crl_handMiddleEnd__R
49             crl_handRing__R
50                 crl_handRing01__R
51                     crl_handRing02__R
52                         crl_handRingEnd__R
53             crl_handPinky__R
54                 crl_handPinky01__R
55                     crl_handPinky02__R
56                         crl_handPinkyEnd__R
57             crl_thigh__L
58             crl_leg__L
59                 crl_foot__L
60                     crl_toe__L
61                         crl_toeEnd__L
62             crl_thigh__R
63             crl_leg__R
64                 crl_foot__R
65                     crl_toe__R
66                         crl_toeEnd__R

```

Manually control walker bones

Following is a detailed step-by-step example of how to change the bone transforms of a walker from the CARLA Python API

Connect to the simulator Import neccessary libraries used in this example

```

1 import carla
2 import random

```

Initialize the carla client

```

1 client = carla.Client('127.0.0.1', 2000)
2 client.set_timeout(2.0)

```

Spawn a walker Spawn a random walker at one of the map's spawn points

```
1 world = client.get_world()
2 blueprint = random.choice(self.world.get_blueprint_library().filter('walker.*'))
3 spawn_points = world.get_map().get_spawn_points()
4 spawn_point = random.choice(spawn_points) if spawn_points else carla.Transform()
5 world.try_spawn_actor(blueprint, spawn_point)
```

Control walker skeletons A walker's skeleton can be modified by passing an instance of the WalkerBoneControl class to the walker's apply_control function. The WalkerBoneControl class contains the transforms of the bones to be modified. Its bone_transforms member is a list of tuples of value pairs where the first value is the bone name and the second value is the bone transform. The apply_control function can be called on every tick to animate a walker's skeleton. The location and rotation of each transform is relative to its parent. Therefore when a parent bone's transform is modified, the transforms of the child bones in model space will also be changed relatively.

In the example below, the rotations of the walker's hands are set to be 90 degrees around the forward axis and the locations are set to the origin.

```
1 control = carla.WalkerBoneControl()
2 first_tuple = ('crl_hand_R', carla.Transform(rotation=carla.Rotation(roll=90)))
3 second_tuple = ('crl_hand_L', carla.Transform(rotation=carla.Rotation(roll=90)))
4 control.bone_transforms = [first_tuple, second_tuple]
5 world.player.apply_control(control)
```

8.4 Generate maps with OpenStreetMap

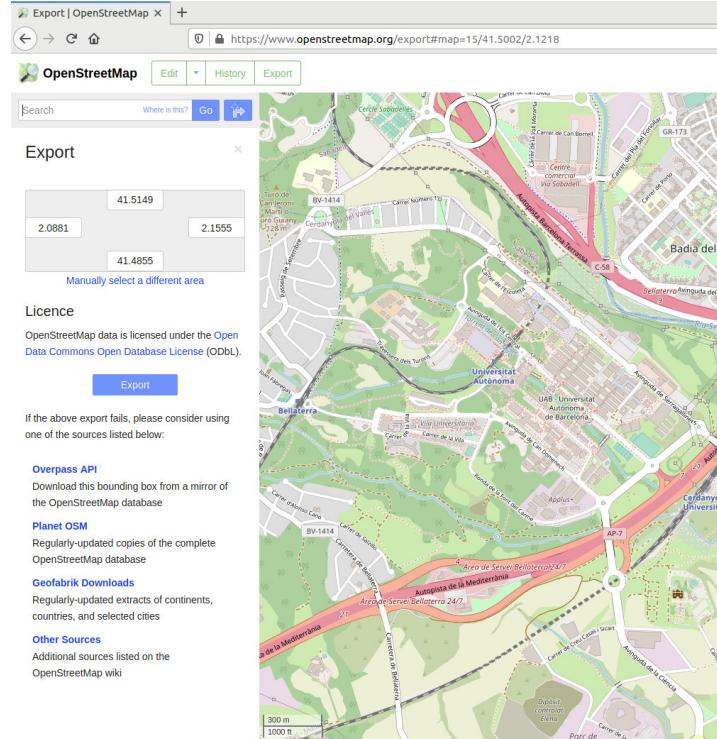
OpenStreetMap is an open license map of the world developed by contributors. Sections of these map can be exported to a .osm file, the OpenStreetMap format, which is essentially an XML. CARLA can convert this file to OpenDRIVE format and ingest it as any other OpenDRIVE map using theOpenDRIVE Standalone Mode. The process is quite straightforward.

- 1- Obtain a map with OpenStreetMap
- 2- Convert to OpenDRIVE format
- 3- Import into CARLA

1- Obtain a map with OpenStreetMap

The first thing to do is use OpenStreetMap to generate the file containing the map information.

1.1. Go to openstreetmap.org. If the map is not properly visualized, try changing the layer in the right panel.



1.2 Search for a desired location and zoom in to a specific area.



Due to the Unreal Engine limitations, CARLA can ingest maps of a limited size (large cities like Paris push the limits of the engine). Additionally, the bigger the map, the longer the conversion to OpenDRIVE will take.

1.3. Click on Export in the upper left side of the window. The **Export** pannel will open.

1.4. Click on Manually select a different area in the **Export** pannel.

1.5. Select a custom area by dragging the corners of the square area in the viewport.

1.6. Click the Export button in the **Export** pannel, and save the map information of the selected area as a **.osm** file.

2- Convert to OpenDRIVE format

CARLA can read the content in the **.osm** file generated with OpenStreetMap, and convert it to OpenDRIVE format so that it can be ingested as a CARLA map. This can be done using the following classes in the PythonAPI.

- **carla.Osm2Odr** – The class that does the conversion. It takes the content of the **.osm** parsed as strind, and returns a string containing the resulting **.xodr**.
 - **osm_file** — The content of the initial **.osm** file parsed as string.
 - **settings** — A **carla.Osm2OdrSettings** object containing the settings to parameterize the conversion.
- **carla.Osm2OdrSettings** – Helper class that contains different parameters used during the conversion.
 - **use_offsets (default False)** — Determines whereas the map should be generated with an offset, thus moving the origin from the center according to that offset.
 - **offset_x (default 0.0)** — Offset in the X axis.
 - **offset_y (default 0.0)** — Offset in the Y axis.
 - **default_lane_width (default 4.0)** — Determines the width that lanes should have in the resulting XODR file.
 - **elevation_layer_height (default 0.0)** — Determines the height separating elements in different layers, used for overlapping elements. Read more on layers.

The input and output of the conversion are not the **.osm** and **.xodr** files itself, but their content. For said reason, the code should be similar to the following.

```
1 # Read the .osm data
2 f = open("path/to/osm/file", 'r')
3 osm_data = f.read()
```

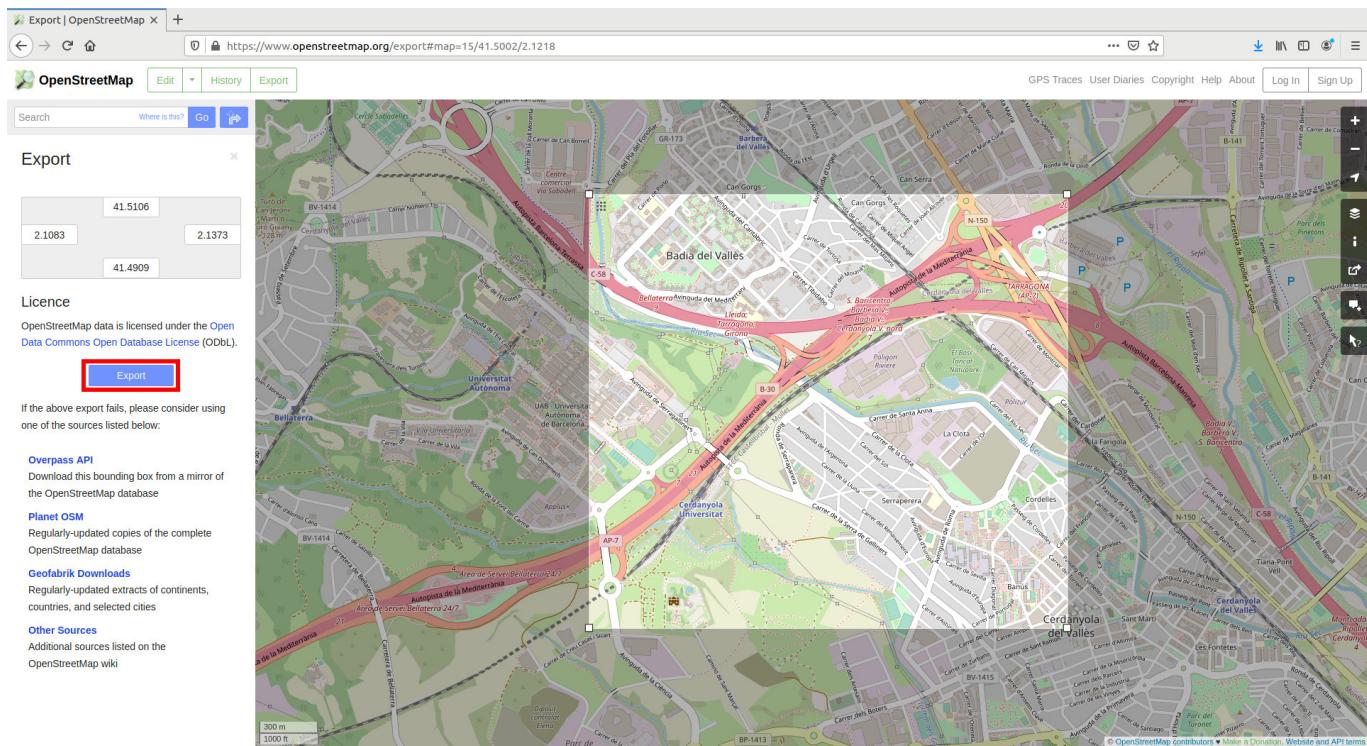


Figure 50: openstreetmap_area

```

4 f.close()
5
6
7 # Define the desired settings. In this case, default values.
8 settings = carla.Osm2OdrSettings()
9 # Convert to .xodr
10 xodr_data = carla.Osm2Odr.convert(osm_data, settings)
11
12
13 # save opendrive file
14 f = open("path/to/output/file", 'w')
15 f.write(xodr_data)
16 f.close()

```

The resulting file contains the road information in OpenDRIVE format.

3- Import into CARLA

Finally, the OpenDRIVE file can be easily ingested in CARLA using the OpenDRIVE Standalone Mode.

a) Using your own script — Call for `client.generate_opendrive_world()` through the API. This will generate the new map, and block the simulation until it is ready. Use the `carla.OpendriveGenerationParameters` class to set the parameterization of the mesh generation.

```

1 vertex_distance = 2.0 # in meters
2 max_road_length = 500.0 # in meters
3 wall_height = 0.0      # in meters
4 extra_width = 0.6      # in meters
5 world = client.generate_opendrive_world(
6     xodr_xml, carla.OpendriveGenerationParameters(
7         vertex_distance=vertex_distance,
8         max_road_length=max_road_length,
9         wall_height=wall_height,
10        additional_width=extra_width,
11        smooth_junctions=True,

```

```
12     enable_mesh_visibility=True))
```



‘wall height = 0.0’ is strongly recommended. OpenStreetMap defines lanes in opposing directions as different roads. If walls are generated, this result in wall overlapping and undesired collisions.

- b) Using `config.py` — The script can load an OpenStreetMap file directly into CARLA using a new argument.
`python3 config.py --osm-file=/path/to/OSM/file`



[`client.generate.opendrive world()`]([`python api.md carla.Client.generate.opendrive world`]) requires the content of the OpenDRIVE file parsed as string , and allows parameterization. On the contrary, ‘`config.py`’ script needs the path to the ‘`xodr`’ file and always uses default parameters.

Either way, the map should be ingested automatically in CARLA and the result should be similar to this.



Outcome of the CARLA map generation using OpenStreetMap.



The roads generated end abruptly in the borders of the map. This will cause the TM to crash when vehicles are not able to find the next waypoint. To avoid this, the OSM mode is set to True by default ([`set osm mode()`]([`python api.md carlatrafficmanager`])). This will show a warning, and destroy vehicles when necessary.

That is all there is to know about how to use OpenStreetMap to generate CARLA maps.

For issues and doubts related with this topic can be posted in the CARLA forum.

8.5 Retrieve simulation data

Learning an efficient way to retrieve simulation data is essential in CARLA. This holistic tutorial is advised for both, newcomers and more experienced users. It starts from the very beginning, and gradually dives into the many options available in CARLA.

First, the simulation is initialized with custom settings and traffic. An ego vehicle is set to roam around the city, optionally with some basic sensors. The simulation is recorded, so that later it can be queried to find the highlights. After that, the original simulation is played back, and exploited to the limit. New sensors can be added to retrieve consistent data. The weather conditions can be changed. The recorder can even be used to test specific scenarios with different outputs.

- **Overview**
- **Set the simulation**
 - Map setting
 - Weather setting
- **Set traffic**
 - CARLA traffic and pedestrians
 - SUMO co-simulation traffic
- **Set the ego vehicle**
 - Spawn the ego vehicle
 - Place the spectator
- **Set basic sensors**
 - RGB camera
 - Detectors
 - Other sensors
- **Set advanced sensors**
 - Depth camera
 - Semantic segmentation camera
 - LIDAR raycast sensor
 - Radar sensor
- **No-rendering-mode**
 - Simulate at a fast pace
 - Manual control without rendering
- **Record and retrieve data**
 - Start recording
 - Capture and record
 - Stop recording
- **Exploit the recording**
 - Query the events
 - Choose a fragment
 - Retrieve more data
 - Change the weather
 - Try new outcomes
- **Tutorial scripts**

Overview

There are some common mistakes in the process of retrieving simulation data. Flooding the simulator with sensors, storing useless data, or struggling to find a specific event are some examples. However, some outlines to this process can be provided. The goal is to ensure that data can be retrieved and replicated, and the simulation can be examined and altered at will.



This tutorial uses the CARLA 0.9.8 deb package . There may be minor changes depending on your CARLA version and installation, specially regarding paths.

The tutorial presents a wide set of options for the different steps. All along, different scripts will be mentioned. Not all of them will be used, it depends on the specific use cases. Most of them are already provided in CARLA for generic purposes.

- **config.py** changes the simulation settings. Map, rendering options, set a fixed time-step...
 - carla/PythonAPI/util/config.py
- **dynamic_weather.py** creates interesting weather conditions.
 - carla/PythonAPI/examples/dynamic_weather.py
- **spawn_npc.py** spawns some AI controlled vehicles and walkers.
 - carla/PythonAPI/examples/spawn_npc.py
- **manual_control.py** spawns an ego vehicle, and provides control over it.
 - carla/PythonAPI/examples/manual_control.py

However, there are two scripts mentioned along the tutorial that cannot be found in CARLA. They contain the fragments of code cited. This serves a twofold purpose. First of all, to encourage users to build their own scripts. It is important to have full understanding of what the code is doing. In addition to this, the tutorial is only an outline that may, and should, vary a lot depending on user preferences. These two scripts are just an example.

- **tutorial_ego.py** spawns an ego vehicle with some basic sensors, and enables autopilot. The spectator is placed at the spawning position. The recorder starts at the very beginning, and stops when the script is finished.
- **tutorial_replay.py** reenacts the simulation that **tutorial_ego.py** recorded. There are different fragments of code to query the recording, spawn some advanced sensors, change weather conditions, and reenact fragments of the recording.

The full code can be found in the last section of the tutorial. Remember these are not strict, but meant to be customized. Retrieving data in CARLA is as powerful as users want it to be.



This tutorial requires some knowledge of Python.

Set the simulation

The first thing to do is set the simulation ready to a desired environment.

Run CARLA.

```
1 cd /opt/carla/bin
2 ./CarlaUE.sh
```

Map setting Choose a map for the simulation to run. Take a look at themap documentation to learn more about their specific attributes. For the sake of this tutorial, **Town07** is chosen.

Open a new terminal. Change the map using the **config.py** script.

```
1 cd /opt/carla/PythonAPI/utils
2 python3 config.py --map Town01
```

This script can enable different settings. Some of them will be mentioned during the tutorial, others will not. Hereunder there is a brief summary.

Optional arguments in config.py

```
1 -h, --help                  show this help message and exit
2 --host H                     IP of the host CARLA Simulator (default: localhost)
3 -p P, --port P               TCP port of CARLA Simulator (default: 2000)
4 -d, --default                set default settings
5 -m MAP, --map MAP           load a new map, use --list to see available maps
6 -r, --reload-map            reload current map
7 --delta-seconds S           set fixed delta seconds, zero for variable frame rate
8 --fps N                      set fixed FPS, zero for variable FPS (similar to
                               --delta-seconds)
9
10 --rendering                 enable rendering
11 --no-rendering              disable rendering
12 --no-sync                   disable synchronous mode
13 --weather WEATHER           set weather preset, use --list to see available
                               presets
14
15 -i, --inspect                inspect simulation
16 -l, --list                   list available options
17 -b FILTER, --list-blueprints FILTER
                               list available blueprints matching FILTER (use '*' to
                               list them all)
18
19 -x XODR_FILE_PATH, --xodr-path XODR_FILE_PATH
                               load a new map with a minimum physical road
                               representation of the provided OpenDRIVE
20
21
22
```



Aerial view of Town07

Weather setting Each town is loaded with a specific weather that fits it, however this can be set at will. There are two scripts that offer different approaches to the matter. The first one sets a dynamic weather that changes conditions over time. The other sets custom weather conditions. It is also possible to code weather conditions. This will be covered later when changing weather conditions.

- **To set a dynamic weather.** Open a new terminal and run **dynamic_weather.py**. This script allows to set the ratio at which the weather changes, being 1.0 the default setting.

```
1 cd /opt/carla/PythonAPI/examples
2
3 python3 dynamic_weather.py --speed 1.0
```

- **To set custom conditions.** Use the script **environment.py**. There are quite a lot of possible settings. Take a look at the optional arguments, and the documentation for carla.WeatherParameters.

```
1 cd /opt/carla/PythonAPI/util
2 python3 environment.py --clouds 100 --rain 80 --wetness 100 --puddles 60 --wind 80 --fog 50
```

Optional arguments in environment.py

| | | |
|----|--------------------------------|-----------------------------------------------------|
| 1 | -h, --help | show this help message and exit |
| 2 | --host H | IP of the host server (default: 127.0.0.1) |
| 3 | -p P, --port P | TCP port to listen to (default: 2000) |
| 4 | --sun SUN | Sun position presets [sunset day night] |
| 5 | --weather WEATHER | Weather condition presets [clear overcast rain] |
| 6 | --altitude A, -alt A | Sun altitude [-90.0, 90.0] |
| 7 | --azimuth A, -azm A | Sun azimuth [0.0, 360.0] |
| 8 | --clouds C, -c C | Clouds amount [0.0, 100.0] |
| 9 | --rain R, -r R | Rain amount [0.0, 100.0] |
| 10 | --puddles Pd, -pd Pd | Puddles amount [0.0, 100.0] |
| 11 | --wind W, -w W | Wind intensity [0.0, 100.0] |
| 12 | --fog F, -f F | Fog intensity [0.0, 100.0] |
| 13 | --fogdist Fd, -fd Fd | Fog Distance [0.0, inf) |
| 14 | --wetness Wet, -wet Wet | Wetness intensity [0.0, 100.0] |
| 15 | | |



Weather changes applied

Set traffic

Simulating traffic is one of the best ways to bring the map to life. It is also necessary to retrieve data for urban environments. There are different options to do so in CARLA.

CARLA traffic and pedestrians The CARLA traffic is managed by the Traffic Manager module. As for pedestrians, each of them has their own carla.WalkerAIController.

Open a new terminal, and run **spawn_npc.py** to spawn vehicles and walkers. Let's just spawn 50 vehicles and the same amount of walkers.

```
1 cd /opt/carla/PythonAPI/examples  
2 python3 spawn_npc.py -n 50 -w 50 --safe
```

Optional arguments in **spawn_npc.py**

```
1 -h, --help          show this help message and exit  
2 --host H           IP of the host server (default: 127.0.0.1)  
3 -p P, --port P     TCP port to listen to (default: 2000)  
4 -n N, --number-of-vehicles N  
                  number of vehicles (default: 10)  
6 -w W, --number-of-walkers W  
                  number of walkers (default: 50)  
8 --safe             avoid spawning vehicles prone to accidents  
9 --filterv PATTERN vehicles filter (default: "vehicle.*")  
10 --filterw PATTERN pedestrians filter (default: "walker.pedestrian.*")  
11 -tm_p P, --tm-port P port to communicate with TM (default: 8000)  
12 --sync             Synchronous mode execution
```



Vehicles spawned to simulate traffic.

SUMO co-simulation traffic CARLA can run a co-simulation with SUMO. This allows for creating traffic in SUMO that will be propagated to CARLA. This co-simulation is bidirectional. Spawning vehicles in CARLA will do so in SUMO. Specific docs on this feature can be found [here](#).

This feature is available for CARLA 0.9.8 and later, in **Town01**, **Town04**, and **Town05**. The first one is the most stable.



The co-simulation will enable synchronous mode in CARLA. Read the documentation to find out more about this.

- First of all, install SUMO.

```
1 sudo add-apt-repository ppa:sumo/stable  
2 sudo apt-get update  
3 sudo apt-get install sumo sumo-tools sumo-doc
```

- Set the environment variable SUMO_HOME.

```
1 echo "export SUMO_HOME=/usr/share/sumo" >> ~/.bashrc && source ~/.bashrc
```

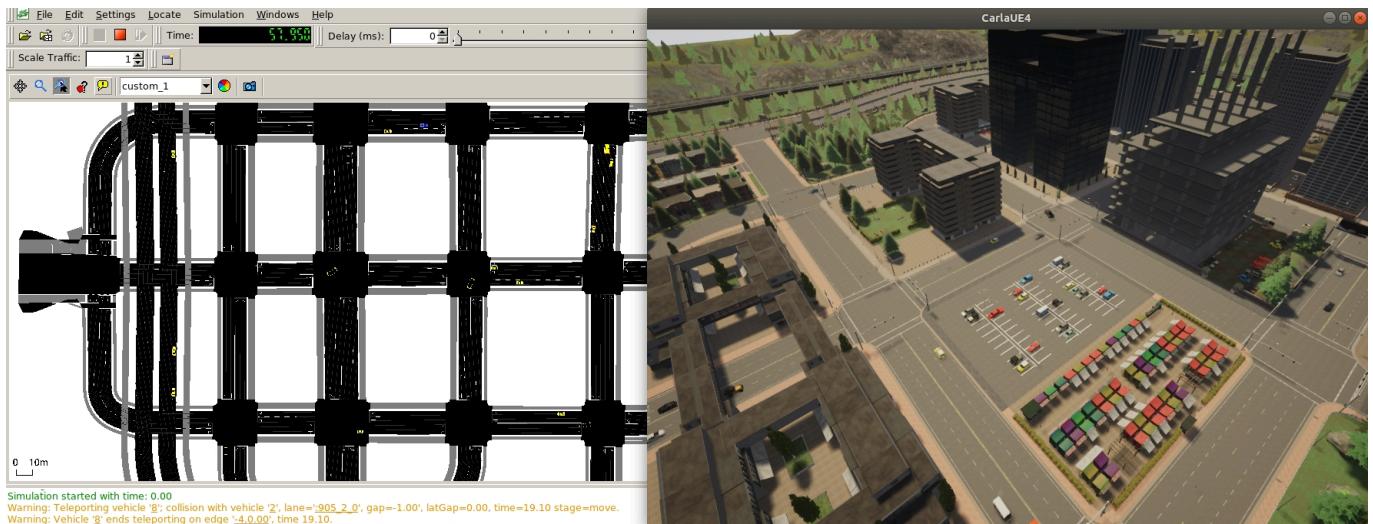
- With the CARLA server on, run the SUMO-CARLA synchrony script.

```
1 cd ~/carla/Co-Simulation/Sumo  
2 python3 run_synchronization.py examples/Town01.sumocfg --sumo-gui
```

- A SUMO window should have opened. **Press Play** in order to start traffic in both simulations.

```
1 > "Play" on SUMO window.
```

The traffic generated by this script is an example created by the CARLA team. By default it spawns the same vehicles following the same routes. These can be changed by the user in SUMO.



SUMO and CARLA co-simulating traffic.



Right now, SUMO co-simulation is a beta feature. Vehicles do not have physics nor take into account CARLA traffic lights.

Set the ego vehicle

From now up to the moment the recorder is stopped, there will be some fragments of code belonging to `tutorial_ego.py`. This script spawns the ego vehicle, optionally some sensors, and records the simulation until the user finishes the script.

Spawn the ego vehicle Vehicles controlled by the user are commonly differentiated in CARLA by setting the attribute `role_name` to `ego`. Other attributes can be set, some with recommended values.

Hereunder, a Tesla model is retrieved from the blueprint library, and spawned with a random recommended colour. One of the recommended spawn points by the map is chosen to place the ego vehicle.

```

1 # --
2 # Spawn ego vehicle
3 # --
4 ego_bp = world.get_blueprint_library().find('vehicle.tesla.model3')
5 ego_bp.set_attribute('role_name','ego')
6 print('\nEgo role_name is set')
7 ego_color = random.choice(ego_bp.get_attribute('color').recommended_values)
8 ego_bp.set_attribute('color',ego_color)
9 print('\nEgo color is set')
10
11 spawn_points = world.get_map().get_spawn_points()
12 number_of_spawn_points = len(spawn_points)
13
14 if 0 < number_of_spawn_points:
15     random.shuffle(spawn_points)
16     ego_transform = spawn_points[0]
17     ego_vehicle = world.spawn_actor(ego_bp,ego_transform)
18     print('\nEgo is spawned')
19 else:
20     logging.warning('Could not find any spawn points')
```

Place the spectator The spectator actor controls the simulation view. Moving it via script is optional, but it may facilitate finding the ego vehicle.

```

1 # --
2 # Spectator on ego position
3 # --
```

```

4 spectator = world.get_spectator()
5 world_snapshot = world.wait_for_tick()
6 spectator.set_transform(ego_vehicle.get_transform())

```

Set basic sensors

The process to spawn any sensor is quite similar.

1. Use the library to find sensor blueprints. **2.** Set specific attributes for the sensor. This is crucial. Attributes will shape the data retrieved. **3.** Attach the sensor to the ego vehicle. **The transform is relative to its parent.** The carla.AttachmentType will determine how the position of the sensor is updated. **4.** Add a `listen()` method. This is the key element. A `lambda` method that will be called each time the sensor listens for data. The argument is the sensor data retrieved.

Having this basic guideline in mind, let's set some basic sensors for the ego vehicle.

RGB camera TheRGB camera generates realistic pictures of the scene. It is the sensor with more settable attributes of them all, but it is also a fundamental one. It should be understood as a real camera, with attributtes such as `focal_distance`, `shutter_speed` or `gamma` to determine how it would work internally. There is also a specific set of attributtes to define the lens distortion, and lots of advanced attributes. For example, the `lens_circle_multiplier` can be used to achieve an effect similar to an eyefish lens. Learn more about them in the documentation.

For the sake of simplicity, the script only sets the most commonly used attributes of this sensor.

- `image_size_x` and `image_size_y` will change the resolution of the output image.
- `fov` is the horizontal field of view of the camera.

After setting the attributes, it is time to spawn the sensor. The script places the camera in the hood of the car, and pointing forward. It will capture the front view of the car.

The data is retrieved as a `carla.Image` on every step. The `listen` method saves these to disk. The path can be altered at will. The name of each image is coded to be based on the simulation frame where the shot was taken.

```

1 # --
2 # Spawn attached RGB camera
3 # --
4 cam_bp = None
5 cam_bp = world.get_blueprint_library().find('sensor.camera.rgb')
6 cam_bp.set_attribute("image_size_x",str(1920))
7 cam_bp.set_attribute("image_size_y",str(1080))
8 cam_bp.set_attribute("fov",str(105))
9 cam_location = carla.Location(2,0,1)
10 cam_rotation = carla.Rotation(0,180,0)
11 cam_transform = carla.Transform(cam_location,cam_rotation)
12 ego_cam = world.spawn_actor(cam_bp,cam_transform,attach_to=ego_vehicle,
    attachment_type=carla.AttachmentType.Rigid)
13 ego_cam.listen(lambda image: image.save_to_disk('tutorial/output/%.6d.jpg' % image.frame))

```



RGB camera output

Detectors These sensors retrieve data when the object they are attached to registers a specific event. There are three type of detector sensors, each one describing one type of event.

Collision detector. Retrieves collisions between its parent and other actors. — Lane invasion detector. — Registers when its parent crosses a lane marking. ***Obstacle detector.** Detects possible obstacles ahead of its parent.

The data they retrieve will be helpful later when deciding which part of the simulation is going to be reenacted. In fact, the collisions can be explicitly queried using the recorder. This is prepared to be printed.

Only the obstacle detector blueprint has attributes to be set. Here are some important ones.

- **sensor_tick** sets the sensor to retrieve data only after x seconds pass. It is a common attribute for sensors that retrieve data on every step.
- **distance and hit-radius** shape the debug line used to detect obstacles ahead.
- **only_dynamics** determines if static objects should be taken into account or not. By default, any object is considered.

The script sets the obstacle detector to only consider dynamic objects. If the vehicle collides with any static object, it will be detected by the collision sensor.

```

1 # --
2 # Add collision sensor to ego vehicle.
3 # --
4
5 col_bp = world.get_blueprint_library().find('sensor.other.collision')
6 col_location = carla.Location(0,0,0)
7 col_rotation = carla.Rotation(0,0,0)
8 col_transform = carla.Transform(col_location,col_rotation)
9 ego_col = world.spawn_actor(col_bp,col_transform,attach_to=ego_vehicle,
   attachment_type=carla.AttachmentType.Rigid)
10 def col_callback(colli):
11     print("Collision detected:\n"+str(colli)+"\n")
12 ego_col.listen(lambda colli: col_callback(colli))
13
14 # --
15 # Add Lane invasion sensor to ego vehicle.
16 # --
```

```

17
18 lane_bp = world.get_blueprint_library().find('sensor.other.lane_invasion')
19 lane_location = carla.Location(0,0,0)
20 lane_rotation = carla.Rotation(0,0,0)
21 lane_transform = carla.Transform(lane_location, lane_rotation)
22 ego_lane = world.spawn_actor(lane_bp, lane_transform, attach_to=ego_vehicle,
   attachment_type=carla.AttachmentType.Rigid)
23 def lane_callback(lane):
24     print("Lane invasion detected:\n"+str(lane)+"\n")
25 ego_lane.listen(lambda lane: lane_callback(lane))
26
27 # --
28 # Add Obstacle sensor to ego vehicle.
29 # --
30
31 obs_bp = world.get_blueprint_library().find('sensor.other.obstacle')
32 obs_bp.set_attribute("only_dynamics",str(True))
33 obs_location = carla.Location(0,0,0)
34 obs_rotation = carla.Rotation(0,0,0)
35 obs_transform = carla.Transform(obs_location, obs_rotation)
36 ego_obs = world.spawn_actor(obs_bp, obs_transform, attach_to=ego_vehicle,
   attachment_type=carla.AttachmentType.Rigid)
37 def obs_callback(obs):
38     print("Obstacle detected:\n"+str(obs)+"\n")
39 ego_obs.listen(lambda obs: obs_callback(obs))

```

```

Lane invasion detected:
LaneInvasionEvent(frame=800, timestamp=11.812701)

Obstacle detected:
ObstacleDetectionEvent(frame=800, timestamp=11.812701, other_actor=0x7f99ec7f10c0)

Collision detected:
CollisionEvent(frame=796, timestamp=11.556151, other_actor=0x7f99ec002ed0)

```

Output for detector sensors

Other sensors Only two sensors of this category will be considered for the time being.

GNSS sensor. Retrieves the geolocation of the sensor. **IMU sensor.** Comprises an accelerometer, a gyroscope, and a compass.

To get general measures for the vehicle object, these two sensors are spawned centered to it.

The attributes available for these sensors mostly set the mean or standard deviation parameter in the noise model of the measure. This is useful to get more realistic measures. However, in **tutorial_ego.py** only one attribute is set.

- **sensor_tick.** As these measures are not supposed to vary significantly between steps, it is okay to retrieve the data every so often. In this case, it is set to be printed every three seconds.

```

1 # --
2 # Add GNSS sensor to ego vehicle.
3 # --
4
5 gnss_bp = world.get_blueprint_library().find('sensor.other.gnss')
6 gnss_location = carla.Location(0,0,0)
7 gnss_rotation = carla.Rotation(0,0,0)
8 gnss_transform = carla.Transform(gnss_location, gnss_rotation)
9 gnss_bp.set_attribute("sensor_tick",str(3.0))
10 ego_gnss = world.spawn_actor(gnss_bp, gnss_transform, attach_to=ego_vehicle,
   attachment_type=carla.AttachmentType.Rigid)
11 def gnss_callback(gnss):

```

```

12     print("GNSS measure:\n"+str(gnss)+"\n")
13 ego_gnss.listen(lambda gnss: gnss_callback(gnss))
14
15 # --
16 # Add IMU sensor to ego vehicle.
17 # --
18
19 imu_bp = world.get_blueprint_library().find('sensor.other.imu')
20 imu_location = carla.Location(0,0,0)
21 imu_rotation = carla.Rotation(0,0,0)
22 imu_transform = carla.Transform(imu_location,imu_rotation)
23 imu_bp.set_attribute("sensor_tick",str(3.0))
24 ego_imu = world.spawn_actor(imu_bp,imu_transform,attach_to=ego_vehicle,
    attachment_type=carla.AttachmentType.Rigid)
25 def imu_callback(imu):
26     print("IMU measure:\n"+str(imu)+"\n")
27 ego_imu.listen(lambda imu: imu_callback(imu))

```

```

GNSS measure:
GnssMeasurement(frame=523, timestamp=23.775793, lat=49.000134, lon=7.999751, alt=-0.004627)

IMU measure:
IMUMeasurement(frame=525, timestamp=23.873660, accelerometer=Vector3D(x=0.507325, y=-0.756483, z=9.800218), gyroscope=Vector3D(x=-0.001625, y=0.002050, z=-0.193408), compass=3.879089)

```

GNSS and IMU sensors output

Set advanced sensors

The script `tutorial_replay.py`, among other things, contains definitions for more sensors. They work in the same way as the basic ones, but their comprehension may be a bit harder.

Depth camera Thedepth camera generates pictures of the scene that map every pixel in a grayscale depth map. However, the output is not straightforward. The depth buffer of the camera is mapped using a RGB color space. This has to be translated to grayscale to be comprehensible.

In order to do this, simply save the image as with the RGB camera, but apply a `carla.ColorConverter` to it. There are two conversions available for depth cameras.

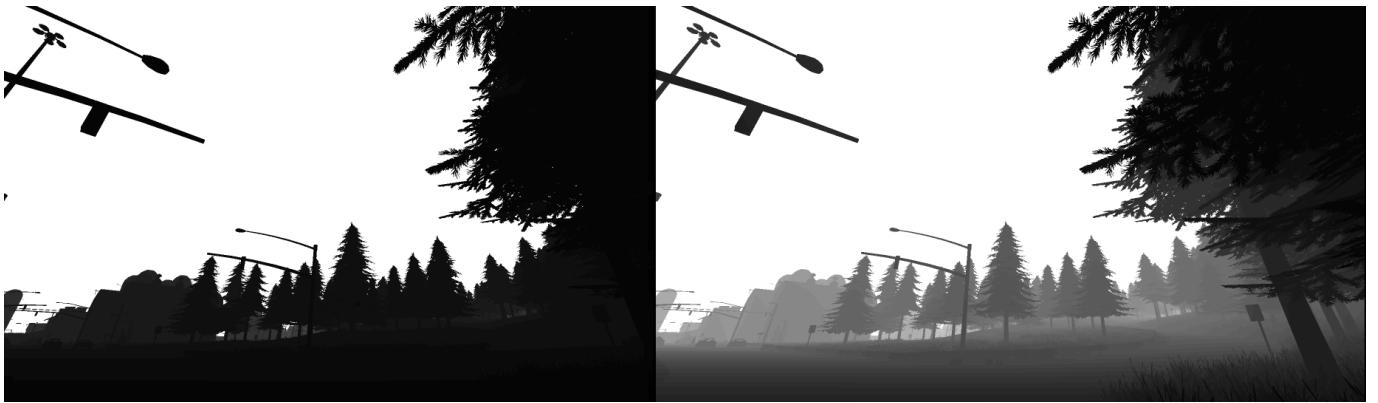
- `carla.ColorConverter.Depth` translates the original depth with milimetric precision.
- `carla.ColorConverter.LogarithmicDepth` also has milimetric granularity, but provides better results in close distances and a little worse for further elements.

The attributes for the depth camera only set elements previously stated in the RGB camera: `fov`, `image_size_x`, `image_size_y` and `sensor_tick`. The script sets this sensor to match the previous RGB camera used.

```

1 # --
2 # Add a Depth camera to ego vehicle.
3 # --
4 depth_cam = None
5 depth_bp = world.get_blueprint_library().find('sensor.camera.depth')
6 depth_location = carla.Location(2,0,1)
7 depth_rotation = carla.Rotation(0,180,0)
8 depth_transform = carla.Transform(depth_location,depth_rotation)
9 depth_cam = world.spawn_actor(depth_bp,depth_transform,attach_to=ego_vehicle,
    attachment_type=carla.AttachmentType.Rigid)
10 # This time, a color converter is applied to the image, to get the semantic segmentation view
11 depth_cam.listen(lambda image: image.save_to_disk('tutorial/new_depth_output/%.6d.jpg' %
    image.frame,carla.ColorConverter.LogarithmicDepth))

```



Depth camera output. Simple conversion on the left, logarithmic on the right.

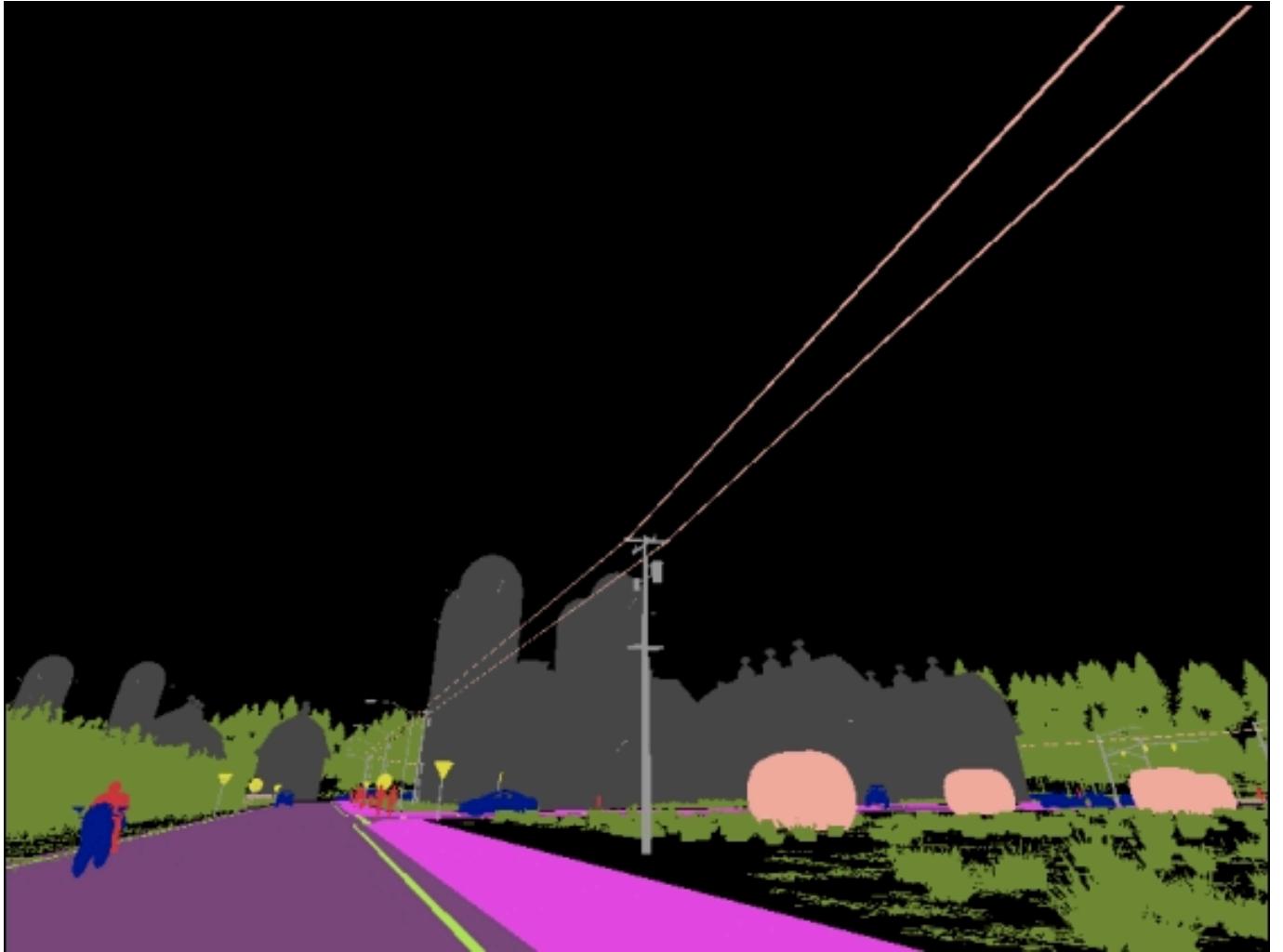
Semantic segmentation camera The semantic segmentation camera renders elements in scene with a different color depending on how these have been tagged. The tags are created by the simulator depending on the path of the asset used for spawning. For example, meshes tagged as `Pedestrians` are spawned with content stored in `Unreal/CarlaUE4/Content/Static/Pedestrians`.

The output is an image, as any camera, but each pixel contains the tag encoded in the red channel. This original image must be converted using `ColorConverter.CityScapesPalette`. New tags can be created, read more in the documentation.

The attributes available for this camera are exactly the same as the depth camera. The script also sets this to match the original RGB camera.

```

1 # --
2 # Add a new semantic segmentation camera to my ego
3 # --
4 sem_cam = None
5 sem_bp = world.get_blueprint_library().find('sensor.camera.semantic_segmentation')
6 sem_bp.set_attribute("image_size_x",str(1920))
7 sem_bp.set_attribute("image_size_y",str(1080))
8 sem_bp.set_attribute("fov",str(105))
9 sem_location = carla.Location(2,0,1)
10 sem_rotation = carla.Rotation(0,180,0)
11 sem_transform = carla.Transform(sem_location,sem_rotation)
12 sem_cam = world.spawn_actor(sem_bp,sem_transform,attach_to=ego_vehicle,
   attachment_type=carla.AttachmentType.Rigid)
13 # This time, a color converter is applied to the image, to get the semantic segmentation view
14 sem_cam.listen(lambda image: image.save_to_disk('tutorial/new_sem_output/%.6d.jpg' %
   image.frame,carla.ColorConverter.CityScapesPalette))
```



Semantic segmentation camera output

LIDAR raycast sensor The LIDAR sensor simulates a rotating LIDAR. It creates a cloud of points that maps the scene in 3D. The LIDAR contains a set of lasers that rotate at a certain frequency. The lasers raycast the distance to impact, and store every shot as one single point.

The way the array of lasers is disposed can be set using different sensor attributes.

- **upper_fov** and **lower_fov** the angle of the highest and the lowest laser respectively.
- **channels** sets the amount of lasers to be used. These are distributed along the desired *fov*.

Other attributes set the way this points are calculated. They determine the amount of points that each laser calculates every step: **points_per_second** / (**FPS** * **channels**).

- **range** is the maximum distance to capture.
- **points_per_second** is the amount of points that will be obtained every second. This quantity is divided between the amount of **channels**.
- **rotation_frequency** is the amount of times the LIDAR will rotate every second.

The point cloud output is described as a carla.LidarMeasurement. It can be iterated as a list of carla.Location or saved to a .ply standart file format.

```

1 # --
2 # Add a new LIDAR sensor to my ego
3 # --
4 lidar_cam = None
5 lidar_bp = world.get_blueprint_library().find('sensor.lidar.ray_cast')
6 lidar_bp.set_attribute('channels',str(32))
7 lidar_bp.set_attribute('points_per_second',str(90000))
8 lidar_bp.set_attribute('rotation_frequency',str(40))
9 lidar_bp.set_attribute('range',str(20))

```

```

10 lidar_location = carla.Location(0,0,2)
11 lidar_rotation = carla.Rotation(0,0,0)
12 lidar_transform = carla.Transform(lidar_location,lidar_rotation)
13 lidar_sen = world.spawn_actor(lidar_bp,lidar_transform,attach_to=ego_vehicle)
14 lidar_sen.listen(lambda point_cloud: point_cloud.save_to_disk('tutorial/new_lidar_output/%.6d.ply' %
    point_cloud.frame))

```

The `.ply` output can be visualized using **Meshlab**.

1. Install Meshlab.

```

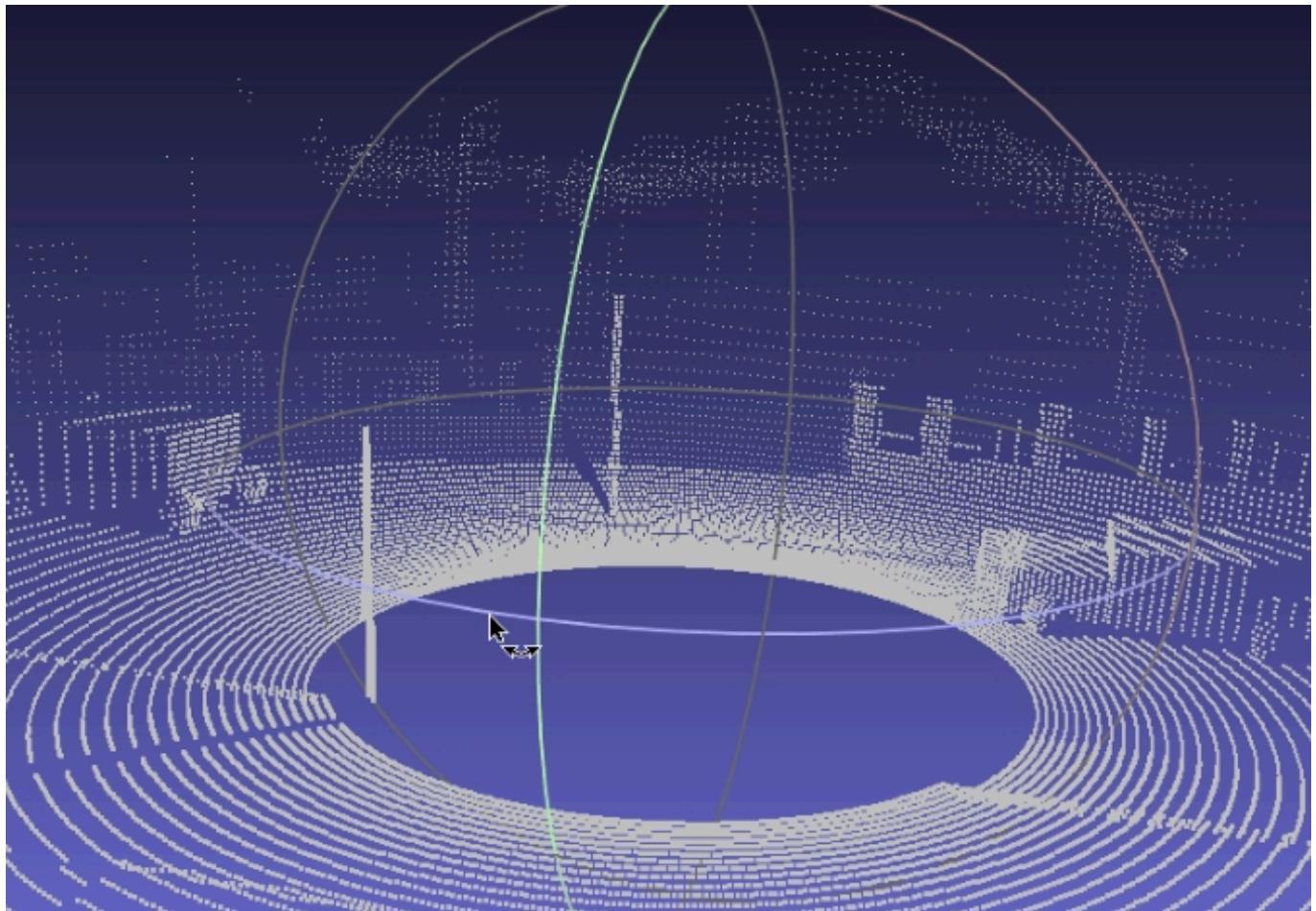
1 sudo apt-get update -y
2 sudo apt-get install -y meshlab

```

2. Open Meshlab.

```
1 meshlab
```

3. Open one of the `.ply` files. **File > Import mesh...**



LIDAR output after being processed in Meshlab.

Radar sensor Theradar sensor is similar to de LIDAR. It creates a conic view, and shoots lasers inside to raycast their impacts. The output is a `carla.RadarMeasurement`. It contains a list of the `carla.RadarDetection` retrieved by the lasers. These are not points in space, but detections with data regarding the sensor: `azimuth`, `altitude`, `sensor` and `velocity`.

The attributes of this sensor mostly set the way the lasers are located.

- `horizontal_fov` and `vertical_fov` determine the amplitude of the conic view.
- `channels` sets the amount of lasers to be used. These are distributed along the desired `fov`.
- `range` is the maximum distance for the lasers to raycast.

- **points_per_second** sets the the amount of points to be captured, that will be divided between the channels stated.

The script places the sensor on the hood of the car, and rotated a bit upwards. That way, the output will map the front view of the car. The **horizontal_fov** is incremented, and the **vertical_fov** diminished. The area of interest is specially the height were vehicles and walkers usually move on. The **range** is also changed from 100m to 10m, in order to retrieve data only right ahead of the vehicle.

The callback is a bit more complex this time, showing more of its capabilities. It will draw the points captured by the radar on the fly. The points will be colored depending on their velocity regarding the ego vehicle.

- **Blue** for points approaching the vehicle.
- **Red** for points moving away from it.
- **White** for points static regarding the ego vehicle.

```

1 # --
2 # Add a new radar sensor to my ego
3 # --
4 rad_cam = None
5 rad_bp = world.get_blueprint_library().find('sensor.other.radar')
6 rad_bp.set_attribute('horizontal_fov', str(35))
7 rad_bp.set_attribute('vertical_fov', str(20))
8 rad_bp.set_attribute('range', str(20))
9 rad_location = carla.Location(x=2.0, z=1.0)
10 rad_rotation = carla.Rotation(pitch=5)
11 rad_transform = carla.Transform(rad_location,rad_rotation)
12 rad_ego = world.spawn_actor(rad_bp,rad_transform,attach_to=ego_vehicle,
   attachment_type=carla.AttachmentType.Rigid)
13 def rad_callback(radar_data):
14     velocity_range = 7.5 # m/s
15     current_rot = radar_data.transform.rotation
16     for detect in radar_data:
17         azi = math.degrees(detect.azimuth)
18         alt = math.degrees(detect.altitude)
19         # The 0.25 adjusts a bit the distance so the dots can
20         # be properly seen
21         fw_vec = carla.Vector3D(x=detect.depth - 0.25)
22         carla.Transform(
23             carla.Location(),
24             carla.Rotation(
25                 pitch=current_rot.pitch + alt,
26                 yaw=current_rot.yaw + azi,
27                 roll=current_rot.roll)).transform(fw_vec)
28
29     def clamp(min_v, max_v, value):
30         return max(min_v, min(value, max_v))
31
32     norm_velocity = detect.velocity / velocity_range # range [-1, 1]
33     r = int(clamp(0.0, 1.0, 1.0 - norm_velocity) * 255.0)
34     g = int(clamp(0.0, 1.0, 1.0 - abs(norm_velocity)) * 255.0)
35     b = int(abs(clamp(- 1.0, 0.0, - 1.0 - norm_velocity)) * 255.0)
36     world.debug.draw_point(
37         radar_data.transform.location + fw_vec,
38         size=0.075,
39         life_time=0.06,
40         persistent_lines=False,
41         color=carla.Color(r, g, b))
42 rad_ego.listen(lambda radar_data: rad_callback(radar_data))

```



Radar output. The vehicle is stopped at a traffic light, so the static elements in front of it appear in white.

No-rendering mode

Theno-rendering mode can be useful to run an initial simulation that will be later played again to retrieve data. Especially if this simulation has some extreme conditions, such as dense traffic.

Simulate at a fast pace Disabling the rendering will save up a lot of work to the simulation. As the GPU is not used, the server can work at full speed. This could be useful to simulate complex conditions at a fast pace. The best way to do so would be by setting a fixed time-step. Running an asynchronous server with a fixed time-step and no rendering, the only limitation for the simulation would be the inner logic of the server.

The same `config.py` used to set the map can disable rendering, and set a fixed time-step.

```
1 cd /opt/carla/PythonAPI/utils  
2 python3 config.py --no-rendering --delta-seconds 0.05 # Never greater than 0.1s
```



| Read the documentation before messing around with synchrony and time-step.

Manual control without rendering The script `PythonAPI/examples/no_rendering_mode.py` provides an overview of the simulation. It creates a minimalistic aerial view with Pygame, that will follow the ego vehicle. This could be used along with `manual_control.py` to generate a route with barely no cost, record it, and then play it back and exploit it to gather data.

```
1 cd /opt/carla/PythonAPI/examples  
2 python3 manual_control.py
```

```
1 cd /opt/carla/PythonAPI/examples  
2 python3 no_rendering_mode.py --no-rendering
```

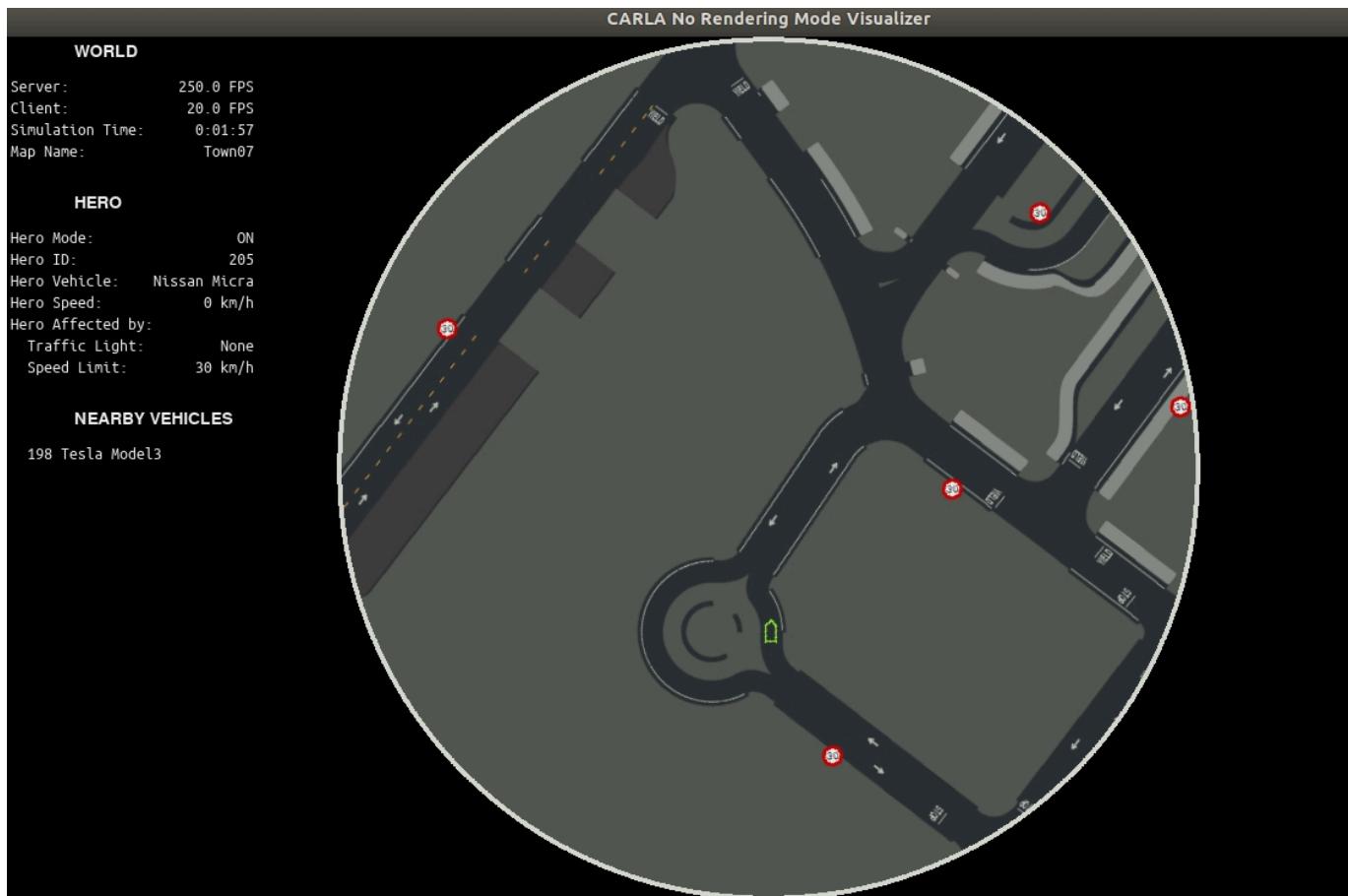
Optional arguments in `no_rendering_mode.py`

| | | |
|---|----------------------------|--------------------------------------------|
| 1 | <code>-h, --help</code> | show this help message and exit |
| 2 | <code>-v, --verbose</code> | print debug information |
| 3 | <code>--host H</code> | IP of the host server (default: 127.0.0.1) |

```

4 -p P, --port P      TCP port to listen to (default: 2000)
5 --res WIDTHxHEIGHT window resolution (default: 1280x720)
6 --filter PATTERN   actor filter (default: "vehicle.*")
7 --map TOWN          start a new episode at the given TOWN
8 --no-rendering     switch off server rendering
9 --show-triggers    show trigger boxes of traffic signs
10 --show-connections show waypoint connections
11 --show-spawn-points show recommended spawn points

```



no_rendering_mode.py working in Town07



In this mode, GPU-based sensors will retrieve empty data. Cameras are useless, but other sensors such as detectors will work properly.

Record and retrieve data

Start recording The `_recorder` can be started at anytime. The script does it at the very beginning, in order to capture everything, including the spawning of the first actors. If no path is detailed, the log will be saved into `CarlaUE4/Saved`.

```

1 # --
2 # Start recording
3 # --
4 client.start_recorder('~/tutorial/recorder/recording01.log')

```

Capture and record There are many different ways to do this. Mostly it goes down as either let it roam around or control it manually. The data for the sensors spawned will be retrieved on the fly. Make sure to check it while recording, to make sure everything is set properly.

- **Enable the autopilot.** This will register the vehicle to the Traffic Manager. It will roam around the city endlessly. The script does this, and creates a loop to prevent the script from finishing. The recording will go on

until the user finishes the script. Alternatively, a timer could be set to finish the script after a certain time.

```
1 # --
2 # Capture data
3 # --
4 ego_vehicle.set_autopilot(True)
5 print('\nEgo autopilot enabled')
6
7 while True:
8     world_snapshot = world.wait_for_tick()
```

- **Manual control.** Run the script `PythonAPI/examples/manual_control.py` in a client, and the recorder in another one. Drive the ego vehicle around to create the desired route, and stop the recorder when finished. The `tutorial_ego.py` script can be used to manage the recorder, but make sure to comment other fragments of code.

```
1 cd /opt/carla/PythonAPI/examples
2 python3 manual_control.py
```



To avoid rendering and save up computational cost, enable no rendering mode . The script `'/PythonAPI/examples/no rendering mode.py'` does this while creating a simple aerial view.

Stop recording The stop call is even simpler than the start call was. When the recorder is done, the recording will be saved in the path stated previously.

```
1 # --
2 # Stop recording
3 # --
4 client.stop_recorder()
```

Exploit the recording

So far, a simulation has been recorded. Now, it is time to examine the recording, find the most remarkable moments, and work with them. These steps are gathered in the script, `tutorial_replay.py`. The outline is structured in different segments of code commented.

It is time to run a new simulation.

```
1 ./CarlaUE4.sh
```

To reenact the simulation,choose a fragment and run the script containing the code for the playback.

```
1 python3 tuto_replay.py
```

Query the events The different queries are detailed in the [recorder documentation](#). In summary, they retrieve data for specific events or frames. Use the queries to study the recording. Find the spotlight moments, and trace what can be of interest.

```
1 # --
2 # Query the recording
3 # --
4 # Show only the most important events in the recording.
5 print(client.show_recorder_file_info("~/tutorial/recorder/recording01.log",False))
6 # Show actors not moving 1 meter in 10 seconds.
7 print(client.show_recorder_actors_blocked("~/tutorial/recorder/recording01.log",10,1))
8 # Filter collisions between vehicles 'v' and 'a' any other type of actor.
9 print(client.show_recorder_collisions("~/tutorial/recorder/recording01.log",'v','a'))
```



The recorder does not need to be on, in order to do the queries.

```

Frame 3 at 0.0960511 seconds
Create 412: vehicle.tesla.model3 (1) at (-1588.33, -6608.36, 225)
  number_of_wheels = 4
  sticky_control = true
  object_type =
  color = 180,180,180
  role_name = ego

```

Query showing important events. This is the frame where the ego vehicle was spawned.

```

Map: Town07
Date: 03/31/20 17:06:21

```

| Time | Id | Actor | Duration |
|------|-----|-------------------------------|----------|
| 202 | 412 | vehicle.tesla.model3 | 100 |
| 203 | 269 | vehicle.jeep.wrangler_rubicon | 99 |

Query showing actors blocked. In this simulation, the ego vehicle remained blocked for 100 seconds.

```

Map: Town07
Date: 03/31/20 17:06:21

```

| Time | Types | Id | Actor 1 | Id | Actor 2 |
|------|-------|-----|----------------------|----|---------|
| 67 | v o | 412 | vehicle.tesla.model3 | 0 | |

Query showing a collision between the ego vehicle and an object of type “other”.



Getting detailed file info for every frame can be overwhelming. Use it after other queries to know where to look at.

Choose a fragment After the queries, it may be a good idea play some moments of the simulation back, before messing around. It is very simple to do so, and it could be really helpful. Know more about the simulation. It is the best way to save time later.

The method allows to choose the beginning and ending point of the playback, and an actor to follow.

```

1 # --
2 # Reenact a fragment of the recording
3 # --
4 client.replay_file("~/tutorial/recorder/recording01.log",45,10,0)

```

Here is a list of possible things to do now.

- **Use the information from the queries.** Find out the moment and the actors involved in an event, and play that again. Start the recorder a few seconds before the event.
- **Follow different actors.** Different perspectives will show new events that are not included in the queries.
- **Rom around with a free spectator view.** Set the `actor_id` to 0, and get a general view of the simulation. Be wherever and whenever wanted thanks to the recording.



When the recording stops, the simulation doesn't. Walkers will stand still, and vehicles will continue roaming around. This may happen either if the log ends, or the playback gets to the ending point stated.

Retrieve more data The recorder will recreate in this simulation, the exact same conditions as the original. That ensures consistent data within different playbacks.

Gather a list of the important moments, actors and events. Add sensors whenever needed and play the simulation back. The process is exactly the same as before. The script **tutorial_replay.py** provides different examples that have been thoroughly explained in the [Set advanced sensors](#) section. Others have been explained in the section [Set basic sensors](#).

Add as many sensors as needed, wherever they are needed. Play the simulation back as many times as desired and retrieve as much data as desired.

Change the weather The recording will recreate the original weather conditions. However, these can be altered at will. This may be interesting to compare how does it affect sensors, while maintaining the rest of events the same.

Get the current weather and modify it freely. Remember that `carla.WeatherParameters` has some presets available. The script will change the environment to a foggy sunset.

```
1 # --
2 # Change weather for playback
3 #
4 weather = world.get_weather()
5 weather.sun_altitude_angle = -30
6 weather.fog_density = 65
7 weather.fog_distance = 10
8 world.set_weather(weather)
```

Try new outcomes The new simulation is not strictly linked to the recording. It can be modified anytime, and even when the recorder stops, the simulation goes on.

This can be profitable for the user. For instance, collisions can be forced or avoided by playing back the simulation a few seconds before, and spawning or destroying an actor. Ending the recording at a specific moment can also be useful. Doing so, vehicles may take different paths.

Change the conditions and mess with the simulation. There is nothing to lose, as the recorder grants that the initial simulation can always be reenacted. This is the key to exploit the full potential of CARLA.

Tutorial scripts

Hereunder are the two scripts gathering the fragments of code for this tutorial. Most of the code is commented, as it is meant to be modified to fit specific purposes.

`tutorial_ego.py`

```
1
2 import glob
3 import os
4 import sys
5 import time
6
7 try:
8     sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
9         sys.version_info.major,
10        sys.version_info.minor,
11        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
12 except IndexError:
13     pass
14
15 import carla
16
17 import argparse
18 import logging
19 import random
20
21
22 def main():
23     argparser = argparse.ArgumentParser(
24         description=__doc__)
```

```

25     argparser.add_argument(
26         '--host',
27         metavar='H',
28         default='127.0.0.1',
29         help='IP of the host server (default: 127.0.0.1)')
30     argparser.add_argument(
31         '-p', '--port',
32         metavar='P',
33         default=2000,
34         type=int,
35         help='TCP port to listen to (default: 2000)')
36     args = argparser.parse_args()
37
38     logging.basicConfig(format='%(levelname)s: %(message)s', level=logging.INFO)
39
40     client = carla.Client(args.host, args.port)
41     client.set_timeout(10.0)
42
43     try:
44
45         world = client.get_world()
46         ego_vehicle = None
47         ego_cam = None
48         ego_col = None
49         ego_lane = None
50         ego_obs = None
51         ego_gnss = None
52         ego_imu = None
53
54         # --
55         # Start recording
56         # --
57         """
58         client.start_recorder('~/tutorial/recorder/recording01.log')
59         """
60
61         # --
62         # Spawn ego vehicle
63         # --
64         """
65         ego_bp = world.get_blueprint_library().find('vehicle.tesla.model3')
66         ego_bp.set_attribute('role_name', 'ego')
67         print('\nEgo role_name is set')
68         ego_color = random.choice(ego_bp.get_attribute('color').recommended_values)
69         ego_bp.set_attribute('color', ego_color)
70         print('\nEgo color is set')
71
72         spawn_points = world.get_map().get_spawn_points()
73         number_of_spawn_points = len(spawn_points)
74
75         if 0 < number_of_spawn_points:
76             random.shuffle(spawn_points)
77             ego_transform = spawn_points[0]
78             ego_vehicle = world.spawn_actor(ego_bp, ego_transform)
79             print('\nEgo is spawned')
80         else:
81             logging.warning('Could not found any spawn points')
82         """
83
84         # --
85         # Add a RGB camera sensor to ego vehicle.

```

```

86     # --
87     """
88     cam_bp = None
89     cam_bp = world.get_blueprint_library().find('sensor.camera.rgb')
90     cam_bp.set_attribute("image_size_x",str(1920))
91     cam_bp.set_attribute("image_size_y",str(1080))
92     cam_bp.set_attribute("fov",str(105))
93     cam_location = carla.Location(2,0,1)
94     cam_rotation = carla.Rotation(0,180,0)
95     cam_transform = carla.Transform(cam_location,cam_rotation)
96     ego_cam = world.spawn_actor(cam_bp,cam_transform,attach_to=ego_vehicle,
97         attachment_type=carla.AttachmentType.Rigid)
98     ego_cam.listen(lambda image: image.save_to_disk('~/tutorial/output/%.6d.jpg' % image.frame))
99     """
100
101    # --
102    # Add collision sensor to ego vehicle.
103    """
104    col_bp = world.get_blueprint_library().find('sensor.other.collision')
105    col_location = carla.Location(0,0,0)
106    col_rotation = carla.Rotation(0,0,0)
107    col_transform = carla.Transform(col_location,col_rotation)
108    ego_col = world.spawn_actor(col_bp,col_transform,attach_to=ego_vehicle,
109        attachment_type=carla.AttachmentType.Rigid)
110    def col_callback(colli):
111        print("Collision detected:\n"+str(colli)+"\n")
112    ego_col.listen(lambda colli: col_callback(colli))
113    """
114
115    # --
116    # Add Lane invasion sensor to ego vehicle.
117    """
118    lane_bp = world.get_blueprint_library().find('sensor.other.lane_invasion')
119    lane_location = carla.Location(0,0,0)
120    lane_rotation = carla.Rotation(0,0,0)
121    lane_transform = carla.Transform(lane_location,lane_rotation)
122    ego_lane = world.spawn_actor(lane_bp,lane_transform,attach_to=ego_vehicle,
123        attachment_type=carla.AttachmentType.Rigid)
124    def lane_callback(lane):
125        print("Lane invasion detected:\n"+str(lane)+"\n")
126    ego_lane.listen(lambda lane: lane_callback(lane))
127    """
128
129    # --
130    # Add Obstacle sensor to ego vehicle.
131    """
132    obs_bp = world.get_blueprint_library().find('sensor.other.obstacle')
133    obs_bp.set_attribute("only_dynamics",str(True))
134    obs_location = carla.Location(0,0,0)
135    obs_rotation = carla.Rotation(0,0,0)
136    obs_transform = carla.Transform(obs_location,obs_rotation)
137    ego_obs = world.spawn_actor(obs_bp,obs_transform,attach_to=ego_vehicle,
138        attachment_type=carla.AttachmentType.Rigid)
139    def obs_callback(obs):
140        print("Obstacle detected:\n"+str(obs)+"\n")
141    ego_obs.listen(lambda obs: obs_callback(obs))
142    """

```

```

143     # --
144     # Add GNSS sensor to ego vehicle.
145     # --
146     """
147     gnss_bp = world.get_blueprint_library().find('sensor.other.gnss')
148     gnss_location = carla.Location(0,0,0)
149     gnss_rotation = carla.Rotation(0,0,0)
150     gnss_transform = carla.Transform(gnss_location,gnss_rotation)
151     gnss_bp.set_attribute("sensor_tick",str(3.0))
152     ego_gnss = world.spawn_actor(gnss_bp,gnss_transform,attach_to=ego_vehicle,
153         attachment_type=carla.AttachmentType.Rigid)
153     def gnss_callback(gnss):
154         print("GNSS measure:\n"+str(gnss)+"\n")
155         ego_gnss.listen(lambda gnss: gnss_callback(gnss))
156     """
157
158     # --
159     # Add IMU sensor to ego vehicle.
160     # --
161     """
162     imu_bp = world.get_blueprint_library().find('sensor.other.imu')
163     imu_location = carla.Location(0,0,0)
164     imu_rotation = carla.Rotation(0,0,0)
165     imu_transform = carla.Transform(imu_location,imu_rotation)
166     imu_bp.set_attribute("sensor_tick",str(3.0))
167     ego_imu = world.spawn_actor(imu_bp,imu_transform,attach_to=ego_vehicle,
168         attachment_type=carla.AttachmentType.Rigid)
168     def imu_callback(imu):
169         print("IMU measure:\n"+str(imu)+"\n")
170         ego_imu.listen(lambda imu: imu_callback(imu))
171     """
172
173     # --
174     # Place spectator on ego spawning
175     # --
176     """
177     spectator = world.get_spectator()
178     world_snapshot = world.wait_for_tick()
179     spectator.set_transform(ego_vehicle.get_transform())
180     """
181
182     # --
183     # Enable autopilot for ego vehicle
184     # --
185     """
186     ego_vehicle.set_autopilot(True)
187     """
188
189     # --
190     # Game loop. Prevents the script from finishing.
191     # --
192     while True:
193         world_snapshot = world.wait_for_tick()
194
195     finally:
196         # --
197         # Stop recording and destroy actors
198         # --
199         client.stop_recorder()
200         if ego_vehicle is not None:
201             if ego_cam is not None:

```

```

202         ego_cam.stop()
203         ego_cam.destroy()
204     if ego_col is not None:
205         ego_col.stop()
206         ego_col.destroy()
207     if ego_lane is not None:
208         ego_lane.stop()
209         ego_lane.destroy()
210     if ego_obs is not None:
211         ego_obs.stop()
212         ego_obs.destroy()
213     if ego_gnss is not None:
214         ego_gnss.stop()
215         ego_gnss.destroy()
216     if ego_imu is not None:
217         ego_imu.stop()
218         ego_imu.destroy()
219     ego_vehicle.destroy()
220
221 if __name__ == '__main__':
222
223     try:
224         main()
225     except KeyboardInterrupt:
226         pass
227     finally:
228         print('\nDone with tutorial_ego.')

```

tutorial_replay.py

```

1
2 import glob
3 import os
4 import sys
5 import time
6 import math
7 import weakref
8
9 try:
10     sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
11         sys.version_info.major,
12         sys.version_info.minor,
13         'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
14 except IndexError:
15     pass
16
17 import carla
18
19 import argparse
20 import logging
21 import random
22
23 def main():
24     client = carla.Client('127.0.0.1', 2000)
25     client.set_timeout(10.0)
26
27     try:
28
29         world = client.get_world()
30         ego_vehicle = None
31         ego_cam = None

```

```

32     depth_cam = None
33     depth_cam02 = None
34     sem_cam = None
35     rad_ego = None
36     lidar_sen = None
37
38     # --
39     # Query the recording
40     # --
41     """
42     # Show the most important events in the recording.
43     print(client.show_recorder_file_info("~/tutorial/recorder/recording05.log",False))
44     # Show actors not moving 1 meter in 10 seconds.
45     #print(client.show_recorder_actors_blocked("~/tutorial/recorder/recording04.log",10,1))
46     # Show collisions between any type of actor.
47     #print(client.show_recorder_collisions("~/tutorial/recorder/recording04.log",'v','a'))
48     """
49
50     # --
51     # Reenact a fragment of the recording
52     # --
53     """
54     client.replay_file("~/tutorial/recorder/recording03.log",0,30,0)
55     """
56
57     # --
58     # Set playback simulation conditions
59     # --
60     """
61     ego_vehicle = world.get_actor(322) #Store the ID from the simulation or query the recording
       to find out
62     """
63
64     # --
65     # Place spectator on ego spawning
66     # --
67     """
68     spectator = world.get_spectator()
69     world_snapshot = world.wait_for_tick()
70     spectator.set_transform(ego_vehicle.get_transform())
71     """
72
73     # --
74     # Change weather conditions
75     # --
76     """
77     weather = world.get_weather()
78     weather.sun_altitude_angle = -30
79     weather.fog_density = 65
80     weather.fog_distance = 10
81     world.set_weather(weather)
82     """
83
84     # --
85     # Add a RGB camera to ego vehicle.
86     # --
87     """
88     cam_bp = None
89     cam_bp = world.get_blueprint_library().find('sensor.camera.rgb')
90     cam_location = carla.Location(2,0,1)
91     cam_rotation = carla.Rotation(0,180,0)

```

```

92     cam_transform = carla.Transform(cam_location,cam_rotation)
93     cam_bp.set_attribute("image_size_x",str(1920))
94     cam_bp.set_attribute("image_size_y",str(1080))
95     cam_bp.set_attribute("fov",str(105))
96     ego_cam = world.spawn_actor(cam_bp,cam_transform,attach_to=ego_vehicle,
97         attachment_type=carla.AttachmentType.Rigid)
98     ego_cam.listen(lambda image: image.save_to_disk('~/tutorial/new_rgb_output/%.6d.jpg' %
99             image.frame))
100    """
101
102    # --
103    # Add a Logarithmic Depth camera to ego vehicle.
104    # --
105    """
106    depth_cam = None
107    depth_bp = world.get_blueprint_library().find('sensor.camera.depth')
108    depth_bp.set_attribute("image_size_x",str(1920))
109    depth_bp.set_attribute("image_size_y",str(1080))
110    depth_bp.set_attribute("fov",str(105))
111    depth_location = carla.Location(2,0,1)
112    depth_rotation = carla.Rotation(0,180,0)
113    depth_transform = carla.Transform(depth_location,depth_rotation)
114    depth_cam = world.spawn_actor(depth_bp,depth_transform,attach_to=ego_vehicle,
115        attachment_type=carla.AttachmentType.Rigid)
116    # This time, a color converter is applied to the image, to get the semantic segmentation view
117    depth_cam.listen(lambda image: image.save_to_disk('~/tutorial/de_log/%.6d.jpg' %
118                    image.frame,carla.ColorConverter.LogarithmicDepth))
119    """
120    # --
121    # Add a Depth camera to ego vehicle.
122    # --
123    """
124    depth_cam02 = None
125    depth_bp02 = world.get_blueprint_library().find('sensor.camera.depth')
126    depth_bp02.set_attribute("image_size_x",str(1920))
127    depth_bp02.set_attribute("image_size_y",str(1080))
128    depth_bp02.set_attribute("fov",str(105))
129    depth_location02 = carla.Location(2,0,1)
130    depth_rotation02 = carla.Rotation(0,180,0)
131    depth_transform02 = carla.Transform(depth_location02,depth_rotation02)
132    depth_cam02 = world.spawn_actor(depth_bp02,depth_transform02,attach_to=ego_vehicle,
133        attachment_type=carla.AttachmentType.Rigid)
134    # This time, a color converter is applied to the image, to get the semantic segmentation view
135    depth_cam02.listen(lambda image: image.save_to_disk('~/tutorial/de/%.6d.jpg' %
136                    image.frame,carla.ColorConverter.Depth))
137    """
138    # --
139    # Add a new semantic segmentation camera to ego vehicle
140    # --
141    """
142    sem_cam = None
143    sem_bp = world.get_blueprint_library().find('sensor.camera.semantic_segmentation')
144    sem_bp.set_attribute("image_size_x",str(1920))
145    sem_bp.set_attribute("image_size_y",str(1080))
146    sem_bp.set_attribute("fov",str(105))
147    sem_location = carla.Location(2,0,1)
148    sem_rotation = carla.Rotation(0,180,0)
149    sem_transform = carla.Transform(sem_location,sem_rotation)
150    sem_cam = world.spawn_actor(sem_bp,sem_transform,attach_to=ego_vehicle,
151        attachment_type=carla.AttachmentType.Rigid)

```

```

146 # This time, a color converter is applied to the image, to get the semantic segmentation view
147 sem_cam.listen(lambda image: image.save_to_disk('~/tutorial/new_sem_output/%.6d.jpg' %
148     image.frame,carla.ColorConverter.CityScapesPalette))
149 """
150 #
151 # Add a new radar sensor to ego vehicle
152 #
153 """
154 rad_cam = None
155 rad_bp = world.get_blueprint_library().find('sensor.other.radar')
156 rad_bp.set_attribute('horizontal_fov', str(35))
157 rad_bp.set_attribute('vertical_fov', str(20))
158 rad_bp.set_attribute('range', str(20))
159 rad_location = carla.Location(x=2.8, z=1.0)
160 rad_rotation = carla.Rotation(pitch=5)
161 rad_transform = carla.Transform(rad_location,rad_rotation)
162 rad_ego = world.spawn_actor(rad_bp,rad_transform,attach_to=ego_vehicle,
163     attachment_type=carla.AttachmentType.Rigid)
164 def rad_callback(radar_data):
165     velocity_range = 7.5 # m/s
166     current_rot = radar_data.transform.rotation
167     for detect in radar_data:
168         azi = math.degrees(detect.azimuth)
169         alt = math.degrees(detect.altitude)
170         # The 0.25 adjusts a bit the distance so the dots can
171         # be properly seen
172         fw_vec = carla.Vector3D(x=detect.depth - 0.25)
173         carla.Transform(
174             carla.Location(),
175             carla.Rotation(
176                 pitch=current_rot.pitch + alt,
177                 yaw=current_rot.yaw + azi,
178                 roll=current_rot.roll)).transform(fw_vec)
179
180     def clamp(min_v, max_v, value):
181         return max(min_v, min(value, max_v))
182
183     norm_velocity = detect.velocity / velocity_range # range [-1, 1]
184     r = int(clamp(0.0, 1.0, 1.0 - norm_velocity) * 255.0)
185     g = int(clamp(0.0, 1.0, 1.0 - abs(norm_velocity)) * 255.0)
186     b = int(abs(clamp(- 1.0, 0.0, - 1.0 - norm_velocity)) * 255.0)
187     world.debug.draw_point(
188         radar_data.transform.location + fw_vec,
189         size=0.075,
190         life_time=0.06,
191         persistent_lines=False,
192         color=carla.Color(r, g, b))
193     rad_ego.listen(lambda radar_data: rad_callback(radar_data))
194 """
195 #
196 # Add a new LIDAR sensor to ego vehicle
197 #
198 """
199 lidar_cam = None
200 lidar_bp = world.get_blueprint_library().find('sensor.lidar.ray_cast')
201 lidar_bp.set_attribute('channels',str(32))
202 lidar_bp.set_attribute('points_per_second',str(90000))
203 lidar_bp.set_attribute('rotation_frequency',str(40))
204 lidar_bp.set_attribute('range',str(20))

```

```

205     lidar_location = carla.Location(0,0,2)
206     lidar_rotation = carla.Rotation(0,0,0)
207     lidar_transform = carla.Transform(lidar_location,lidar_rotation)
208     lidar_sen =
209         world.spawn_actor(lidar_bp,lidar_transform,attach_to=ego_vehicle,attachment_type=carla.AttachmentTy
210         lidar_sen.listen(lambda point_cloud:
211             point_cloud.save_to_disk('/home/adas/Desktop/tutorial/new_lidar_output/%.6d.ply' %
212             point_cloud.frame))
213             """
214             # --
215             # Game loop. Prevents the script from finishing.
216             # --
217             while True:
218                 world_snapshot = world.wait_for_tick()
219
220             finally:
221                 # --
222                 # Destroy actors
223                 # --
224                 if ego_vehicle is not None:
225                     if ego_cam is not None:
226                         ego_cam.stop()
227                         ego_cam.destroy()
228                     if depth_cam is not None:
229                         depth_cam.stop()
230                         depth_cam.destroy()
231                     if sem_cam is not None:
232                         sem_cam.stop()
233                         sem_cam.destroy()
234                     if rad_ego is not None:
235                         rad_ego.stop()
236                         rad_ego.destroy()
237                     if lidar_sen is not None:
238                         lidar_sen.stop()
239                         lidar_sen.destroy()
240                         ego_vehicle.destroy()
241                         print('\nNothing to be done.')
242
243 if __name__ == '__main__':
244     try:
245         main()
246     except KeyboardInterrupt:
247         pass
248     finally:
249         print('\nDone with tutorial_replay.')

```

That is a wrap on how to properly retrieve data from the simulation. Make sure to play around, change the conditions of the simulator, experiment with sensor settings. The possibilities are endless.

Visit the forum to post any doubts or suggestions that have come to mind during this reading.

9 Tutorials (assets)

9.1 Add a new map

Users can create their own maps, and run CARLA using these. The creation of the map object is quite independent from CARLA. Nonetheless, the process to ingest it has been refined to be automatic. Thus, the new map can be used in CARLA almost out-of-the-box.

- **Introduction**
- **Create a map with RoadRunner**
 - Export from RoadRunner
- **Map ingestion in a CARLA package**
- **Map ingestion in a build from source**
 - Modify pedestrian navigation
- **Deprecated ways to import a map**

Introduction

RoadRunner is the recommended software to create a map due to its simplicity. Some basic steps on how to do it are provided in the next section. The resulting map should consist of a `.fbx` and a `.xodr` with the mesh and road network information respectively.

The process of the map ingestion has been simplified to minimize the users' intervention. For said reason, there are certain steps have been automatized.

- **Package .json file and folder structure.** Normally packages need a certain folder structure and a `.json` file describing them to be imported. However, as regards the map ingestion, this can be created automatically during the process.
- **Traffic signs and traffic lights.** The simulator will generate the traffic lights, stops, and yields automatically when running. These will be created according to their `.xodr` definition. The rest of landmarks present in the road map will not be physically on scene, but they can be queried using the API.
- **Pedestrian navigation.** The ingestion will generate a `.bin` file describing the pedestrian navigation. It is based on the sidewalks and crosswalks that appear in the OpenDRIVE map. This can only be modified if working in a build from source.



If a map contains additional elements besides the ‘`fbx`’ and ‘`xodr`’, the package has to be prepared manually.

The map ingestion process differs, depending if the package is destined to be in a CARLA package (e.g., 0.9.9) or a build from source.

There are other ways to import a map into CARLA, which are now deprecated. They require the user to manually set the map ready. Nonetheless, as they may be useful for specific cases when the user wants to customize a specific setting, they are listed in the last section of this tutorial.

Create a map with RoadRunner

RoadRunner is an accessible and powerful software from Vector Zero to create 3D scenes. There is a trial version available at their site, and an installation guide.

The process is quite straightforward, but there are some things to take into account.

- **Center the map in (0,0).**
- **Create the map definition.** Take a look at the official tutorials.
- **Check the map validation.** Take a close look at all connections and geometries.

Once the map is ready, click on the `OpenDRIVE Preview Tool` button to visualize the OpenDRIVE road network. Give one last check to everything. Once the map is exported, it cannot be modified.



OpenDrive Preview Tool makes it easier to test the integrity of the map. If there is any error with junctions, click on ‘Maneuver Tool’, and ‘Rebuild Maneuver Roads’.

Export from RoadRunner **1. Export the scene using the CARLA option.** `File/Export/CARLA(.fbx+.xml+.xodr)`

2. Leave Export individual Tiles unchecked. This will generate only one `.fbx` with all the pieces. It makes easier to keep track of the map.

3. Click Export.

This will generate a `mapname.fbx` and `mapname.xodr` files within others. There is more detailed information about how to export to CARLA in VectorZero's documentation.

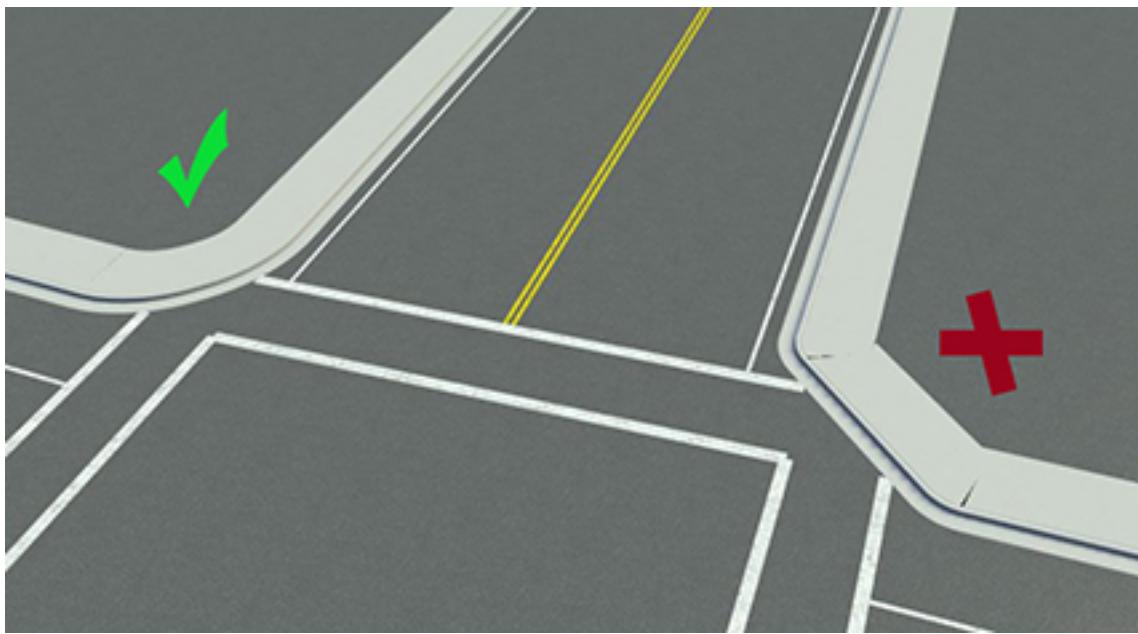


Figure 51: CheckGeometry



Figure 52: checkopen



Make sure that the .xodr and the .fbx files have the same name.

Map ingestion in a CARLA package

This is the recommended method to import a map into a CARLA package. It will run a Docker image of Unreal Engine to import the files, and export them as a standalone package. The Docker image takes 4h and 400GB to be built. However, this is only needed the first time.

1. **Build a Docker image of Unreal Engine.** Follow these instructions to build the image.
2. **Change permissions on the input folder.** If no .json file is provided, the Docker will try to create it on the input folder. To be successful, said folder must have all permissions enabled for others.

```
1 #Go to the parent folder, where the input folder is contained
2 chmod 777 input_folder
```



This is not necessary if the package is prepared manually, and contains a ‘.json’ file.

2. **Run the script to cook the map.** In the folder ~/carla/Util/Docker there is a script that connects with the Docker image previously created, and makes the ingestion automatically. It only needs the path for the input and output files, and the name of the package to be ingested. If no .json is provided, the name must be `map_package`.

```
1 python3 docker_tools.py --input ~/path_to_input_folder --output ~/path_to_output_folder --packages
  map_package
```



If the argument ‘–package <package name>’ is not provided, the Docker will make a package of CARLA.

3. **Locate the package.** The Docker should have generated the package `map_package.tar.gz` in the output path. This is the standalone package for the assets.

4. Import the package into CARLA.

- **On Windows** extract the package in the `WindowsNoEditor` folder.
- **On Linux** move the package to the `Import` folder, and run the script to import it.

```
1 cd Util
2 ./ImportAssets.sh
```

5. **Change the name of the package folder.** Two packages cannot have the same name in CARLA. Go to `Content` and find the package. Change the name if necessary, to use one that identifies it.

Map ingestion in a build from source

This method is meant to be used if working with the source version of CARLA. Place the maps to be imported in the `Import` folder. The script will make the ingestion, but the pedestrian navigation will have to be generated after that. Make sure that the name of the .xodr and .fbx files are the same for each of the maps being imported. Otherwise, the script will not recognize them as a map.

There are two parameters to be set.

- **Name of the package.** By default, the script ingest the map or maps in a package named `map_package`. This could lead to error the second time an ingestion is made, as two packages cannot have the same name. **It is highly recommended to change the name of the package.**

```
1 ARGS="--package package_name"
```

- **Usage of CARLA materials.** By default, the maps imported will use CARLA materials, but this can be changed using a flag.

```
1 ARGS="--no-carla-materials"
```

Check that there is an `.fbx` and a `.xodr` for each map in the `Import` folder, and make the ingestion.

```
1 make import ARGS="--package package_name --no-carla-materials"
```

After the ingestion, only the pedestrian navigation is yet to be generated. However there is an optional step that can be done before that.

- **Create new spawning points.** Place them over the road, around 0.5/1m so the wheels do not collide with the ground. These will be used in scripts such as `spawn_npc.py`.

Generate pedestrian navigation The pedestrian navigation is managed using a `.bin`. However, before generating it, there are two things to be done.

- **Add crosswalk meshes.** Crosswalks defined inside the `.xodr` remain in the logic of the map, but are not visible. For each of them, create a plane mesh that extends a bit over both sidewalks connected. **Place it overlapping the ground, and disable its physics and rendering.**



To generate new crosswalks, change the name of the mesh to ‘Road Crosswalk’. Avoid doing so if the crosswalk is in the ‘`xodr`’. Otherwise, it will be duplicated.



Figure 53: ue_crosswalks

- **Customize the map.** It is common to modify the map after the ingestion. Props such as trees, streetlights or grass zones are added, probably interfering with the pedestrian navigation. Make sure to have the desired result before generating the pedestrian navigation. Otherwise, it will have to be generated again.

Now that the version of the map is final, it is time to generate the pedestrian navigation file.

1. Select the **Skybox object** and add a tag `NoExport` to it. Otherwise, the map will not be exported, as the size would be too big.
2. Check the name of the meshes. By default, pedestrians will be able to walk over sidewalks, crosswalks, and grass (with minor influence over the rest).

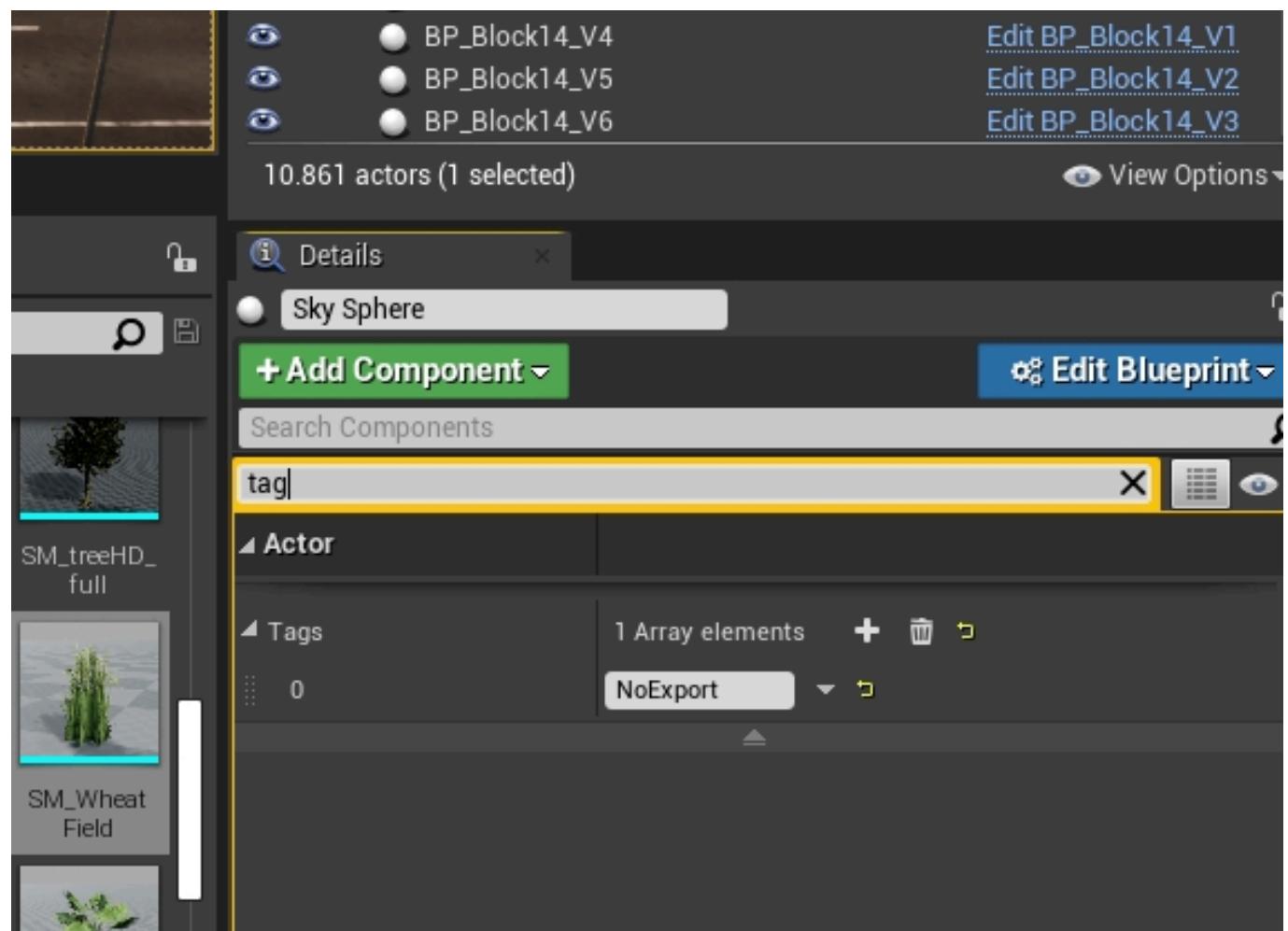


Figure 54: ue_skybox_no_export

- Sidewalk = Road_Sidewalk.
 - Crosswalk = Road_Crosswalk.
 - Grass = Road_Grass.
3. Name these planes following the common format Road_Crosswalk_mapname.
 4. Press G to deselect everything, and export the map. File > Export CARLA.... A `map_file.obj` file will be created in Unreal/CarlaUE4/Saved.
 5. Move the `map_file.obj` and the `map_file.xodr` to `Util/DockerUtils/dist`.
 6. Run the following command to generate the navigation file.
 - Windows

```
1 build.bat map_file # map_file has no extension
```

 - Linux

```
1 ./build.sh map_file # map_file has no extension
```
 7. Move the `.bin` into the `Nav` folder of the package that contains the map.

Deprecated ways to import a map

There are other ways to import a map used in previous CARLA releases. These required to manually cook the map and prepare everything, so they are now deprecated. However, they are explained below in case they are needed.

Prepare the package manually A package needs to follow a certain folder structure and contain a `.json` file describing it. This steps can be saved under certains circumstances, but doing it manually will always work.

Read how to prepare the folder structure and `.json` file

Create the folder structure **1. Create a folder inside carla/Import.** The name of the folder is not relevant.

2. Create different subfolders for each map to import.

3. Move the files of each map to the corresponding subfolder. A subfolder will contain a specific set of elements.

- The mesh of the map in a `.fbx`.
- The OpenDRIVE definition in a `.xodr`.
- Optionally, the textures required by the asset.

For instance, an `Import` folder with one package containing two maps should have a structure similar to the one below.

```
1 Import
2
3 Package01
4   Package01.json
5   Map01
6     Asphalt1_Diff.jpg
7     Asphalt1_Norm.jpg
8     Asphalt1_Spec.jpg
9     Grass1_Diff.jpg
10    Grass1_Norm.jpg
11    Grass1_Spec.jpg
12    LaneMarking1_Diff.jpg
13    LaneMarking1_Norm.jpg
14    LaneMarking1_Spec.jpg
15    Map01.fbx
16    Map01.xodr
17  Map02
18    Map02.fbx
```

World Outliner

| Label | Type |
|----------------------------|-----------------------------------|
| Road_Sidewalk_Town10HD43 | StaticMeshActor |
| Road_Sidewalk_Town10HD44 | StaticMeshActor |
| Road_Sidewalk_Town10HD45 | StaticMeshActor |
| Road_Sidewalk_Town10HD46 | StaticMeshActor |
| Road_Sidewalk_Town10HD47 | StaticMeshActor |
| Road_Sidewalk_Town10HD48 | StaticMeshActor |
| Road_Sidewalk_Town10HD49 | StaticMeshActor |
| Road_Sidewalk_Town10HD50 | StaticMeshActor |
| Road_Sidewalk_Town10HD51 | StaticMeshActor |
| Road_Sidewalk_Town10HD52 | StaticMeshActor |
| Road_Sidewalk_Town10HD53 | StaticMeshActor |
| Road_Sidewalk_Town10HD54 | StaticMeshActor |
| Road_Sidewalk_Town10HD55 | StaticMeshActor |
| Road_Sidewalk_Town10HD56 | StaticMeshActor |
| Road_Sidewalk_Town10HD57 | StaticMeshActor |
| Roa | ID Name: Road_Sidewalk_Town10HD57 |
| Roa | StaticMeshActor |
| Roa | StaticMeshActor |
| Road_Sidewalk_Town10HD60 | StaticMeshActor |
| Road_Sidewalk_Town10HD61 | StaticMeshActor |
| Road_Sidewalk_Town10HD62 | StaticMeshActor |
| Road_Sidewalk_Town10HD63 | StaticMeshActor |
| Road_Sidewalk_Town10HD64 | StaticMeshActor |
| SM_Road_SideWalk_Town10HD | StaticMeshActor |
| SM_Road_SideWalk_Town10HD2 | StaticMeshActor |
| SM_Road_SideWalk_Town10HD3 | StaticMeshActor |
| SM_Road_SideWalk_Town10HD4 | StaticMeshActor |
| SM_Road_SideWalk_Town10HD5 | StaticMeshActor |
| SM_Road_SideWalk_Town10HD6 | StaticMeshActor |
| SM_Road_SideWalk_Town10HD7 | StaticMeshActor |
| SM_Road_SideWalk_Town10HD8 | StaticMeshActor |

10.861 actors (1 selected) View Options ▾

Figure 55: ue_meshes

Create the JSON description Create a `.json` file in the root folder of the package. Name the file after the package. Note that this will be the distribution name. The content of the file will describe a JSON array of **maps** and **props** with basic information for each of them.

Maps need the following parameters.

- **name** of the map. This must be the same as the `.fbx` and `.xodr` files.
- **source** path to the `.fbx`.
- **use_carla_materials**. If **True**, the map will use CARLA materials. Otherwise, it will use RoadRunner materials.
- **xodr** Path to the `.xodr`.

Props are not part of this tutorial. The field will be left empty. There is another tutorial on how to add new props.

In the end, the `.json` should look similar to the one below.

```
1 {
2   "maps": [
3     {
4       "name": "Map01",
5       "source": "./Map01/Map01.fbx",
6       "use_carla_materials": true,
7       "xodr": "./Map01/Map01.xodr"
8     },
9     {
10       "name": "Map02",
11       "source": "./Map02/Map02.fbx",
12       "use_carla_materials": false,
13       "xodr": "./Map02/Map02.xodr"
14     }
15   ],
16   "props": [
17   ]
18 }
```

RoadRunner plugin import This software provides specific plugins for CARLA. Get those and follow some simple steps to get the map.

Read RoadRunner plugin import guide



These importing tutorials are deprecated. There are new ways to ingest a map to simplify the process.

Plugin installation These plugins will set everything ready to be used in CARLA. It makes the import process more simple.

1. Locate the plugins in RoadRunner's installation folder `/usr/bin/VectorZero/Tools/Unreal/Plugins`.
2. Copy those folders to the CarlaUE4 plugins directory `/carla/Unreal/CarlaUE4/Plugins/`.
3. Rebuild the plugin following the instructions below.

- a) **Rebuild on Windows.**
 - Right-click the `.uproject` file and **Generate Visual Studio project files**.
 - Open the project and build the plugins.
- b) **Rebuild on Linux.**
 - Run the following command.

```
1 > UE4_ROOT/GenerateProjectFiles.sh -project="carla/Unreal/CarlaUE4/CarlaUE4.uproject" -game -engine
```

4. **Restart Unreal Engine.** Make sure the checkbox is on for both plugins **Edit > Plugins**.

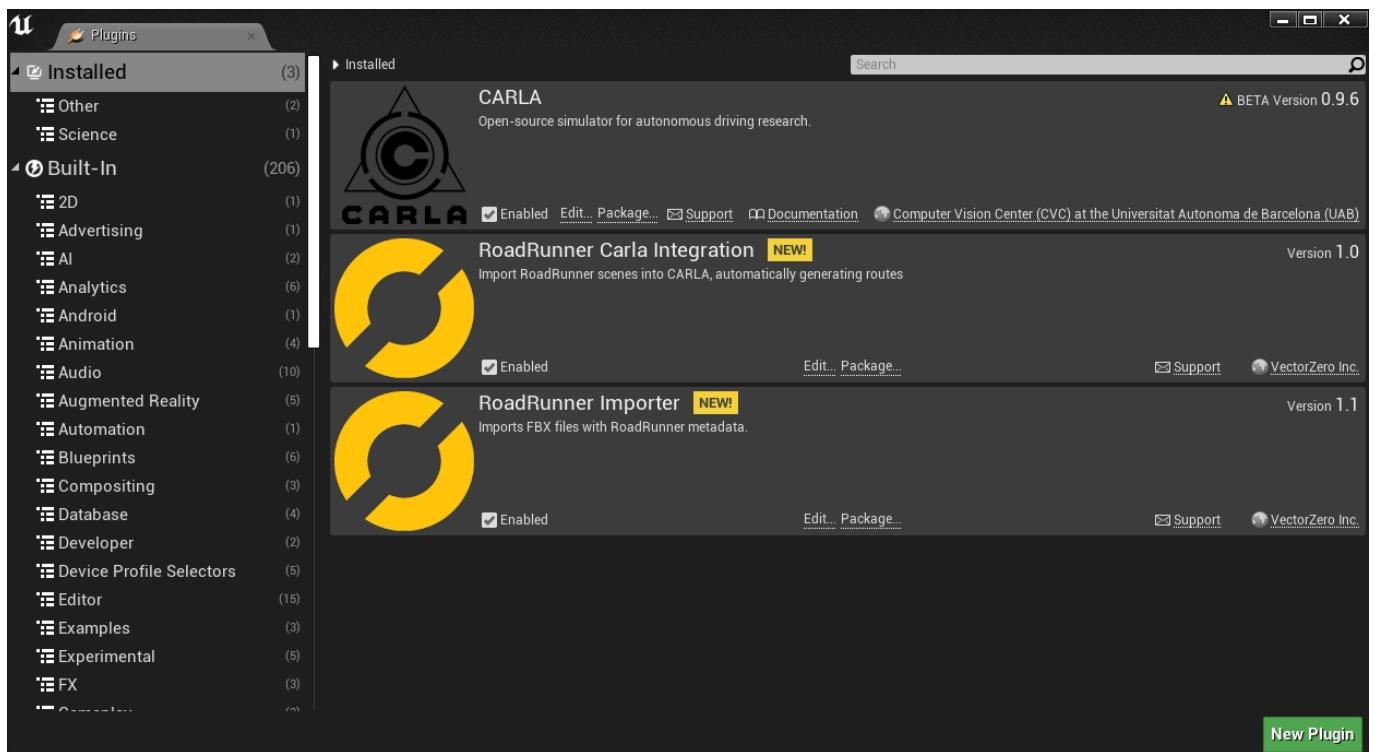


Figure 56: rr_ue_plugins

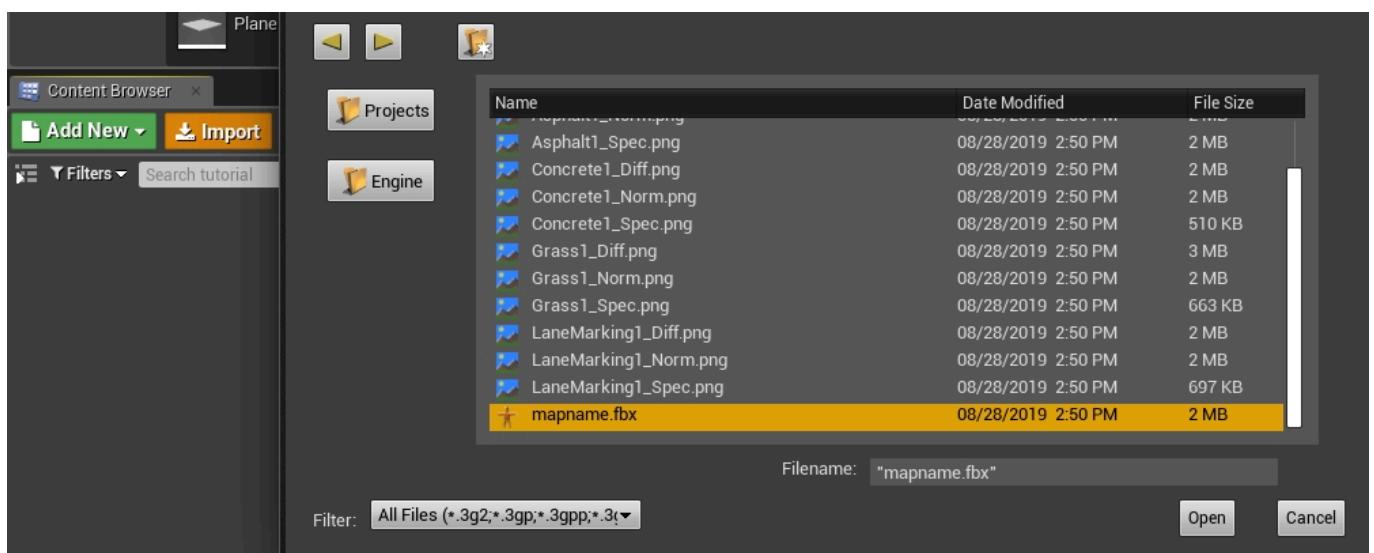


Figure 57: ue_import

Import map 1. Import the *mapname.fbx* file to a new folder under /Content/Carla/Maps with the Import button.

2. Set Scene > Hierarchy Type to Create One Blueprint Asset (selected by default). 3. Set Static Meshes > Normal Import Method to Import Normals.

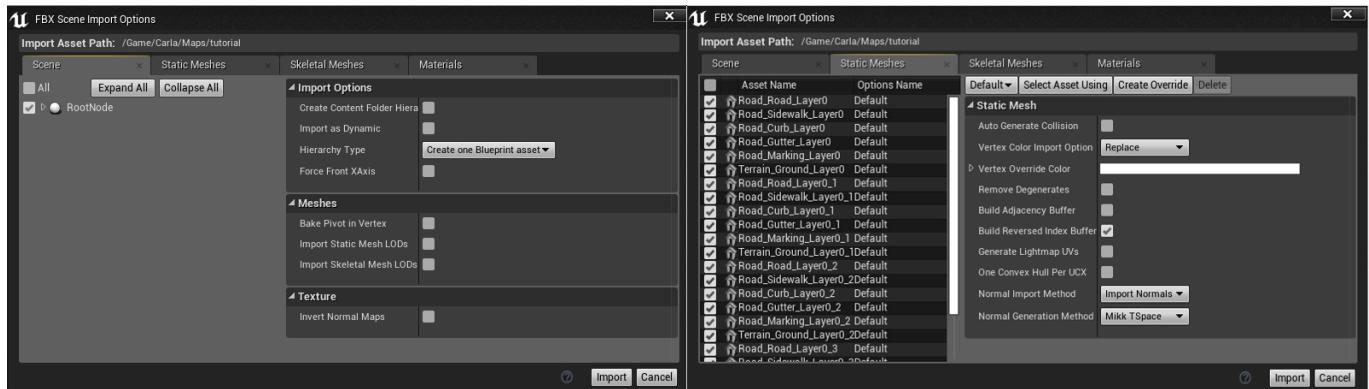
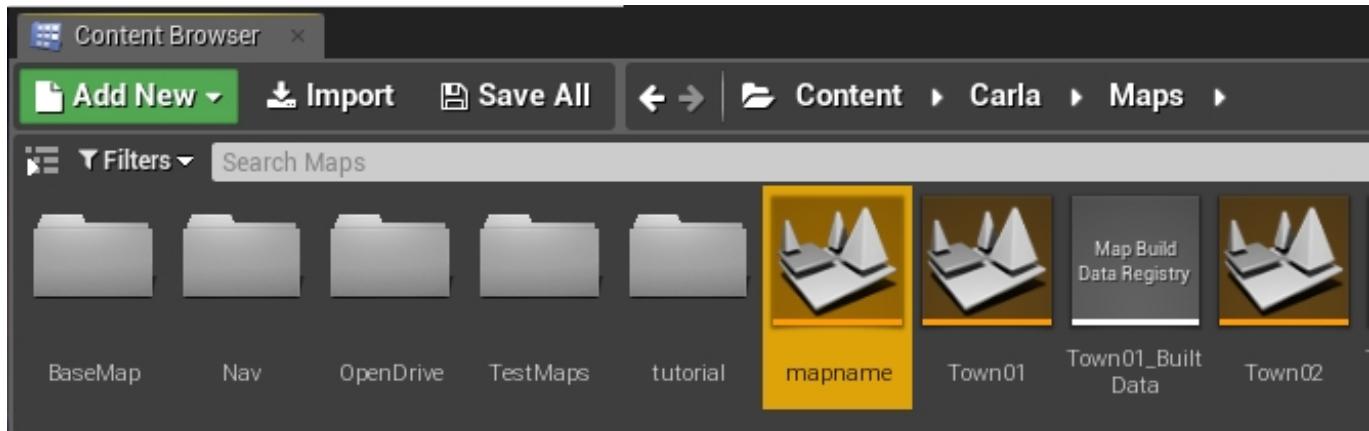


Figure 58: ue_import_options

4. Click Import. 5. Save the current level File > Save Current As... > *mapname*.

The new map should now appear next to the others in the Unreal Engine Content Browser.



The tags for semantic segmentation will be assigned by the name of the asset. And the asset moved to the corresponding folder in 'Content/Carla/PackageName/Static'. To change these, move them manually after imported.

Manual import This process requires to go through all the process manually. From importing *.fbx* and *.xodr* to setting the static meshes.

Read manual import guide



These importing tutorials are deprecated. There are new ways to ingest a map to simplify the process.

This is the generic way to import maps into Unreal Engine using any *.fbx* and *.xodr* files. As there is no plugin to ease the process, there are many settings to be done before the map is available in CARLA.

1. Create a new level with the Map name in Unreal Add New > Level under Content/Carla/Maps. 2. Copy the illumination folder and its content from the BaseMap Content/Carla/Maps/BaseMap, and paste it in the new level. Otherwise, the map will be in the dark.

Import binaries 1. Import the *mapname.fbx* file to a new folder under /Content/Carla/Maps with the Import button. Make sure the following options are unchecked.

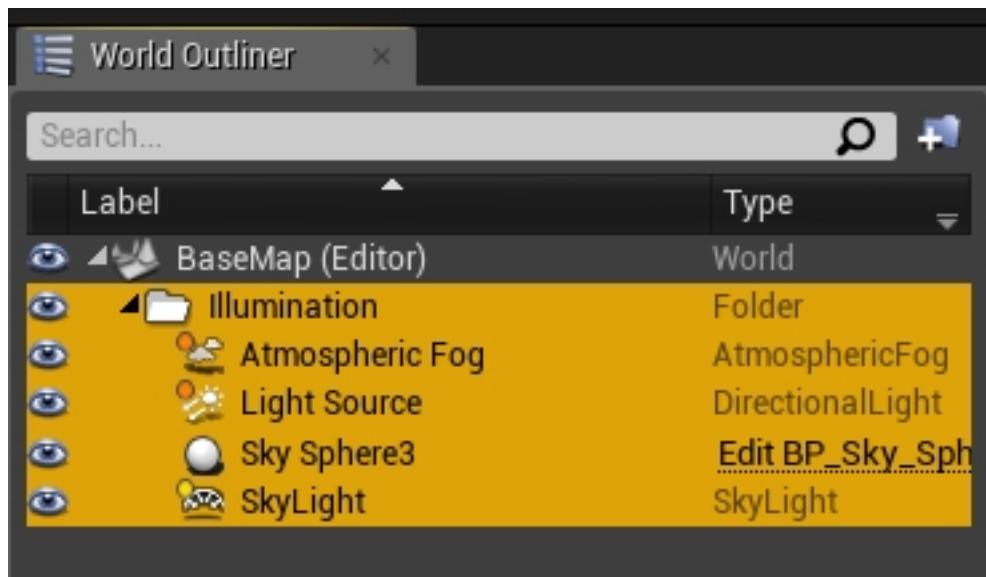


Figure 59: ue_illumination

- Auto Generate Collision
- Combine Meshes
- Force Front xAxis
- Normal Import Method - *To import normals*

2. Check the following options.

- Convert Scene Unit
- *To import materials and textures.*
 - Material Import Method - *To create new materials*
 - Import Textures

3. Check that the static meshes have appeared in the chosen folder.

4. Drag the meshes into the level.

5. Center the meshes at point (0,0,0) when Unreal finishes loading.

6. Generate collisions. Otherwise, pedestrians and vehicles will fall into the abyss.

- Select the meshes meant to have colliders.
- Right-click Asset Actions > Bulk Edit via Property Matrix....
- Search for *collision* in Property's Matrix search box.
- Change Collision complexity from Project Default to Use Complex Collision As Simple.
- Go to File > Save All.

7. Move the static meshes from Content/Carla/Maps/mapfolder to the corresponding Carla/Static subsequent folder. This will be meaningful for the semantic segmentation ground truth.

- Terrain/mapname
- Road/mapname
- RoadLines/mapname

1 Content

2 Carla

```

3     Blueprints
4     Config
5     Exported Maps
6     HDMaps
7     Maps
8     Static
9         Terrain
10            mapname

```

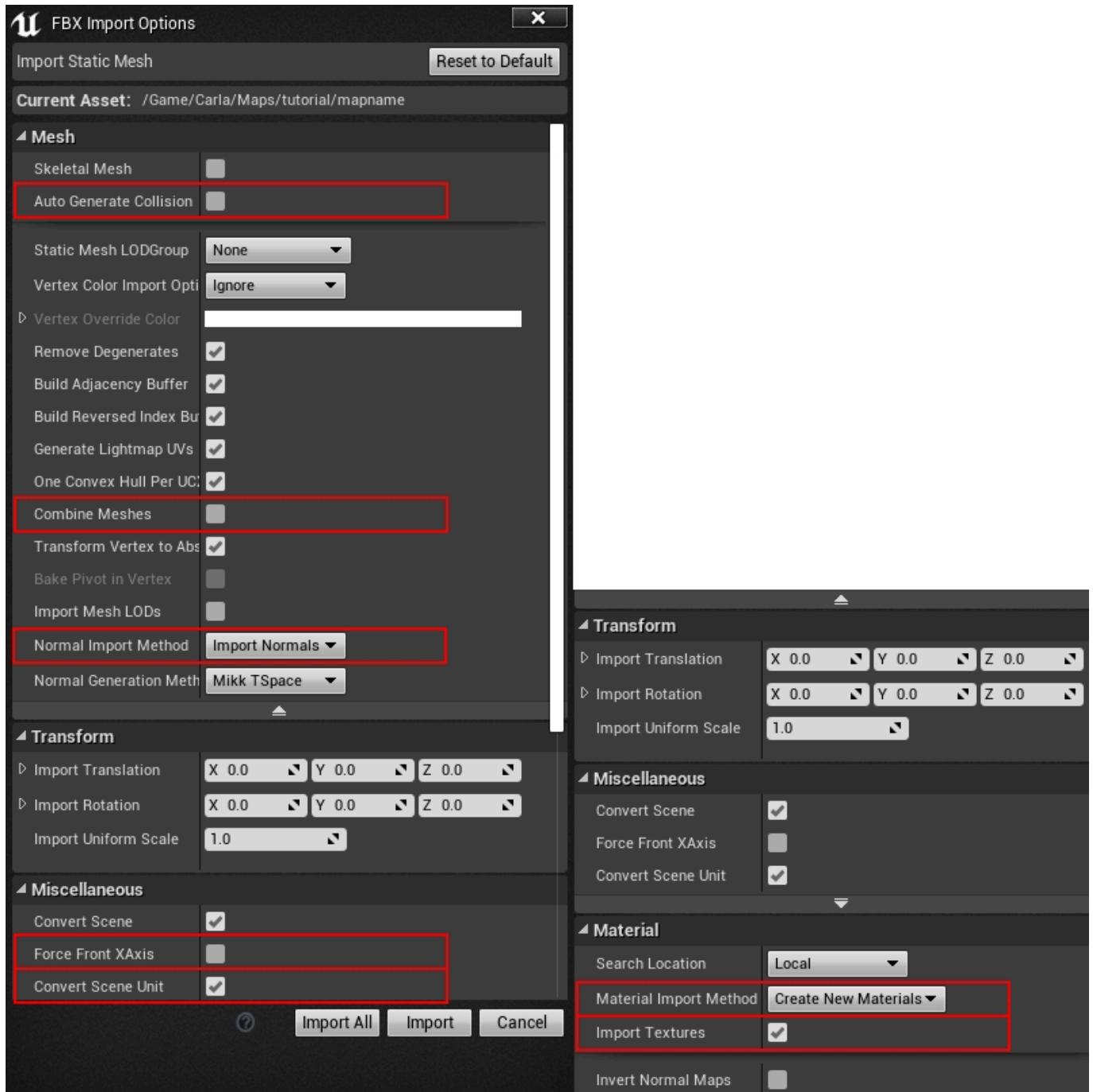


Figure 60: ue_import_file

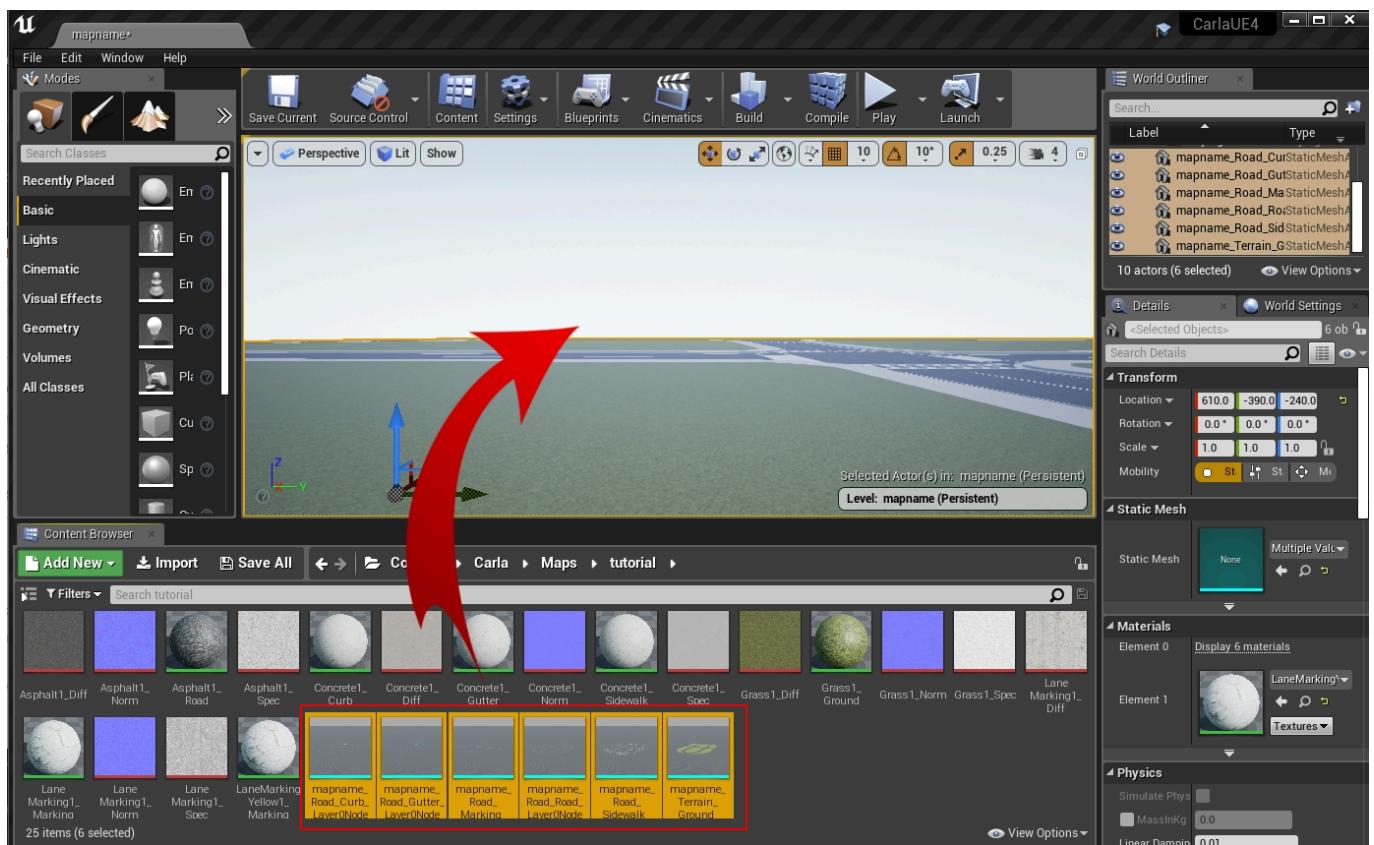


Figure 61: ue_mesches



Figure 62: Transform_Map

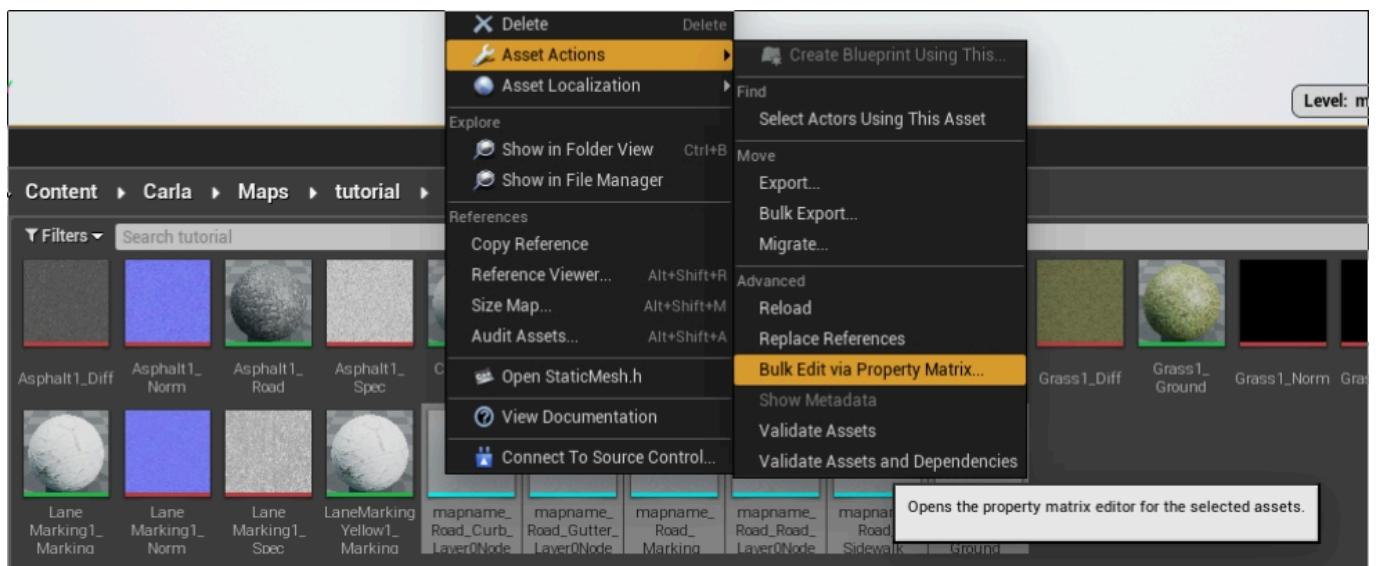


Figure 63: ue_selectmesh_collision

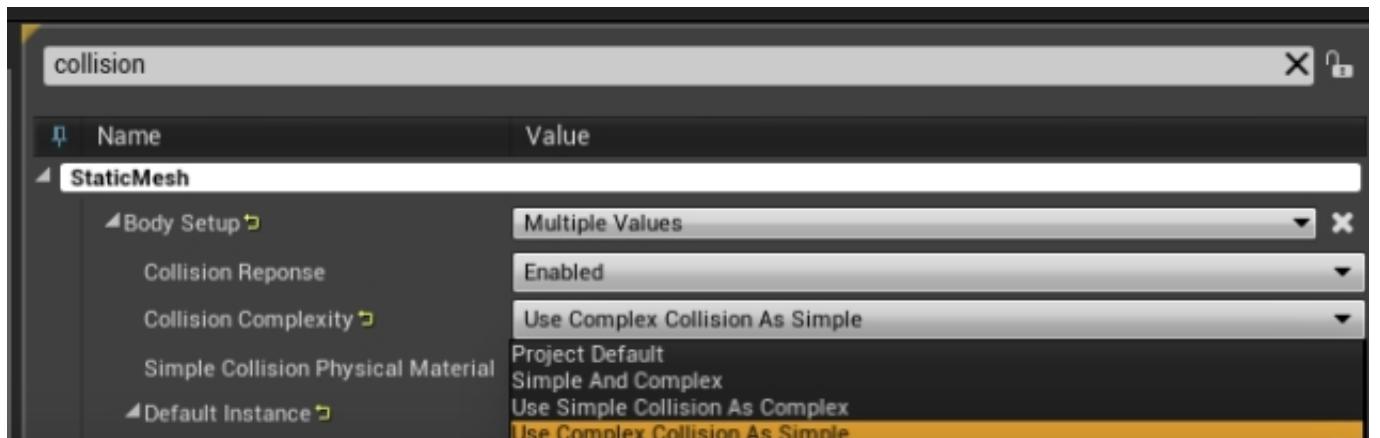


Figure 64: ue_collision_complexity

```

11             Static Meshes
12
13     Road
14         mapname
15             Static Meshes
16
17     RoadLines
18         |   mapname
19         |       Static Meshes
20     Sidewalks
21         mapname
22             Static Meshes

```

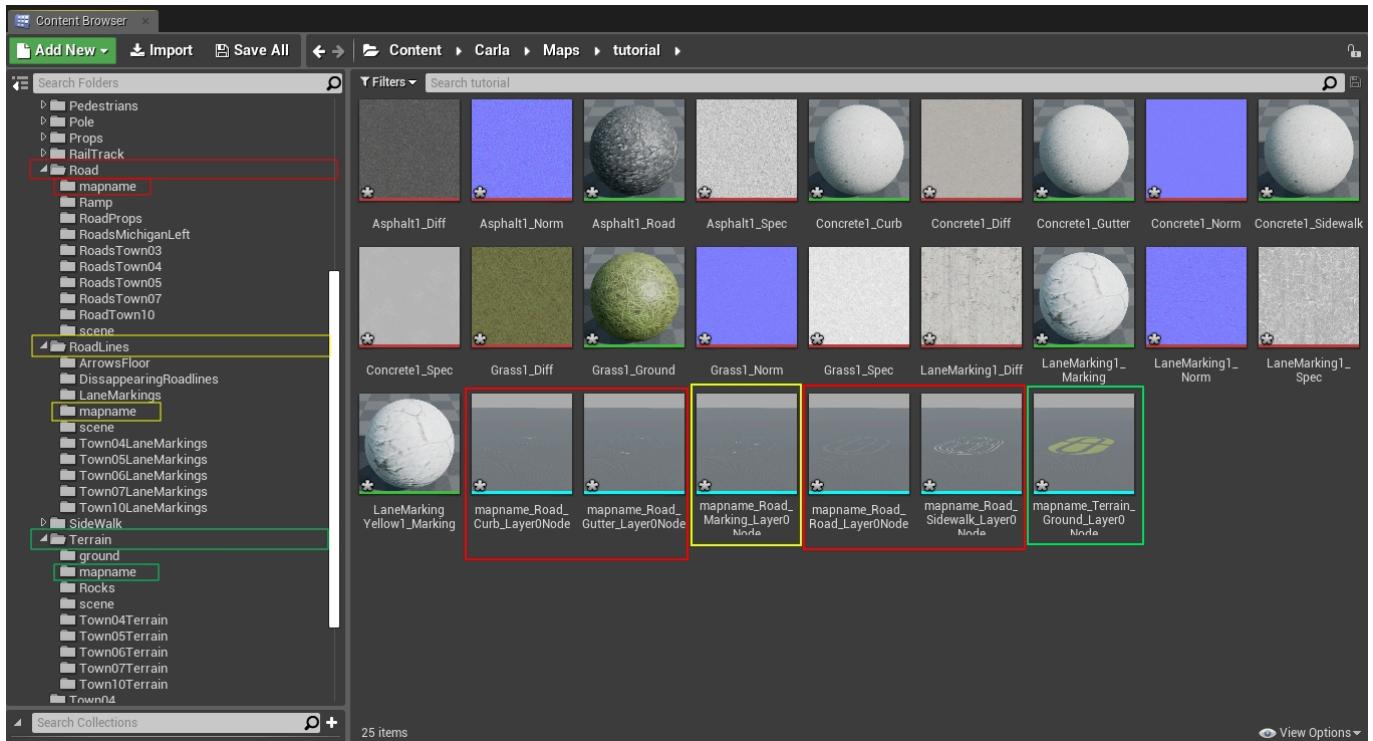


Figure 65: ue____semantic_segmentation

Import OpenDRIVE files **1.** Copy the .xodr file inside the Content/Carla/Maps/OpenDrive folder. **2.** Open the Unreal level. Drag the *Open Drive Actor* inside the level. It will read the level's name. Search the Opendrive file with the same name and load it.

Set traffic and pedestrian behaviour This software provides specific plugins for CARLA. Get those and follow some simple steps to get the map.

Read traffic and pedestrian setting guide



These importing tutorials are deprecated. There are new ways to ingest a map to simplify the process.

Set traffic behavior Once everything is loaded into the level, it is time to create traffic behavior.

1. Click on the *Open Drive Actor*. 2. Check the following boxes in the same order.

- Add Spawners.
- (*Optional for more spawn points*) On Intersections.
- Generate Routes.

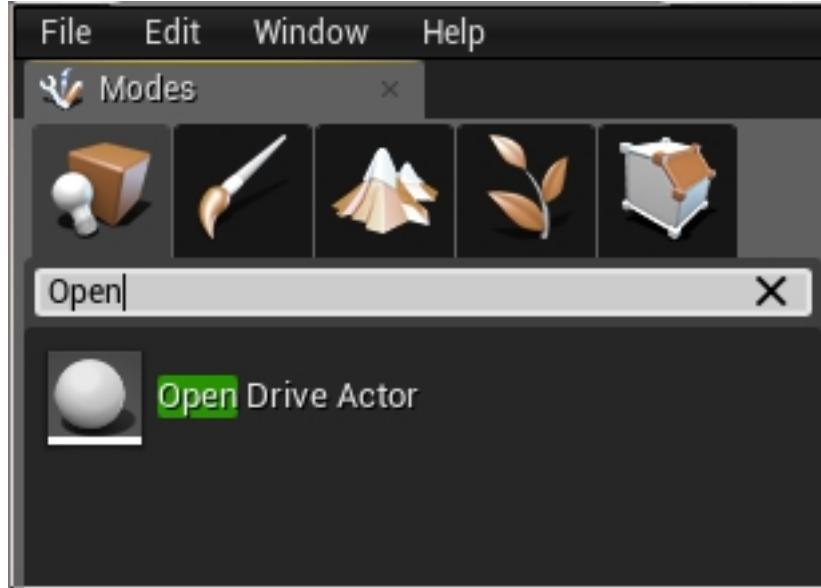


Figure 66: ue_opendrive_actor

This will generate a series of *RoutePlanner* and *VehicleSpawnPoint* actors. These are used for vehicle spawning and navigation.

Traffic lights and signs Traffic lights and signs must be placed all over the map.

1. Drag traffic light/sign actors into the level and place them. 2. Adjust the [trigger volume][triggerlink] for each of them. This will determine their area of influence. [triggerlink]: python_api.md#carla.TrafficSign.trigger_volume

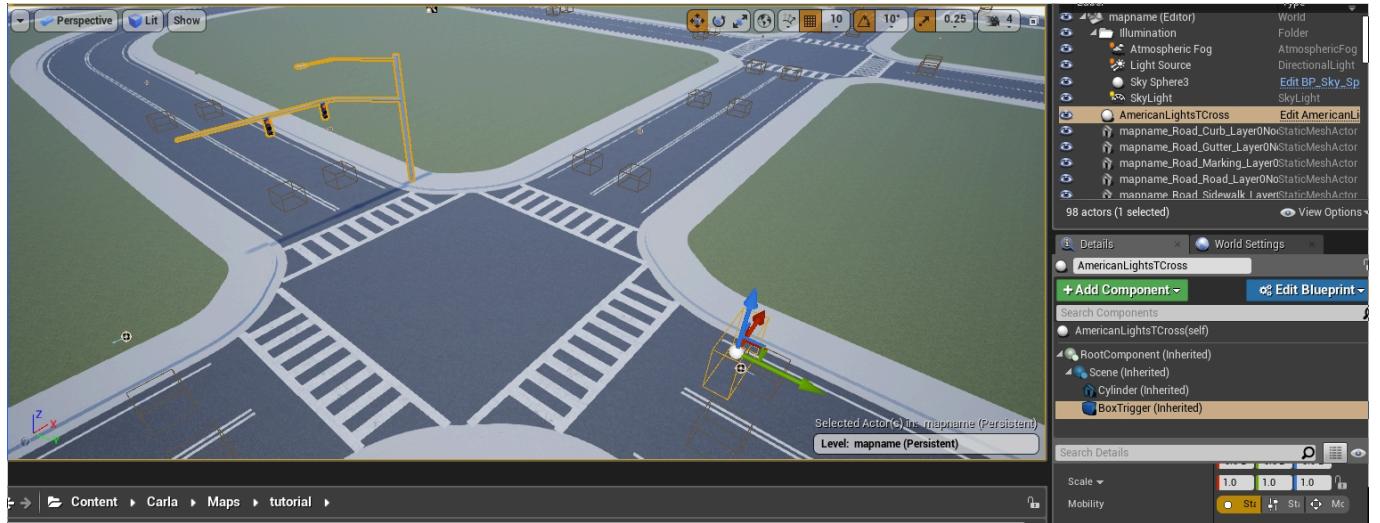


Figure 67: ue_trafficlight

3. In junctions, drag a traffic light group actor into the level. Assign to it all the traffic lights involved and configure their timing. Make sure to understand how do traffic lights work.

4. Test traffic light timing and traffic trigger volumes. This may need trial and error to fit perfectly.

Example: Traffic Signs, Traffic lights and Turn based stop.

Add pedestrian navigation In order to prepare the map for pedestrian navigation, there are some settings to be done before exporting it.

1. Select the Skybox object and add a tag NoExport to it. Otherwise, the map will not be exported, as the size would be too big. Any geometry that is not involved or interfering in the pedestrian navigation can be tagged also as **NoExport**.

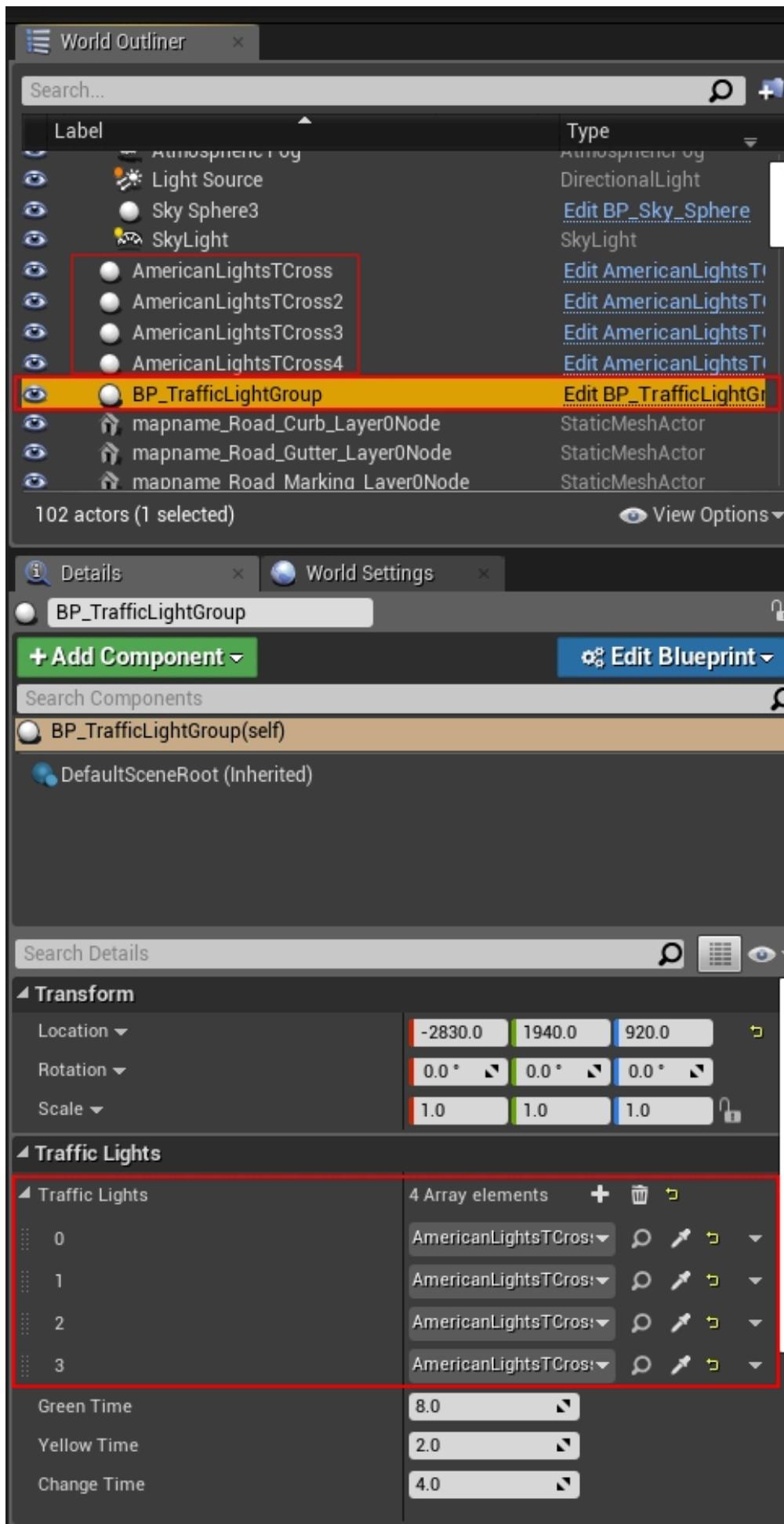


Figure 68: ue_tl_group
260



Figure 69: ue_tlsigns_example

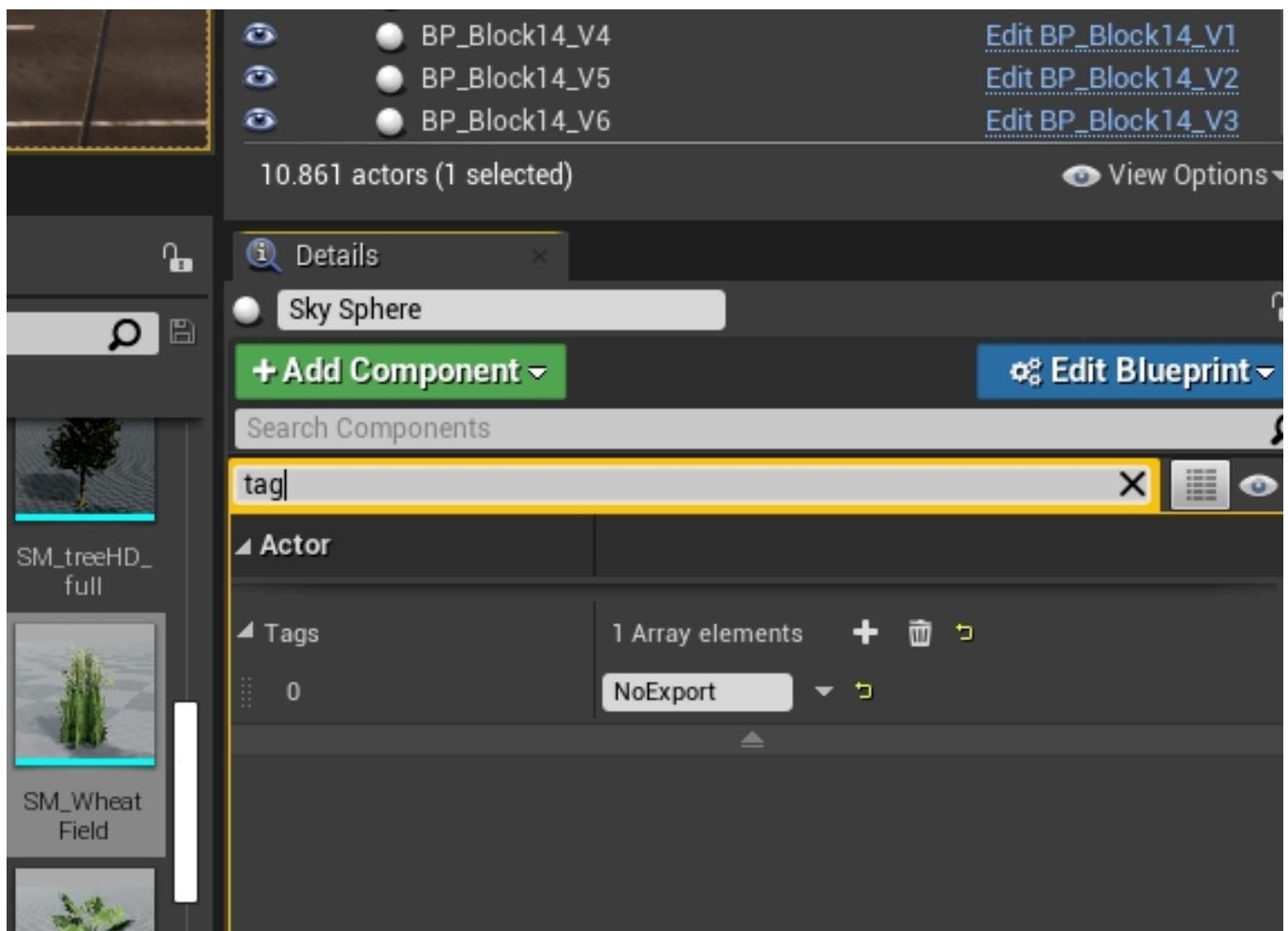
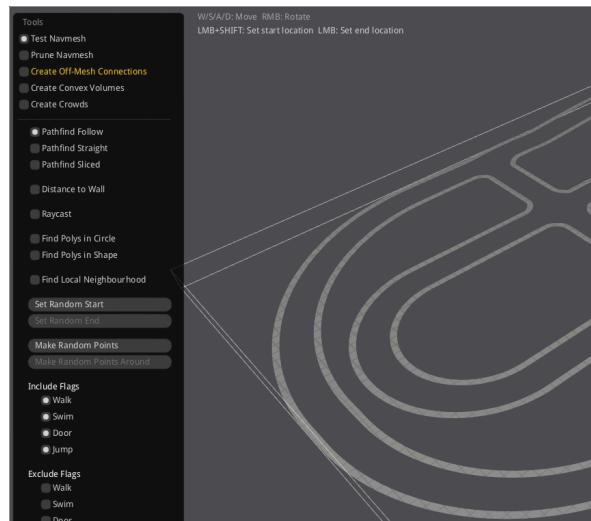


Figure 70: ue_skybox_no_export

2. Check the name of the meshes. By default, pedestrians will be able to walk over sidewalks, crosswalks, and grass (with minor influence over the rest).
 3. Crosswalks have to be manually created. For each of them, create a plane mesh that extends a bit over both sidewalks connected. **Place it overlapping the ground, and disable its physics and rendering.**
 4. Name these planes following the common format `Road_Crosswalk_mapname`.
 5. Press `G` to deselect everything, and export the map. `File > Export CARLA....`
 6. Run `RecastDemo ./RecastDemo`.
- Select **Solo Mesh** from the **Sample** parameter's box.



- Select the `mapname.obj` file from the **Input Mesh** parameter's box.

7. Click on the **Build** button. **8.** Once the build has finished, click on the **Save** button. **9.** Change the **filename** of the binary file generated at `RecastDemo/Bin` to `mapname.bin`. **10.** Drag the `mapname.bin` file into the **Nav** folder under `Content/Carla/Maps`.

That comprises the process to create and import a new map into CARLA. If during the process any doubts arise, feel free to post these in the forum.

9.2 Add a new vehicle

This tutorial covers step by step the process to add a new vehicle model. From the very moment the model is finished, to a simple test in CARLA.

- **Add a 4 wheeled vehicle**
 - Bind the skeleton
 - Import and prepare the vehicle
- **Add a 2 wheeled vehicle**



This tutorial only applies to users that work with a build from source, and have access to the Unreal Editor.

Add a 4 wheeled vehicle

Bind the skeleton CARLA provides a general skeleton for 4-wheeled vehicles that must be used by all of them. The position of the bones can be changed, but rotating them, adding new ones or changing the hierarchy will lead to errors.

1. **Download the skeleton** from the CARLA repositories. Here is the link to [General4WheeledVehicleSkeleton](#), the reference skeleton for 4 wheeled vehicles.
2. **Import the skeleton** into the 3D project of the new vehicle. This could be in Maya, Blender or whichever software used for modelling.
3. **Bind the bones** to the corresponding portions of the mesh. Make sure to center the wheels' bones within the mesh.
 - Front left wheel — `Wheel_Front_Left`.

World Outliner

| Label | Type |
|----------------------------|-----------------------------------|
| Road_Sidewalk_Town10HD43 | StaticMeshActor |
| Road_Sidewalk_Town10HD44 | StaticMeshActor |
| Road_Sidewalk_Town10HD45 | StaticMeshActor |
| Road_Sidewalk_Town10HD46 | StaticMeshActor |
| Road_Sidewalk_Town10HD47 | StaticMeshActor |
| Road_Sidewalk_Town10HD48 | StaticMeshActor |
| Road_Sidewalk_Town10HD49 | StaticMeshActor |
| Road_Sidewalk_Town10HD50 | StaticMeshActor |
| Road_Sidewalk_Town10HD51 | StaticMeshActor |
| Road_Sidewalk_Town10HD52 | StaticMeshActor |
| Road_Sidewalk_Town10HD53 | StaticMeshActor |
| Road_Sidewalk_Town10HD54 | StaticMeshActor |
| Road_Sidewalk_Town10HD55 | StaticMeshActor |
| Road_Sidewalk_Town10HD56 | StaticMeshActor |
| Road_Sidewalk_Town10HD57 | StaticMeshActor |
| Roa | ID Name: Road_Sidewalk_Town10HD57 |
| Roa | StaticMeshActor |
| Roa | StaticMeshActor |
| Road_Sidewalk_Town10HD60 | StaticMeshActor |
| Road_Sidewalk_Town10HD61 | StaticMeshActor |
| Road_Sidewalk_Town10HD62 | StaticMeshActor |
| Road_Sidewalk_Town10HD63 | StaticMeshActor |
| Road_Sidewalk_Town10HD64 | StaticMeshActor |
| SM_Road_SideWalk_Town10HD | StaticMeshActor |
| SM_Road_SideWalk_Town10HD2 | StaticMeshActor |
| SM_Road_SideWalk_Town10HD3 | StaticMeshActor |
| SM_Road_SideWalk_Town10HD4 | StaticMeshActor |
| SM_Road_SideWalk_Town10HD5 | StaticMeshActor |
| SM_Road_SideWalk_Town10HD6 | StaticMeshActor |
| SM_Road_SideWalk_Town10HD7 | StaticMeshActor |
| SM_Road_SideWalk_Town10HD8 | StaticMeshActor |

10.861 actors (1 selected) View Options ▾

Figure 71: ue_meshes



Figure 72: ue_crosswalks

- **Front right wheel** — Wheel_Front_Right.
- **Rear left wheel** — Wheel_Rear_Left.
- **Rear right wheel** — Wheel_Rear_Right.
- **Rest of the mesh** — VehicleBase.



| Do not add new bones, change their names or the hierarchy.

4. Export the result.

Select all the meshes and the base of the skeleton and export as .fbx.

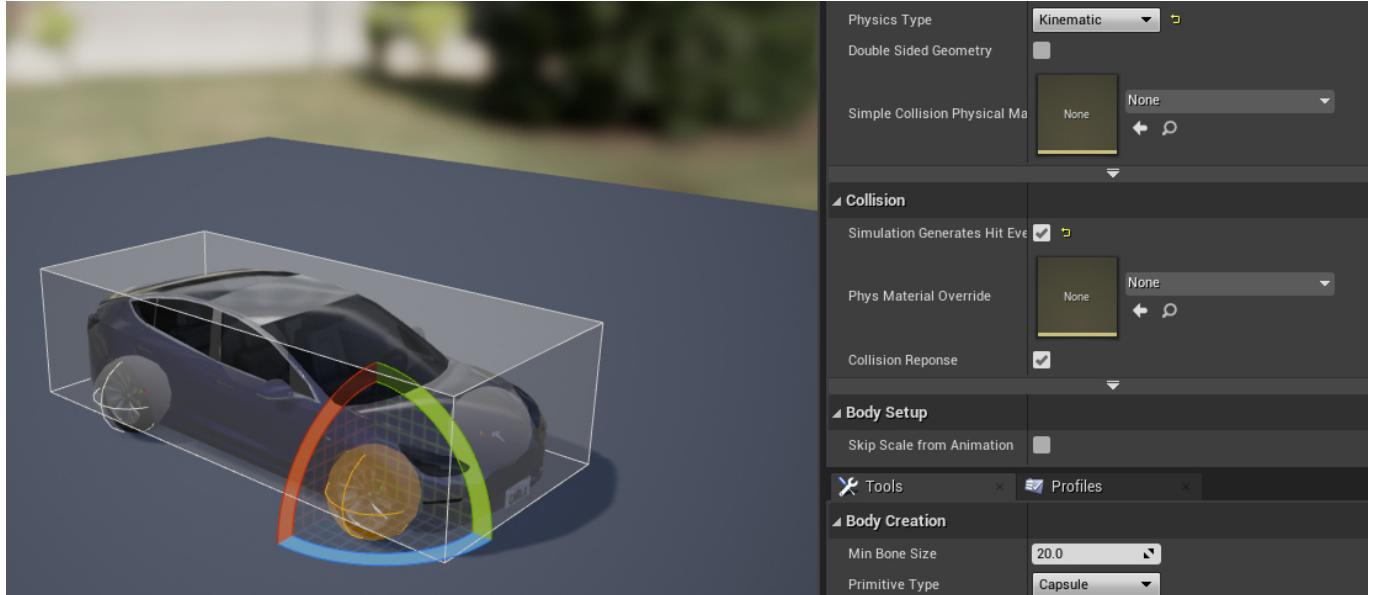
Import and prepare the vehicle

1. Create a new folder named <vehicle_name> in Content/Carla/Static/Vehicle.
2. Import the .fbx in its folder. The Skelletal Mesh will appear along with two new files, <vehicle_name>_PhysicsAsset and <vehicle_name>_Skeleton.
 - 2.1 - *Import Content Type* — Geometry and Skinning Weights.
 - 2.2 - *Normal Import Method* — Import Normals.
 - 2.3 - *Material import method* — Optionally choose Do not create materials and uncheck Import Textures to avoid Unreal creating its own default materials.



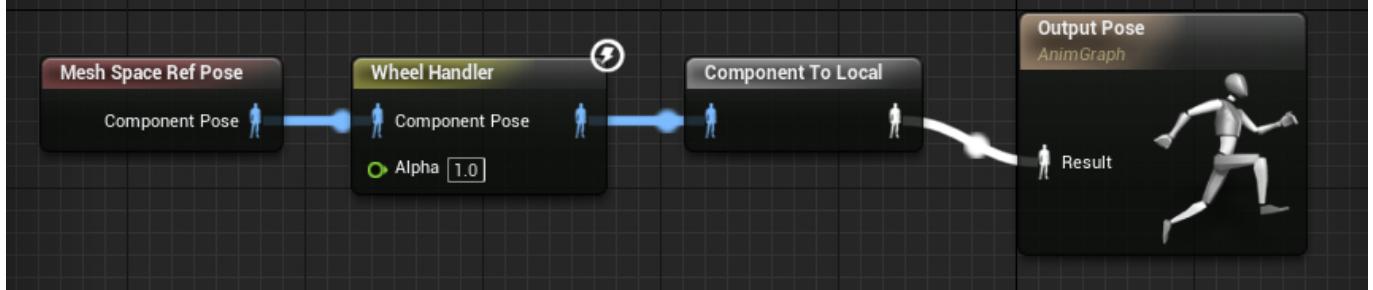
| If Unreal does not create the vehicle materials, these will have to be created manually.

3. Open <vehicle_name>_PhysicsAssets to set the vehicle colliders.
 - 3.1 - Change the wheels' colliders — Select a sphere and adjust it to the shape of the wheel.
 - 3.2 - Change the wheels' *Physics Type* — Select Kinematic for all of them.
 - 3.3 - Change the general collider — Select a box and adjust it to the shape of the vehicle.
 - 3.4 - Enable *Simulation Generates Hit Event* — Check it for all of the physics' bodies.



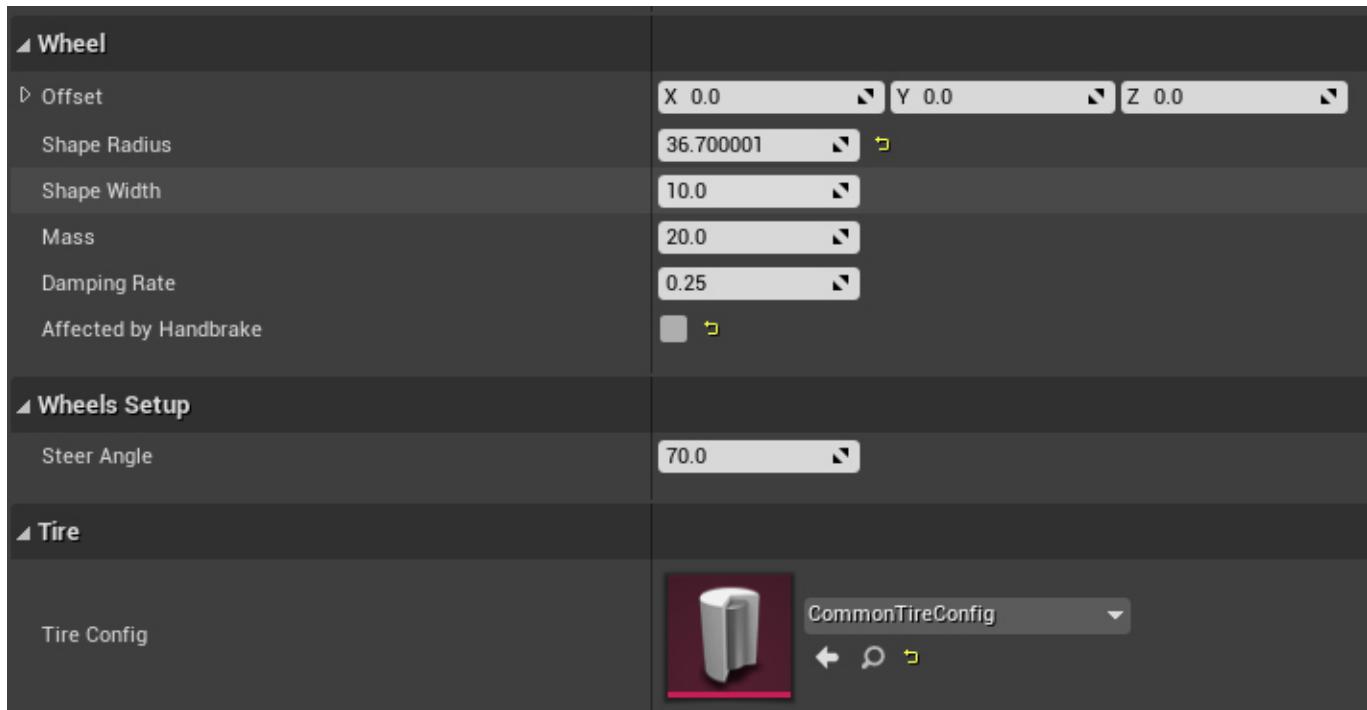
Step 3, set colliders.

- 4. **Create the Animation Blueprint.** In the new vehicle folder, click right and go to **Create advanced asset/Animation/Animation blueprint**.
 - 4.1 - **Parent Class** — `VehicleAnimInstance`.
 - 4.2 - **Skeleton** — `<vehicle_name>_Skeleton`.
 - 4.3 - **Rename the blueprint** — `BP_<vehicle_name>_anim`.
 - 4.4 - **Copy an existing Animation Blueprint** — Go to `Content/Carla/Static/Vehicle` and choose any vehicle folder. Open its Animation Blueprint and copy the content from the *AnimGraph*.
 - 4.5 - **Compile the Animation Blueprint** — Connect the content in the blueprint and click the button **Compile** on the top left corner.



Step 4.5, connect the blueprint.

- 5. **Create a folder for the vehicle blueprints.** Go to `Content/Carla/Blueprints/Vehicles` and create a new folder `<vehicle_name>`.
- 6. **Create blueprints for the wheels.** Inside the folder, right-click and go to **Create advanced assets/Blueprints class**. Create two blueprints derived from `VehicleWheel`, one named `<vehicle_name>_FrontWheel` and the other `<vehicle_name>_RearWheel`.
 - 6.1 - **Shape radius** — Exactly the radius, not diameter, of the wheel mesh.
 - 6.2 - **Rig Config** — `CommonTireConfig`.
 - 6.3 - **On the front wheel** — Set **Steer Angle**, default is 70. Uncheck **Affected by Handbrake**.
 - 6.4 - **On the rear wheel** — Set **Steer Angle** to 0. Check **Affected by Handbrake**.

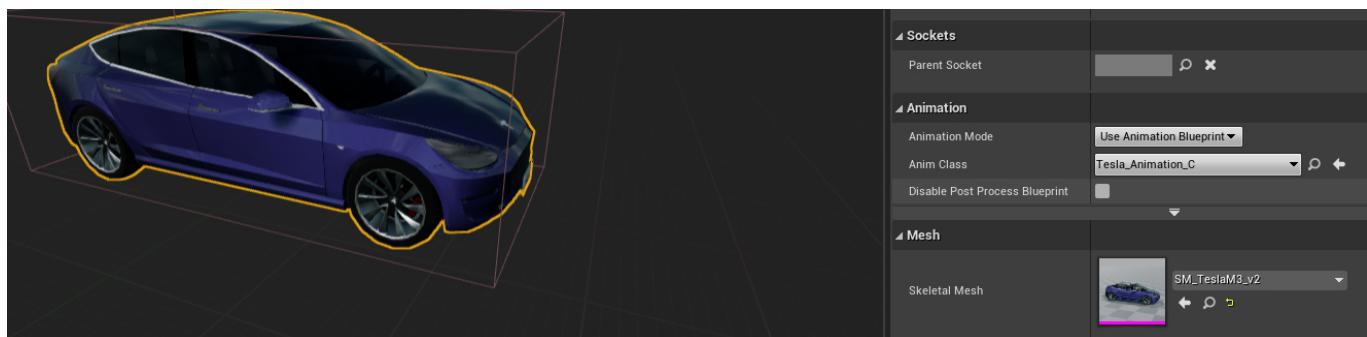


Step 6.3, front wheel setup.

- 7. Create a blueprint for the vehicle. Inside the folder, create another blueprint derived from BaseVehiclePawn and named BP_<vehicle_name>.
 - 7.1 - **Mesh** — Choose the skeletal mesh of the vehicle.
 - 7.2 - **Anim class** — Choose the Animation blueprint created in step 4.
 - 7.3 - **Vehicle bound** — Adjust it to include the whole volume of the vehicle.

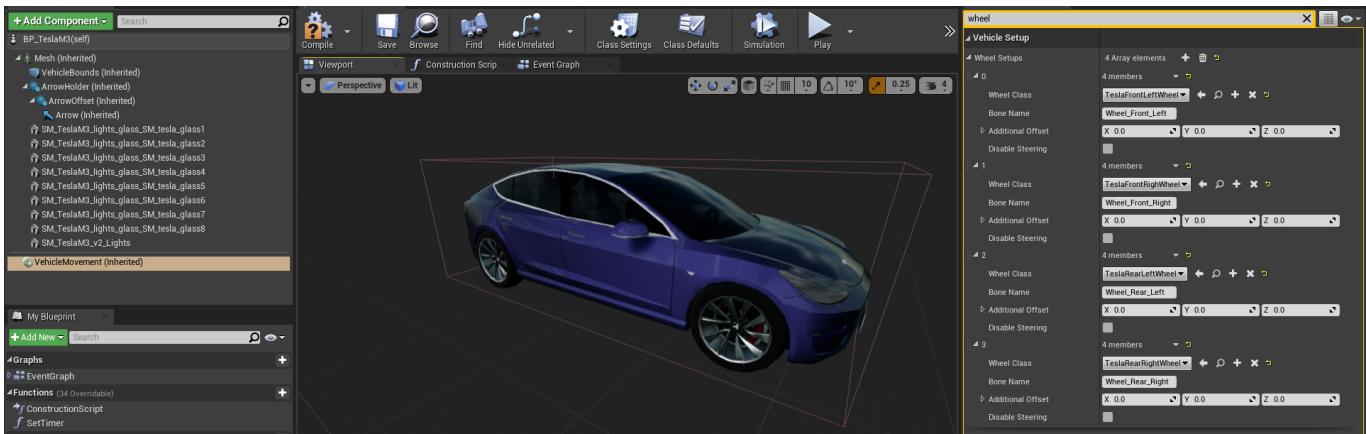


These options appear in the menu *Components* on the left side of the window.



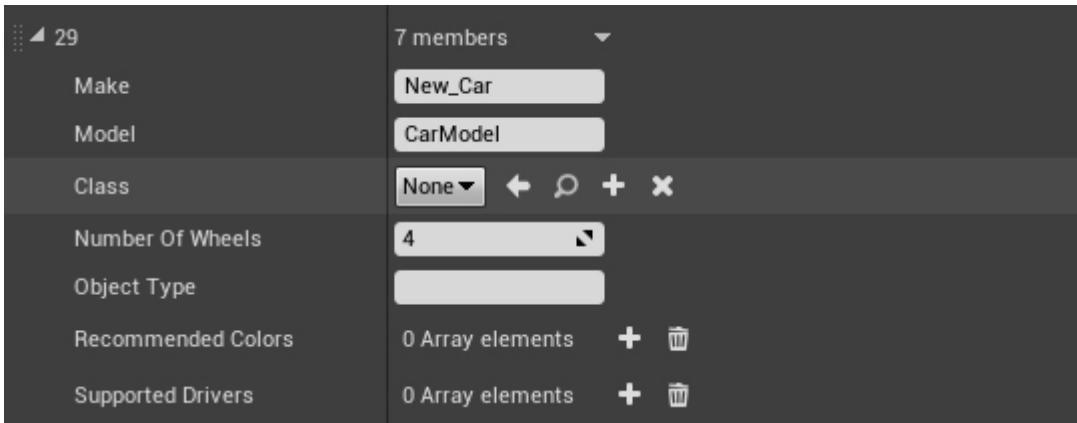
Step 6.3, create the blueprint.

- 8. Pair the wheels with their blueprint. In Vehicle Movement/Wheel Setups expand the menu and prepare each wheel.
 - 8.1 - **Wheel_Front_Left** — <vehicle_model>_FrontWheel
 - 8.2 - **Wheel_Front_Right** — <vehicle_model>_FrontWheel
 - 8.3 - **Wheel_Rear_Left** — <vehicle_model>_RearWheel
 - 8.4 - **Wheel_Rear_Right** — <vehicle_model>_RearWheel



Step 8, pair the wheels.

- **9 - Compile the blueprint** — Click the button **Compile** on the top left corner.
- **10 - Add the vehicle.** In Content/Carla/Blueprint/Vehicle, open the **VehicleFactory** and add a new element to the array of vehicles.
 - **10.1 - Make** — Choose a name to be used in Unreal.
 - **10.2 - Model** — Choose the name to be used in the blueprint library in CARLA.
 - **10.3 - Class** — BP_<vehicle_name>.
 - **10.4 - Recommended colours** — Optionally, provide a set of recommended colours for the vehicle.



Step 10, add the new vehicle.

- **11. Test the vehicle.** Launch CARLA, open a terminal in PythonAPI/examples and run the following command.

```
1 python3 manual_control.py --filter <model_name> # The name used in step 10.2
```

Add a 2 wheeled vehicle

Adding 2 wheeled vehicles is similar to adding a 4 wheeled one but due to the complexity of the animation you'll need to set up additional bones to guide the driver's animation. Here is the link to the reference skeleton for 2 wheeled vehicles.

As with the 4 wheeled vehicles, orient the model towards positive "x" and every bone axis towards positive x and with the z axis facing upwards.

```
1 Bone Setup:
2   - Bike_Rig:           # The origin point of the mesh. Place it in the point 0 of the
     scenecomment
3   - BikeBody:          # The model's body centre.
4     - Pedals:           # If the vehicle is a bike bind the pedalier to this bone, will
       rotate with the bike acceleration.
5     - RightPedal:        # Sets the driver's feet position and rotates with the pedalier if
       the vehicle is a bike.
6     - LeftPedal:         #
7     - RearWheel:         # Rear Wheel of the vehicle
```

```

8   - Handler:           # Rotates with the frontal wheel of the vehicle bind the vehicle
9     handler to it.
10  - HandlerMidBone:    # Positioned over the front wheel bone to orient the handler with the
11    wheel
12  - HandlerRight:      # Sets the position of the driver's hand, no need to bind it to
13    anything.
14  - HandlerLeft:       # ^
15  - Frontwheel:        # Frontal wheel of the vehicle.
16  - RightHelperRotator: # This four additional bones are here for an obsolete system of
17    making the bike stable by using aditional invisible wheels
     - RightHelprWheel:   # ^
     - LeftHelperRotator: # ^
     - LeftHelperWheel:   # ^
     - Seat:              # Sets the position of the drivers hip bone. No need to bind it to
                           anything but place it carefully.

```

1. Import fbx as Skelletal Mesh to its own folder inside `Content/Carla/Static/Vehicles/2Wheeled`. When importing select “General2WheeledVehicleSkeleton” as skelleton A Physics asset should be automatically created and linked.

2. Tune the Physics asset. Delete the automatically created ones and add boxes to the `BikeBody` bone trying to match the shape as possible, make sure generate hit events is enabled. Add a sphere for each wheel and set their “Physics Type” to “Kinematic”.

3. Create folder `Content/Blueprints/Vehicles/<vehicle-model>`

4. Inside that folder create two blueprint classes derived from “VehicleWheel” class. Call them `<vehicle-model>_FrontWheel` and `<vehicle-model>_RearWheel`. Set their “Shape Radius” to exactly match the mesh wheel radius (careful, radius not diameter). Set their “Tire Config” to “CommonTireConfig”. On the front wheel uncheck “Affected by Handbrake” and on the rear wheel set “Steer Angle” to zero.

5. Inside the same folder create a blueprint class derived from `Base2WheeledVehicle` call it `<vehicle-model>`. Open it for edit and select component “Mesh”, setup the “Skeletal Mesh” and the “Anim Class” to the corresponding ones. Then select the `VehicleBounds` component and set the size to cover vehicle’s area as seen from above.

6. Select component “VehicleMovement”, under “Vehicle Setup” expand “Wheel Setups”, setup each wheel.

- **0:** Wheel Class=`<vehicle-model>_FrontWheel`, Bone Name=`FrontWheel`
- **1:** Wheel Class=`<vehicle-model>_FrontWheel`, Bone Name=`FrontWheel`
- **2:** Wheel Class=`<vehicle-model>_RearWheel`, Bone Name=`RearWheel`
- **3:** Wheel Class=`<vehicle-model>_RearWheel`, Bone Name=`RearWheel` (You’ll notice that we are basically placing two wheels in each bone. The vehicle class unreal provides does not support vehicles with wheel numbers different from 4 so we had to make it believe the vehicle has 4 wheels)

7. Select the variable “is bike” and tick it if your model is a bike. This will activate the pedalier rotation. Leave unmarked if you are setting up a motorbike.

8. Find the variable back Rotation and set it as it fit better select the component `SkeletalMesh` (The driver) and move it along x axis until its in the seat position.

9. Test it, go to `CarlaGameMode` blueprint and change “Default Pawn Class” to the newly created bike blueprint.

9.3 Add new props

Props are the assets that populate the scene, besides the map, and the vehicles. That includes streetlights, buildings, trees, and much more. The simulator can ingest new props anytime in a simple process. This is really useful to create customized environments in a map.

Prepare the package Create the folder structure * Create the JSON description * **Ingestion in a CARLA package** * **Ingestion in a build from source**

Prepare the package

Create the folder structure **1. Create a folder inside carla/Import.** The name of the folder is not relevant.

2. Create the subfolders. There should be one general subfolder for all the props, and inside of it, as many subfolders as props to import.

3. Move the files of each prop to the corresponding subfolder. A prop subfolder will contain the .fbx mesh, and optionally, the textures required by it.

For instance, an Import folder with two separate packages should have a structure similar to the one below.

```
1 Import
2
3 Package01
4     Package01.json
5     Props
6         Prop01
7             Prop01_Diff.png
8             Prop01_Norm.png
9             Prop01_Spec.png
10            Prop01.fbx
11        Prop02
12            Prop02.fbx
13 Package02
14     Packag02.json
15     Props
16         Prop03
17             Prop03.fbx
```

Create the JSON description Create a .json file in the root folder of the package. Name the file after the package. Note that this will be the distribution name. The content of the file will describe a JSON array of **maps** and **props** with basic information for each of them.

Maps are not part of this tutorial, so this definition will be empty. There is a specific tutorial to **add a new map**.

Props need the following parameters.

- **name** of the prop. This must be the same as the .fbx.
- **source** path to the .fbx.
- **size** estimation of the prop. The possible values are listed here.
 - tiny
 - small
 - medium
 - big
 - huge
- **tag** value for the semantic segmentation. If the tag is misspelled, it will be read as **Unlabeled**.
 - Bridge
 - Building
 - Dynamic
 - Fence
 - Ground
 - GuardRail
 - Other
 - Pedestrian
 - Pole
 - RailTrack
 - Road
 - RoadLine
 - SideWalk
 - Sky
 - Static
 - Terrain
 - TrafficLight
 - TrafficSign
 - Unlabeled
 - Vegetation
 - Vehicles
 - Wall
 - Water

In the end, the `.json` should look similar to the one below.

```
1 {
2   "maps": [
3   ],
4   "props": [
5     {
6       "name": "MyProp01",
7       "size": "medium",
8       "source": "./Props/Prop01/Prop01.fbx",
9       "tag": "SemanticSegmentationTag01"
10    },
11    {
12      "name": "MyProp02",
13      "size": "small",
14      "source": "./Props/Prop02/Prop02.fbx",
15      "tag": "SemanticSegmentationTag02"
16    }
17  ]
18 }
```



Packages with the same name will produce an error.

Ingestion in a CARLA package

This is the method used to ingest the props into a CARLA package such as CARLA 0.9.8.

A Docker image of Unreal Engine will be created. It acts as a black box that automatically imports the package into the CARLA image, and generates a distribution package. The Docker image takes 4h and 400GB to be built. However, this is only needed the first time.

1. **Build a Docker image of Unreal Engine.** Follow these instructions to build the image.
2. **Run the script to cook the props.** In the folder `~/carla/Util/Docker` there is a script that connects with the Docker image previously created, and makes the ingestion automatically. It only needs the path for the input and output files, and the name of the package to be ingested.

```
1 python3 docker_tools.py --input ~/path_to_package --output ~/path_for_output_assets
  --package=Package01
```

3. **Locate the package.** The Docker should have generated the package `Package01.tar.gz` in the output path. This is the standalone package for the assets.

4. Import the package into CARLA.

- **On Windows** extract the package in the `WindowsNoEditor` folder.
- **On Linux** move the package to the `Import` folder, and run the script to import it.

```
1 cd Util
2 ./ImportAssets.sh
```



There is an alternative on Linux. Move the package to the 'Import' folder and run the script 'Util/ImportAssets.sh' to extract the package.

Ingestion in a build from source

This is the method to import the props into a CARLA build from source.

The JSON file will be read to place the props inside the `Content` in Unreal Engine. Furthermore, it will create a `Package1.Package.json` file inside the package's `Config` folder. This will be used to define the props in the blueprint library, and expose them in the Python API. It will also be used if the package is exported as a standalone package.

When everything is ready, run the command.

```
1 make import
```



Make sure that the package is inside the ‘Import’ folder in CARLA.

That is all there is to know about the different ways to import new props into CARLA. If there are any doubts, feel free to post these in the forum.

9.4 Create distribution packages for assets

It is a common practice in CARLA to manage assets with standalone packages. Keeping them aside allows to reduce the size of the build. These asset packages can be easily imported into a CARLA package anytime. They also become really useful to easily distribute assets in an organized way.

- **Export a package from the UE4 Editor**
- **Import assets into a CARLA package**

Export a package from the UE4 Editor

Once assets are imported into Unreal, users can generate a **standalone package** for them. This will be used to distribute the content to CARLA packages such as 0.9.8.

To export packages, simply run the command below.

```
1 make package ARGS="--packages=Package1,Package2"
```

This will create a standalone package compressed in a `.tar.gz` file for each of the packages listed. The files will be saved in `Dist` folder on Linux, and `/Build/UE4Carla/` on Windows.



As an alternative, the [Docker method](tuto A add map.md via-docker) will create the standalone package without the need of having Unreal Engine in the system.

Import assets into a CARLA package

A standalone package is contained in a `.tar.gz` file. The way this is extracted depends on the platform.

- **On Windows** extract the compressed file in the main root CARLA folder.
- **On Linux** move the compressed file to the `Import` folder and run the following script.

```
1 cd Import  
2 ./ImportAssets.sh
```



Standalone packages cannot be directly imported into a CARLA build. Follow the tutorials to import [props](tuto A add props.md), [maps](tuto A add map.md) or [vehicles](tuto A add vehicle.md).

That sums up how to create and use standalone packages in CARLA. If there is any unexpected issue, feel free to post in the forum.

9.5 Map customization tools

There are several tools provided by the CARLA team that allow users to edit maps at will from the Unreal Editor. This tutorial introduces the most relevant tools, according to their purpose.

- **Serial meshes**
 - BP_RepSpline
 - BP_Spline
 - BP_Wall
 - BP_SplinePowerLine

- **Procedural buildings**
 - Building structure
 - Structure modifications
- **Weather customization**
 - BP_Weather
 - BP_Sky



This tutorial only applies to users that work with a build from source, and have access to the Unreal Editor.

Add serial meshes

There is a series of blueprints in `Carla/Blueprints/LevelDesign` that are useful to add props aligned in one direction. All of them use a series of meshes, and a Bezier curve that establishes the path where the props are placed.

There are differences between them, that make them fit specific purposes. However, they all work the same way. Only the parametrization presents differences.

- **Initialize the series.** The blueprints need a **Static Mesh** that will be repeated. Initially, only one element will appear, standing on the starting point of a Bezier curve with two nodes, beginning and ending.
- **Define the path.** Press **Alt** over one of the nodes, to create a new one and modify the curve. A new mesh will appear on every node of the curve, and the space between nodes will be filled with elements **separated by a distance** measure. Adjust the curve using the weights on every node.
- **Customize the pattern.** This is where the blueprints present differences between each other.



New props will probably interfere with the mesh navigation. If necessary, rebuild that as explained [here](tuto A add map.md generate-pedestrian-navigation) after doing these changes.

BP_RepSpline The blueprint **BP_RepSpline** adds **individual** elements along the path defined by a Bezier curve. There are some specific parameters that change the serialization.

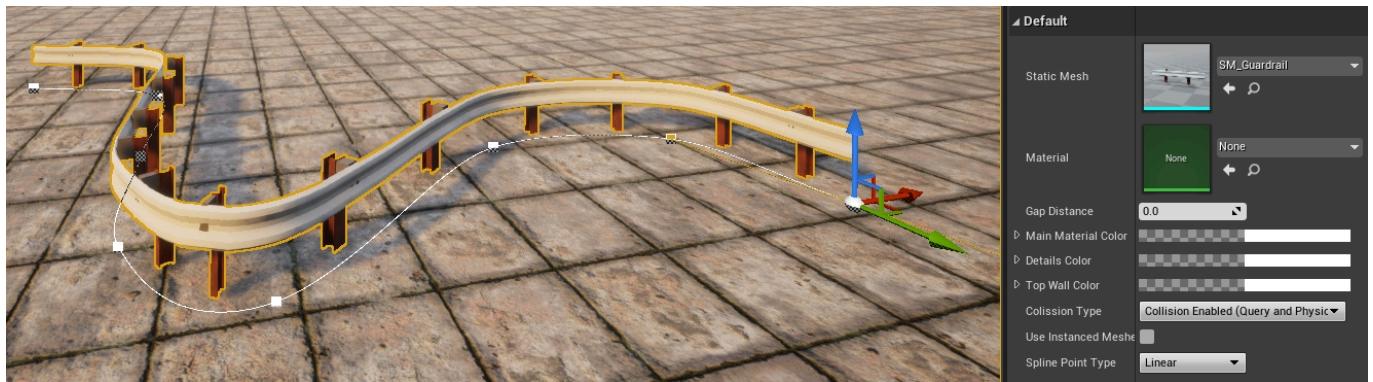
- **Distance between** — Set the distance between elements.
- **Offset rotation** — Set a fixed rotation for the different axis.
- **Random rotation** — Set a range of random rotations for the different axis.
- **Offset translation** — Set a range of random locations along the different axis.
- **Max Number of Meshes** — Set the maximum amount of elements that will be placed between nodes of the curve.
- **World aligned ZY** — If selected, the elements will be vertically aligned regarding the world axis.
- **EndPoint** — If selected, an element will be added in the ending node of the curve.
- **Collision enabled** — Set the type of collisions enabled for the meshes.



BP_RepSpline example.

BP_Spline The blueprint **BP_Spline** adds **connected** elements **strictly** following the path defined by a Bezier curve. The mesh will be warped to fit the path created.

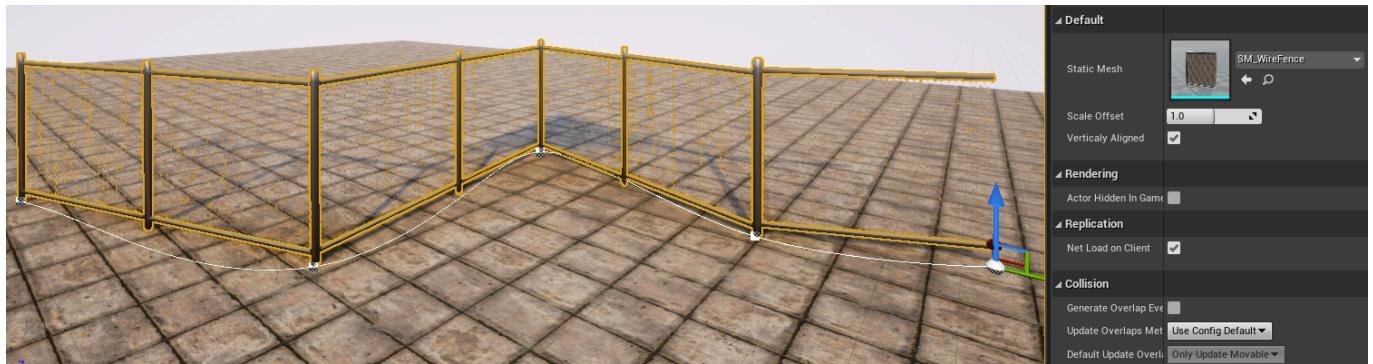
- **Gap distance** — Add a separation between elements.



BP_Spline example.

BP_Wall The blueprint **BP_Wall** adds **connected** elements along the path defined by a Bezier curve. The mesh will not be warped to fit the curve, but the nodes will be respected.

- **Distance between** — Set the distance between elements.
- **Vertically aligned** — If selected, the elements will be vertically aligned regarding the world axis.
- **Scale offset** — Scale the length of the mesh to round out the connection between elements.



BP_Wall example.

BP_SplinePowerLine The blueprint **BP_SplinePowerLine** adds **electricity poles** along the path defined by a Bezier curve, and **connects them with power lines**.

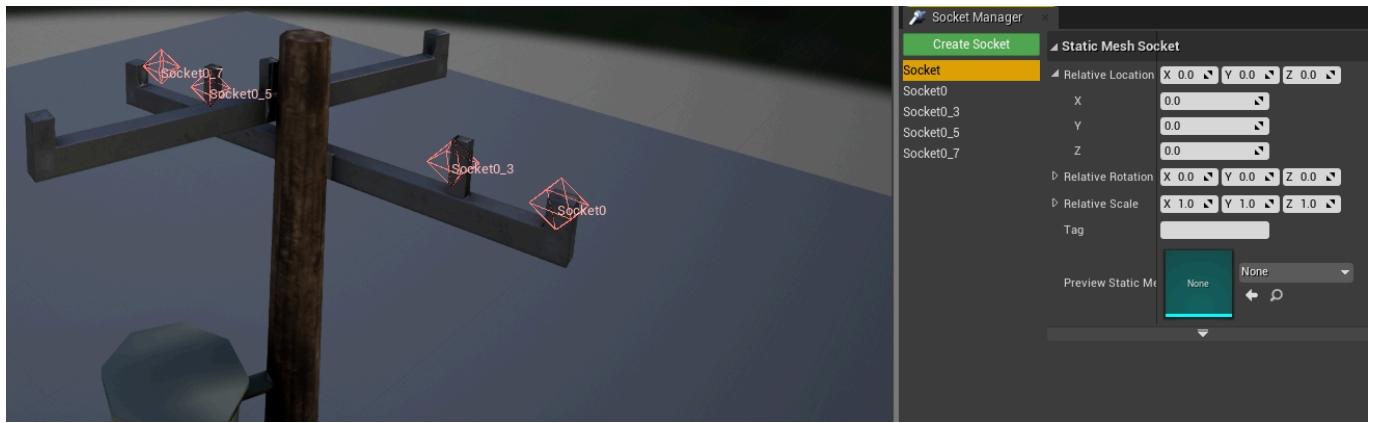
This blueprint can be found in **Carla/Static/Pole**. This blueprint allows to set an **array of meshes** to repeat, to provide variety.



BP_SplinePowerLine example.

The power line that connects the pole meshes can be customized.

- **Choose the mesh** that will be used as wire.
- **Edit the tension** value. If 0, the power lines will be straight. The bigger the value, the looser the connection.
- **Set the sockets**. Sockets are empty meshes that represent the connection points of the power line. A wire is created from socket to socket between poles. The amount of sockets can be changed inside the pole meshes.



Visualization of the sockets for BP_SplinePowerLine.



The amount of sockets and their names should be consistent between poles. Otherwise, visualization issues may arise.

Procedural buildings

The blueprint **BP_Procedural_Building** in `Content/Carla/Blueprints/LevelDesign` creates a realistic building using key meshes that are repeated along the structure. For each of them, the user can provide an array of meshes that will be used at random for variety. The meshes are only created once, and the repetitions will be instances of the same to save up costs.



Blueprints can be used instead of meshes, to allow more variety and customization for the building. Blueprints can use behaviour trees to set illumination inside the building, change the materials used, and much more.

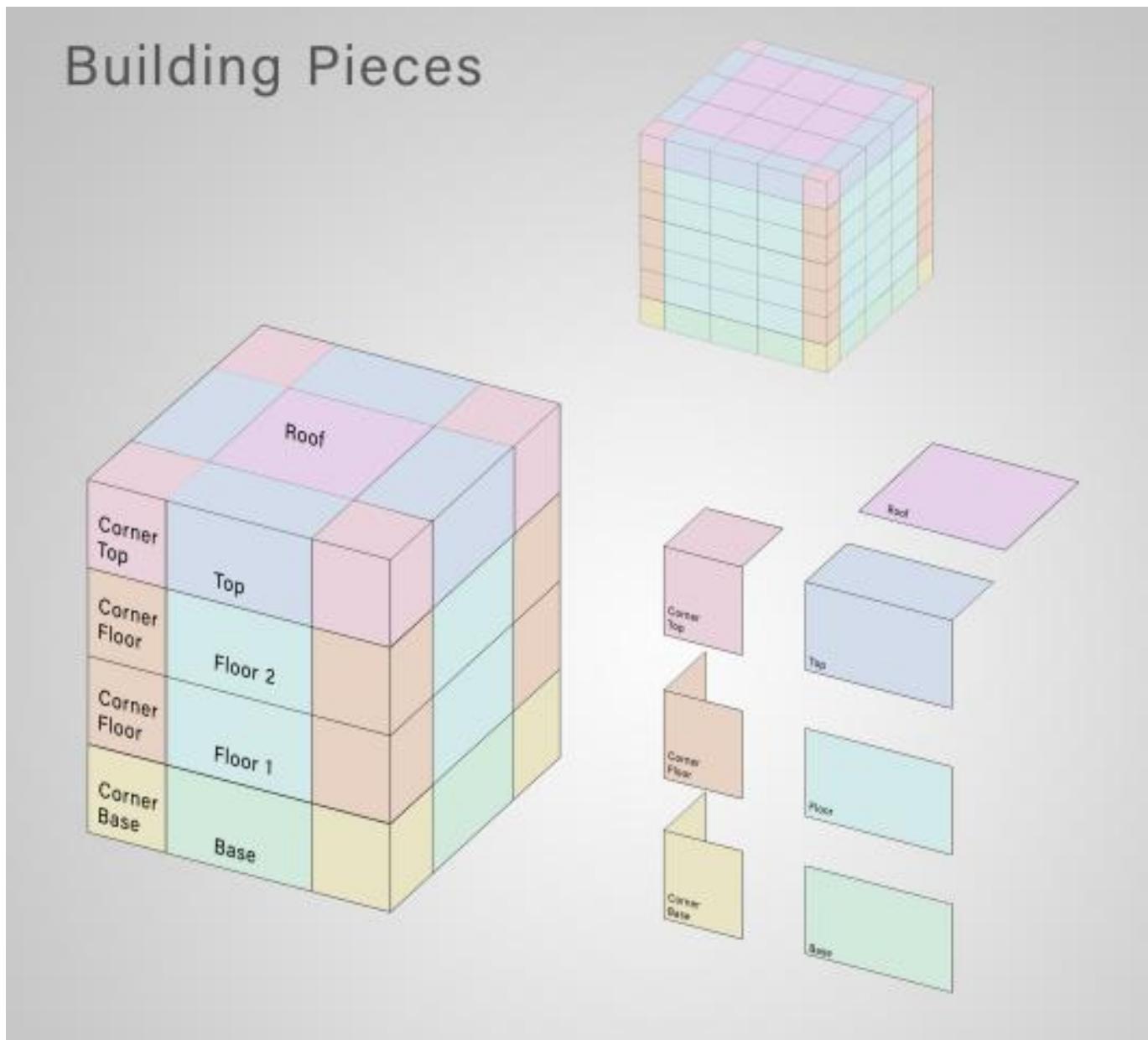
Building structure The key meshes will be updated everytime a change is made, and the building will disappear. Enable **Create automatically** or click on **Create Building** to see the new result.

These key meshes can be perceived as pieces of the building's structure. They can be grouped in four categories.

- **Base** — The ground floor of the building.
- **Body** — The middle floors of the building.
- **Top** — The highest floor of the building.
- **Roof** — Additional mesh that used to fill the spaces in the top floor.

For each of them, except the **Roof**, there is a mesh to fill the center of the floor, and a **Corner** mesh that will be placed on the sides of the floor. The following picture represents the global structure.

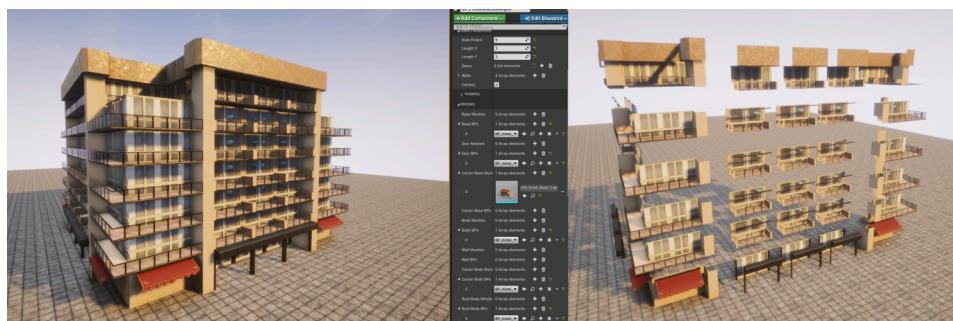
Building Pieces



Visualization of the building structure.

The **Base parameters** set the dimensions of the building.

- **Num Floors** — Floors of the building. Repetitions of the **Body** meshes.
- **Length X and Length Y** — Area of the building. Repetitions of the central meshes for each side of the building.



Example of BP_Procedural_Building.

Structure modifications There are some additional options to modify the general structure of the building.

- **Disable corners** — If selected, no corner meshes will be used.

- **Use full blocks** — If selected, the structure of the building will use only one mesh per floor. No corners nor repetitions will appear in each floor.
- **Doors** — Meshes that appear in the ground floor, right in front of the central meshes. The amount of doors and their location can be set. 0 is the initial position, 1 the next base repetition, and so on.
- **Walls** — Meshes that substitute one or more sides of the building. For example, a plane mesh can be used to paint one side of the building.



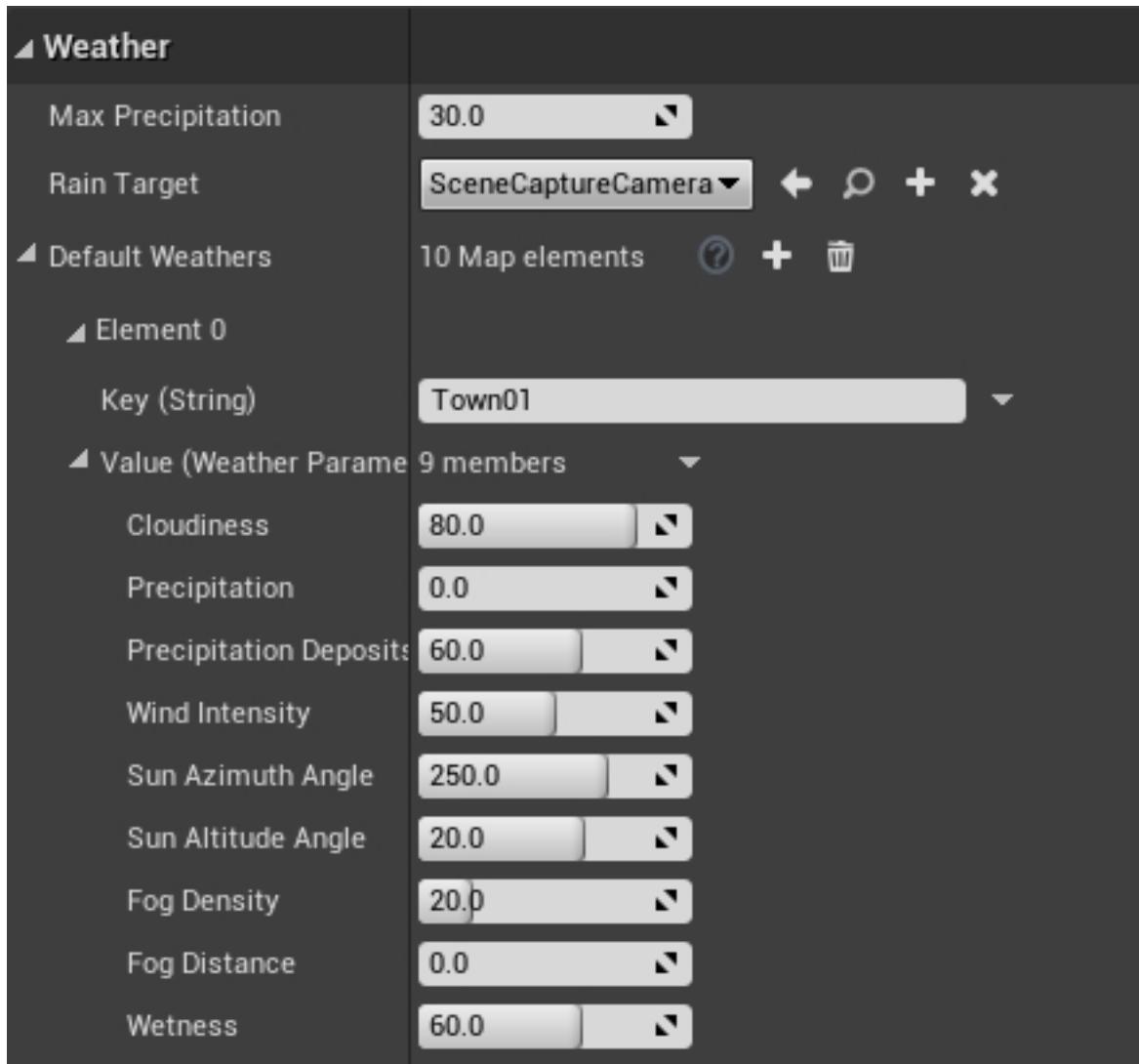
On the left, a building with no corners and one door. On the right, a building with a wall applied to one side of the building. The wall is a texture with no fire escape.

Weather customization

The weather can be easily customized by the users in CARLA using the PythonAPI. However, there is some configuration that users can do in order to set the default weather for a map. The weather parameters available for configuration by the following blueprints, are the same accessible from the API. These are described here.

BP_Weather This blueprint is loaded into the world when the simulation starts. It contains the default weather parameters for every map, and these can be modified at will.

1. Open the **BP_Weather** in Content/Carla/Blueprints/Weather.
2. Go to the **Weather** group in the blueprint.
3. Choose the desired town and modify the parameters.



Array containing default weather parameters for every CARLA map. Town01 opened.

BP_Sky This blueprint groups all the weather parameters. It can be loaded into the scene when there is no CARLA server running, and used to test different configurations before setting a new default weather.

1. Find the BP_Sky in Content/Carla/Blueprints/Weather.
2. Load the blueprint in the scene. Drag it into the scene view.
3. Edit the weather parameters. The weather in the scene will be updated accordingly.



If more than one blueprint is loaded into the scene, the weather will be duplicated with weird results, such as having two suns.

That is all there is so far, regarding the different map customization tools available in CARLA.

Open CARLA and mess around for a while. If there are any doubts, feel free to post these in the forum.

9.6 Material customization

The CARLA team prepares every asset to run under certain default settings. However, users that work in a build from source can modify these to best suit their needs.

- Car materials
- Customize car materials
 - Exterior properties
- Building material

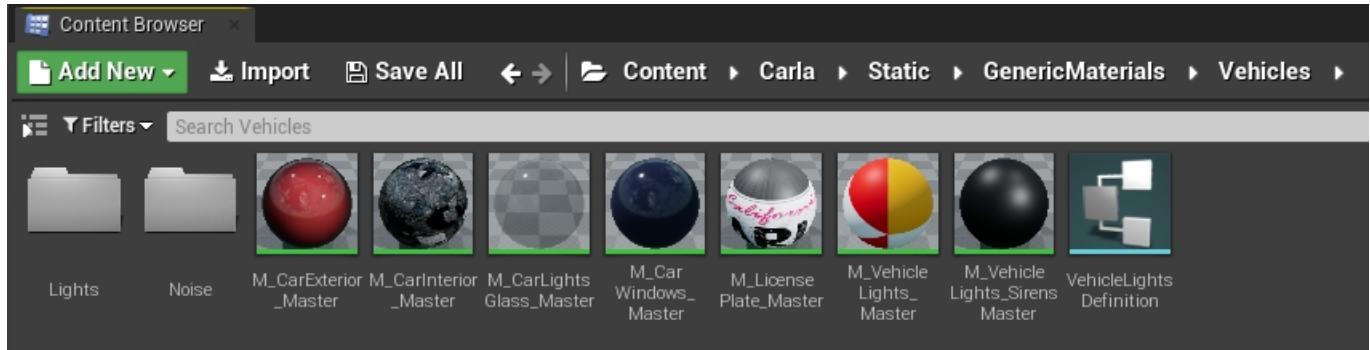
- **Customize a building material**
- **Customize the road**
 - Create a group material
 - Change the appearance of the road
 - Update the appearance of lane markings



This tutorial only applies to users that work with a build from source, and have access to the Unreal Editor.

Car materials

In CARLA, there is a set of master materials that are used as templates for the different parts of the vehicle. An instance of these is created for each vehicle model, and then changed to the desired result. The master materials can be found in `Content/Carla/Static/GenericMaterials/Vehicles`, and these are the following.

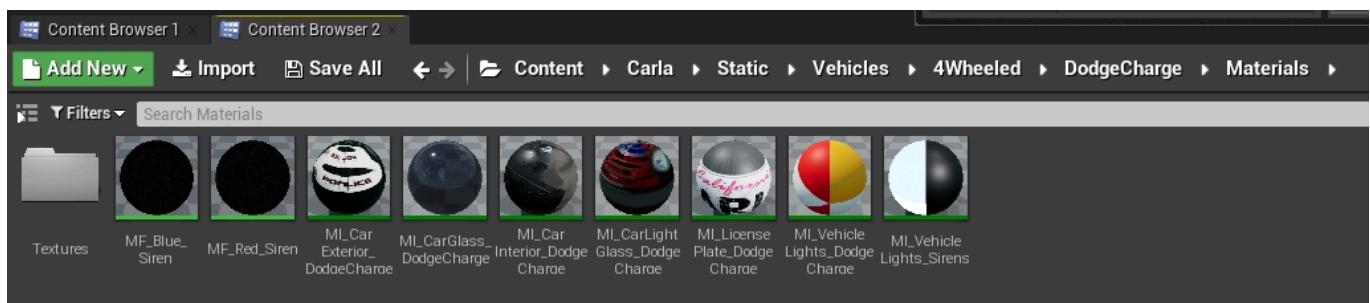


Master materials applied to cars.

- **M_CarExterior_Master** — Material applied to the body of the car.
- **M_CarInterior_Master** — Material applied to the inside of the car.
- **M_CarLightsGlass_Master** — Material applied to the glass covering car lights.
- **M_CarWindows_Master** — Material applied to the windows.
- **M_CarLicensePlate_Master** — Material applied to the license plate.
- **M_VehicleLights_Master** — Material applied to the car lights as an emissive texture.
- **M_VehicleLights_Sirens_Master** — Material applied to the sirens, if applicable.

Customize car materials

Create instances of the master materials and store them in the corresponding folder for the new model. Here is an example of the instances created for the police car available in the blueprint library, `vehicle.dodge_charger.police`.



Instanced materials for the police car blueprint.

Generic documentation for materials and how to work with them can be found in the UE Docs. All the materials can be modified to a great extent, but only the exterior one has properties worth mentioning. Others have certain properties that can be changed, such as opacity and color in glass materials, but it is not recommended to do so, except for specific purposes.

Exterior properties The exterior material is applied to the body of the car, and it is the one that can be customized the most.

- **Base color** — Base color of the bodywork.
- **Tint shade** — Tint color which visibility varies depending on the angle of visualization.



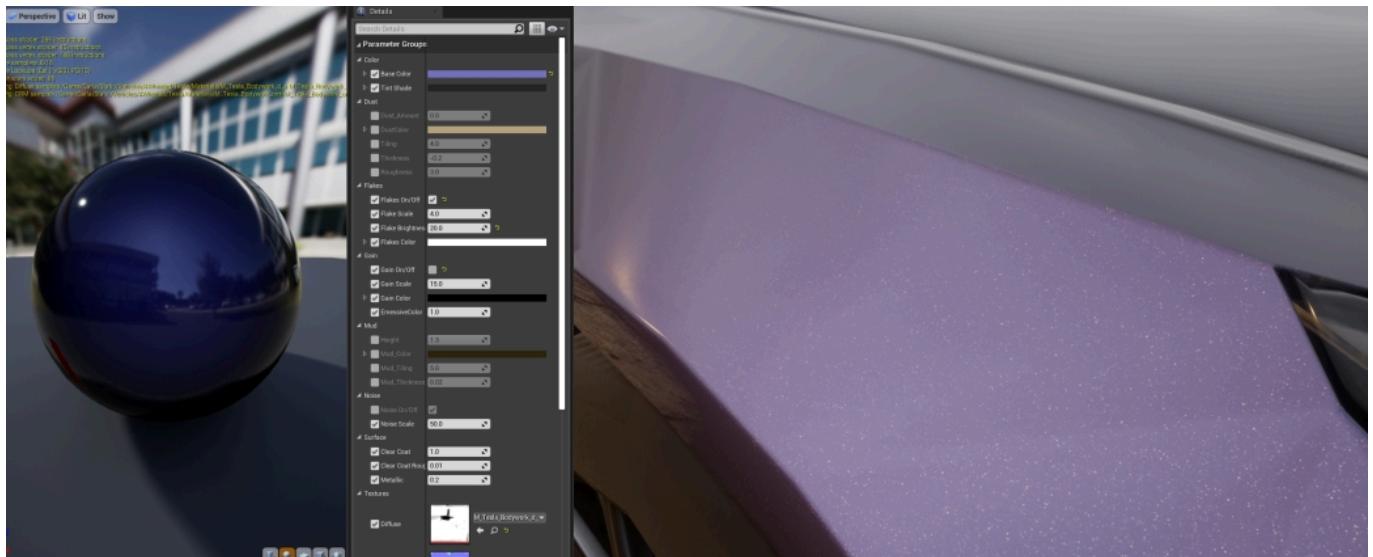
Red car with pink tint. On the left, tint is disabled, on the right, enabled.

- **Dust** — A texture of dirt applied to the car. Dust is meant to pile on top of the geometry, and it is barely noticeable in the bottom parts. If the geometry is rotated, the dust will appear on the parts of the vehicle that are on top.
 - **Amount** — Opacity of the texture.
 - **Color** — Base color of the dust texture.
 - **Tiling** — Size and repetition of the dust texture pattern.
 - **Thickness** — Density of the dust.
 - **Roughness** — Decrease of the car's metallic reflections due to dust.



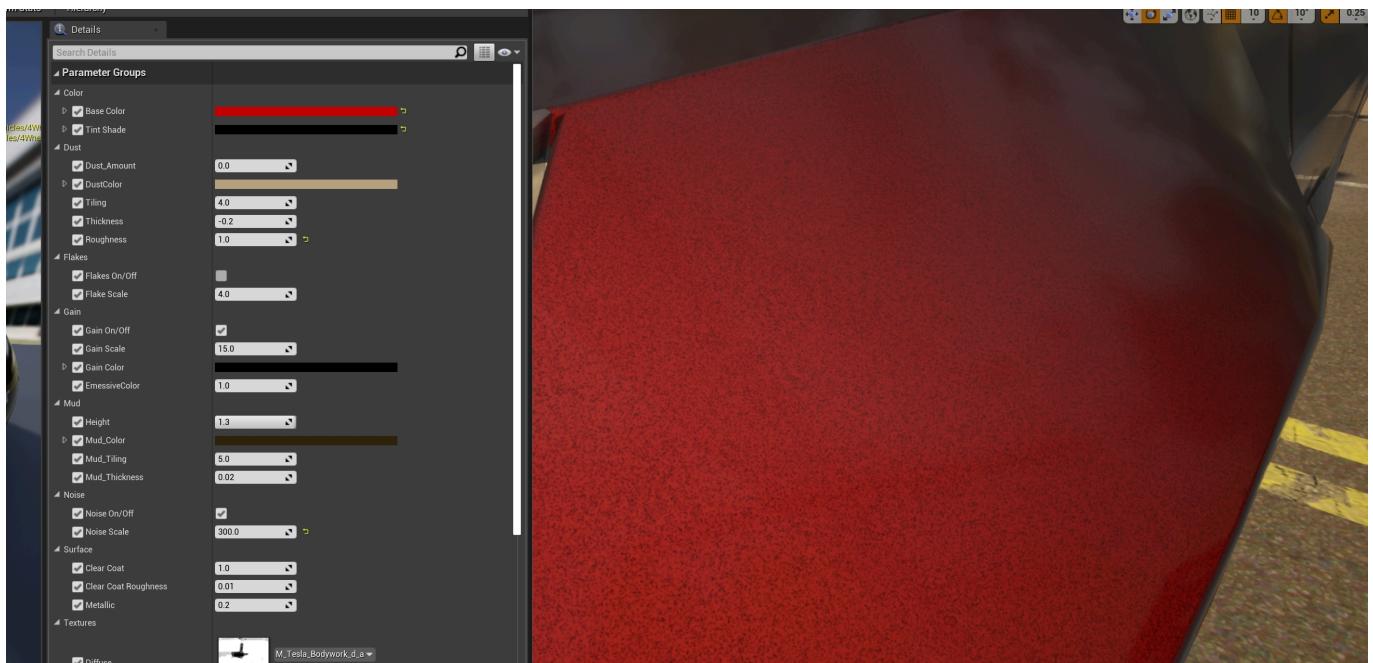
Dust property in a car's material.

- **Flakes** — Sparkling flakes to the metallic paint of the car.
 - **On/Off** — Enables or disables the feature.
 - **Scale** — Size of the flakes.
 - **Brightness** — Intensity of the sparkle.
 - **Color** — Base color of the particles.



Flakes property in a car's material.

- **Gain** — Noise to the base paint of the car.
 - **On/Off** — Enables or disables the feature.
 - **Scale** — Size of the gain.
 - **Color** — Base color of the gain.



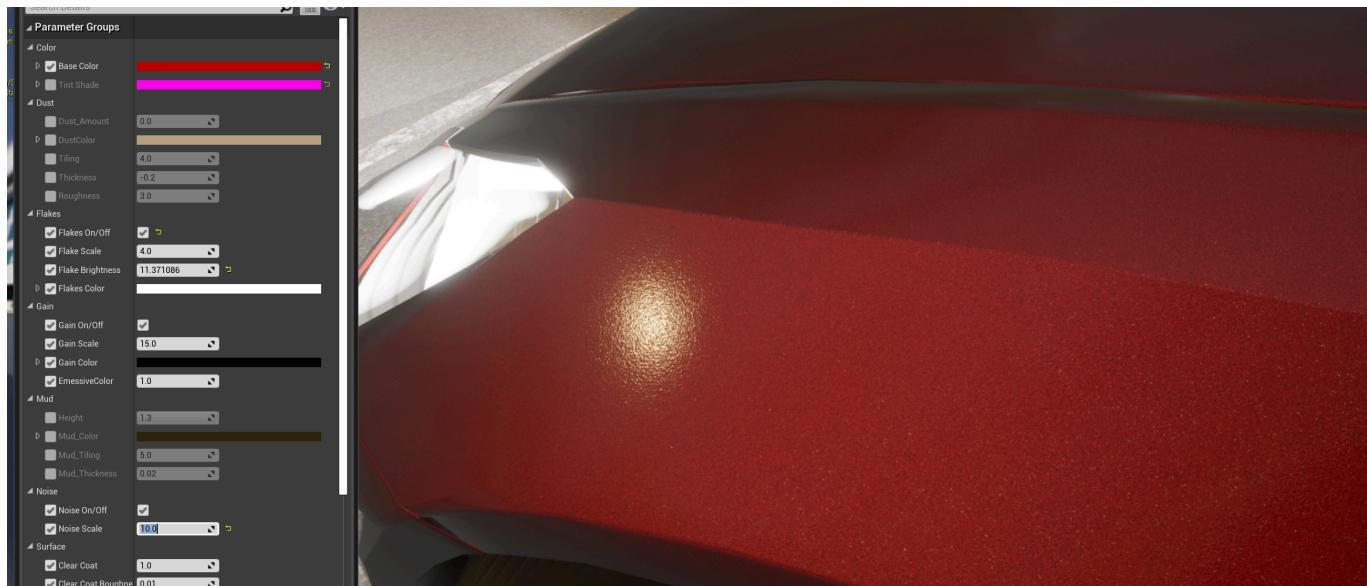
Gain property in a car's material.

- **Mud** — A texture of mud applied to the car. Mud appears from the bottom to top of the car.
 - **Height** — Portion of the car where mud appears.
 - **Mud_Color** — Base color of the mud texture.
 - **Mud_Tiling** — Size and repetition of the mud texture pattern.
 - **Mud_Thickness** — Density of the mud.



Mud property in a car's material.

- **Noise** — Noise applied to the normal of the material. Creates an orange peel effect.
 - **On/Off** — Enables or disables the feature.
 - **Scale** — Size of the bumps created by the alteration of the normal map.



Noise property in a car's material.

- **Surface** — Gloss and transparent coating applied to the vehicle's paint. This last step in automotive paint.
 - **ClearCoat** — Opacity of the coating.
 - **ClearCoat_Brightness** — Glossiness of the resulting material.
 - **ClearCoat_Metallic** — Reflection of the resulting material.



Visualization of the Surface coating applied to a material.

Building materials

The materials applied to buildings are made of four basic textures that are combined to determine the basic properties of the material.

- **Diffuse** — Contains the basic painting of the material.
 - RGB — Channels with the base colors.
 - Alpha — This channel defines a mask that allows to modify the color of the portions in white. This is useful to create some variations from the same material.
- **ORME** — Maps different properties of the material using specific channels.
 - Ambient occlusion — Contained in the R channel.
 - Roughness — Contained in the G channel.
 - Metallic map — Contained in the B channel.
 - Emissive mask — Contained in the Alpha channel. This mask allows to change the emissive color and intensity of the portions in white.
- **Normal** — Contains the normal map of the material.
 - RGB — The normal map information.
- **Emissive** — If applicable, this texture is used to set the emissive base colors of the texture.
 - RGB — Color information for the emissive elements in the texture.

Customize a building material

Similarly to car materials, a building material can be greatly changed if desired, but it is only recommended if the user has some expertise with Unreal Engine. However, there is some customization available for the two main shaders that buildings use.

- **Glass shader** — `M_CarWindows_Master`.
 - Opacity — Enable color changes on the white area on the **Diffuse Alpha** texture.
 - Color — Tint to be applied based on the white area on the **Diffuse Alpha** texture.
- **Building shader** — `M_Building_Master`
 - Change Color — Enable color changes on the white area on the **Diffuse Alpha** texture.
 - Color — Tint to be applied based on the white area on the **Diffuse Alpha** texture.
 - Emissive Texture — Enable the usage of an **Emissive** texture.
 - EmissiveColor — Tint to be applied based on the white area on the **ORME Emissive mask** texture.
 - Emissive atenuance — Factor that divides the intensity stated in **BP_Lights** to obtain proper emissive values.
 - RoughnessCorrection — Changes the intensity of the roughness map.
 - MetallicCorrection — Changes the intensity of the metallic map.
 - NormalFlatness — Changes the intensity of the normal map.

Customize the road

RoadPainter is a tool that uses OpenDRIVE information to paint roads. To be able to do so, a blueprint is used, which uses a master material, a render target of the road as canvas, and additional decals and meshes. The master material groups a collection of materials that will be used by the blueprint, using brushes to blend their application. This makes for an easy way to change drastically the appearance of the road. The initial geometry of the road is painted like a canvas. There is no need to apply photometry techniques nor consider the UVs of the geometry. The result is achieved simply by blending textures and creating masks.

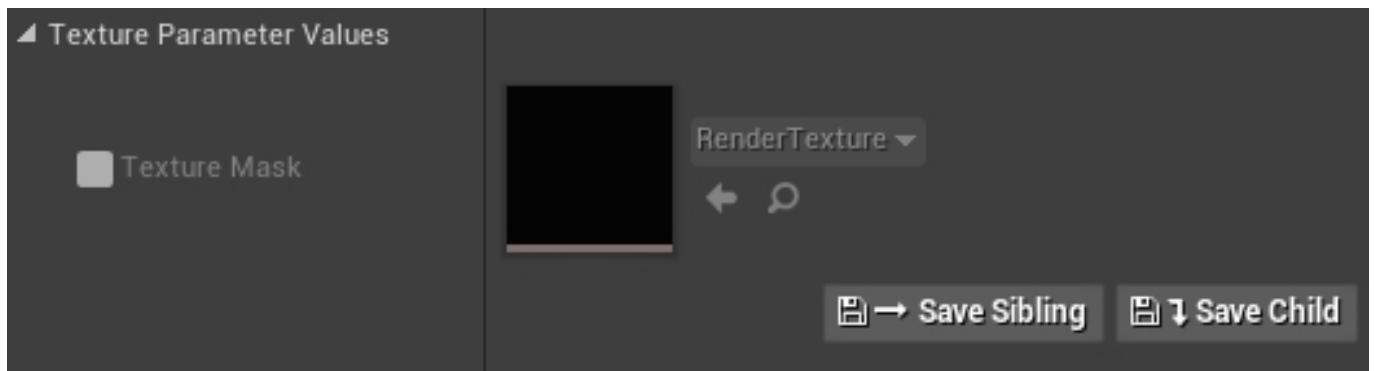
Create a group material First of all, a group material will be created. This will contain the instances of the materials the road will be painted with.

1. Create an instance of the RoadMaster material. It can be found in Game/Carla/Static/GenericMaterials/RoadPaint



Panel to set materials to be applied on the road.

2. Set the **RenderTarget**. Create a render target for the road map that is being used. In **Texture Parameter Values** enable **Texture mask** and add the texture.



Panel where the Render Target should be set.

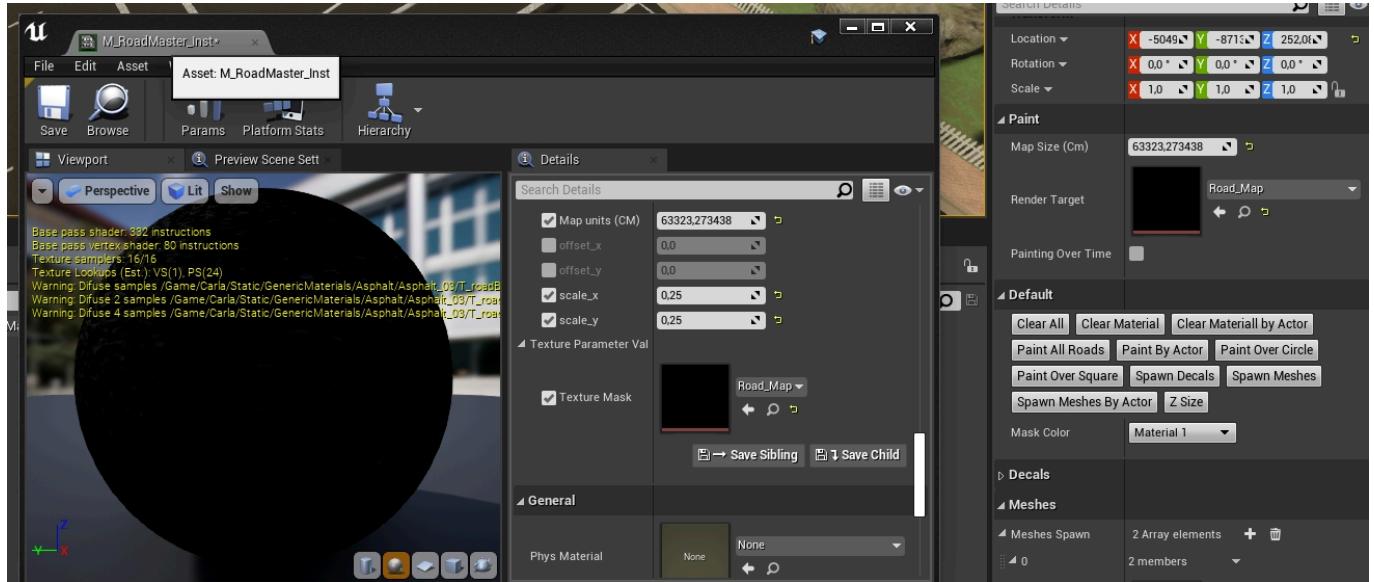
3. Set the **map size**. In **Scalar Parameter Values** the field **Map units (CM)**. If the size is not known, use a temporary value. The real size of the map can be retrieved later, when using the blueprint.
4. Choose the materials to be applied. There is space for one base material and three additional materials that will be used to paint over the base one. The painting information will be stored in the RGB channels of a RenderTarget, one channel per material. Add them to the fields **Base Material**, **Material 1**, **Material 2**, and **Material 3**.

Change the appearance of the road The appearance of the road is changed using a blueprint provided by CARLA. The blueprint will paint the road using the materials passed, combined with some brush settings. Additionally, decals and meshes can be added so that the final result is more detailed. This tool takes into account road information and will paint the elements using the orientation of the lane, unless an offset is stated.



This tool does not interfere with the weather settings that change the road's appearance, such as wetness or precipitation.

- Create an instance of the RoadPainter blueprint.** It can be found in `Carla/Blueprints/LevelDesign`. This blueprint determines how is the road being painted, and how are the materials in `RoadMaster` being used.
- Set the RenderTarget and the Map size.** In the Paint category. These must be the same as in the `RoadMaster` material.



Panels in group material and road blueprint where the Render Target and Map Size values should match.

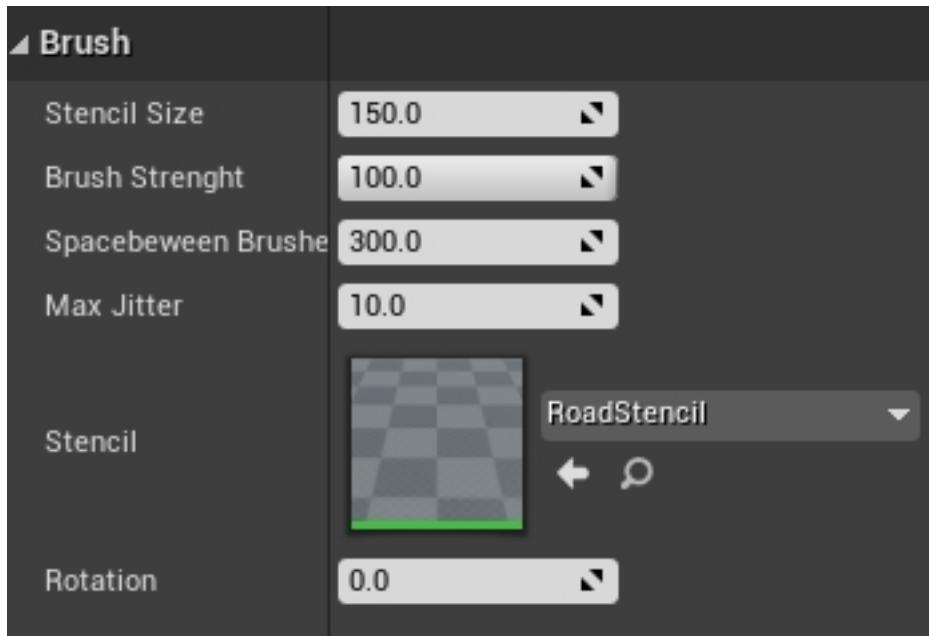


The ‘Z-size’ option in the ‘Default’ panel will give you the exact size of the map. Increase this by a little and make sure both, the blueprint and the master material have the same value.

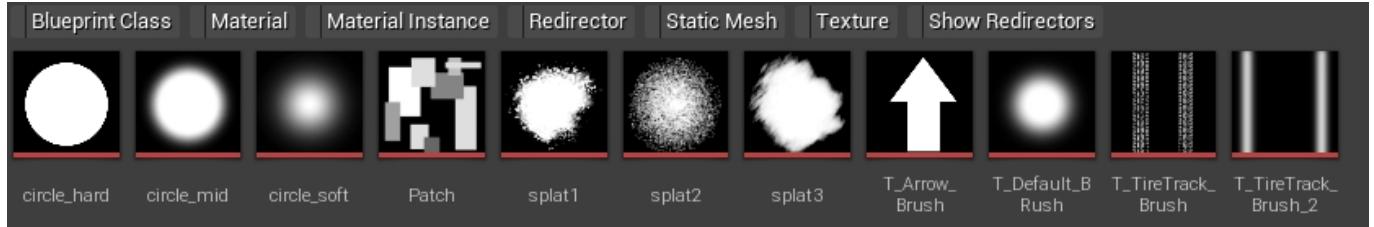
- Apply the group material.** Select all the road meshes and apply the instance.

- For each material, set the brush to be used.** There are different brushes available in `GenericMaterials/RoadStencil/`. The materials will be applied over the road using the brush and parameters stated here.

- **Stencil size** — Size of the brush.
- **Brush strength** — Roughness of the outline.
- **Spacebetween Brushes** — Distance between strokes.
- **Max Jitter** — Size variation of the brush between strokes.
- **Stencil** — The brush to be used.
- **Rotation** — Rotation applied to the stroke.



Brush panel.



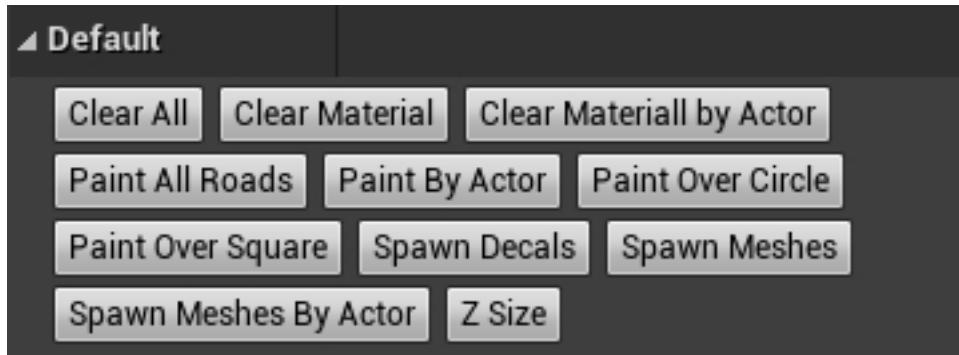
Different types of brushes.

5. For each material, apply it to the desired portions of the road. In the Default section, there is a series of buttons to choose how materials are applied.

- Paint all roads — Applies the brush to all the road.
- Paint by actor — Paints a specific actor being selected.
- Paint over circle — Paints using a circular pattern, useful to provide variation.
- Paint over square — Paints using a square pattern, useful to provide variation.

This section also contains options to erase the changes applied.

- Clear all — Erases all the painting applied by the blueprint.
- Clear materials — Remove the materials currently active.
- Clear material by actor — Removes material closest to the actor selected.



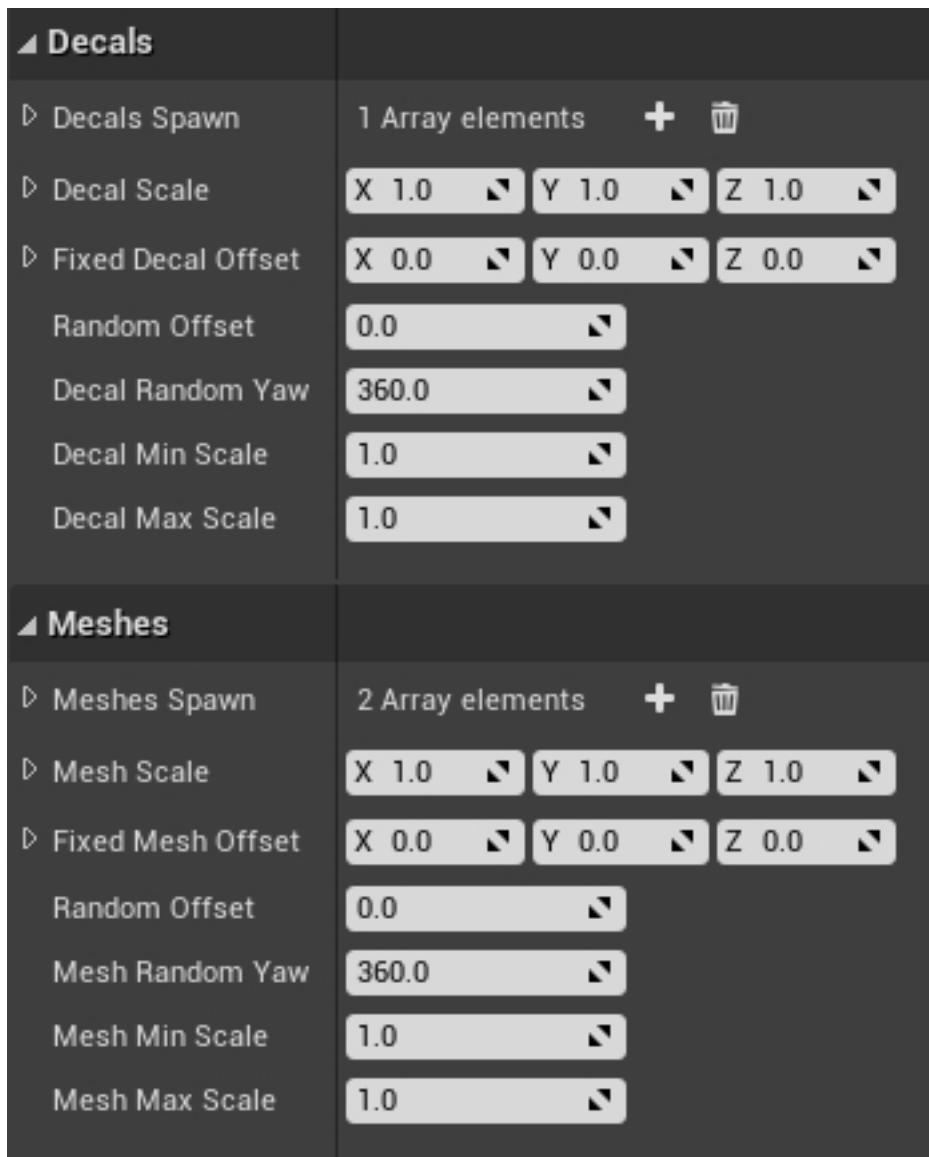
Different painting and erasing options.



Move the editor view a little for the changes to be visible.

5. Add decals and meshes. Add the elements to the corresponding array and set some basic parameters for how should these be spawned over the road.

- Decal/Mesh Scale — Scale of the decal/mesh per axis.
- Fixed Decal/Mesh Offset — Deviation from the center of the lane per axis.
- Decal/Mesh Random Offset — Max deviation from the center of the lane per axis.
- Decal/Mesh Random Yaw — Max random yaw rotation.
- Decal/Mesh Min Scale — Minimum random scale applied to the decal/mesh.
- Decal/Mesh Max Scale — Max random scale applied to the decal/mesh.



Decals and Meshes panels.

6. **Spawn the decals and meshes.** Use the buttons **Spawn decals** and **Spawn meshes** to do so.
7. **Play around.** Try different settings, materials and brushes to obtain completely different results. This tool allows to create different presets for the road appearance and changing between them simply by changing the group material and updating a few settings.

Here is an example of the how the appearance of the road changes along the previous steps.



Example of base road material.



Example after material 1 is applied.



Example after material 2 is applied.



Example after material 3 is applied.



Example after decals are applied.



Example after meshes are applied.

Update the appearance of lane markings After the road's appearance has been customized, the lane markings should be updated accordingly to get a realistic look. The process to do so, is really simple.

1. Create a copy of the group material. This has to be the same as it was for the road. **2. Select the lane marking meshes.** **3. Mark them as lane markings.** In Static Switch Parameter Values enable LaneMark. This will update the lane markings to be consistent to the painting applied to the road. **4. Choose the color of the lane marking.** In Texture set the LaneColor to the value desired. This will set the lane markings selected to have a base paint of the stated color.

That is a wrap on the most remarkable ways users can customize the materials of vehicles and buildings.

Any doubts that may arise are more than welcomed in the forum.

9.7 How to model vehicles

- **4-wheeled Vehicles**
 - Modelling
 - Naming materials
 - Texturing
 - Rigging
 - LODs

4-Wheeled Vehicles

Modelling Vehicles must have a minimum of 10.000 and a maximum of 17.000 Tris approximately. We model the vehicles using the size and scale of actual cars. The bottom part of the vehicle consists of a plane adjusted to the Bodywork.

The vehicle must be divided in 6 materials:

1. **BodyWork:** The Bodywork includes the chassis, doors, car handle, and front and back parts of the vehicle. The BodyWork material is controlled by Unreal Engine. You can add logos and some details, but remember, all the details will be painted by Unreal using the same color. Use the alpha channel if you want to paint details with a different color.
2. **Wheels:** Model the Wheels with hubcaps and add details to the tire with Substance. In the UV, add the tires and the hubcaps separately.
3. **Interior:** The Interior includes the seats, the steering wheel, and the bottom of the vehicle. You don't need to add much detail here.
4. **Details:** Lights, logos, exhaust pipes, protections, and grille.
5. **Glass:** Light glasses, windows, etc. This material is controlled by Unreal.
6. **LicencePlate:** Put a rectangular plane with this size 29-12 cm, for the licence Plate. We assign the license plate texture.

Naming materials

- M(Material)_“CarName”_Bodywork(part of car)
- M_“CarName”_Wheel
- M_“CarName”_Interior
- M_“CarName”_Details
- M_“CarName”_Glass
- M_“CarName”_LicencePlate

Texturing The size of the textures is 2048x2048.

- T_“CarName”_PartOfMaterial_d (BaseColor)
- T_“CarName”_PartOfMaterial_n (Normal)
- T_“CarName”_PartOfMaterial_orm (OcclusionRoughnessMetallic)
- **EXAMPLE:** Type of car Tesla Model 3

TEXTURES * T_Tesla3_BodyWork_d * T_Tesla3_BodyWork_n * T_Tesla3_BodyWork_orm

MATERIAL * M_Tesla3_BodyWork

Rigging The easiest way is to copy the “General4WheeledVehicleSkeleton” present in our project, either by exporting it and copying it to your model or by creating your skeleton using the same bone names and orientation.

The model and every bone must be oriented towards positive X axis with the Z axis facing upwards.

Bone Setup:

Vhehicle_Base: The origin point of the mesh, place it in the point (0,0,0) of the scene.

- Wheel_Front_Left: Set the joint's position in the middle of the Wheel.
- Wheel_Front_Right: Set the joint's position in the middle of the Wheel.
- Wheel_Rear_Left: Set the joint's position in the middle of the Wheel.
- Wheel_Rear_Right: Set the joint's position in the middle of the Wheel.

LODs All vehicle LODs must be made in Maya or other 3D software. Because Unreal does not generate LODs automatically, you can adjust the number of Tris to make a smooth transitions between levels.

- *Level 0* - Original
- *Level 1* - Deleted 2.000/2.500 Tris (*Do not delete the interior and steering wheel*)
- *Level 2* - Deleted 2.000/2.500 Tris (*Do not delete the interior*)
- *Level 3* - Deleted 2.000/2.500 Tris (*Delete the interior*)
- *Level 4* - Simple shape of a vehicle.

10 Tutorials (developers)

10.1 How to upgrade content

Our content resides on a separate Git LFS repository. As part of our build system, we generate and upload a package containing the latest version of this content tagged with the current date and commit. Regularly, we upgrade the CARLA repository with a link to the latest version of the content package. This document contains the manual steps necessary to update this link to the latest version.

1. **Copy the tag of the content package you wish to link.** This tag can be found by looking at the package name generated in the artifacts section of the latest Jenkins build, e.g., 20190617_086f97f.tar.gz.
2. **Paste the tag in ContentVersions.txt.** Edit ContentVersions.txt by pasting the tag at the end of the file, e.g. Latest: 20190617_086f97f (without the .tar.gz part).
3. **Open a Pull Request.** Commit the changes and open a new Pull Request.

10.2 How to add a new sensor

This tutorial explains the basics for adding a new sensor to CARLA. It provides the necessary steps to implement a sensor in Unreal Engine 4 (UE4) and expose its data via CARLA's Python API. We'll follow all the steps by creating a new sensor as an example.

- **Prerequisites**
- **Introduction**
- **Creating a new sensor**
 - 1- Sensor actor
 - 2- Sensor data serializer
 - 3- Sensor data object
 - 4- Register your sensor
 - 5- Usage example
- **Appendix**
 - Reusing buffers
 - Sending data asynchronously
 - Client-side sensors

Prerequisites

In order to implement a new sensor, you'll need to compile CARLA source code, for detailed instructions on how to achieve this see [Building from source](#).

This tutorial also assumes the reader is fluent in C++ programming.

Introduction

Sensors in CARLA are a special type of actor that produce a stream of data. Some sensors produce data continuously, every time the sensor is updated, other produce data only after certain events. For instance, a camera produces an image on every update, but a collision sensor is only triggered in the event of a collision.

Although most sensors compute their measurements in the server side (UE4), it's worth noticing that some sensors run in the client-side only. An example of such sensor is the LaneInvasion, it notifies every time a lane mark has been crossed. For further details see Appendix: Client-side sensors.

In this tutorial, we'll be focusing on server-side sensors.

In order to have a sensor running inside UE4 sending data all the way to a Python client, we need to cover the whole communication pipeline.

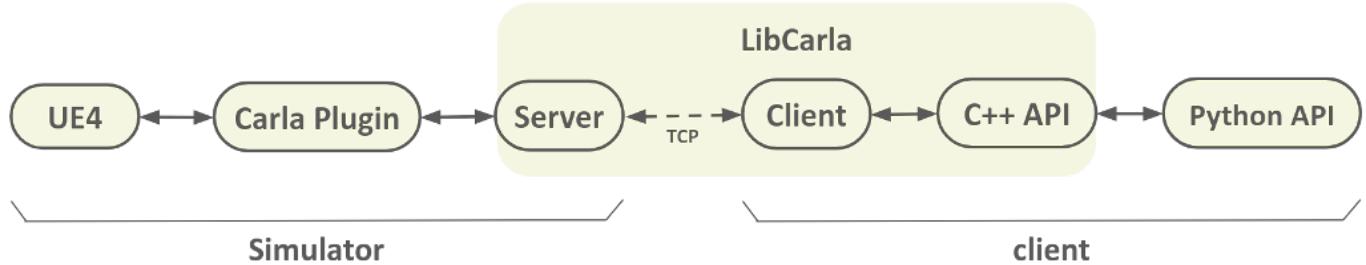


Figure 73: Communication pipeline

Thus we'll need the following classes covering the different steps of the pipeline

- **Sensor actor** Actor in charge of measuring and/or simulating data. Running in Carla plugin using UE4 framework. Accessible by the user as Sensor actor.
- **Serializer** Object containing methods for serializing and deserializing the data generated by the sensor. Running in LibCarla, both server and client.
- **Sensor data** Object representing the data generated by the sensor. This is the object that will be passed to the final user, both in C++ and Python APIs.



To ensure best performance, sensors are registered and dispatched using a sort of "compile-time plugin system" based on template meta-programming. Most likely, the code won't compile until all the pieces are present.

Creating a new sensor

[Full source code here.](#)

We're going to create a sensor that detects other actors around our vehicle. For that we'll create a trigger box that detects objects within, and we'll be reporting status to the client every time a vehicle is inside our trigger box. Let's call it *Safe Distance Sensor*.

For the sake of simplicity we're not going to take into account all the edge cases, nor it will be implemented in the most efficient way. This is just an illustrative example.

1- Sensor actor This is the most complicated class we're going to create. Here we're running inside Unreal Engine framework, knowledge of UE4 API will be very helpful but not indispensable, we'll assume the reader has never worked with UE4 before.

Inside UE4, we have a similar hierarchy as we have in the client-side, **ASensor** derives from **AActor**, and an actor is roughly any object that can be dropped into the world. **AActor** has a virtual function called **Tick** that we can use to update our sensor on every simulator update. Higher in the hierarchy we have **UObject**, base class for most of UE4 classes. It is important to know that objects deriving from **UObject** are handle via pointers and are garbage collected when they're no longer referenced. Class members pointing to **UObjects** need to be marked with **UPROPERTY** macros or they'll be garbage collected.

Let's start.



Figure 74: Trigger box

This class has to be located inside Carla plugin, we'll create two files for our new C++ class

- Unreal/CarlaUE4/Plugins/Carla/Source/Carla/Sensor/SafeDistanceSensor.h
- Unreal/CarlaUE4/Plugins/Carla/Source/Carla/Sensor/SafeDistanceSensor.cpp

At the very minimum, the sensor is required to inherit `ASensor`, and provide a static method `GetSensorDefinition`; but we'll be overriding also the `Set`, `SetOwner`, and `Tick` methods. This sensor also needs a trigger box that will be detecting other actors around us. With this and some required boiler-plate UE4 code, the header file looks like

```

1 #pragma once
2
3 #include "Carla/Sensor/Sensor.h"
4
5 #include "Carla/Actor/ActorDefinition.h"
6 #include "Carla/Actor/ActorDescription.h"
7
8 #include "Components/BoxComponent.h"
9
10 #include "SafeDistanceSensor.generated.h"
11
12 UCLASS()
13 class CARLA_API ASafeDistanceSensor : public ASensor
14 {
15     GENERATED_BODY()
16
17 public:
18     ASafeDistanceSensor(const FObjectInitializer &ObjectInitializer);
19
20     static FActorDefinition GetSensorDefinition();
21
22     void Set(FActorDescription &ActorDescription) override;
23
24     void SetOwner(AActor *Owner) override;
25
26     void Tick(float DeltaSeconds) override;
27
28 private:
29 }
```

```

31 UPROPERTY()
32 UBoxComponent *Box = nullptr;
33 };

```

In the cpp file, first we'll need some includes

```

1 #include "Carla.h"
2 #include "Carla/Sensor/SafeDistanceSensor.h"
3
4 #include "Carla/Actor/ActorBlueprintFunctionLibrary.h"
5 #include "Carla/Game/CarlaEpisode.h"
6 #include "Carla/Util/BoundingBoxCalculator.h"
7 #include "Carla/Vehicle/CarlaWheeledVehicle.h"

```

Then we can proceed to implement the functionality. The constructor will create the trigger box, and tell UE4 that we want our tick function to be called. If our sensor were not using the tick function, we can disable it here to avoid unnecessary ticks

```

1 ASafeDistanceSensor::ASafeDistanceSensor(const FObjectInitializer &ObjectInitializer)
2   : Super(ObjectInitializer)
3 {
4   Box = CreateDefaultSubobject<UBoxComponent>(TEXT("BoxOverlap"));
5   Box->SetupAttachment(RootComponent);
6   Box->SetHiddenInGame(true); // Disable for debugging.
7   Box->SetCollisionProfileName(FName("OverlapAll"));
8
9   PrimaryActorTick.bCanEverTick = true;
10 }

```

Now we need to tell Carla what attributes this sensor has, this is going to be used to create a new blueprint in our blueprint library, users can use this blueprint to configure and spawn this sensor. We're going to define here the attributes of our trigger box, in this example we'll expose only X and Y safe distances

```

1 FActorDefinition ASafeDistanceSensor::GetSensorDefinition()
2 {
3   auto Definition = UActorBlueprintFunctionLibrary::MakeGenericSensorDefinition(
4     TEXT("other"),
5     TEXT("safe_distance"));
6
7   FActorVariation Front;
8   Front.Id = TEXT("safe_distance_front");
9   Front.Type = EActorAttributeType::Float;
10  Front.RecommendedValues = { TEXT("1.0") };
11  Front.bRestrictToRecommended = false;
12
13  FActorVariation Back;
14  Back.Id = TEXT("safe_distance_back");
15  Back.Type = EActorAttributeType::Float;
16  Back.RecommendedValues = { TEXT("0.5") };
17  Back.bRestrictToRecommended = false;
18
19  FActorVariation Lateral;
20  Lateral.Id = TEXT("safe_distance_lateral");
21  Lateral.Type = EActorAttributeType::Float;
22  Lateral.RecommendedValues = { TEXT("0.5") };
23  Lateral.bRestrictToRecommended = false;
24
25  Definition.Variations.Append({ Front, Back, Lateral });
26
27  return Definition;
28 }

```

With this, the sensor factory is able to create a *Safe Distance Sensor* on user demand. Immediately after the sensor is created, the `Set` function is called with the parameters that the user requested

```

1 void ASafeDistanceSensor::Set(const FActorDescription &Description)
2 {
3     Super::Set(Description);
4
5     float Front = UActorBlueprintFunctionLibrary::RetrieveActorAttributeToFloat(
6         "safe_distance_front",
7         Description.Variations,
8         1.0f);
9     float Back = UActorBlueprintFunctionLibrary::RetrieveActorAttributeToFloat(
10        "safe_distance_back",
11        Description.Variations,
12        0.5f);
13    float Lateral = UActorBlueprintFunctionLibrary::RetrieveActorAttributeToFloat(
14        "safe_distance_lateral",
15        Description.Variations,
16        0.5f);
17
18    constexpr float M_TO_CM = 100.0f; // Unit conversion.
19
20    float LocationX = M_TO_CM * (Front - Back) / 2.0f;
21    float ExtentX = M_TO_CM * (Front + Back) / 2.0f;
22    float ExtentY = M_TO_CM * Lateral;
23
24    Box->SetRelativeLocation(FVector{LocationX, 0.0f, 0.0f});
25    Box->SetBoxExtent(FVector{ExtentX, ExtentY, 0.0f});
26 }
```

Note that the `set` function is called before UE4's `BeginPlay`, we won't use this virtual function here, but it's important for other sensors.

Now we're going to extend the box volume based on the bounding box of the actor that we're attached to. For that, the most convenient method is to use the `SetOwner` virtual function. This function is called when our sensor is attached to another actor.

```

1 void ASafeDistanceSensor::SetOwner(AActor *Owner)
2 {
3     Super::SetOwner(Owner);
4
5     auto BoundingBox = UBoundingBoxCalculator::GetActorBoundingBox(Owner);
6
7     Box->SetBoxExtent(BoundingBox.Extent + Box->GetUnscaledBoxExtent());
8 }
```

The only thing left to do is the actual measurement, for that we'll use the `Tick` function. We're going to look for all the vehicles currently overlapping our box, and we'll send this list to client

```

1 void ASafeDistanceSensor::Tick(float DeltaSeconds)
2 {
3     Super::Tick(DeltaSeconds);
4
5     TSet<AActor *> DetectedActors;
6     Box->GetOverlappingActors(DetectedActors, ACarlaWheeledVehicle::StaticClass());
7     DetectedActors.Remove(GetOwner());
8
9     if (DetectedActors.Num() > 0)
10    {
11        auto Stream = GetDataStream(*this);
12        Stream.Send(*this, GetEpisode(), DetectedActors);
13    }
14 }
```



In production-ready sensors, the ‘Tick’ function should be very optimized, specially if the sensor sends big chunks of data. This function is called every update in the game thread thus significantly affects the performance of the simulator.

Ok, a couple of things going on here that we haven’t mentioned yet, what’s this stream?

Every sensor has a data stream associated. This stream is used to send data down to the client, and this is the stream you subscribe to when you use the `sensor.listen(callback)` method in the Python API. Every time you send here some data, the callback on the client-side is going to be triggered. But before that, the data is going to travel through several layers. First of them will be the serializer that we have to create next. We’ll fully understand this part once we have completed the `Serialize` function in the next section.

2- Sensor data serializer This class is actually rather simple, it’s only required to have two static methods, `Serialize` and `Deserialize`. We’ll add two files for it, this time to LibCarla

- `LibCarla/source/carla/sensor/s11n/SafeDistanceSerializer.h`
- `LibCarla/source/carla/sensor/s11n/SafeDistanceSerializer.cpp`

Let’s start with the `Serialize` function. This function is going to receive as arguments whatever we pass to the `Stream.Send(...)` function, with the only condition that the first argument has to be a sensor and it has to return a buffer.

```
1 static Buffer Serialize(const Sensor &, ...);
```

A `carla::Buffer` is just a dynamically allocated piece of raw memory with some convenient functionality, we’re going to use it to send raw data to the client.

In this example, we need to write the list of detected actors to a buffer in a way that it can be meaningful in the client-side. That’s why we passed the `episode` object to this function.

The `UCarlaEpisode` class represent the current *episode* running in the simulator, i.e. the state of the simulation since last time we loaded a map. It contains all the relevant information to Carla, and among other things, it allows searching for actor IDs. We can send these IDs to the client and the client will be able to recognise these as actors

```
1 template <typename SensorT, typename EpisodeT, typename ActorListT>
2 static Buffer Serialize(
3     const SensorT &,
4     const EpisodeT &episode,
5     const ActorListT &detected_actors) {
6     const uint32_t size_in_bytes = sizeof(ActorId) * detected_actors.Num();
7     Buffer buffer{size_in_bytes};
8     unsigned char *it = buffer.data();
9     for (auto *actor : detected_actors) {
10         ActorId id = episode.FindActor(actor).GetActorId();
11         std::memcpy(it, &id, sizeof(ActorId));
12         it += sizeof(ActorId);
13     }
14     return buffer;
15 }
```

Note that we templatize the UE4 classes to avoid including these files within LibCarla.

This buffer we’re returning is going to come back to us, except that this time in the client-side, in the `Deserialize` function packed in a `RawData` object

```
1 static SharedPtr<SensorData> Deserialize(RawData &&data);
```

We’ll implement this method in the cpp file, and it’s rather simple

```
1 SharedPtr<SensorData> SafeDistanceSerializer::Deserialize(RawData &&data) {
2     return SharedPtr<SensorData>(new data::SafeDistanceEvent(std::move(data)));
3 }
```

except for the fact that we haven’t defined yet what’s a `SafeDistanceEvent`.

3- Sensor data object We need to create a data object for the users of this sensor, representing the data of a *safe distance event*. We'll add this file to

- LibCarla/source/carla/sensor/data/SafeDistanceEvent.h

This object is going to be equivalent to a list of actor IDs. For that, we'll derive from the Array template

```

1 #pragma once
2
3 #include "carla/rpc/ActorId.h"
4 #include "carla/sensor/data/Array.h"
5
6 namespace carla {
7 namespace sensor {
8 namespace data {
9
10 class SafeDistanceEvent : public Array<rpc::ActorId> {
11 public:
12
13     explicit SafeDistanceEvent(RawData &&data)
14         : Array<rpc::ActorId>(std::move(data)) {}
15     };
16
17 } // namespace data
18 } // namespace sensor
19 } // namespace carla

```

The Array template is going to reinterpret the buffer we created in the `Serialize` method as an array of actor IDs, and it's able to do so directly from the buffer we received, without allocating any new memory. Although for this small example may seem a bit overkill, this mechanism is also used for big chunks of data; imagine we're sending HD images, we save a lot by reusing the raw memory.

Now we need to expose this class to Python. In our example, we haven't add any extra methods, so we'll just expose the methods related to Array. We do so by using Boost.Python bindings, add the following to *PythonAPI/carla/-source/libcarla/SensorData.cpp*.

```

1 class_<
2     csd::SafeDistanceEvent,                                // actual type.
3     bases<csd::SensorData>,                            // parent type.
4     boost::noncopyable,                                  // disable copy.
5     boost::shared_ptr<csd::SafeDistanceEvent>           // use as shared_ptr.
6     >("SafeDistanceEvent", no_init)                      // name, and disable construction.
7     .def("__len__", &csd::SafeDistanceEvent::size)
8     .def("__iter__", iterator<csd::SafeDistanceEvent>())
9     .def("__getitem__", +[](const csd::SafeDistanceEvent &self, size_t pos) -> cr::ActorId {
10         return self.at(pos);
11     })
12 ;

```

Note that `csd` is an alias for the namespace `carla::sensor::data`.

What we're doing here is exposing some C++ methods in Python. Just with this, the Python API will be able to recognise our new event and it'll behave similar to an array in Python, except that cannot be modified.

4- Register your sensor Now that the pipeline is complete, we're ready to register our new sensor. We do so in *LibCarla/source/carla/sensor/SensorRegistry.h*. Follow the instruction in this header file to add the different includes and forward declarations, and add the following pair to the registry

```
1 std::pair<ASafeDistanceSensor *, s11n::SafeDistanceSerializer>
```

With this, the sensor registry now can do its magic to dispatch the right data to the right serializer.

Now recompile CARLA, hopefully everything goes ok and no errors. Unfortunately, most of the errors here will be related to templates and the error messages can be a bit cryptic.

```
1 make rebuild
```

5- Usage example Finally, we have the sensor included and we have finished recompiling, our sensor by now should be available in Python.

To spawn this sensor, we simply need to find it in the blueprint library, if everything went right, the sensor factory should have added our sensor to the library

```
1 blueprint = blueprint_library.find('sensor.other.safe_distance')
2 sensor = world.spawn_actor(blueprint, carla.Transform(), attach_to=vehicle)
```

and now we can start listening for events by registering a callback function

```
1 world_ref = weakref.ref(world)
2
3 def callback(event):
4     for actor_id in event:
5         vehicle = world_ref().get_actor(actor_id)
6         print('Vehicle too close: %s' % vehicle.type_id)
7
8 sensor.listen(callback)
```

This callback is going to execute every update that another vehicle is inside our safety distance box, e.g.

```
1 Vehicle too close: vehicle.audi.a2
2 Vehicle too close: vehicle.mercedes-benz.coupe
```

That's it, we have a new sensor working!

Appendix

Reusing buffers In order to optimize memory usage, we can use the fact that each sensor sends buffers of similar size; in particular, in the case of cameras, the size of the image is constant during execution. In those cases, we can save a lot by reusing the allocated memory between frames.

Each stream contains a *buffer pool* that can be used to avoid unnecessary memory allocations. Remember that each sensor has a stream associated thus each sensor has its own buffer pool.

Use the following to retrieve a buffer from the pool

```
1 auto Buffer = Stream.PopBufferFromPool();
```

If the pool is empty, it returns an empty buffer, i.e. a buffer with no memory allocated. In that case, when you resize the buffer new memory will be allocated. This will happen a few times during the first frames. However, if a buffer was retrieved from the pool, its memory will go back to the pool once the buffer goes out of the scope. Next time you get another buffer from the pool, it'll contain the allocated piece of memory from the previous buffer. As you can see, a buffer object acts actually as a smart pointer to a contiguous piece of raw memory. As long as you don't request more memory than the currently allocated, the buffer reuses the memory. If you request more, then it'll have to delete the current memory and allocate a bigger chunk.

The following snippet illustrates how buffers work

```
1 Buffer buffer;
2 buffer.reset(1024u); // (size 1024 bytes, capacity 1024 bytes) -> allocates
3 buffer.reset(512u); // (size 512 bytes, capacity 1024 bytes)
4 buffer.reset(2048u); // (size 2048 bytes, capacity 2048 bytes) -> allocates
```

Sending data asynchronously Some sensors may require to send data asynchronously, either for performance or because the data is generated in a different thread, for instance, camera sensors send the images from the render thread.

Using the data stream asynchronously is perfectly fine, as long as the stream itself is created in the game thread. For instance

```
1 void MySensor::Tick(float DeltaSeconds)
2 {
3     Super::Tick(DeltaSeconds);
4 }
```

```

5 auto Stream = GetDataStream(*this);
6
7 std::async(std::launch::async, [Stream=std::move(Stream)]() {
8     auto Data = ComputeData();
9     Stream.Send(*this, Data);
10 });
11 }

```

Client-side sensors Some sensors do not require the simulator to do their measurements, those sensors may run completely in the client-side freeing the simulator from extra computations. Examples of such sensors is the *LaneInvasion* sensors.

The usual approach is to create a “dummy” sensor in the server-side, just so the simulator is aware that such actor exists. However, this dummy sensor doesn’t tick nor sends any sort of data. Its counterpart on the client-side however, registers a “on tick” callback to execute some code on every new update. For instance, the *LaneInvasion* sensor registers a callback that notifies every time a lane mark has been crossed.

It is very important to take into account that the “on tick” callback in the client-side is executed concurrently, i.e., the same method may be executed simultaneously by different threads. Any data accessed must be properly synchronized, either with a mutex, using atomics, or even better making sure all the members accessed remain constant.

10.3 Customize vehicle suspension

This tutorial covers the basics of the suspension system for CARLA vehicles, and how are these implemented for the different vehicles available. Use this information to access the suspension parameterization of a vehicle in Unreal Engine, and customize it at will.

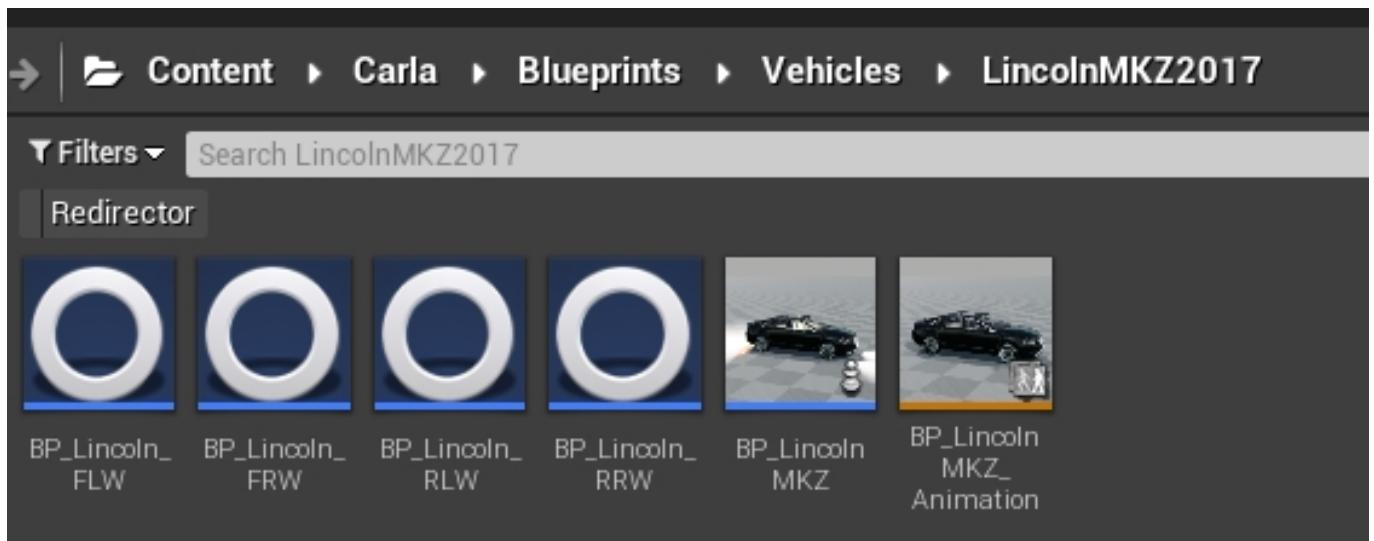
- **Basics of the suspension system**
- **Suspension groups**
 - Coupe
 - Off-road
 - Truck
 - Urban
 - Van

Basics of the suspension system

The suspension system of a vehicle is defined by the wheels of said vehicle. Each wheel has an independent blueprint with some parameterization, which includes the suspension system.

These blueprints can be found in `Content/Carla/Blueprints/Vehicles/<vehicle_name>`. They are named such as: `BP_<vehicle_name>_<F/R><R/L>W`.

- F or R is used for front or rear wheels correspondingly.
- R or L is used for right or left wheels correspondingly.

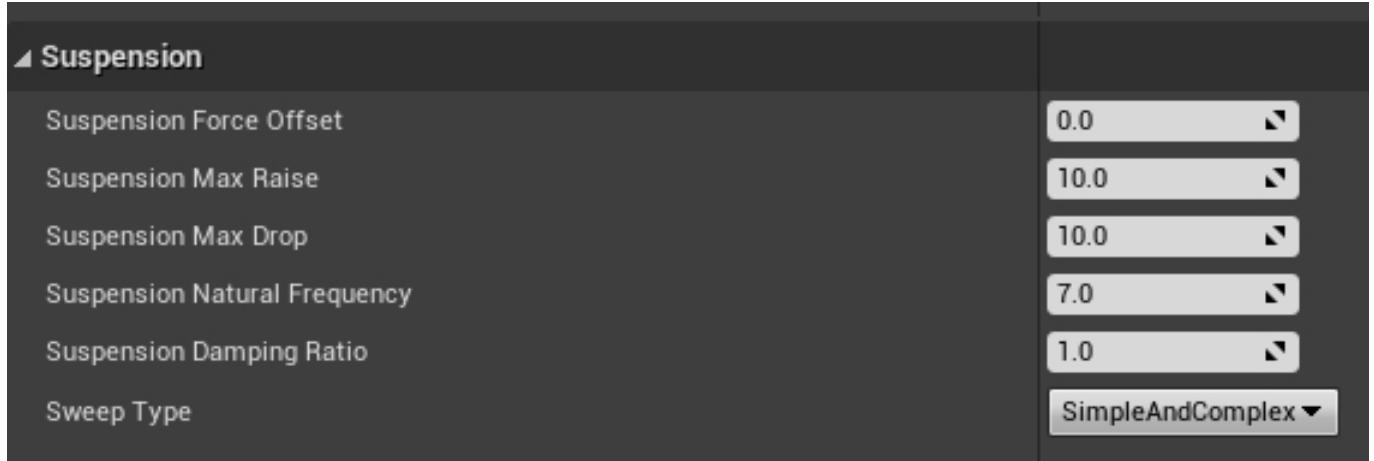


In this example, the blueprint of the front left wheel of the Audi A2 is named as BP_AudiA2_FLW.

`shape_radius` for the wheel to rest over the road, neither hovering nor inside of it.

Inside the blueprint, there is a section with some parameterization regarding the suspension of the wheel. Here are their definitions as described in Unreal Engine.

- **Suspension Force Offset** — Vertical offset from where suspension forces are applied (along Z axis).
- **Suspension Max Raise** — How far the wheel can go above the resting position.
- **Suspension Max Drop** — How far the wheel can drop below the resting position.
- **Suspension Natural Frequency** — Oscillation frequency of the suspension. Standard cars have values between 5 and 10.
- **Suspension Damping Ratio** — The rate at which energy is dissipated from the spring. Standard cars have values between 0.8 and 1.2. Values <1 are more sluggish, values >1 are more twitchy.
- **Sweep Type** — Whether wheel suspension considers simple, complex or both.



The Suspension panel inside a wheel blueprint.



By default, all the wheels of a vehicle have the same parameterization in CARLA. The following explanations will be covered per vehicle, instead of per wheel.

Suspension groups

According to their system suspension, vehicles in CARLA can be classified in five groups. All the vehicles in a group have the same parameterization, as they are expected to have a similar behaviour on the road. The suspension of a vehicle can be modified at will, and is no subject to these five groups. However understanding these, and observing their behaviour in the simulation can be of great use to define a custom suspension.

The five groups are: *Coupe*, *Off-road*, *Truck*, *Urban*, and *Van*. In closer observation, the parameterization of these groups follows a specific pattern.

Stiff suspension

Coupe

Urban

Van

Off-road

Truck

Soft suspension

When moving from a soft to a stiff suspension, there are some clear tendencies in the parameterization.

- **Decrease of Suspension Max Raise and Suspension Max Drop** — Stiff vehicles are meant to drive over plane roads with no bumps. For the sake of aerodynamics, the chassis is not supposed to move greatly, but remain constantly close to the ground.

- **Increase of Suspension Damping Ratio** — The absorption of the bouncing by the dampers is greater for stiff vehicles.

Coupe Vehicles with the stiffest suspension.

Parameterization

Vehicles

Suspension Force Offset — 0.0 Suspension Max Raise — 7.5 Suspension Max Drop — 7.5 Suspension Natural Frequency — 9.5 Suspension Damping Ratio — 1.0 Sweep Type — SimpleAndComplex

vehicle.audi.tt vehicle.lincoln.mkz2017 vehicle.mercedes-benz.coupe vehicle.seat.leon vehicle.tesla.model3

Off-road Vehicles with a soft suspension.

Parameterization

Vehicles

Suspension Force Offset — 0.0 Suspension Max Raise — 15.0 Suspension Max Drop — 15.0 Suspension Natural Frequency — 7.0 Suspension Damping Ratio — 0.5 Sweep Type — SimpleAndComplex

vehicle.audi.etron vehicle.jeep.wrangler_rubicon vehicle.nissan.patrol vehicle.tesla.cybertruck

Truck Vehicles with the softest suspension.

Parameterization

Vehicles

Suspension Force Offset — 0.0 Suspension Max Raise — 17.0 Suspension Max Drop — 17.0 Suspension Natural Frequency — 6.0 Suspension Damping Ratio — 0.4 Sweep Type — SimpleAndComplex

vehicle.carlamotors.carlacola

Urban Vehicles with a soft suspension.

Parameterization

Vehicles

Suspension Force Offset — 0.0 Suspension Max Raise — 8.0 Suspension Max Drop — 8.0 Suspension Natural Frequency — 9.0 Suspension Damping Ratio — 0.8 Sweep Type — SimpleAndComplex

vehicle.audi.a2 vehicle.bmw.grandtourer vehicle.chevrolet.impala vehicle.citroen.c3 vehicle.dodge_charger.police vehicle.mini.cooperst vehicle.mustang.mustang vehicle.nissan.micra vehicle.toyota.prius

Van Vehicles with a middle-ground suspension.

Parameterization

Vehicles

Suspension Force Offset — 0.0 Suspension Max Raise — 9.0 Suspension Max Drop — 9.0 Suspension Natural Frequency — 8.0 Suspension Damping Ratio — 0.8 Sweep Type — SimpleAndComplex

vehicle.volksvagen.t2

Use the forum to post any doubts, issues or suggestions regarding this topic.

Here are some advised readings after this one.

10.4 Generate detailed colliders

This tutorial explains how to create more accurate collision boundaries for vehicles (relative to the original shape of the object). These can be used as physics collider, compatible with collision detection, or as a secondary collider used by raycast-based sensors such as the LIDAR to retrieve more accurate data. New colliders can be integrated into CARLA so that all the community can benefit from these. Find out more about how to contribute to the content repository [here](#).

There are two approaches to create the new colliders, but they are not completely equivalent.

- **Raycast colliders** — This approach requires some basic 3D modelling skills. A secondary collider is added to the vehicle so that raycast-based sensors such as the LIDAR retrieve more precise data.
- **Physics colliders** — This approach follows the tutorial created by the contributor **Yan Kaganovsky / yankagan** to create a mesh with no need of manual modelling. This mesh is then used as main collider for the vehicle, for physics and sensor detection (unless a secondary collider is added).
- **Raycast colliders**
 - 1-Export the vehicle FBX
 - 2-Generate a low density mesh
 - 3-Import the mesh into UE
 - 4-Add the mesh as collider

- **Physics colliders**

- 0-Prerequisites
- 1-Define custom collision for wheels in Unreal Editor
- 2-Export the vehicle as FBX
- 3 to 4-Import to Blender and create custom boundary
- 5-Export from Blender to FBX
- 6 to 8-Import collider and define physics

Raycast colliders

1-Export the vehicle FBX First of all, the original mesh of the vehicle is necessary to be used as reference. For the sake of learning, this tutorial exports the mesh of a CARLA vehicle. **1.1** open CARLA in UE and go to Content/Carla/Static/Vehicles/4Wheeled/<model_of_vehicle>. **1.2** Press right-click on SM_<model_of_vehicle> to export the vehicle mesh as FBX.

2-Generate a low density mesh **2.1** Open a 3D modelling software and, using the original mesh as reference, model a low density mesh that stays reliable to the original.

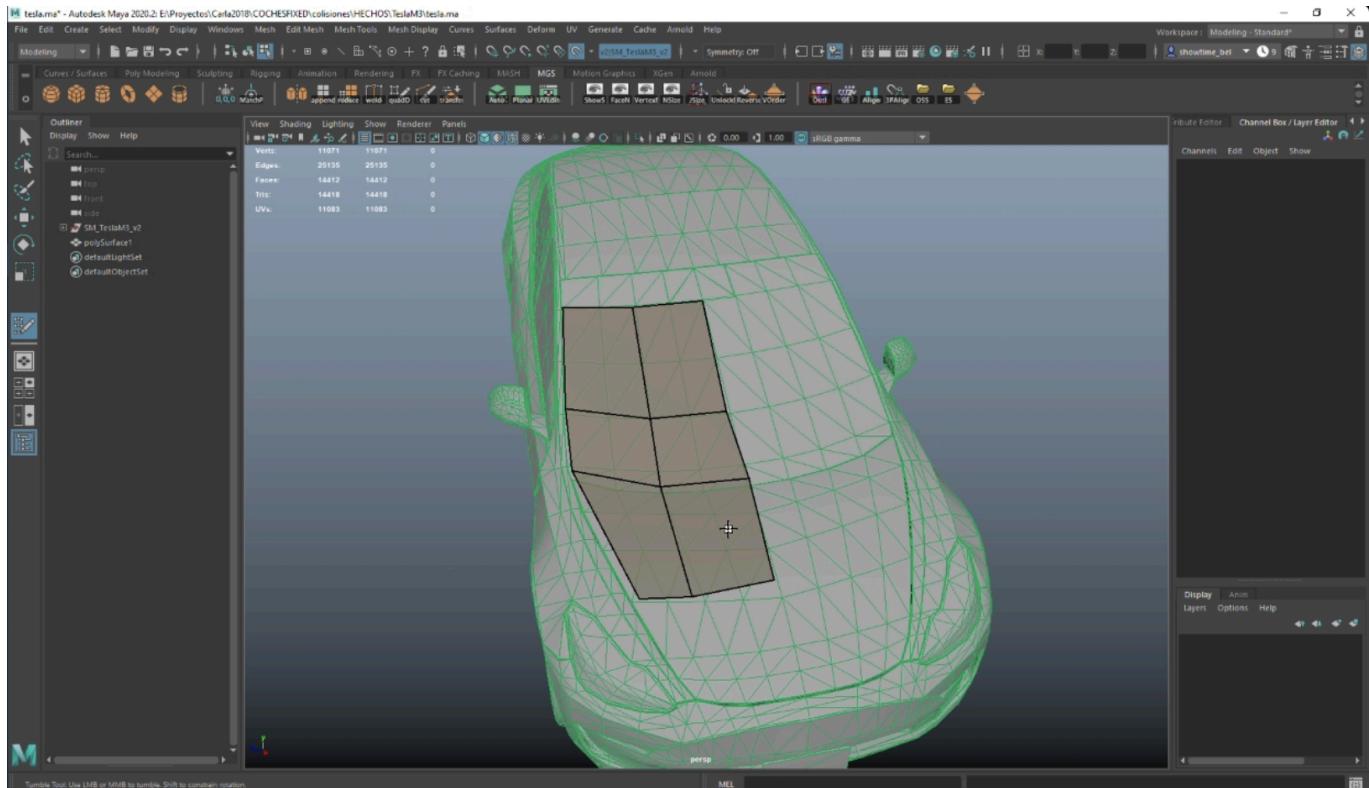


Figure 75: manual_meshgen

2.2 Save the new mesh as FBX. Name de mesh as sm_sc_<model_of_vehicle>.fbx. E.g. sm_sc_audiTT.fbx.



As for the wheels and additional elements such as roofs, mudguards, etc. the new mesh should follow the geometry quite accurately. Placing simple cubes will not do it.

3-Import the mesh into UE

3.1 Open CARLA in UE and go to Content/Carla/Static/Vehicles/4Wheeled/<model_of_vehicle>.fbx

3.2 Press right-click to import the new mesh SM_sc_<model_of_vehicle>.fbx.

4-Add the mesh as collider

4.1 Go to Content/Carla/Blueprints/Vehicles/<model_of_vehicle> and open the blueprint of the vehicle named as BP_<model_of_vehicle>.

4.2 Select the CustomCollision element and add the SM_sc_<model_of_vehicle>.fbx in the Static mesh property.

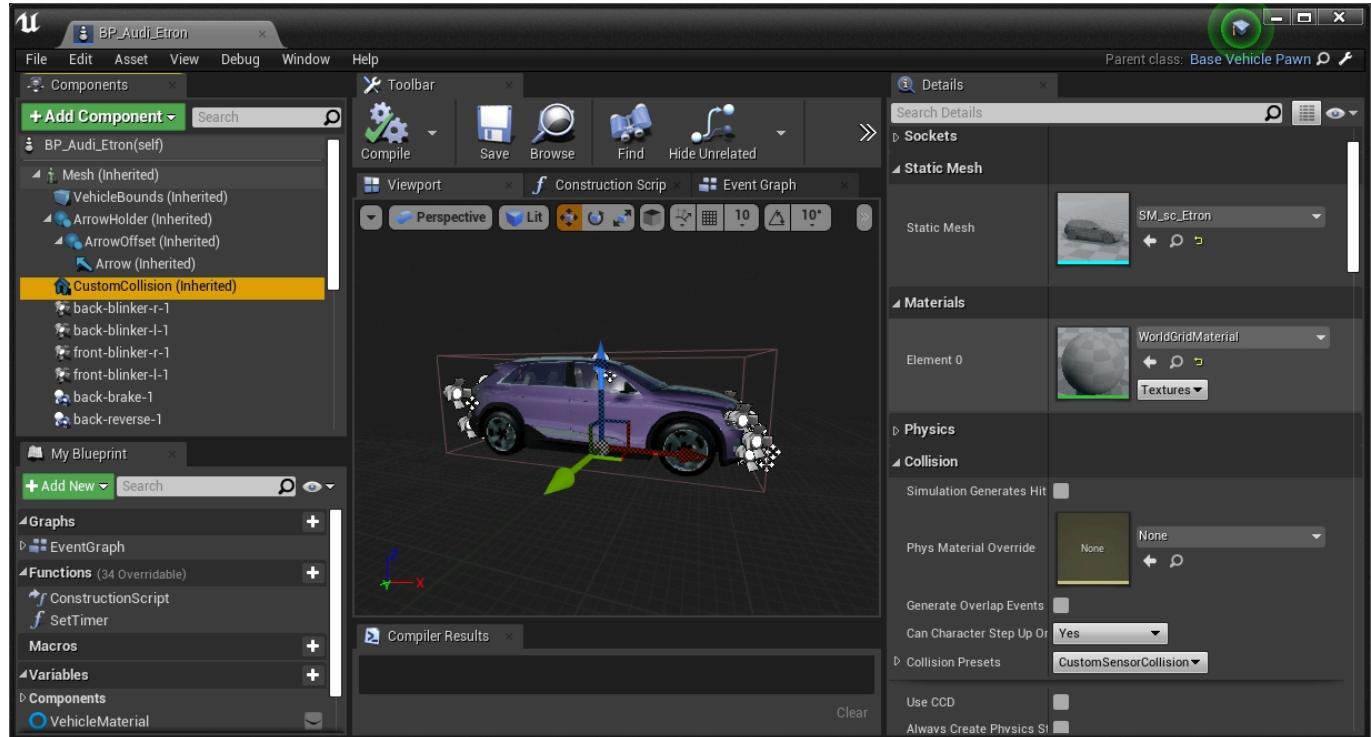


Figure 76: manual_customcollision

4.3 Press Compile in the toolbar above and save the changes.



For vehicles such as motorbikes and bicycles, change the collider mesh of the vehicle itself using the same component, ‘CustomCollision’.

Physics colliders



This tutorial is based on a [contribution](<https://bitbucket.org/yankagan/carla-content/wiki/Home>) made by [yankagan](<https://github.com/yankagan>) ! The contributor also wants to acknowledge Francisco E for the tutorial on [how to import custom collisions in UE](<https://www.youtube.com/watch?v=SEH4f0HrCDM>).

This video shows the results achieved after following this tutorial.

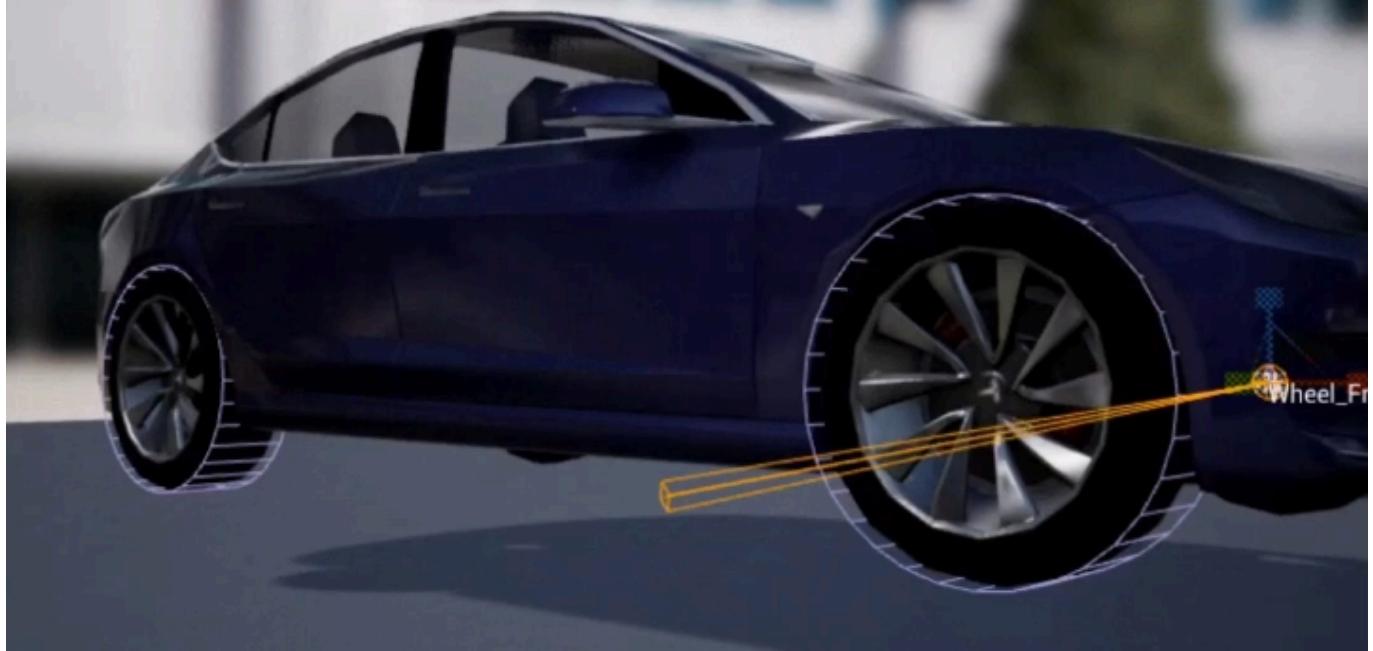
0-Prerequisites

- **Build CARLA from source** onLinux orWindows.
- **Blender 2.80 or newer** from the official site for free (open-source 3D modelling software).
- **VHACD Plugin for Blender** following the using the instructions in here. This plugin automatically creates an approximation of a selected object using a collection of convex hulls. [Read more](#).



This [series](<https://www.youtube.com/watch?v=ppASl6yaguU>) and [Udemy course](<https://www.udemy.com/course/blender-3d-from-zero-to-hero/?pmtag=MRY1010>) may be a good introduction to Blender for newcomers.

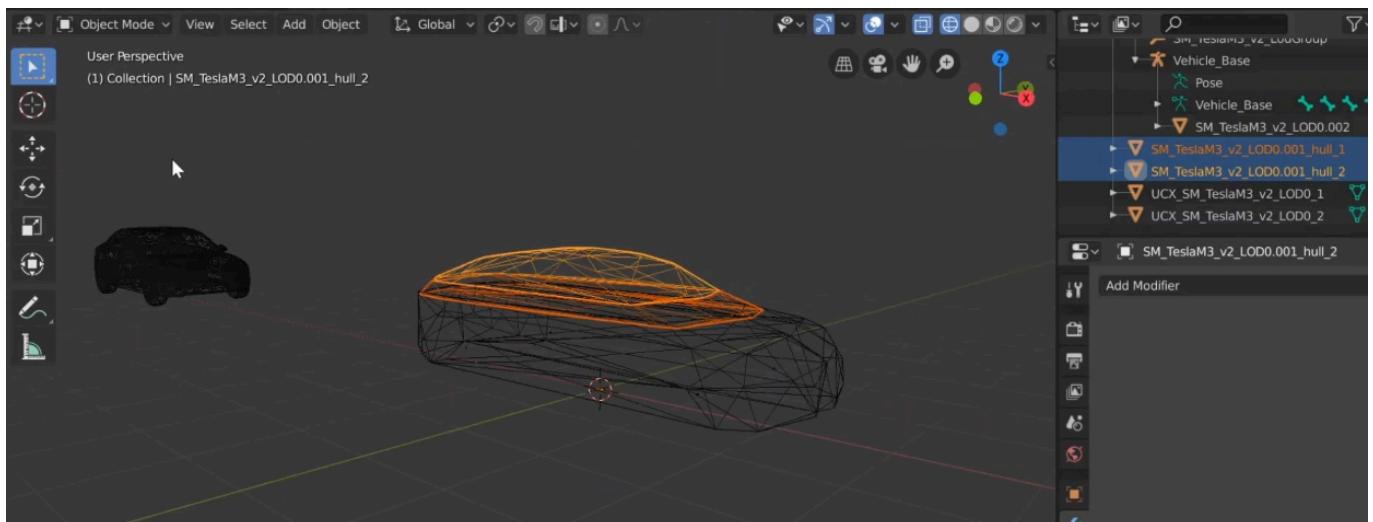
1-Define custom collision for wheels in Unreal Editor **Step 1.** (*in UE*) — Add collision boundaries for the wheels. The steps are detailed in the following video.



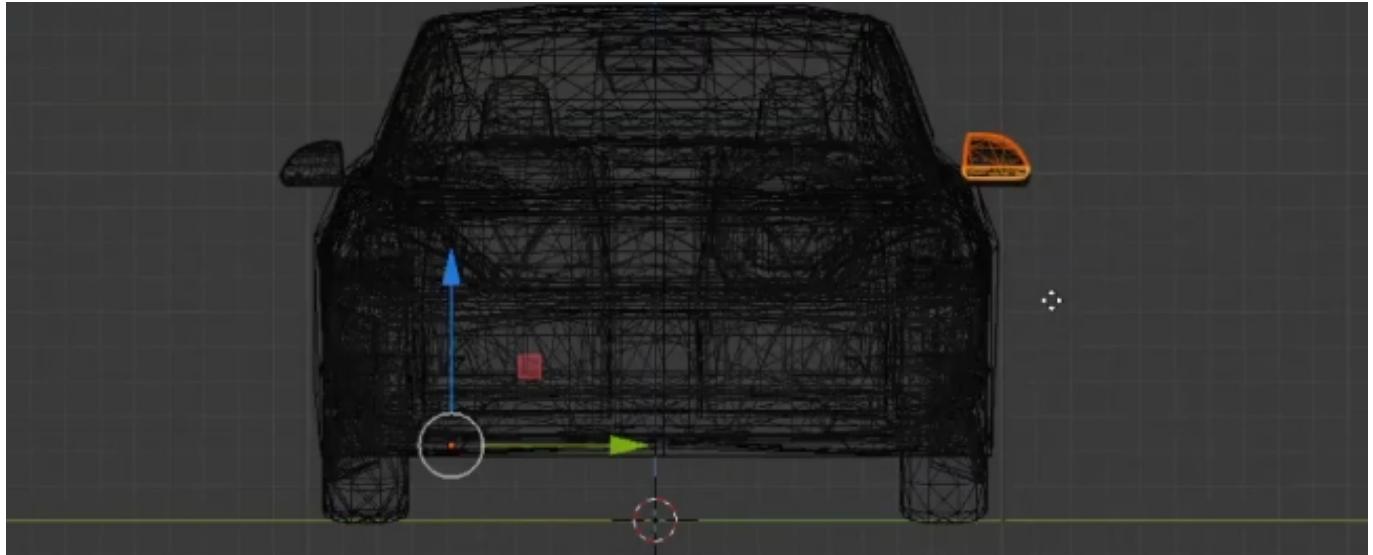
2-Export the vehicle as FBX **Step 2.** (*in UE*) — Export the skeletal mesh of a vehicle to an FBX file. **2.1** Go to Content/Carla/Static/Vehicles/4Wheeled/<model_of_vehicle>. **2.2** Press right-click on SM_<model_of_vehicle> to export the vehicle mesh as FBX.

3 to 4-Import to Blender and create custom boundary **Step 3.** (*in Blender*) — Import the FBX file into Blender. **Step 4.** (*in Blender*) — Add convex hull meshes to form the new collision boundary (UE requirement for computational efficiency). This is the hardest step. If the entire car is selected, the collision boundary created by VHACD will be imprecise and messy. It will contain sharp edges which will mess-up the drive on the road. It's important that the wheels have smooth boundaries around them. Using convex decomposition on the car's body the mirrors would still not look right. For computer vision, the details of the vehicle are important. For said reason, these step has been divided into two parts.

4.1 Cut out the bottom parts of the wheels, the side mirrors and the top part of the car's body to create the first boundary using the VHACD tool. Cut out the bottom half of the car to create the second boundary (top part of the car) using the VHACD tool.

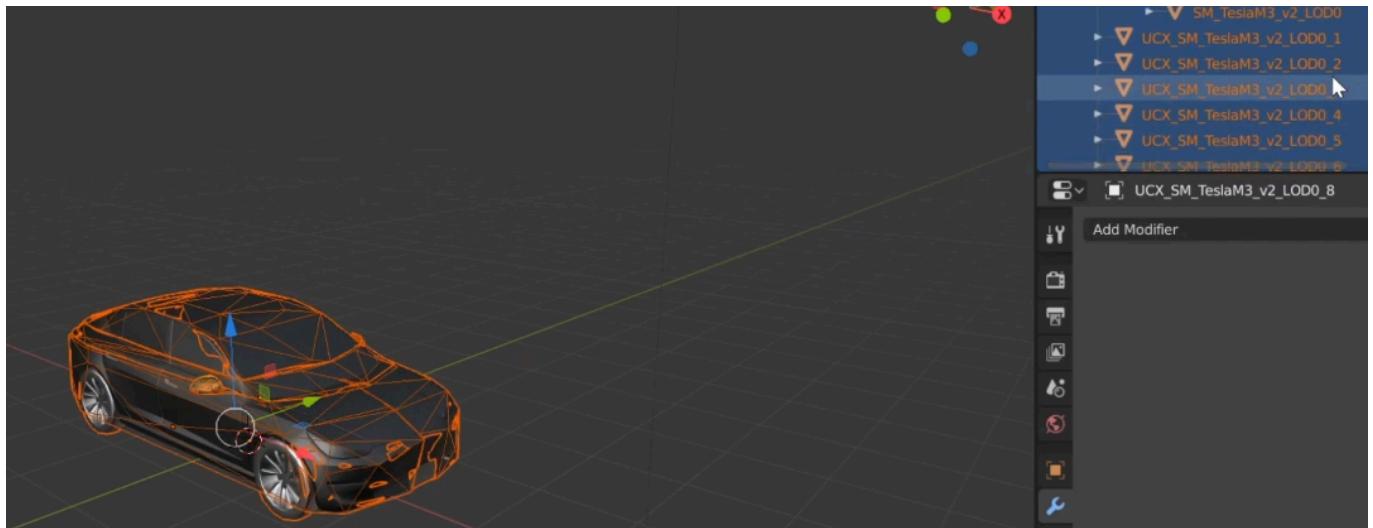


4.2 Create separate boundaries for side mirrors using the VHACD tool.

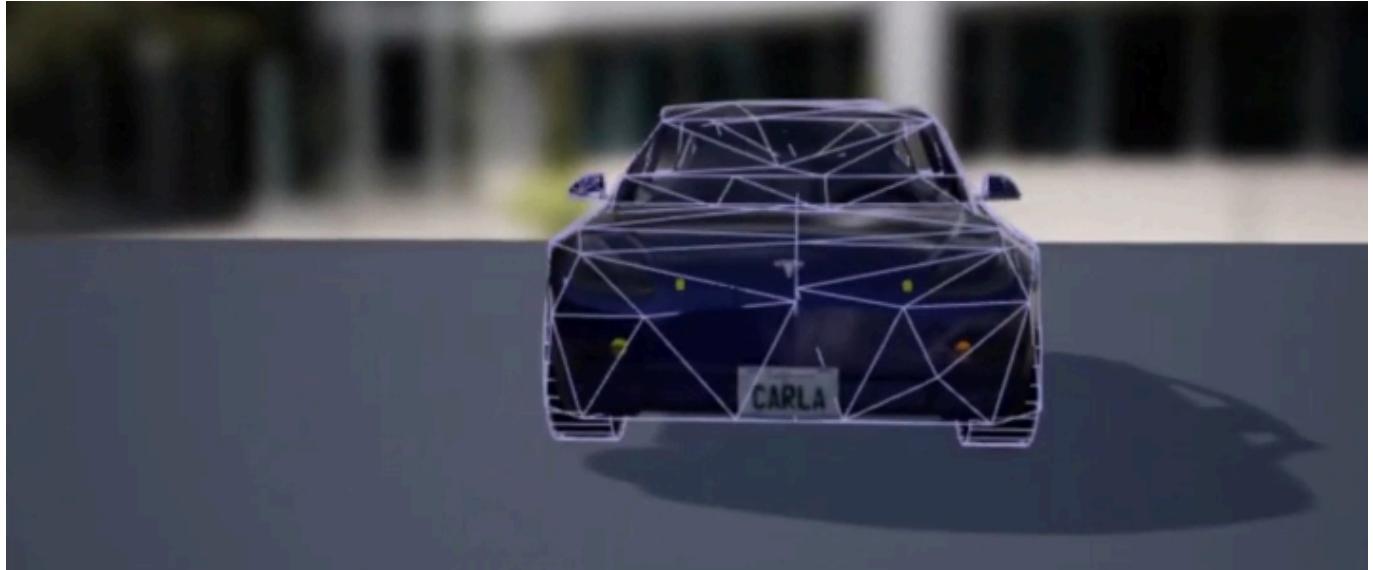


Be very careful about naming the object. Each boundary should have begin with ‘UCX ‘, and the rest of the name has to be exactly the same as the original mesh.

5-Export from Blender to FBX **Step 5.** (*in Blender*) — Export the custom collision boundaries into an FBX file. **5.1** Select only the original vehicle and all the newly added objects for collision. **5.2** In the export menu, check selected objects and select only “Mesh”.



6 to 8-Import collider and define physics **Step 6.** (*in UE*) — Import the new FBX into CARLA as an Unreal asset file (static mesh). **Step 7.** (*in UE*) — Import the custom collider into the physics asset for the specific vehicle, so that it is used for computations. **Step 8.** (*in UE*) — Create constraints that connect the different joints and define the physics of all parts.



That is a wrap on how to change the default colliders for vehicles in CARLA.

Open CARLA and mess around for a while. If there are any doubts, feel free to post these in the forum.

10.5 How to make a release

This document is meant for developers that want to publish a new release.

1. **Make sure content is up-to-date.** See Upgrade the content.
2. **Increase CARLA version where necessary.** Increase version in the following files: *DefaultGame.ini*, *Carla.uplugin*, *setup.py*, *ContentVersions.txt*. Grep for the current version to make sure you don't miss any references.
3. **Clean up CHANGELOG.md.** Make sure the change log is up-to-date, reword and reorganize if necessary; take into account which items can be more important to the users.
4. **Commit changes and add a new tag.** Once all your changes are committed, add a new tag with `git tag -a X.X.X` (replacing X.X.X by latest version). Add the changelog of this version as tag message.
5. **Tag content repo.** Add a similar tag to the content repository at the exact commit as in *ContentVersions.txt*.
6. **Push changes.** Push all the changes to both repositories, to push tags you may need to use `git push --tags`. Create a Pull Request if necessary.
7. **Edit GitHub release.** Go to GitHub releases and create a new release on top of the newly created tag. Wait until Jenkins has finished publishing the builds with the latest version and add the download links to the newly created release.

10.6 How to generate the pedestrian navigation info

The pedestrians to walk need information about the map in a specific format. That file that describes the map for navigation is a binary file with extension .BIN, and they are saved in the **Nav** folder of the map. Each map needs a .BIN file with the same name that the map, so automatically can be loaded with the map.

This .BIN file is generated from the Recast & Detour library and has all the information that allows pathfinding and crow management.

If we need to generate this .BIN file for a custom map, we need to follow this process:

- Export the map meshes
- Rebuild the .BIN file with RecastBuilder
- Copy the .BIN file in a **Nav/** folder with the map

Export meshes

We have several types of meshes for navigation. The meshes need to be identified as one of those types, using specific nomenclature.

Type
 Start with
 Description
 Ground
 Road_Sidewalk
 Pedestrians can walk over these meshes freely (sidewalks...).
 Grass
 Road_Crosswalk
 Pedestrians can walk over these meshes but as a second option if no ground is found.
 Road
 Road_Grass
 Pedestrians won't be allowed to walk on it unless we specify some percentage of pedestrians that will be allowed.
 Crosswalk
 Road_Road, Road_Curb, Road_Gutter, Road_Marking
 Pedestrians can cross the roads only through these meshes.
 Block
 Any other name
 Pedestrians will avoid these meshes always (are obstacles like traffic lights, trees, houses...).
 For instance, all road meshes need to start with `Road_Road` e.g: `Road_Road_Mesh_1`, `Road_Road_Mesh_2...`
 This nomenclature is used by RoadRunner when it exports the map, so we are just following the same.
 Also, we don't need to export meshes that are not of interest by the navigation system, like landscapes where pedestrians will not walk, or UE4's *sky domes*, rivers, lakes and so on. We can **tag** any mesh with the name **NoExport** and then the exporter will ignore that mesh.
 Once we have all meshes with the proper nomenclature and tagged the ones that we want to ignore, we can call the Carla Exporter that is found under the `File > Actors > Carla Exporter`:
 The `.OBJ` file will be saved with the name of the map with extension `OBJ` in the `CarlaUE4/Saved` folder of UE4.

Rebuild the navigation binary

With Recast & Detour library comes an executable file that needs to be used to generate the final `.BIN` file. The executable uses by default the parameters to work on Carla, and you only need to pass as parameter the `.OBJ` you exported above, and the `.BIN` will be created.

You can find the executable, on the `Carla/Build/recast-{version}-install/bin/` folder, already compiled when Carla was installed.

Run:

```
1 $ RecastBuilder test.object
```

And it will create a `test.bin` file, that needs to be copied to the map folder, inside the `Nav/` folder.

For example, this would be the structure of files of a Test map:

```
1 Test/
2   Test.umap
3   Nav/
4     Test.bin
5   OpenDrive/
6     Test.xodr
```

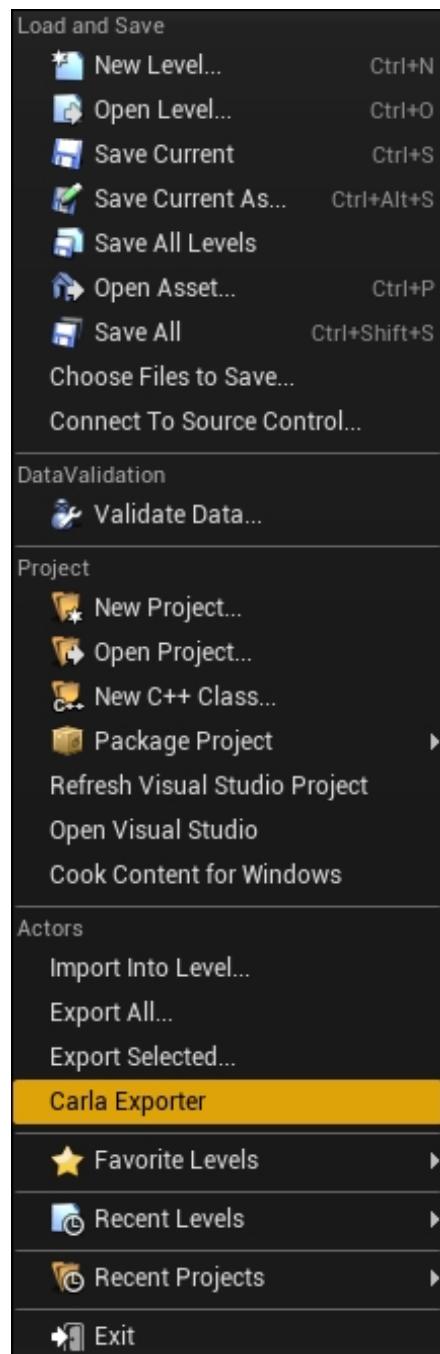


Figure 77: Carla Exporter

11 Contributing

11.1 Contributing to CARLA

The CARLA team is glad to accept contributions from anybody willing to collaborate. There are different ways to contribute to the project, depending on the capabilities of the contributor. The team will work as much as possible so that contributions are successfully integrated in CARLA.

Take a look and don't hesitate!

- **Report bugs**
- **Request features**
- **Code contributions**
 - Learn about Unreal Engine
 - Before getting started
 - Coding standard
 - Submission
 - Checklist
- **Art contributions**
- **Docs contributions**

Report bugs

Issues can be reported in the issue section on GitHub. Before reporting a new bug, make sure to do some checkups.

1. **Check if the bug has been reported.** Look it up in that same issue section on GitHub.
2. **Read the docs.** Make sure that the issue is a bug, not a misunderstanding on how CARLA supposed to work. Read the pages related to the issue in the Documentation and take a look at the FAQ page.

Request features

Ideas for new features are also a great way to contribute. Any suggestion that could improve the users' experience can be submitted in the corresponding GitHub section here.

Code contributions

Before starting hands-on on coding, please check out the issue board to check what is the team already working on, to avoid overlapping. In case of doubt or to discuss how to proceed, please contact one of us (or send an email to carla.simulator@gmail.com).

In order to start working, fork the CARLA repository, and clone said fork in your computer. Remember to keep your fork in sync with the original repository.

Learn about Unreal Engine A basic introduction to C++ programming with UE4 can be found at Unreal's C++ Programming Tutorials. There are other options online, some of them not free of charge. The Unreal C++ Course at Udemy it's pretty complete and there are usually offers that make it very affordable.

Before getting started Check out the CARLA Design document to get an idea on the different modules that compose CARLA. Choose the most appropriate one to hold the new feature. Feel free to contact the team in the Discord server in case any doubt arises during the process.

Coding standard Follow the current coding standard when submitting new code.

Submission Contributions and new features are not merged directly to the `master` branch, but to an intermediate branch named `dev`. This Gitflow branching model makes it easier to maintain a stable master branch. This model requires a specific workflow for contributions.

- Always keep your `dev` branch updated with the latest changes.
- Develop the contribution in child branch from `dev` named as `username/name_of_the_contribution`.
- Once the contribution is ready, submit a pull-request from your branch to `dev`. Try to be as descriptive as possible when filling the description. Note that there are some checks that the new code is required to pass before merging. The checks are automatically run by the continuous integration system. A green tick mark will appear if the checks are successful. If a red mark, please correct the code accordingly.

Once the contribution is merged in `dev`, it can be tested with the rest of new features. By the time of the next release, the `dev` branch will be merged to `master`, and the contribution will be available and announced.

Checklist

- [] Your branch is up-to-date with the `dev` branch and tested with latest changes.
- [] Extended the README/documentation, if necessary.
- [] Code compiles correctly.
- [] All tests passing with `make check`.

Art contributions

Art contributions include vehicles, walkers, maps or any other type of assets to be used in CARLA. These are stored in a BitBucket repository, which has some account space limitations. For said reason, the contributor will have to get in touch with the CARLA team, and ask them to create a branch on the content repository for the contributions.

1. **Create a BitBucket account.** Visit the Bitbucket page.
2. **Contact the art team to get access to the content repository.** Join the Discord server. Go to the **Contributors** channel and request for access to the content repository.
3. **A branch will be created for each contributor.** The branch will be named as `contributors/contributor_name`. All the contributions made by said user should be made in that corresponding branch.
4. **Build CARLA.** In order to contribute, a CARLA build is necessary. Follow the instructions to build either in Linux or Windows.
5. **Download the content repository.** Follow the instructions to update the content in here.
6. **Update the branch to be in sync with master.** The branch should always be updated with the latest changes in master.
7. **Upload the contribution.** Do the corresponding changes and push the branch to origin.
8. **Wait for the art team to check it up.** Once the contribution is uploaded, the team will check that everything is prepared to be merged with master.

Docs contributions

If some documentation is missing, vague or imprecise, it can be reported as with any other bug (read the previous section on how to report bugs). However, users can contribute by writing documentation.

The documentation is written with a mix of Markdown and HTML tags, with some extra CSS code for features such as tables or the town slider. Follow the steps below to start writing documentation.



To submit docs contributions, follow the same workflow explained right above for code contributions.
To sum up, contributions are made in a child branch from 'dev' and merged to said branch.

1. **Build CARLA from source.** Follow the steps in the docs to build on Linux or Windows.

2. **Install MkDocs.** MkDocs is a static site generator used to build documentation.

```
1 sudo pip install mkdocs
```

3. **Visualize the docs.** In the main CARLA folder, run the following command and click the link that appears in the terminal (`http://127.0.0.1:8000`) to open a local visualization of the documentation.

```
1 mkdocs serve
```

4. **Create a git branch.** Make sure to be in the `dev` branch (updated to latest changes) when creating a new one.

```
1 git checkout -b <contributor_name>/<branch_name>
```

5. **Write the docs.** Edit the files following the guidelines in the documentation standard page.

6. **Submit the changes.** Create a pull request in the GitHub repository, and add one of the suggested reviewers. Try to be as descriptive as possible when filling the pull-request description.

7. Wait for review. The team will check if everything is ready to be merged or any changes are needed.



The local repository must be updated with the latest updates in the ‘dev’ branch.

11.2 Contributor Covenant Code of Conduct

- Our pledge
- Our standards
- Our responsibilities
- Scope
- Enforcement
- Attribution

Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others’ private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at info-carla@osvf.org. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project’s leadership.

Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at <https://www.contributor-covenant.org/version/1/4/code-of-conduct.html>

11.3 Coding standard

- General
- Python
- C++

General

- Use spaces, not tabs.
- Avoid adding trailing whitespace as it creates noise in the diffs.

Python

- Comments should not exceed 80 columns, code should not exceed 120 columns.
- All code must be compatible with Python 2.7 and 3.7.
- Pylint should not give any error or warning (few exceptions apply with external classes like `numpy` and `pygame`, see our `.pylintrc`).
- Python code follows PEP8 style guide (use `autopep8` whenever possible).

C++

- Comments should not exceed 80 columns, code may exceed this limit a bit in rare occasions if it results in clearer code.
- Compilation should not give any error or warning (`clang++-8 -Wall -Wextra -std=C++14 -Wno-missing-braces`).
- The use of `throw` is forbidden, use `carla::throw_exception` instead.
- Unreal C++ code (CarlaUE4 and Carla plugin) follow the Unreal Engine's Coding Standard with the exception of using spaces instead of tabs.
- LibCarla uses a variation of Google's style guide.
- Uses of `try-catch` blocks should be surrounded by `#ifndef LIBCARLA_NO_EXCEPTIONS` if the code is used in the server-side.

11.4 Documentation Standard

This document will serve as a guide and example of some rules that need to be followed in order to contribute to the documentation.

- Docs structure
- Rules
- Exceptions

Docs structure

We use a mix of markdown and HTML tags to customize the documentation along with an `extra.css` file. To update Python API docs, instead of directly modifying the Markdown you need to edit the corresponding YAML files inside `carla/PythonAPI/docs/` and run `doc_gen.py` or `make PythonAPI.docs`.

This will re-generate the respective Markdown files inside `carla/Docs/`, which can then be fed into `mkdocs`.

Rules

- Leave always an empty line between sections and at the end of the document.
- Writting should not exceed 100 columns, except for HTML related content, markdown tables, code snippets and referenced links.
- If an inline link exceeds the limit, use referenced `[name] [reference_link]` markdown notation `[reference_link]: https://` rather than `[name] (https://)`.
- Use `
` to make inline jumps rather than leaving two spaces at the end of a line.
- Use `<h1>Title</h1>` at the beginning of a new page in order to make a Title or `<hx>Heading<hx>` to make a heading that `won't show` on the navigation bar.
- Use 'underlining a Heading or #' hierarchy to make headings and show them in the navigation bar.

Exceptions

- Documentation generated via Python scripts like PythonAPI reference
Handy markdown cheatsheet.

Acknowledgements

This work is done by Infotiv under VALU3S project.

This project has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 876852. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Austria, Czech Republic, Germany, Ireland, Italy, Portugal, Spain, Sweden, Turkey.