CARLA: Open-source Simulator for Autonomous Driving Research

Nestor Subiron
Content

1. Introduction to CARLA architecture
2. Client API example: data acquisition
3. Highlighted features
4. Coming soon
Introduction to CARLA
CARLA Simulator

- **Open-source**
  All source code, 3D models, and maps fully open and redistributable.

- **Flexible API**
  Programmatic control over all the aspects of the simulation.

- **Autonomous driving sensor suite**
  Configurable sensors and ground-truth data.

- **Integration**
  ROS and Autoware integrated via our ROS-bridge.

[https://github.com/carla-simulator](https://github.com/carla-simulator)
## New API

<table>
<thead>
<tr>
<th>Feature</th>
<th>0.8.x</th>
<th>0.9.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-client</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Multi-agent</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Network error proof</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Reset episode</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Pedestrians</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Synchronous mode</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>No-rendering mode</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>OpenDRIVE navigation</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Map ingestion</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Logging and playback</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
CARLA: Simulator + Client API

- Simulator computes the physics and renders the scene.
- Users control the simulator from the Client API.
All the functionality is available for both Python and C++

- The C++ API provides a lower-level fast-performance communication.
- The Python API provides convenient bindings for easy-to-use communication.
CARLA Modules

Modules

- Render: Unreal Engine 4
- Physics: PhysX
- Networking: RPC + LibCarla
- Navigation: Scene graph based on OpenDRIVE
- Traffic: Agents & Autopilot
Modules: Unreal Engine 4

Game development framework

- AAA quality and performance render
- Lighting and post-processing
- Level editor
- Able to render multiple cameras in parallel
Modules: PhysX

Configurable vehicles

- Mass and center of mass
- Engine and transmission setup
- Steering and torque curves
- Per wheel setup: tire friction, steer, damping rate

Client API can modify at runtime.
- Connect and control via RPC protocol, flexible but slow for big messages.
- Sensor data sent via the streaming server, unidirectional but fast.
- Transparent to the user.
Modules: OpenDRIVE Navigation

Navigation API based on OpenDRIVE standard.

Vehicles are able to navigate newly imported maps containing an OpenDRIVE file.
We should distinguish between two types of navigation agents in Carla:

- **Client-side agents**
  - Scalable, able to distribute load among different machines.
  - Python, easy and quick to modify.

- **Server-side Autopilot**
  - Less overhead, but load goes to simulator.
  - Not configurable.
Client API example:
Data acquisition
Python API usage example:

Data acquisition:

- Set up environment, choose map and weather conditions.
- Add background traffic and ego-vehicle.
- Set up sensors.
- Acquire measurements and ground-truth.
Client API: Basics

- **World:** Currently loaded map.
- **Blueprint:** Specifications for creating an actor.
- **Actor:** Anything that plays a role in the simulation.
  - **Sensor:** Special type of actor that streams data.

```python
actor = world.spawn_actor(blueprint, transform)
actor.foo()
```
Loading the world

Connect to the simulator

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

Retrieve active world

```python
world = client.get_world()
```

Reload active world

```python
world = client.reload_world()
```

Load a world by map name

```python
world = client.load_world('Town05')
```
Changing the weather

Parametrized weather and lighting conditions, fully controllable at run-time

```
world.set_weather(WeatherParameters(precipitation=70))
```
The blueprint library

- The list of all available blueprints is kept in the blueprint library.
- Blueprints contain the information necessary to create a new actor.
- Provides methods for filtering and finding blueprints.
- Includes: sensors, vehicles, pedestrians, and static props.

```python
blueprint_library = world.get_blueprint_library()
traffic_cone = blueprint_library.find('static.prop.traffic_cone')
sensors = blueprint_library.filter('sensor.*')
```
Spawning actors

CARLA provides basic transform objects

- carla.Location(x, y, z) (in meters)
- carla.Rotation(pitch, yaw, roll) (in degrees)
- carla.Transform(carla.Location, carla.Rotation)

Important: CARLA uses left-handed coordinate axis
Spawning vehicles in autopilot

- Find the blueprint.
- Use a recommended spawn point.
- Enable autopilot.

```python
for blueprint in blueprint_library.filter('vehicle.*'):
    spawn_point = random.choice(world.get_map().get_spawn_points())
    vehicle = world.spawn_actor(blueprint, spawn_point)
    vehicle.set_autopilot(True)
```
Spawning ego-vehicle

- The ego-vehicle is spawned as any other vehicle.
- Tagged as “hero” role.

```python
blueprint = blueprint_library.find('vehicle.ford.mustang')
blueprint.set_attribute('role_name', 'hero')
vehicle = world.spawn_actor(blueprint, spawn_point)

vehicle.apply_control(VehicleControl(steer=0.0, throttle=0.7))
```
Spawning sensors

Sensors, as any other actor, can be spawned anywhere in the world

camera = world.spawn_actor(camera_bp, transform)

They can also be attached to other actors to follow them as they move around

camera = world.spawn_actor(camera_bp, relative_transform, attach_to=my_car)
Fine tuning sensors

Blueprints provide a way to customize the attributes of a sensor too

- Camera resolution, FOV, update frequency, gamma, blur, etc.
- Lidar range, channels, number of points, rotation frequency, etc.

```python
camera_bp.set_attribute('sensor_tick', '0.5')
camera_bp.set_attribute('fov', '120')
camera_bp.set_attribute('image_size_x', '1920')
camera_bp.set_attribute('image_size_y', '1080')
lidar_bp.set_attribute('channels', '64')
lidar_bp.set_attribute('rotation_frequency', '30.0')
```
Everytime a measurement is produced, an event is triggered in the client-side. We can register a callback that is going to be called on each new measurement.

```python
camera.listen(lambda image: image.save_to_disk('%06d.png' % image.frame_number))
```

The callback is executed asynchronously in the background.
Listening to sensors

Different sensors produce different types of data

- Image
- Point cloud
- GNSS coordinate
- Collision event
- Lane invasion event
- Obstacle event
Highlighted features
High-performance traffic simulation

- No-rendering mode for faster than real-time traffic simulation.
- Client-side 2D visualization of the scene built on top of our API.
Logging and playback

- Record the simulation to a file on disk.
- Simulation summary with important events, collisions or blockers.
- Replay only important events.
- Resume simulation at any time, diverging from replay.
- Replay in different weather and lighting conditions.
- Replay at different speeds.
Scenario runner

- Execute complex traffic scenarios.
- Define new scenarios based on behaviour trees.
- Future: support for standard formats as OpenSCENARIO.

https://github.com/carla-simulator/scenario_runner
CARLA loads OpenDRIVE files to a graph in memory providing:

- Easy API for map queries
- Waypoint generation
- Topology

This map can also be created off-line:

```python
map = carla.Map(opendrive_contents)
```
Map creation and ingestion

- Create your own map, e.g. with VectorZero’s RoadRunner.
- Import the map into Carla (geometry + OpenDRIVE).
- Cook the map for distribution.
- Users can import cooked maps into release package.
Importing cooked maps in release package

Cooked maps can be imported into a release package in just two steps:

- **Place the package in** `ExportedMaps` folder
- **Run** `ImportMaps.sh` script

The new map then becomes accessible from Client API, even if imported at run-time.
CARLA Docker image

Docker image publicly available at DockerHub

- `docker pull carlasim/carla:latest`

**Use** `NVIDIA_VISIBLE_DEVICES` **to select GPU**

https://hub.docker.com/r/carlasim/carla
Coming soon
Pedestrian navigation

Exposing new API to control pedestrians

- “Go to location” commands
- Stop-resume navigation

Efficient navigation algorithms

- Navigation mesh for quick path finding.
- DeTour avoidance to avoid collisions.
- Different weights for roads and sidewalks.
Pedestrian skeleton control

Simulate realistic poses for pedestrian detection

- Support for controlling each individual bone from Client API.
- 64 bones: 24 body + hands and eyes.
Thanks!