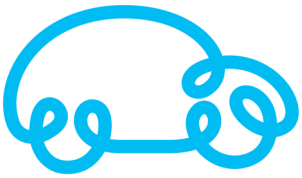
Dream-like simulation abilities for automated cars



DREAMS4CARS

Grant Agreement No. 731593

Deliverable:	D5.5 – System abilities (open data)
Dissemination level:	PU – Public
Delivery date:	31/December/2019
Status:	Final

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731593

Deliverable Title	System abilities (open data)			
WP number and title	WP5 Agent evolution, evaluation of ability levels and final assessment of the technology			
Lead Editor	Mauro Da Lio, UNITN			
Contributors	Rafael Math, DFKI			
	Gastone Pietro Rosati Papini, Alice Plebe, UNITN			
Creation Date	29/October/2019		**Version number**	0.9
Deliverable Due Date	31/December/2019		**Actual Delivery Date**	23/December/2019
Nature of deliverable	**X**	R - Report		
		DEM – Demonstrator, pilot, prototype, plan designs		
		DEC – Websites, patents filing, press&media actions		
	X	O – Other – Software, technical diagram		
Dissemination Level / Audience	**X**	PU – Public, fully open		
		CO - Confidential, restricted under conditions set out in MGA		
		CI – Classified, information as referred to in Commission Decision 2001/844/EC		

Version	Date	Modified by	Comments
0.1	29/October/2019	Rafael Math	First draft for discussion
0.2	16/November/2019	Mauro Da Lio	Arrangement of the experimental data section. Section 2.1. Section 1 (introduction and objectives).
0.3	22/November/2019	Rafael Math	Added simulation examples
0.4	01/December/2019	Mauro Da Lio	Section 2.2
0.5	13/December/2019	Rafael Math	Minor changes
0.6	15/December/2019	Alice Plebe	Minor changes on language and some comments on unclear parts
0.7	17/December/2019	Gastone Pietro Rosati Papini	Minor comments on the unclear parts.
0.8	19/December/2019	Rafael Math	Changes requested by peer reviewers
0.9	23/December/2019	Mauro Da Lio	Final version for submission

EXECUTIVE SUMMARY

This deliverable is the public version of D5.4 and documents data that is made publicly available at the end of the project. This includes two types of data:

1) Experimental data (and examples of data use) from test vehicles.
2) Simulation tools, simulation examples and simulation data.

The provided material includes the licensed Codriver agent for hands-on testing in the virtual prototyping environment OpenDS.

Table of Contents

List of Figures

1 Introduction and Objectives

1.1 Introduction

According to the Data Management Plans (D7.5, D7.6 and D7.7) Dreams4Cars is committed to produce Open Research Data.

This deliverable documents data that is made accessible at the end of the project. This includes essentially two types of data:

3) Experimental data (and examples of data use) from test vehicles.
4) Simulation tools, simulation examples and simulation data.

Not all data collected by Dreams4Cars is disclosed now (by the end of the project) for several reasons that range from confidentiality of the OEM vehicle data to avoiding anticipated disclosure of materials that is going to be used for exploitation and/or upcoming publications.

Hence, balancing the needs between producing Open Research Data and IPR protection, the following data are published on ZENODO https://zenodo.org/, at the domain for Dreams4Cars Horizon 2020 Project community https://zenodo.org/communities/dreams4cars/ (DOI: 10.5281/zenodo.3582054, 10.5281/zenodo.3582953). After the conclusion of the Dreams4Cars this repository will be maintained updated with any future follow-up related publications and to promote further developments of the D4C concepts.

1.2 Experimental data from test vehicles

Section 2 of this deliverable documents the experimental data collected by the MIA car test vehicle (DOI: 10.5281/zenodo.3582953). We give datasets that are used for training the lateral and longitudinal vehicle forward models and example neural networks and training procedures that produce the forward models.

This corresponds to providing examples for the public version of the final simulation system (D3.3, section 3.1).

1.3 Simulation tools, examples and data

Section 3 of this deliverable documents the OpenDS environment with the final version of the Codriver (DOI: 10.5281/zenodo.3582054). Use of this software is regulated by a license agreement[1]. With this tool we also give example scenarios.
Compared to other free simulators this suite provides:

a) The Codriver agent.
b) A toolchain for automatic generation of driving scenarios.
c) Simulation based on (third-party) OpenDRIVE road description files.
d) Simulation at fixed step size (faster-than-real-time and deterministic simulation).
e) Support of two physics engines: Chrono and Bullet engine.

The latter (e) allows the user to choose from a basic and an advanced vehicle dynamics simulation. The basic vehicle dynamics model (based on the Bullet engine) is highly efficient and more realistic than in other free simulators and may be sufficient for testing behaviours that do not require hard manoeuvres (preventive safety). A more complete vehicle dynamics model (based on the Chrono engine) can be configured for driving in emergency situations with hard manoeuvres (but note that to efficiently drive one vehicle, the matched inverse models must be trained and used: the current inverse dynamics matches the provide examples).
Collectively, the data and tools provided in section 3 correspond to the public version of the agent described in D2.3.

[1] OpenDS is open source. The Codriver library may be used under the conditions listed in the repository.

With the open simulation environment, many research and evaluation activities become possible to external researchers and industrial users, such as, e.g.:

1) The Codriver agent integrated into OpenDS environment can be installed into driving simulators for various realistic studies on human-automated vehicle interactions. The fact that a very realistic and human like codriver agent is used constitutes a significant advancement in these kinds of studies.
2) The Codriver agent in collaboration with the OpenDS environment can be used for Reinforcement Learning framework studies (i.e., learning high-level behavioural parameters) following the lines of D3.3.
3) Developers of automated driving functions can use the agent as a benchmark.
4) Researchers can use the environment to test different types of driving assistance functions.

This list is of course non-exhaustive, because many other uses can be done with the released environment.

2 Experimental data from test vehicles

2.1 Longitudinal dynamics of the MIA car

Here we demonstrate the learning of forward models, following the methods explained in D3.3. The following table lists the data that were collected in 4 different test sessions that correspond to the test pans listed in section 2.1.1 of deliverable D5.1 (Vehicle Dynamics Tests for Learning Forward/Inverse models).

The datasets are stored in ZENODO (DOI: 10.5281/zenodo.3582953) and summarized in Figure 1. Data are stored both in the form of HDF5 files and in the form of Wolfram's .mx files. The datasets contain the relevant signals for the forward models, which have been extracted from the original ROS rosbags recorded during tests. More details concerning the datasets are given in the information sheets (the .docx files) included in the dataset folders.

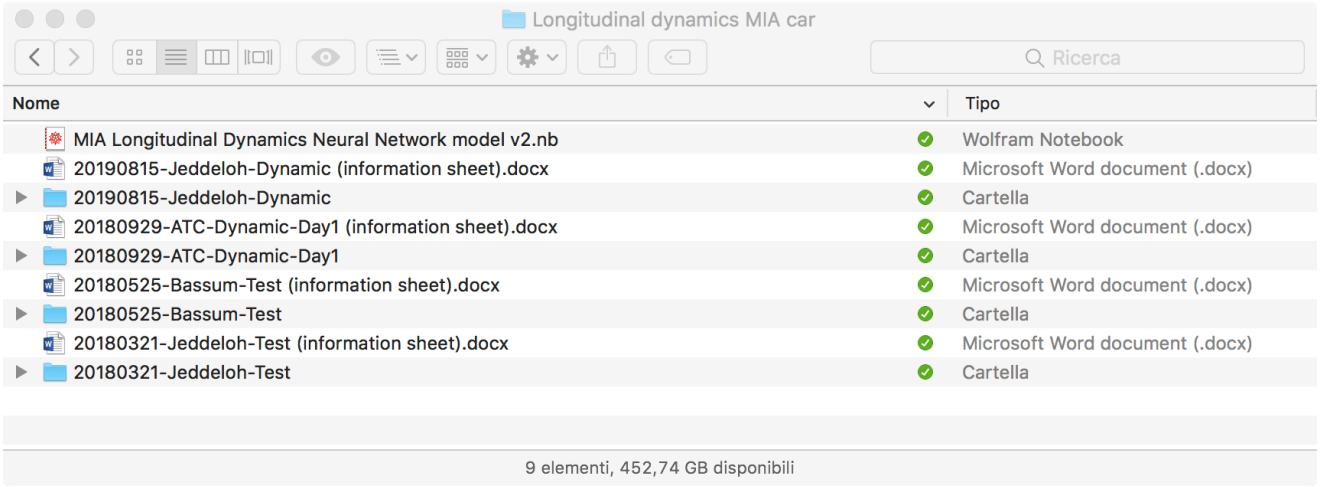


Figure 1: Dataset folder for the longitudinal dynamics of the MIA car.

A Wolfram Mathematica notebook ("MIA Longitudinal Dynamics Neural Network model v2.nb") is provided that opens both file types and guides step by step to the construction, training and evaluation of the vehicle forward model. It can be opened with the free Wolfram CDF player (https://www.wolfram.com/player/) and used as a guide to reproduce the same workflow in any other deep learning framework. For a live demonstration the notebook can be executed with Wolfram Mathematica (a free 15 days trial license is available https://www.wolfram.com/mathematica/trial/). The Mathematica version used for the example is 12.0.

Concerning the 4 datasets, it has to be noted that the data collected in the first two sessions (March and May 2018) have been pre-processed to eliminate various measurement issues (including filtering with a Butterworth filter of order 3 and cut frequency of 1Hz). Hence the first two dataset have a different level of noise than the latter two, and might not represent the exact dynamics of the test vehicle.

The forward/inverse models used in the project have been trained on the third dataset. The fourth dataset is on gravel surface and served to make comparisons with the asphalt surface of the third dataset.

Table 1: Test sessions for collection of data for the learning of forward models

Dataset	Date and place	Contents (HDF5 and MX file formats)
20180321-Jeddeloh-Test	2018/3/21 Jeddeloh (gravel surface)	36 start and stop manoeuvres (throttle steps followed by brake steps). Longitudinal velocity (**filtered**) Longitudinal acceleration (**filtered**) Throttle command Brake command **N.B. data have been pre-processed to filter several issues.**
20180525-Bassum-Test	2018/5/25 Bassum race track (asphalt surface)	24 start and stop manoeuvres (throttle steps followed by brake steps). Longitudinal velocity (**filtered**) Longitudinal acceleration (**filtered**) Throttle command Brake command **N.B. data have been pre-processed to filter several issues.**
20180929-ATC-Dynamic-Day1	2018/9/29 Aldenhoven Testing Center (asphalt surface)	14 brake steps, 14 throttle steps, 7 random throttle input Longitudinal velocity (raw from odometer) Longitudinal acceleration (from IMU) Throttle command Brake command
20190815-Jeddeloh-Dynamic	2019/8/15 Jeddeloh (gravel surface)	36 start and stop manoeuvres Longitudinal velocity (raw from odometer) Longitudinal acceleration (from IMU) Throttle command Brake command

2.2 Lateral dynamics of the MIA car

This section is like the previous one but for the lateral dynamics of the MIA car (with same DOI: 10.5281/zenodo.3582953). The following table lists the data of 3 different test sessions. The first one is made of steering sine and steering step inputs used for training. The second one refers to a driving situation on the

same test track (ATC). The third one is on a different test track which has much narrower curves, as sharp as ~4 m radius (the MIA is a minute car which can afford such small radiuses).

The datasets are stored in ZENODO with same criteria of the previous section (Figure 2).

A Wolfram Mathematica notebook ("MIA Lateral Dynamics Neural Network model v2.nb") is provided that opens both file types and guides step by step to the construction, training and evaluation of the vehicle lateral forward model.

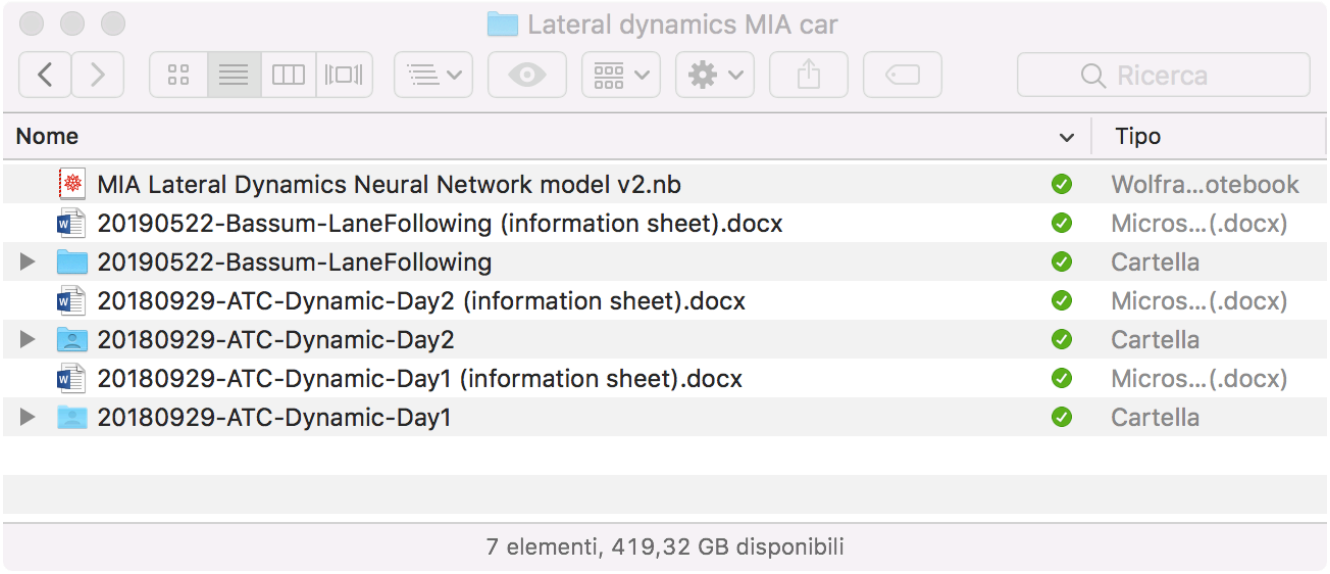


Figure 2: Dataset folder for the Lateral dynamics of the MIA car.

Table 2: Test sessions for collection of data for the learning of lateral forward models

Dataset	Date and place	Contents (HDF5 and MX file formats)
20180929-ATC-Dynamic-Day1	2018/9/29 Aldenhoven Testing Center (asphalt surface)	12 sinusoidal sweep steering inputs with variable amplitude and frequency (from 0.2 to 1.5 Hz). 6 steering step inputs with variable amplitude. Longitudinal velocity (raw from odometer) Steering wheel angle Yaw rate (from IMU)
20180930-ATC-Dynamic-Day2	2018/9/30 Aldenhoven Testing Center (asphalt surface)	2 general course recordings Longitudinal velocity (raw from odometer) Steering wheel angle Yaw rate (from IMU)

20190522-Bassum-LaneFollowing	2019/5/22 Bassum race track (asphalt surface)	15 general course recordings (with narrow curves) Longitudinal velocity (raw from odometer) Steering wheel angle Yaw rate (from IMU)

3 Simulation Tools, Examples, and Data

This chapter deals with open data generated by the simulation environment (DOI: 10.5281/zenodo.3582054).

Section 3.1 explains how to use the automatized terrain and road generation toolchain in order to create complex driving situations for simulation with OpenDS.

Section 3.2 presents selected examples which have been described in D5.1 – Test Plans, Methods, and Metrics as simulation fidelity tests. These examples have been created with the help of the toolchain described in Section 3.1.

An overview of the simulation software implemented in *Dreams4Cars,* which has been made publicly available, is given in Section 3.3.

3.1 Generation of Driving Scenarios

In the following, we present a toolchain to create road networks aligned to realistic terrain models for the simulation with *OpenDS*, the central component of the *Dreams4Cars* simulation environment. The toolchain consists of several individual applications transforming a simple road description file (RDf) into a 3D terrain and road network model with semantic information about the road and lane geometry. Each application in the toolchain can be accessed from the command line, reading the output of its predecessor and providing input to its successor. Figure 3 depicts the complete toolchain including flow of data.

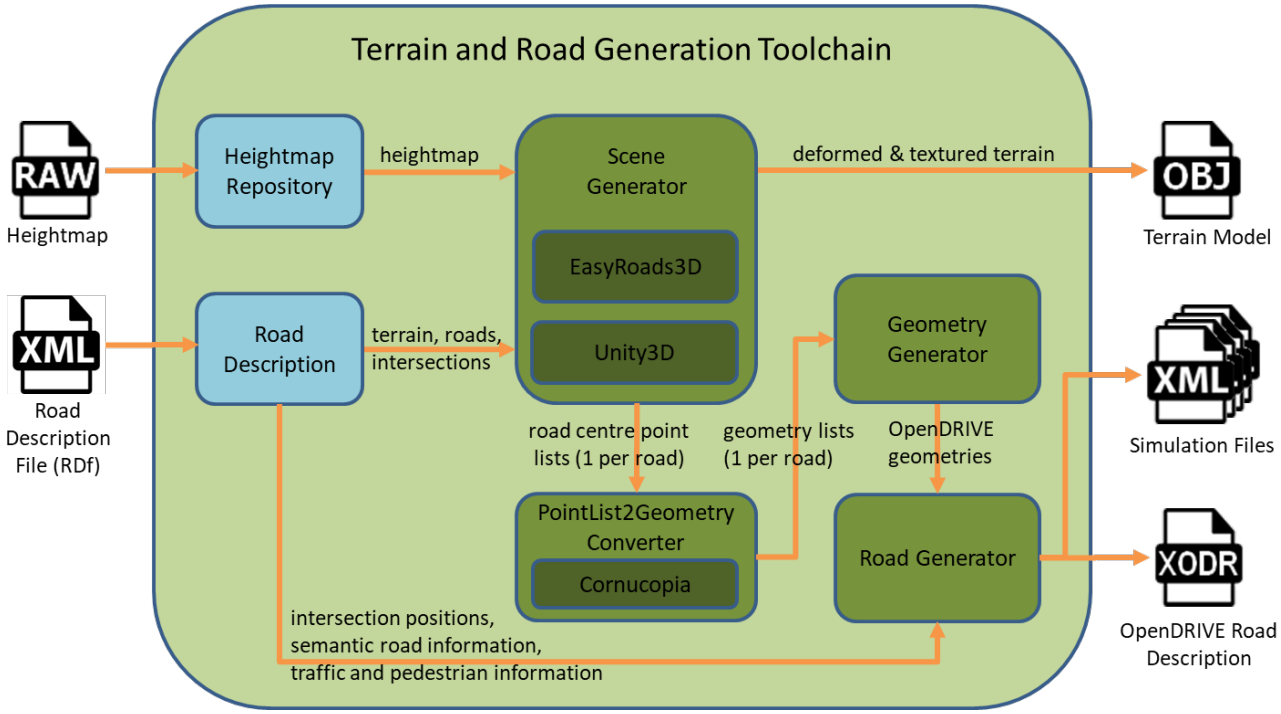


Figure 3: Building blocks of the Terrain and Road Generation Toolchain

The complete road generation process, as shown in Figure 3, starts when a RDf is passed to the terrain generation component (*Scene Generator*), which extracts terrain and road network information from this file. Accordingly, a 16-bit heightmap (in RAW format) from the central heightmap repository is loaded to generate a 3D terrain model. On top of this model, roads are placed, deforming the initial terrain if necessary. The (deformed) terrain is saved in a simulator-readable format (Wavefront OBJ) and for each road segment a list of 2D coordinates is generated representing the road reference line, which usually is equal to the centreline. Each coordinate list is filled with the absolute position data of the original reference line (approx. one coordinate per meter) and is forwarded to the geometry generation tools (cf. Figure 3). In the first geometry generation tool (*PointList2Geometry Converter*), all the point lists are processed sequentially by the *Cornucopia* algorithm to generate one set of curve primitives out of every point list. In the next tool (*Geometry Generator*), each set of curve primitives is converted into an OpenDRIVE geometry and, finally, the *Road Generator* puts together the OpenDRIVE geometries according to the initial road description by generating junctions to connect road sections whenever necessary. The result of the *Road Generator* is a road network description in OpenDRIVE format[2], which matches the previously generated terrain. Both terrain and OpenDRIVE files are forwarded to the driving simulator (*OpenDS*) – together with simulation files – and the road generation process terminates. The simulation files are generated from templates and contain typical parameters and settings used to launch OpenDS with the generated terrain and OpenDRIVE files. Table 3 shows the location of the individual tools in the directory structure of the simulation environment.

Tool/Resource	Directory
road description files (RDf)	./input/
automated toolchain scripts	./input/{*Linux_executables/, Windows_executables/*}
heightmaps	./heightmaps/
Scene Generator	./tools/SceneGenerator/SceneGenerator{.exe, .x86, .x86_64}
PointList2Geometry Converter	./tools/PointList2Geometry/P2GConverter{.exe, }
Geometry Generator	./tools/OpenDS_4.9/GeometryGenerator.jar
Road Generator	./tools/OpenDS_4.9/RoadGenerator.jar
OpenDS	./tools/OpenDS_4.9/OpenDS.jar
output folder	./output/

Table 3: Overview of tools/resources and their location in the directory structure

Linux and Windows scripts are available to execute the complete toolchain in order to automatically transform one of the sample RDfs (located in *./input/*) into a 3D model and an OpenDRIVE file by just one mouse click. The Linux and Windows scripts can be found in the folders *./input/Linux_executables/* and *./input/Windows_executables/*, respectively, named after the sample RDfs. Executing a script results in the

[2] At the time of implementation OpenDRIVE Specification Format version 1.4H was the latest version. For further details visit: http://www.opendrive.org

generation of a 3D terrain model, an OpenDRIVE road description file, and five generic simulation files needed to run the scenario by OpenDS. The resulting files are stored in the OpenDS folder (where they are used for simulation) at the following locations:

3D terrain model: *./tools/OpenDS_4.9/assets/Scenes/<current timestamp>/*

OpenDRIVE file: *./tools/OpenDS_4.9/assets/DrivingTasks/Projects/<current timestamp>/*

Simulation files: *./tools/OpenDS_4.9/assets/DrivingTasks/Projects/<current timestamp>/*

Furthermore, these files as well as all intermediate results (point lists, curve primitives, etc.) of the toolchain are copied to *./output/<current timestamp>/*.

Apart from executing the tools of the toolchain by one of the scripts, each tool can be executed individually as described below.

The following sections provides additional details about the road description specification (Section 3.1.1), which is used to create RDfs , and the links of the road generation toolchain (Sections 3.1.2, 3.1.3, and 3.1.4), which processes a road description file. Section 3.1.5 is about the driving simulation software *OpenDS*, which is capable of loading the resulting terrain and road network.

3.1.1 Road Description Specification

This section introduces the specification used to define (almost) arbitrary road networks represented by human-readable XML files. The specification allows the definition of the underlying terrain structure as well as the exact pathway of the road geometry including lane layout, intersections, speed limits, and planned vehicular trajectories (the Codriver agent generates its own trajectory). Furthermore, trajectories of pedestrians can be defined relative to the road geometry. A full description of the specification can be obtained from the XML schema file, which can be found at *./input/roadDescription.xsd*.

The XML structure of a road description file is divided into four main parts: terrain description, road segment description, intersection description, and traffic description (cf. Listing 1), which are described below in more details.

```
<roadDescription>
    <terrain>...</terrain>
    <segments>...</segments>
    <intersections>...</intersections>
    <traffic>
        <codriver>...</codriver>
        <vehicles>...</vehicles>
        <pedestrians>...</pedestrians>
    </traffic>
</roadDescription>
```

Listing 1: Road Description File (RDf) – overview

3.1.1.1 Terrain Description

In this part, terrain properties like extent, heightmap, and starting point of the road network must be specified. The heightmap can be created in advance with a tool of choice; however, we recommend using

L3DT[3], a Windows application for generating terrain maps and textures. Heightmaps must be saved in 16-bit RAW format. Table 4 lists all available properties and Listing 2 gives an example of a terrain description.

Property	Type	Unit	Description
extent/@length[4]	Float	meter	Length of terrain
extent/@width	Float	meter	Width of terrain
extent/@maxHeight	Float	meter	Difference between lowest and highest points
heightmap/@path	String		Path to heightmap file (RAW format)
heightmap/@resolution	Int		Resolution of heightmap (square)
roadStartingPoint/@x	Float	meter	x offset of starting point for road generation
roadStartingPoint/@z	Float	meter	z offset of starting point for road generation

Table 4: Road Description File – terrain properties

```
<terrain>
    <extent length="500" width="500" maxHeight="15" />
    <heightMap path="heightmaps/heightmap_2048_001.raw" resolution="2048"/>
    <roadStartingPoint x="100" z="400" />
</terrain>
```

Listing 2: Road Description File – sample terrain description

The initial road segment will be placed in the road starting point and the road generation will begin in x-direction.

3.1.1.2 Road Segment Description

The second part of the road description specification contains a list of all road segments. Since road networks are represented in terms of trees, each road segment (node) needs to have a unique ID in order to define predecessor/successor relations (edges) to connect the root with the leaves. Figure 4 depicts a sample road tree consisting of five road segments (one root node, one inner node, and three leaf nodes) and one intersection.

[3] Large 3D Terrain Generator, Bundysoft: http://www.bundysoft.com/L3DT/

[4] XML elements are separated by "/" (slash) and attributes are marked with "@"

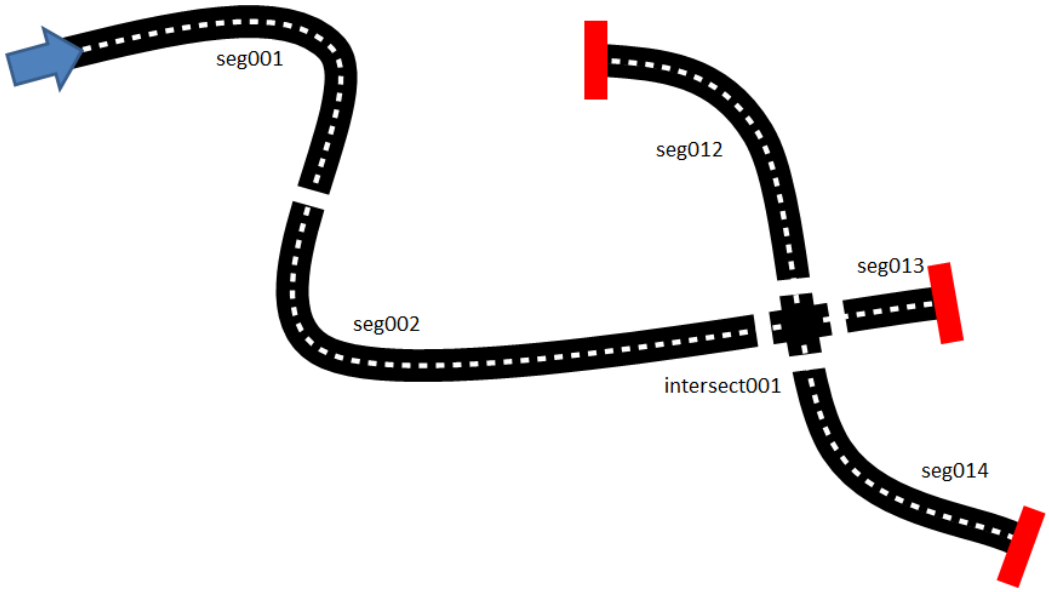


Figure 4: Visual representation of a tree-like road description

Except for the root (Figure 4, *seg001*), which is separately marked as "initial", every segment has one predecessor and except for the leaves (Figure 4, *seg012-seg014*), every segment has one successor. The successor of a segment can be either another segment or an intersection (junction); intersections may have multiple successor segments, however, no succeeding intersection. In the example, *seg001* is succeeded by *seg002*, which again is succeeded by *intersect001*. Be aware that the definition of segments must contain exactly one "initial" segment.

The layout of each segment is defined by a sequence of geometries (curves and straights) which can have arbitrary curvature (angle) and length. Furthermore, each segment contains information about the number of lanes, the lane width, and the speed limit. All available properties of a road segment can be found in Table 5.

Property	Type	Unit	Description
@id	String		Unique ID of road segment
@initial	Boolean	true/false	Whether road segment is root
geometries/geometry/@length	Float	meter	Length of underlying geometry
geometries/geometry/@curvature	Float	degree	Curvature of geometry (straight = 0)
successor/segment/@ref	String		Reference of successor segment
successor/intersection/@ref	String		Reference of successor intersection
laneLayout/noOfLanes	Int		Number of lanes (1, 2, or 4)
laneLayout/width	Float	meter	Width of lane
laneLayout/speedLimit	Float	km/h	Speed limit in generation direction
laneLayout/speedLimitOppositeDirection	Float	km/h	Speed limit in opposite direction
surface/friction	Float		Road friction coefficient (optional)

Table 5: Road Description File – road segment properties

The `curvature` attribute describes the angle (in degrees) of the curve of a "geometry" entry. For instance, a value of "-90" describes a geometry that is curved to the left at a right angle, the value "0" describes a straight geometry, and "20" describes a geometry curved to the right at 20 degrees.

The `length` attribute defines the length (in meters) the given curve has. Changing the length of a curve can also be used to increase/decrease the curve radius.

A sample road segment description (implementing *seg001* of Figure 4) can be found in Listing 3.

```
<segments>
    <segment id="seg001" initial="true">
        <geometries>
            <geometry length="50" curvature="0" />
            <geometry length="50" curvature="120" />
        </geometry>
        <laneLayout>
            <noOfLanes>2</noOfLanes>
            <laneWidth>2.60</laneWidth>
            <speedLimit>80</speedLimit>
        </laneLayout>
        <successor>
            <segment ref="seg002" />
        </successor>
    </segment>
</segments>
```

Listing 3: Road Description File – sample road segment description

The `<successor>` element of segment *seg002* in the previous example (cf. Figure 4) points to intersection *intersect001* and the corresponding line in the road segment description will read as follows: `<segment ref="intersect001" />`.

3.1.1.3 Intersection Description

The intersection part is optional and may contain a list of intersections which must have a unique ID in order to be referenced as the successor of one of the road segments. Furthermore, each intersection must have two (in case of T-junctions) or three (in case of crossroads) successor segments (= outgoing segments) which must be defined in the road segments part (3.1.1.2). Every outgoing segment of an intersection must be assigned to one of the following directions: -90, 0, and 90. Each direction may not be assigned to more than one segment. -90/0/90 denotes that the outgoing segment is connected to the left/straight/right (from the perspective of the incoming road). In case of a crossroad, all four outgoing segments must be provided, in case of a T-junction, one of the outgoing directions may be omitted.

Table 6 represents the properties of an intersection.

Property	Type	Unit	Description
@id	String		Unique ID of intersection
@type	String		(not in use)
outgoingSegment/@ref	String		ID of outgoing road segment
outgoingSegment/@degree	String	degree	Connection direction (-90 = left, 0 = straight, 90 = right)

Table 6: Road Description File – intersection properties

The previous example (cf. Figure 4) contains a four-way intersection (=crossroad) with one incoming road (*seg002*) and three outgoing roads. Segment *seg012* is connected to the left (-90), *seg013* straight ahead (0), and *seg014* to the right (90). Listing 4 shows the corresponding intersection description of *intersect001*.

```
<intersections>
    <intersection id="intersect001" type="crossing">
        <outgoingSegment ref="seg012" degree="-90"/>
        <outgoingSegment ref="seg013" degree="0"/>
        <outgoingSegment ref="seg014" degree="90" />
    </intersection>
</intersections>
```

Listing 4: Road Description File – sample intersection description

3.1.1.4 Traffic Description

This part of the road description specification describes the traffic present in a driving scenario, which includes computer-controlled vehicles and pedestrians as well as the vehicle controlled by the Codriver. In the following subsections, more details about these three types of traffic will be introduced.

3.1.1.4.1 Codriver-controlled Vehicle

The Codriver-controlled vehicle is the substantial element of the simulation. In this subsection, we demonstrate how to set up Codriver parameters like initial position and target position, intended pathways, and interaction with the simulation environment.

The initial position of the Codriver-controlled vehicle must be specified by providing segment ID, lane position and offset from the starting point of the segment. The given segment ID must be equal to one of the segment IDs defined in Section 3.1.1.2. The lane position is given by a non-zero integer value according to Figure 5. Facing the direction of road creation, lanes left of the lane reference line have positive numbers (red lanes) and lanes right of the lane reference line have negative numbers (green lanes). If there is no lane offset (which is usually the case for segments generated by this toolchain), lane reference line and road reference line are equal. The enumeration of the lanes starts at the lane reference line in positive and negative direction. E.g. the position of the first lane to the right will be “-1”, while the position of the fourth lane to the left will be “4” (if it exists). The longitudinal offset must be given in meters from the beginning of the given segment and must not exceed the length of the segment.

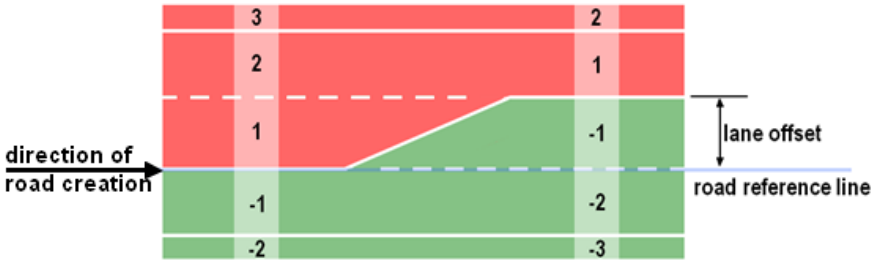


Figure 5: Lane positions of an OpenDRIVE road – with and without lane offset

Optionally, preferred pathways can be specified for driving scenarios containing intersections. For this purpose, every vehicle provides a specific connection list which allows specifying a from/to relation for a given intersection. For instance, a vehicle can be set up to turn right at *intersect001* from *seg002* to *seg014* (see Figure 4).

Furthermore, a target position – consisting of segment ID (e.g. *seg014*), lane position (e.g. *-1*) and offset (e.g. *20*) from the starting point of the segment – and a condition for termination of the simulation may be added. Termination conditions include the Codriver-controlled vehicle reaching its target position and/or reaching the end of any leaf vertex of the road tree. All available vehicle properties can be found in Table 7 and a sample Codriver description implementing the aforementioned values is shown in Listing 5.

Property	Type	Unit	Description
startPosition/@segment	String		ID of initial road segment of Codriver
startPosition/@lane	Int		Initial lane position of Codriver
startPosition/@s	Float	meter	Starting position relative to lane start
connection/@intersectionID	String		ID of intersection
connection/@from	String		ID of source road segment
connection/@to	String		ID of target road segment
targetPosition/@segment	String		ID of initial road segment of Codriver
targetPosition/@lane	Int		Initial lane position of Codriver
targetPosition/@s	Float	meter	Starting position relative to lane start
terminateSimulation/ @onTargetPositionReached	Boolean	true/false	Stop if Codriver reached target position
terminateSimulation/ @onRoadEndReached	Boolean	true/false	Stop if Codriver reached any road end

Table 7: Road Description File – codriver properties

```
<traffic>
    <codriver>
        <startPosition segment="seg001" lane="-1" s="10" />
        <preferredConnections>
                <connection intersectionID="intersect001" from="seg002" to="seg014" />
            </preferredConnections>
        <targetPosition segment="seg014" lane="-1" s="20" />
        <terminateSimulation onTargetPositionReached="true" onRoadEndReached="false" />
    </codriver>
</traffic>
```

Listing 5: Road Description File – sample Codriver description

3.1.1.4.2 Vehicles

Computer-controlled vehicles are optional. If present, a unique ID and a starting position (segment ID, lane position and longitudinal offset (cf. 3.1.1.4.1)) must be specified for each vehicle. Optionally, preferred pathways can be specified for driving scenarios containing intersections (similar to 3.1.1.4.1). Furthermore, an

individual speed limit may be set for each vehicle by the use of the optional <maxSpeed> element. All available vehicle properties can be found in Table 8 and a sample vehicle description is shown in Listing 6.

Property	Type	Unit	Description
@id	String		ID of vehicle
startPosition/@segment	String		ID of initial road segment of vehicle
startPosition/@lane	Int		Initial lane position of vehicle
startPosition/@s	Float	meter	Starting position relative to lane start
connection/@intersectionID	String		ID of intersection
connection/@from	String		ID of source road segment
connection/@to	String		ID of target road segment
maxSpeed	Float	km/h	Individual speed limit of vehicle

Table 8: Road Description File – vehicle properties

```
<traffic>
    <vehicles>
        <vehicle id="car01">
            <maxSpeed>50</maxSpeed>
            <startPosition segment="seg014" lane="1" s="30" />
            <preferredConnections>
                <connection intersectionID="intersect001" from="seg014" to="seg012" />
            </preferredConnections>
        </vehicle>
    <vehicles>
</traffic>
```

Listing 6: Road Description File – sample vehicle description

3.1.1.4.3 Pedestrians

Pedestrians are optional. If present, a unique ID, a start position, and a list of walking targets must be specified for each pedestrian. Optionally, one can specify whether the beginning of the pedestrian's walk will depend on the position of the Codriver-controlled vehicle. Listing 7 shows a sample pedestrian definition.

```
<traffic>
    <pedestrians>
        <pedestrian id="pedestrian01" >
            <startPosition segment="seg002" lateralOffset="-5.3" s="0" />
            <targets>
                <target lateralOffset="-5.3" s="30" speed="4.1" />
                <target lateralOffset="5.3" s="30" speed="2.8" />
                <target lateralOffset="5.3" s="60" speed="4.3" />
                <target lateralOffset="-5.3" s="60" speed="1.5" />
            </targets>
            <triggerPosition segment="seg001" lane="-1" s="20" />
        </pedestrian>
    </pedestrians>
</traffic>
```

Listing 7: Road Description File – sample pedestrian description

Besides the `id` attribute, which needs to be provided with a unique ID, the `<startPosition>` element and the `<targets>` element need also to be present. The `<triggerPosition>` element is optional.

`<startPosition>` represents the initial position of the pedestrian provided by 2D coordinates (lateral offset `lateralOffset` and longitudinal offset `s`) relative to the given road segment (`segment`). All three attributes are required. Since the road segment of the start position serves as a reference system of the pedestrian's position, making the pedestrian change between two road segments (i.e., two distinct coordinate systems) will not be possible. Due to this restriction, the indication of the segment has been omitted in the following `<target>` elements; the positional data implicitly refer to the segment given by `<startPosition>`.

`<targets>` represents a list of 2D way points the pedestrian will process in the given order by walking to the given relative positions (`lateralOffset` and `s`) successively at the given speed (`speed`). At least one `<target>` sub-element with the required attributes `lateralOffset`, `s`, and `speed` must to be provided.

`<triggerPosition>` allows to provide a relative position (`segment`, `lateralOffset`, and longitudinal offset `s`) which needs to be approached by the Codriver vehicle to make the pedestrian start walking towards the first target. If the optional `<triggerPosition>` element is present, `segment`, `lane`, and `s` must be provided; otherwise, the pedestrian will start walking immediately after the starting the simulation. In contrast to the `<target>` elements, the position of the trigger is not limited to the road segment given by `<startPosition>` and, thus, the `segment` attributes of `<startPosition>` and `<triggerPosition>` may differ.

Table 9 shows types and units of all available attributes and sub-elements of a pedestrian element.

Property	Type	Unit	Description
@id	String		ID of pedestrian
startPosition/@segment	String		ID of initial road segment of pedestrian
startPosition/@lateralOffset	Float	meter	Initial lateral offset
startPosition/@s	Float	meter	Initial longitudinal offset
targets/target/@lateralOffset	Float	meter	lateral offset of target
targets/target/@s	Float	meter	longitudinal offset of target
targets/target/@speed	Float	km/h	speed towards target
triggerPosition/@segment	String		Codriver trigger position (segment, lane and longitudinal offset) to start walking
triggerPosition/@lane	Int		
triggerPosition/@s	Float	meter	

Table 9: Road Description File – pedestrian properties

Example. Figure 6 visualizes the example given in Listing 7. The initial position of the pedestrian *pedestrian01* is at the beginning (s=0) of road segment *seg002*. The initial lateral offset is 5.3 meters right of the road reference line (see green dot).

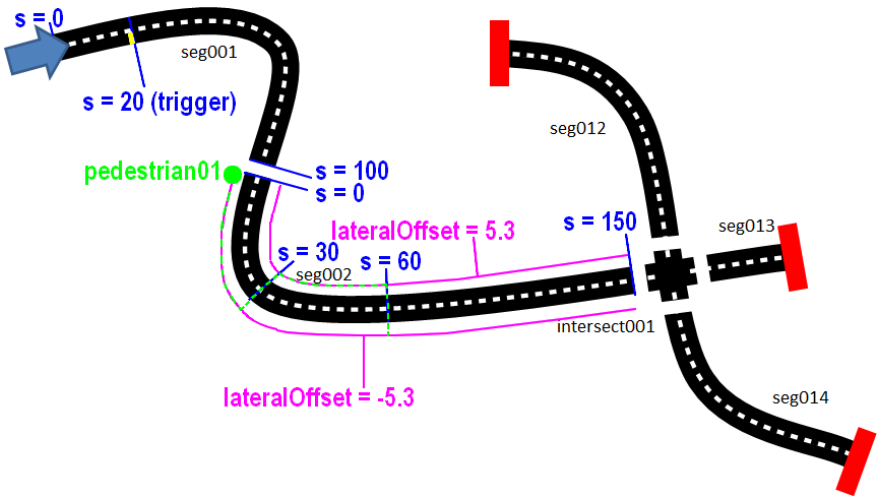


Figure 6: Sketch of the example described in Listing 7

When the codriver vehicle hits the trigger, which is in the right lane (lane=-1) of road segment *seg001* exactly 20 meters (s=20) from the road reference point, the pedestrian will start walking to the first target (green dashed line). The first target is located 30 meters away at s=30. As the lateral offset of the target is also 5.3 meters right of the road reference line, the pedestrian will walk parallel to the road at 4.1 km/h following the curve shape at an exact distance of 5.3 meters.

After reaching the first target, the pedestrian will turn left to cross the road at a right angle and a speed of 2.8 km/h. While crossing the road, the lateral offset will change from -5.3 to +5.3 and the longitudinal offset will remain at s=30. The pedestrian is now at the left of the road reference line.

On the other side of the road, the pedestrian will continue walking parallel to the road at a speed of 4.3 km/h until s=60 has been reached where she will turn right to cross the road again.

After reaching the final target, the pedestrian will rest there till the end of the simulation.

3.1.1.5 Hints and Restrictions

In this section, important information about the concepts explained above are clarified. Please follow these hints:

- Any occurrence of `segment` must always refer to a valid road segment ID.
- `s` must have positive values only; where `s="0"` always points to the beginning of a road.
- Pedestrians:
  - `lateralOffset`: positive values (in meters) denote positions left of the road reference line; negative values denote positions right of the road reference line.
  - A pedestrian will walk from the previous target (or start position) to the current target by interpolating the lateral and longitudinal offsets linearly.
  - The speed of a pedestrian will be constant while moving between two targets and can only be changed when a target has been reached.

For the sake of completeness, we like to point out known limitations of the road description specification:

- The course of the road cannot be specified very precisely as the underlying generation tool (EasyRoads3D Pro) tends to “optimize” the curvature of the road in an unpredictable manner, e.g. angular shaped curves will be rounded.
- Junctions are restricted to either crossroads or T-junctions where the intersecting roads must be connected exactly at a right angle. This is a requirement of the used road generation and terrain deformation tool.

- Road networks can only be defined in a tree-like representation starting with an initial road segment (root), which is connected to all other segments by junctions. Thus, cyclic road networks cannot be defined with this specification format. This limitation traces back to the underlying road generation tool as the precise course of a road cannot be predicted. Enormous efforts would be required to make sure that a road will exactly end at a pre-defined position.

3.1.2 Terrain Generation

This section introduces the terrain generation procedure, which is carried out by the *Scene Generator* (*./tools/SceneGenerator/*), a Unity application based on EasyRoads3D Pro. In the following we explain how terrain generation works and how the tool can be used.

EasyRoads3D Pro is a road network editor extension for Unity[5] which allows rapid generation of custom 3D road models. The user may start with a simple terrain using the Unity terrain generation tools or load a pre-defined heightmap to generate a more complex terrain. On top of the terrain, reference points can be placed by mouse clicks, which will be connected by (curved) road segments in order to construct realistic roads as shown in Figure 7.

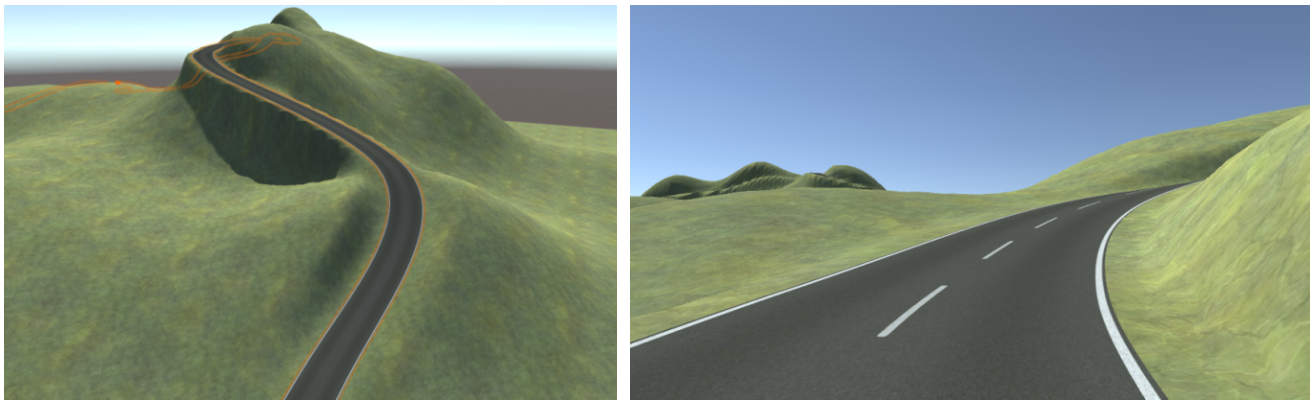


Figure 7: EasyRoads3D Pro – simple road models

Although the software is focused on manual road network generation by a GUI-based editor, EasyRoads3D Pro comes with an API, which is still under development but already allows executing many important functions of the application by Unity Scripts. API access is the fundamental requirement to automatize the road generation process including terrain generation from pre-defined heightmaps, placing intersections, and connecting them with road segments of arbitrary shape and width.

Another impressive feature of EasyRoads3D Pro, which influenced the decision to utilize this software, is the automatic deformation of terrain whenever a new road segment is added to undulating terrain (cf. Figure 8). This allows the creation of negotiable roads with little superelevation and fair inclination angles.

[5] https://unity.com/

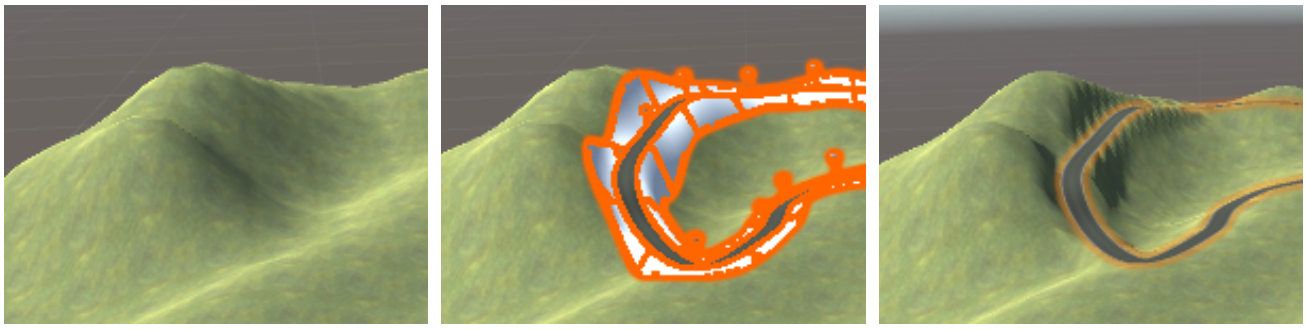


Figure 8: EasyRoads3D Pro – terrain deformation

A limitation of EasyRoads3D Pro at the time of implementation was, that the latest version (3.0 beta) was lacking documentation and the provided source code was obfuscated. Furthermore, the software did not provide API functions for all the features of the manual editor (e.g. roundabouts could not be created, and generated road networks could not be exported to a 3D model format). Instead of exporting 3D models consisting of terrain and road, we therefore decided to export the terrain only and extract the absolute 2D coordinates of a road's centreline to separate text files. The resulting coordinates of the road centrelines could be used as starting point to re-create road geometries and finally project them on top of the terrain model at a later stage. Here, we make it a condition that the road reference line of the geometries to be generated in the next section will be located at the same place on the terrain model where the centreline of the road generated by EasyRoads3D has been located before.

Technically, the terrain generation process consists of the following steps. First, a road description file is loaded and processed by the Scene Generator in order to create a new terrain model given by a heightmap reference in the road description file. After all road segments specified in the RDf have been added to the terrain using EasyRoads3D Pro, the generation process is finalized by deforming the terrain in order to allow a smooth transition between terrain and road. The deformed terrain is provided with a texture and exported by Unity exporting routines to Wavefront (*.obj) format. Furthermore, the pathway of each road segment's centreline is stored as 2D point list with absolute coordinates in a separate text file (named after the road segment ID) before the meshes of the road objects are discarded. The resulting point lists is used by the next link of the toolchain to compute geometries.

A compiled version of the Scene Generator is available for Windows and Linux and can be executed from the command line using the following command:

Windows: `$ SceneGenerator.exe -input <IF> -output <PLF> -terrain <TF> -display on|off`

Linux: `$ SceneGenerator.x86_64 -input <IF> -output <PLF> -terrain <TF> -display on|off`

Where

`-input <IF>`	<IF> is the path to the input file (e.g. *roadDescription.xml*)
`-output <PLF>`	<PLF> is the output folder where the point lists will be stored
`-terrain <TF>`	<TF> is the output file path of the terrain model (OBJ format)
`-display on|off`	indicates whether the generation process will be interrupted to display the terrain model ("on") or whether the application will be closed automatically after computation ("off").

Executing the Scene Generator creates a terrain model (including **.obj*, **.mtl* and texture files) in the given output folder <TF> and a set of text files containing 2D point lists in the given output folder <PLF>. For each road segment contained in the road description, a text file is generated containing coordinates of the segment's centreline. The fact that these text files are named after the corresponding road segment ID, highlights the necessity of providing unique road segment IDs.

The following sample command can be used to run the Scene Generator under Windows in order to process the input file *roadDescription.xml*:

```
$ SceneGenerator.exe -input roadDescription.xml -output output\pointlists\ -terrain output\obj\terrain.obj -display off
```

3.1.3 Point List to Road Reference Line

The following section explains the transformation of a 2D point list into a geometry that represents the reference line of a road. This geometry is described as a sequence of primitives of various types. We differentiate between the following primitives:

- **Line**: a straight line with zero curvature
- **Arc**: a curve with constant non-zero curvature
- **Spiral**: a curve with linear change of curvature (also known as Euler spiral or clothoid)

Disregarding intersections and elevation difference, an arbitrary road can be described by a sequence of the aforementioned primitives. According to the German road construction act, many turns in rural areas are in fact constructed using clothoidal parts between line and arc segments to provide a smooth steering phase when passing the lane section. Furthermore, the OpenDRIVE standard, which is supported by the driving simulation *OpenDS*, makes use of these three concepts to describe complex road shapes.

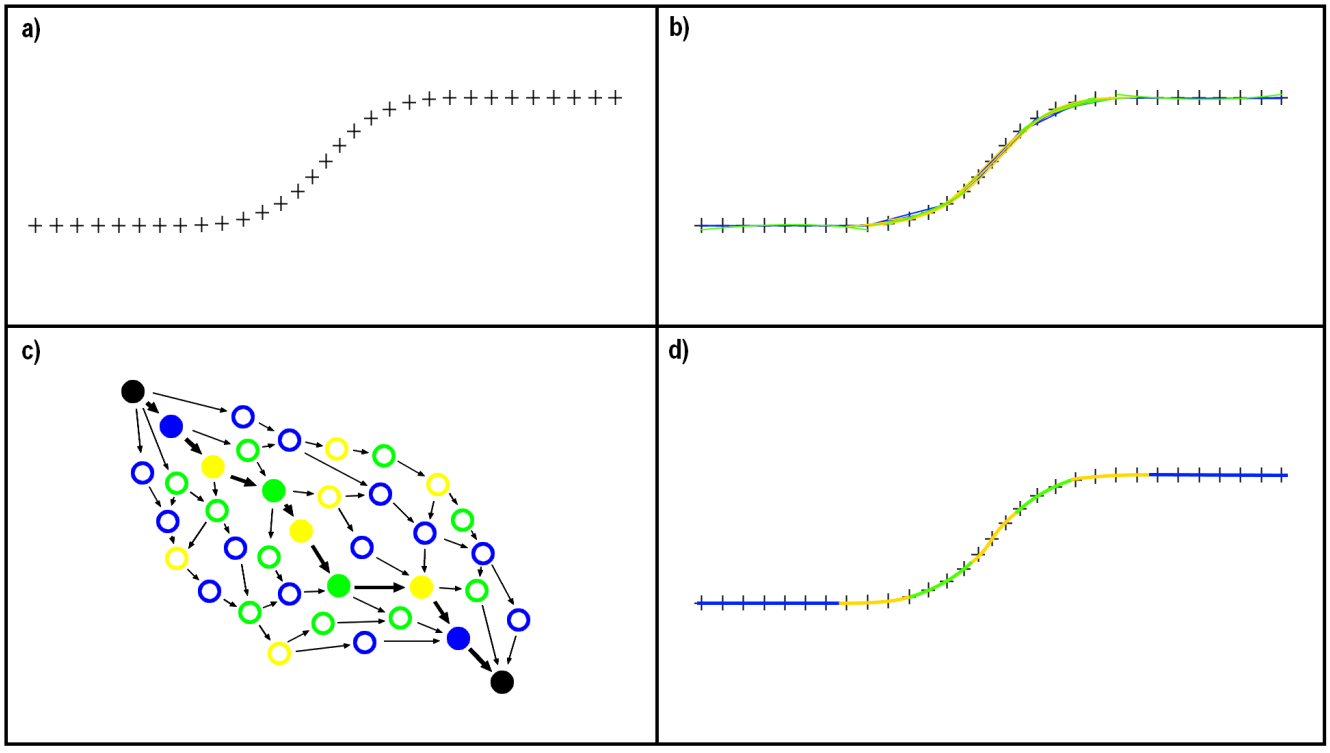


Figure 9: Cornucopia – Fitting primitives to a list of points (colour code: line, arc, spiral)

For the computation of a precise reference line (consisting of line, arc, and spiral segments) from a given point list, the *Cornucopia* library[6] turned out to be the ideal solution. This C++ library is an open-source project developed by Baran et al. [1] at the Massachusetts Institute of Technology. The algorithm is intended to approximate a mouse or tablet sketch stroke with a smooth continuous curve consisting of a set of lines, spirals, and arcs. The input can be derived from 2D coordinates and the resulting geometry concepts exactly meet the requirements of the OpenDRIVE road description format.

The algorithm applies the following processing steps to a list of 2D points to end up with a fair curve approximating the input (cf. Figure 9a):

1. Closed curve detection: determine whether the input curve is almost closed, and, if so, make it precisely closed.
2. Corner detection: determine whether there are sharp corners and remove them.
3. Resampling: to reduce the size of the problem and make the sampling more regular, resample the sketched stroke in a curvature-sensitive way.
4. Primitive fitting: for every contiguous subsequence of samples, fit a candidate line, spiral, and arc. This results in an over-complete set of overlapping primitives (cf. Figure 9b).
5. Graph construction: construct a weighted graph with the primitives as nodes and transitions between primitives as edges, such that weights denote the quality of the transition (cf. Figure 9c). To control the output of the algorithm, the costs for a line, an arc, and a spiral, for G^0, G^1, and G^2 transitions, for inflections (points where the curvature changes sign), the approximation error cost and the penalty for short primitives have been chosen in order to get optimal results for road construction. Table 10 lists the selected cost setup (also known as “Accurate (G2)” pre-set) for all configurable parameters of the *Cornucopia* curve fitting algorithm.
6. Shortest path: find an acceptable shortest path through the graph, validating transitions in the process. This step picks out a high-quality segmentation of the input point list into curve primitives and transitions between them.
7. Merging: enforce the continuity constraints on the chosen primitives by solving a nonlinear program (cf. Figure 9d).

Parameter	Min. Value	Value	Max. Value
Line cost	0	7.5	20
Arc cost	0	9	30
Clothoid cost	0	15	50
G0 cost	0	50	50
G1 cost	0	50	50
G2 cost	0	0	50
Error cost	0	5	10
Shortness cost	0	1	10
Inflection cost	0	20	100

Table 10: Cornucopia – primitive fitting costs (for reference, minimum and maximum values are provided)

[6] https://code.google.com/archive/p/cornucopia-lib

In order to integrate the *Cornucopia* library, we extended the source code by a routine for reading a two-dimensional point list from a given file and a routine for writing the resulting list of primitives to an output file. The output file consists of a sequence of line, arc, and spiral primitives as well as the initial position and orientation of the first primitive. Every subsequent primitive starts in the endpoint of its predecessor with the same orientation as its predecessor in that point. Line primitives only have a length property, arcs additionally have a curvature property, and spirals additionally have a second curvature property – one describing the curvature at the beginning and one at the end. Curvature values in between need to be interpolated linearly.

A compiled version of the modified *Cornucopia* library is available for Windows and Linux under the name *Point2Geometry Converter* (./tools/PointList2Geometry/) and can be executed from the command line as follows:

Windows: `$ P2GConverter.exe <inputFile> <outputFile>`

Linux: `$ P2GConverter <inputFile> <outputFile>`

Where `<inputFile>` and `<outputFile>` represent the paths to the input and output files (optional). If these arguments are not provided, the application will expect a text file named *input.txt* and create a text file named *output.txt* in the same folder (overwriting existing files).

The input list must comply with the pattern depicted in Listing 8 and consist of two or more points, which are represented as two-dimensional floating-point coordinates (x and y) separated by ";". Furthermore, the list must provide exactly one point per line. A sample output of the *Point2Geometry Converter* is shown in Listing 9.

```
x1;y1
x2;y2
x3;y3
  ⋮
xn;yn
```

Listing 8: Point2Geometry Converter – format of the input list

In the next link of the toolchain, the road reference lines of all involved road segments will be merged in order to create a valid OpenDRIVE file.

3.1.4 Road Reference Lines to OpenDRIVE Road Description

In this section we describe how the resulting road reference lines from the previous link of the toolchain are combined to a valid OpenDRIVE file. According to Figure 3, this conversion consists of two steps:

1. *Geometry Generator*: a set of curve primitives is converted to a set of OpenDRIVE geometries
2. *Road Generator*: a set of OpenDRIVE geometries is inserted into one OpenDRIVE template

In the following, both steps are described in more detail.

3.1.4.1 Geometry Generator

In order to transform curve primitives (resulting from the *Point2Geometry Converter*) into OpenDRIVE geometries, all line, spiral, and arc primitives need to be complemented with additional data. Listing 9 shows some sample output of the *Point2Geometry Converter* after fitting a sequence of `line → spiral → arc → spiral → line` primitives to a list of 2D coordinates. The starting point and initial heading of each curve primitive is equal to the end point (and heading) of the preceding primitive – except for the first primitive, which begins at the coordinates and heading given in the `<start>` element.

```
<road>
    <start x="-283.268" y="-201.359" hdg="3.321"/>
    <geometries>
        <line length="0.486" />
        <spiral length="3.174" curvStart="0.000" curvEnd="0.126" />
        <arc length="9.195" curvature="0.126" />
        <spiral length="3.174" curvStart="0.126" curvEnd="0.000" />
        <line length="0.486" />
    </geometries>
</road>
```

Listing 9: Sample output of the Point2Geometry Converter

In more detail, the transformation of the XML representation of Listing 9 into OpenDRIVE format (cf. Listing 10) requires the calculation of absolute position data (x, y, hdg) for each geometry. Furthermore, a consistent s-value representing the offset (in meters) from the beginning of the road reference line needs to be provided. Even though the data shown in Listing 9 and Listing 10 are equivalent, the OpenDRIVE representation (Listing 10) is much more detailed due to redundancy.

```
<planView>
    <geometry s="0.0" x="-283.268" y="-201.359" hdg="3.321" length="0.486">
        <line/>
    </geometry>
    <geometry s="0.486" x="-283.746" y="-201.446" hdg="3.321" length="3.174">
        <spiral curvStart="0.000" curvEnd="0.126"/>
    </geometry>
    <geometry s="3.66" x="-286.819" y="-202.220" hdg="3.522" length="9.195">
        <arc curvature="0.126"/>
    </geometry>
    <geometry s="12.856" x="-291.764" y="-209.356" hdg="4.690" length="3.174">
        <spiral curvStart="0.126" curvEnd="0.000"/>
    </geometry>
    <geometry s="16.031" x="-291.409" y="-212.505" hdg="4.891" length="0.486">
        <line/>
    </geometry>
</planView>
```

Listing 10: OpenDRIVE representation of the sample shown in Listing 9

The tool needed to transform the output of the *Point2Geometry Converter* into OpenDRIVE format is called *Geometry Generator* and is available as executable **.jar* file for Windows, Mac OS, and Linux. It can be found in *./tools/OpenDS_4.9/GeometryGenerator.jar* and can be executed from the command line as follows:

```
$ java -jar GeometryGenerator.jar <inputFile> <outputFile> <headless>
```

Where

`<inputFile>`	is the path to the input file, e.g. *geometryDescription.xml* (optional). An XML schema for the input file can be found in the same folder (*geometryDescription.xsd)*.
`<outputFile>`	is the path to the output file, e.g. *openDrive.xodr* (optional).
`<headless>`	indicates whether the window displaying the result will be suppressed (optional).

If these arguments are not provided, the application will expect a text file named *geometryDescription.xml* in the same folder and create a text file named *openDrive.xodr* in a timestamped subfolder of folder *openDRIVEData*. By default, the application will show the graphical result in a separate window (if not suppressed) as depicted in Figure 10.

The following sample command can be used to run the Geometry Generator in order to process the input file *geometryDescription.xml* and generate the output file *openDrive.xodr* in headless mode (without visual output):

```
$ java -jar GeometryGenerator.jar geometryDescription.xml openDrive.xodr headless
```

Be aware that the implicit execution of the *Geometry Generator* is not part of the road network generation toolchain and the aforementioned command will not be contained in the provided shell scripts (**.bat* and **.sh* files). Instead, the Geometry Generator will be executed internally by the *Road Generator* (cf. 3.1.4.2).

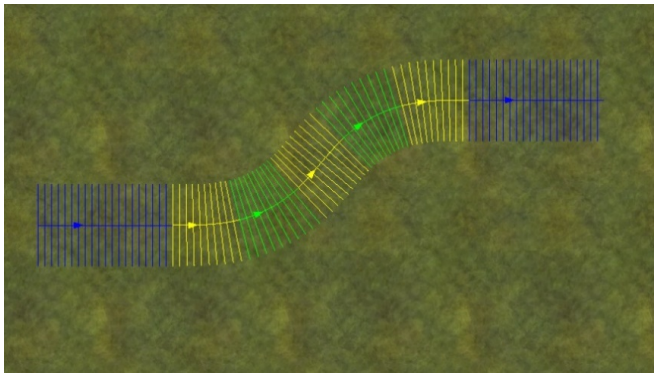


Figure 10: OpenDRIVE representation of a sample road segment (road reference line)

3.1.4.2 Road Generator

After transforming the reference line of each road segment into a separate file containing the OpenDRIVE representation thereof, the last tool of the chain, the *Road Generator*, can be used to merge all these files into one file, the OpenDRIVE road description. For this purpose, the initial road description file (cf. 3.1.1) must be processed a second time since all the information lost during the centre point export (e.g. road width, lane configuration, speed limit, etc.) must be added to the respective OpenDRIVE geometries in the final output file. Furthermore, the original road description contains information about how to connect road segments with each other – either using a direct connection between two adjacent roads (predecessor/successor relation) or using a custom-built intersection area (cf. Figure 11) in order to connect multiple road segments.

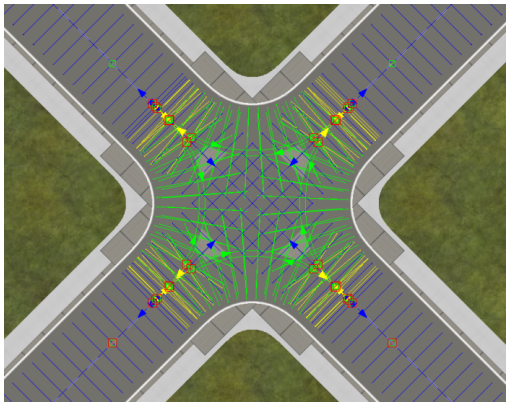


Figure 11: OpenDRIVE representation of a custom-built intersection area

The *Road Generator* is available as executable **.jar* file for Windows, Mac OS, and Linux. It can be found in *./tools/OpenDS_4.9/RoadGenerator.jar* and can be executed from the command line as follows:

```
$ java -jar RoadGenerator.jar <inputFile> <timestamp> <headless>
```

Where

`<inputFile>` is the path to the input file (RDf), e.g. *roadDescription.xml* (optional). An XML schema for the input file can be found in the same folder (*roadDescription.xsd*). Furthermore, for each road segment that is defined in the RDf, a separate XML file (named after the corresponding road segment ID) containing the OpenDRIVE representation thereof must be provided in the main folder of OpenDS.

`<timestamp>` is the timestamp of creation, e.g. *2019-11-22_13:45:20* (optional). This timestamp will be used to create and name a subfolder in the simulator's project folder in order to save the generated driving task files (output).

`<headless>` indicates whether the window displaying the resulting road network will be suppressed (optional).

If these arguments are not provided, the application will expect a text file named *roadDescription.xml* in the same folder and create a subfolder named after the current timestamp in the simulator's project folder (*./tools/OpenDS_4.9/assets/DrivingTasks/Projects/*). By default, the application will show the graphical result in a separate window (if not suppressed) as depicted in Figure 11.

The following sample command can be used to run the *Road Generator* in order to process the input file *roadDescription.xml* and save the output files to a folder named *2019-11-22_13:45:20* in headless mode (without visual output):

```
$ java -jar RoadGenerator.jar roadDescription.xml 2019-11-22_13:45:20 headless
```

The *Road Generator* processes the given road description file and expects a separate XML file for each road segment that is contained in the road description. These XML files must be in the main folder of *OpenDS* and be named after the corresponding road segment ID. The content of each geometry file is first transformed into the OpenDRIVE geometry format by internally executing the *Geometry Generator* for each XML file. After that, the OpenDRIVE file (**.xodr*) is generated from the resulting OpenDRIVE geometries and further information of the road description. Finally, the OpenDRIVE file is copied – together with five generic simulation files – to a timestamped subfolder of the simulator's project folder *./tools/OpenDS_4.9/assets/DrivingTasks/Projects/*. These simulation files are generated from templates and contain typical parameters and settings used to launch OpenDS with the generated terrain and OpenDRIVE files. The terrain file which is the output of the preceding terrain generation process (cf. 3.1.2) is expected to be available in a timestamped subfolder of *./tools/OpenDS_4.9/assets/Scenes/* where both timestamps must be equal. The provided shell scripts *(*.bat* and **.sh* files) ensure that the same timestamp is used throughout the whole generation process of a road network.

3.1.5 OpenDS

Finally, terrain, OpenDRIVE, and simulation files need to be processed by the driving simulator in order to create the final driving environment. When starting *OpenDS*, first, the terrain model is loaded, then, textured 3D meshes are created according to the OpenDRIVE representation of the road network and projected on top of the terrain resulting in an exact match due to the terrain deformation applied at the beginning of the toolchain. Furthermore, the simulation files are processed by *OpenDS*, which facilitates the simulation of

dynamic scene objects (e.g. traffic, obstacles, vehicle pathways, etc.) and allows setting up interaction between them.

The simulation parameters and settings (also known as "driving task") are distributed over several files and consist of the following three concepts: scene, scenario, and interaction which are usually available as XML-files (*scene.xml*, *scenario.xml*, and *interaction.xml*, respectively) and a file containing simulator settings (*settings.xml*). These concepts, their typical file names, and some details about their content are shown in Table 11.

Driving Task Layer	File Name	Description
Scene	scene.xml	(static) objects, geometries, reset points, sounds, images, lights, etc.
Scenario	scenario.xml	weather, driving car, dynamic vehicles, pedestrians, cyclists, etc.
Interaction	interaction.xml	trigger conditions, trigger actions (events)
Settings	settings.xml	general settings, controller/key assignment, codriver settings, etc.

Table 11: OpenDS – layers of a driving task

All four driving task files are generated from templates and can be edited manually in the respective subfolder of *./tools/OpenDS_4.9/assets/DrivingTasks/Projects/* after the generation process has finished and before the simulation has started. If some parameter needs to be changed permanently, it might be wise to edit the corresponding template before starting the generation process. Templates can be found in folder *./tools/OpenDS_4.9/assets/OpenDRIVE/templates/*. For instance, one might want to deactivate logging of the codriver: this could be accomplished for each individual driving task by setting the `<enableLog>` element to *false* in the respective *settings.xml* or for all future driving tasks by doing so in the template *emptySettingsFile.ftlx*.

OpenDS is available as executable **.jar* file for Windows, Mac OS, and Linux. It can be found in *./tools/OpenDS_4.9/OpenDS.jar* and can be executed from the command line as follows:

```
$ java -jar OpenDS.jar <inputFile>
```

Where `<inputFile>` is the path to the project file (optional). If this argument is not provided, the application will prompt a selection screen – given that no project file has been specified in the *startProperties.properties* file in the simulator's main folder. A project file consists of pointers to the OpenDRIVE file and the four driving task files described in Table 11. XML schemas for the validation of these five XML files can be found in *./tools/OpenDS_4.9/assets/DrivingTasks/Schema/*.

The following sample command can be used to run *OpenDS* with a pre-selected driving task file:

```
$ java -jar OpenDS.jar assets/DrivingTasks/Projects/track1/track1.xml
```

The road description specification allows placing any number of computer-controlled vehicles at arbitrary positions on the road. Every vehicle will continuously be driving at its individual maximum speed (without exceeding the general speed limit) following the lane it has initially been assigned to until a bifurcation is reached. In order to resolve ambiguity in pathways, a list of preferred turnings (when approaching to an intersection) can be specified for each vehicle – including the Codriver-controlled car. Figure 12 depicts the road example from the previous steps rendered by *OpenDS* including textured meshes and dynamic scene objects (one Codriver- and two computer-controlled vehicles) interacting with the road network. Each computer-controlled vehicle is set up to follow an individual target point (cf. Figure 12, green dots) which is

moving five meters ahead of the vehicle pivot along the lateral lane centre. Analogously, the trajectory of the Codriver-controlled vehicle (visualized by red and yellow dots in Figure 12) can be computed.

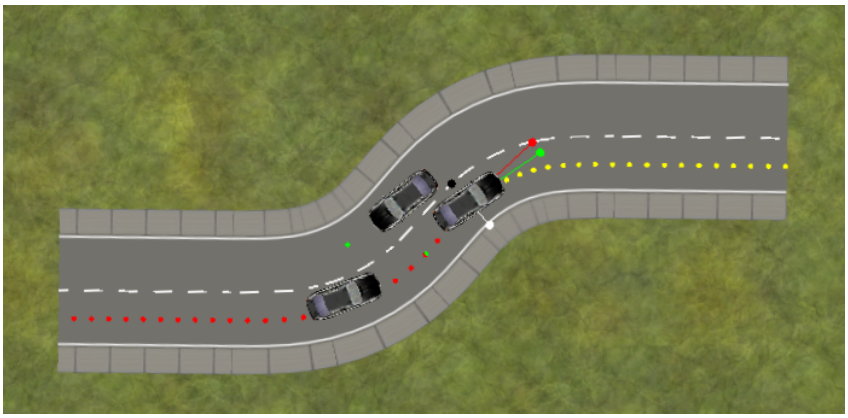


Figure 12: OpenDRIVE representation of a sample road segment (textured) with dynamic scene objects

Figure 13 depicts a more complex terrain and road network model created by the automatized road generation toolchain. The resulting 3D models (terrain, road segments, and junctions) have been generated without human interaction and solely by interpreting the road description specification. As the screenshots demonstrate, arbitrary slopes and junctions in undulating terrain can be generated.

Figure 13: OpenDS simulation of an automatically generated road network (the inset on the top left fram is a representation of the codriver motor space (or motor cortex).

Every simulation that has been triggered by the automatized road generation toolchain starts *OpenDS* with the codriver enabled. If the codriver needs to be disabled (e.g. in order to explore the environment by a human driver), the respective parameter (`<enableConnection>`) can be set to *false* in the corresponding *settings.xml*. For permanent deactivation the template (*emptySettingsFile.ftlx*) could be edited instead. While the codriver is activated, pressing *"M"* will show a graphical representation of the current motor cortex state (cf. Figure 13, upper left).

3.2 Examples of Driving Scenarios

The simulation files of the following sample scenarios (and some more) can be found in the project folder of the simulation environment (*./tools/OpenDS_4.9/assets/DrivingTasks/Projects/*). These can be opened with the provided simulation environment for instant simulation of the respective scenario. For most of the scenarios, a road description file (RDf) is available which has been used to generate driving scenarios from scratch. These files can be found in the input folder of the simulation environment (*./input/*) and be used to re-generate terrain, road network, and the aforementioned driving scenario files. This step is optional, as ready-to-use simulation files are provided for each of the examples. Each example contains a table referring to the respective generation (RDf), simulation, and log files.

The first two sample scenarios (3.2.1 and 3.2.2) demonstrate the features of the automatized terrain and road generation toolchain and the integration of the Chrono physics engine, respectively. The other scenarios implement the situations used for simulation fidelity tests described in Chapter 5 of D5.1 – Test Plans, Methods, and Metrics.

In the folder *./tools/OpenDS_4.9/* a file named *startProperties.properties* can be found. This file is processed every time the simulator is started. Editing this file (by uncommenting the respective line) allows to set up one of the scenarios presented in the following sections. The simulation of the selected scenario can be launched by executing *./tools/OpenDS_4.9/OpenDS.jar*.

3.2.1 Generated Terrain and Road Network Scenario

This scenario demonstrates the basic features of the automatized terrain and road generation toolchain. From a single road network description file, the user can generate a driving scenario including terrain, crossroads, traffic, and pedestrians. Traffic and pedestrians will follow a pre-defined pathway interacting with the Codriver-controlled car. The Bullet engine is used to render physics. Figure 14 (left) shows the layout of the road network, highlighting the positions of the traffic participants, which are shown in Figure 14 and are: Codriver-controlled vehicle (arrow), pedestrian (circle), and traffic vehicle (rectangle).

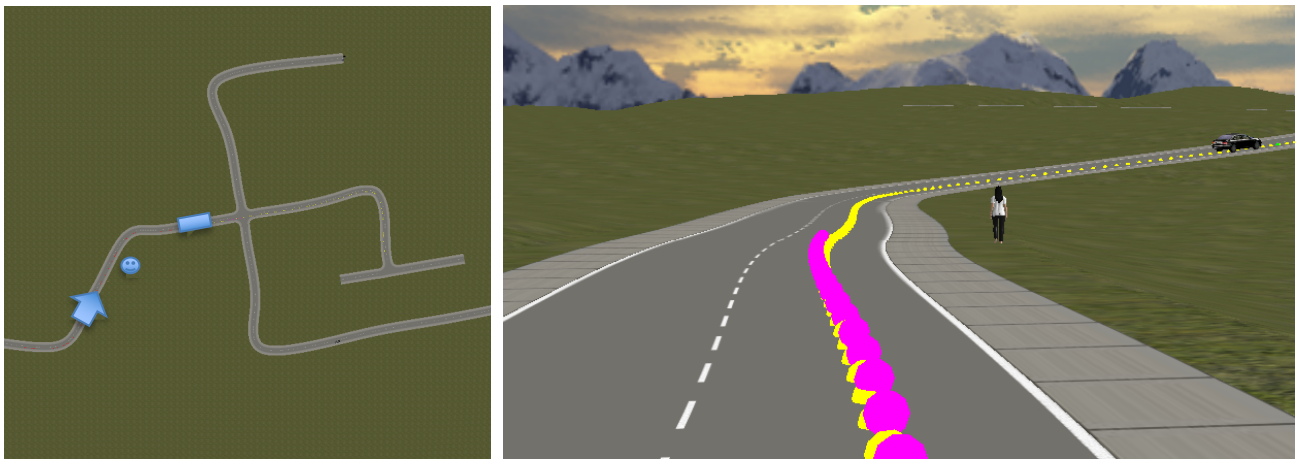


Figure 14: Generated scenario including terrain, crossroads, traffic, and pedestrians

It is possible to disable the codriver. For this, the codriver settings, which can be found in the corresponding *settings.xml* file, must be adjusted as shown in Listing 11. After disabling the codriver and restarting OpenDS, the vehicle can be controlled by keyboard, steering wheel, and pedals.

```
<codriver>
    <enableConnection>false</enableConnection>
</codriver>
```

Listing 11: Disabling the codriver

Table 12 provides the location of the corresponding road description file, the simulation file, and the log data which is produced when running the scenario.

Data	Location
Generation	./input/terrainTest.xml
Simulation	./tools/OpenDS_4.9/assets/DrivingTasks/Projects/TerrainTest/terrainTest.xml
Results	./tools/OpenDS_4.9/log/TerrainTest/

Table 12: Terrain example – Location of description files and resulting data

3.2.2 Chrono Scenario

The terrain and road network used in this scenario is an exact copy of the one in the previous scenario. That is why the generation script (cf. Table 13) is the same. The only difference is the use of the Chrono engine for physics rendering. The user can experience the differences between both engines by driving the steering car manually (instead of having it controlled by the codriver). Differences in the simulation files can be checked out by comparing them.

In general, every scenario can be run with the Chrono physics engine by specifying what part of the terrain to render by the Chrono engine and what Chrono-approved vehicle to use, no matter whether it will be controlled by the Codriver[7] or a human. While the Bullet physics engine can compute collisions of any two scene objects, the Chrono engine is limited to calculate the collision of two specific objects: terrain and vehicle. Thus, the user can select which scene object to use as “terrain” and which scene object to use as “vehicle”. Since the collision computation can be very expensive, a terrain as simple as possible should be used. The terrain must be available in Wavefront (*.obj) format and must be defined in the *scene.xml* (as usual). Furthermore, it must be referenced in the *settings.xml* by its unique ID, as shown in Listing 12. In this way the editor can make sure that only those surface triangles of a given track that are reachable by the car will be added to the Chrono engine.

[7] However, please note that the Codriver installation in OpenDS uses a generic (albeit adaptive) inverse model control that cannot not be perfectly matched to all possible Chrono-vehicle dynamics.

In the real vehicles and in the CarMaker environment used for other parts of Dreams4Cars the inverse models were tailored to the exact vehicle dynamics (e.g., D5.4). To carry out studies concerning control of vehicle with focus on the real dynamics and actuator lags, tailored inverse model would be required.

```
<chrono>
    <terrainModel ref="terrain_45" />
</chrono>
```

Listing 12: Adding terrain to Chrono physics simulation

The vehicle for Chrono can be specified in the *scene.xml* in the usual way. OpenDS can detect whether a Bullet or Chrono vehicle has been specified and load the respective physics engine. If a Bullet vehicle has been given, Chrono will not be started – even if a terrain has been specified as shown in Listing 12. Conversely, if a Chrono vehicle has been specified without selecting a terrain, the vehicle will be simulated in Chrono, however, with no terrain to drive on the vehicle will fall infinitely. Currently, only one vehicle (included in the vehicle selection of OpenDS) has been prepared for the use with Chrono:

Models/Cars/drivingCars/CitroenC4_Chrono/Car.scene

If this vehicle is used, more than 100 vehicle parameters can be adjusted, by modifying the *.json files in the subfolders of: *assets/Chrono/vehicle/*

Table 13 provides the location where to find the corresponding road description file and the simulation file. The generation file (RDf) is the same as in the previous example since the only difference is the use of a Chrono vehicle. After the generation of the terrain and road network, the editor must replace the Bullet vehicle (default) by a Chrono vehicle. This replacement has been done in the provided simulation project.

Data	Location
Generation	./input/terrainTest.xml
Simulation	./tools/OpenDS_4.9/assets/DrivingTasks/Projects/ChronoTest/chronoTest.xml
Results	(not run by codriver)

Table 13: Chrono example – Location of description files

3.2.3 Speed Adaptation

In this test, the Codriver-controlled car will be driven on a two-lane road. The main goal is to test the longitudinal inverse model and low-level control. This includes testing speed limit adaptation. The test is carried out both on a straight and a curvy road.

3.2.3.1 Straight Road

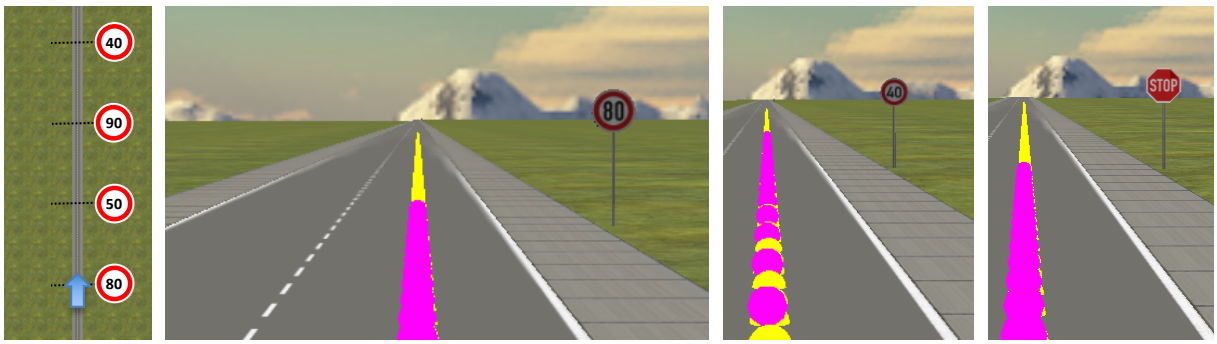


Figure 15: Speed adaptation on a straight road

The Codriver-controlled car is placed at the beginning of a straight two-lane road 4 km long. Every 500 to 1000 meters a new speed limit – implemented in the semantic annotation of the road (OpenDRIVE) – appears. The road signs depicted in Figure 15 are used as visual markers (for the human viewer) only. The Codriver receives the speed limit from the map. The speed limits are in the following order: 30, 80, 50, 100, 40, STOP.

Table 14 provides the location of the corresponding road description file, the simulation file, and the log data which is produced when running the scenario.

Data	Location
Generation	./input/speedAdaptationTest1.xml
Simulation	./tools/OpenDS_4.9/assets/DrivingTasks/Projects/SpeedAdaptationTest1/
Results	./tools/OpenDS_4.9/log/SpeedAdaptationTest1/

Table 14: Speed adaptation example 1 – Location of description files and resulting data

3.2.3.2 Curvy Road

The Codriver is placed at the beginning of a curvy two-lane road of 4 km length. Again, every 500 to 1000 meters a new (OpenDRIVE) speed limit appears. The corresponding scenario is shown in Figure 16. The speed limits are in the following order: 30, 80, 50, 100, 40, STOP.

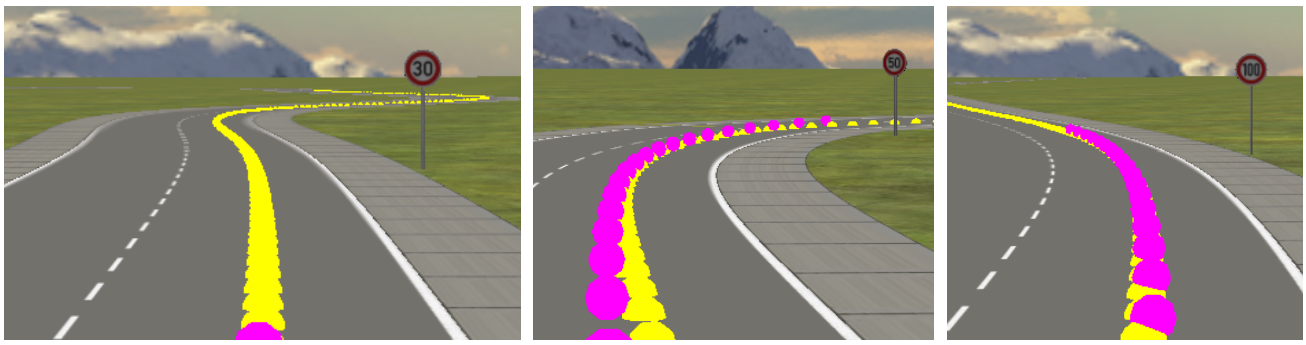


Figure 16: Speed adaptation on a curvy road

In curvy roads the speed choice of the Codriver depends on the speed limits, but, also on the curvature of the road according to [2], [3]. Thus, the Codriver might choose to drive at a lower velocity than the one given by the speed limit.

Table 15 provides the location of the corresponding road description file, the simulation file, and the log data which is produced when running the scenario.

Data	Location
Generation	./input/speedAdaptationTest2.xml
Simulation	./tools/OpenDS_4.9/assets/DrivingTasks/Projects/SpeedAdaptationTest2/
Results	./tools/OpenDS_4.9/log/SpeedAdaptationTest2/

Table 15: Speed adaptation example 2 – Location of description files and resulting data

3.2.4 Car Following

In this test, the Codriver-controlled car follows a leading vehicle on a one-lane road (in order to prevent the Codriver from overtaking). The main goal is to test the car following behaviour in response to different behaviours of the leading vehicle. A particular case is approaching a slower or stationary object. The leading car is set up to change velocity in an unpredictable way. Due to sudden large acceleration phases of the leading car, there are several approaches of the Codriver-controlled car to the leading car. The test is carried out using both a straight and a curvy road scenario.

3.2.4.1 Straight Road

The Codriver-controlled car is placed at the beginning of a straight one-lane road 4 km long with different speed limits. The Codriver car is instructed to ignore the speed limits and to drive at 100 km/h whenever possible. Since the leading vehicle obeys the speed limits exactly, the Codriver-controlled car must adjust its speed in order not to collide.

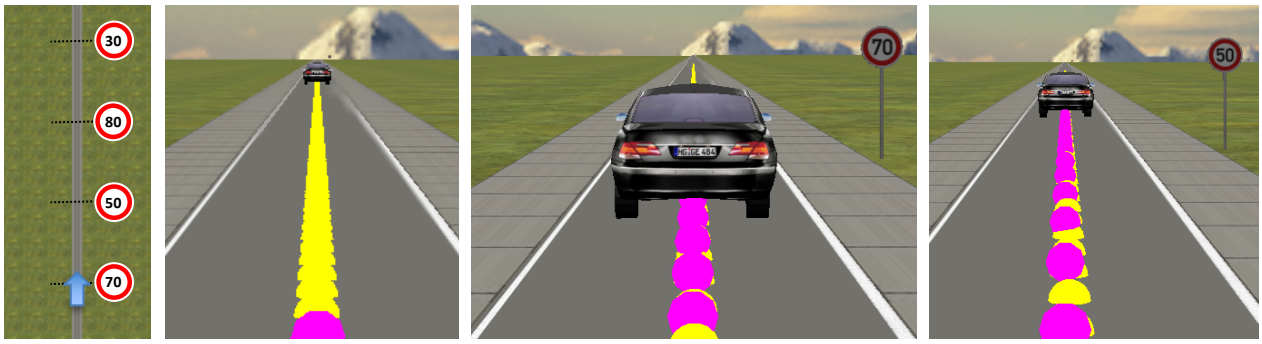


Figure 17: Car following on a straight road

Table 16 provides the location of the corresponding road description file, the simulation file, and the log data which is produced when running the scenario.

Data	Location
Generation	./input/carFollowingTest1.xml
Simulation	./tools/OpenDS_4.9/assets/DrivingTasks/Projects/CarFollowingTest1/carFollowingTest1.xml
Results	./tools/OpenDS_4.9/log/CarFollowingTest1/

Table 16: Car following example 1 – Location of description files and resulting data

3.2.4.2 Curvy Road

In this version, the road is curvy. Apart from this, the setup is exactly the same as described for the straight road (cf. 3.2.4.1). The scenario is shown in Figure 18. While the Codriver ignores the speed limits, it still complies with the speed choice in curves mentioned above,

Table 17 provides the location of the corresponding road description file, the simulation file, and the log data which is produced when running the scenario.

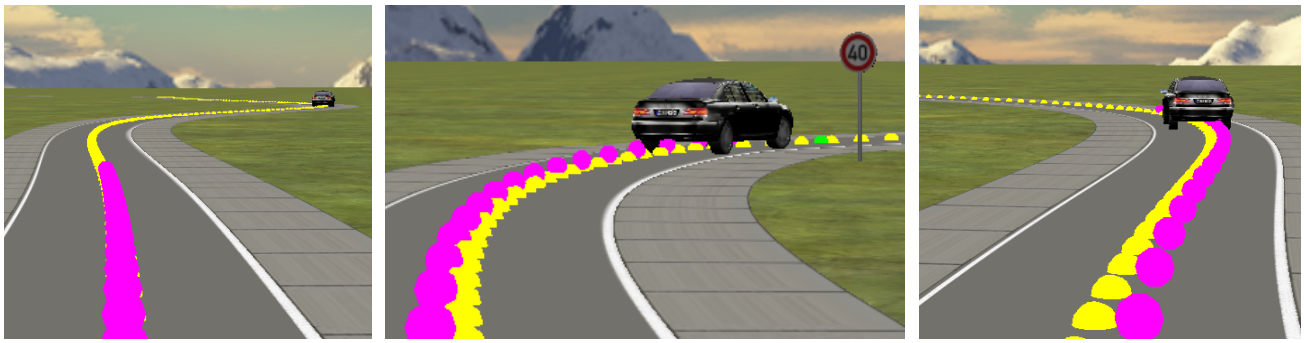


Figure 18: Car following on a curvy road

Data	Location
Generation	./input/carFollowingTest2.xml
Simulation	./tools/OpenDS_4.9/assets/DrivingTasks/Projects/CarFollowingTest2/carFollowingTest2.xml
Results	./tools/OpenDS_4.9/log/CarFollowingTest2/

Table 17: Car following example 2 – Location of description files and resulting data

3.2.5 Pedestrian Approaching

In this test the Codriver-controlled car drives on a straight road. The goal is to test the adaptation of speed in a pedestrian crossing situation. In further tests, the collision avoidance by changing lanes will additionally be examined. Once a pedestrian is approaching the road, the Codriver should open a temporal gap (and/or laterally deviate from the lane) to avoid the pedestrian.

3.2.5.1 Single-lane Road

The single-lane road scenario has been selected in order to test the adaptation of speed exclusively. The car is set up to follow the lane at 50 km/h. Along the road, three pedestrian-crossing events (one every 500 meters) can be found. For the human viewer, the crossing positions are visually marked by a pair of pedestrian-crossing signs. These signs as well as the intended crossing positions are not forwarded to the Codriver. Figure 19 (left) shows a sketch of the track, including the Codriver-controlled vehicle (arrow) and the three pedestrians (circles). The other images of Figure 19 show the three pedestrians.

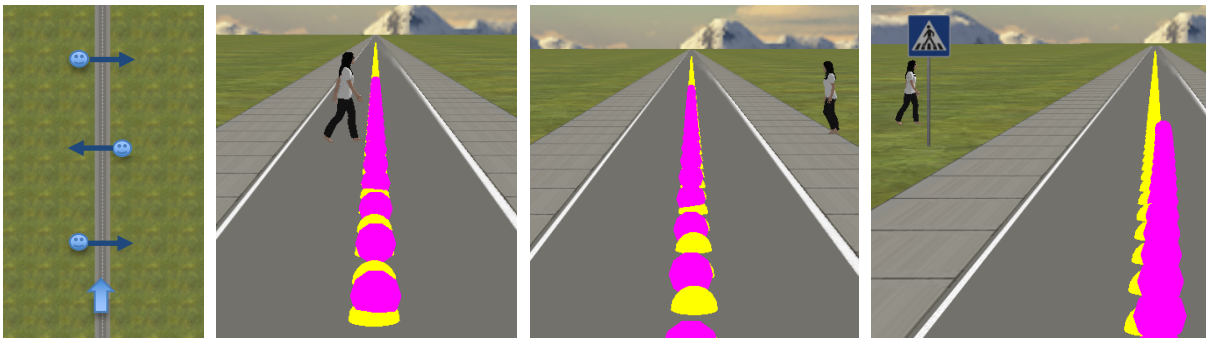


Figure 19: Pedestrian approaching on a one-lane road

The first pedestrian has been set up to cross the road from left to right at a constant speed of 4 km/h without stopping. She arrives at the road centre exactly at the same time as the Codriver-controlled car would arrive

(cf. Figure 19, image 2). The Codriver must slow down and eventually stop the car. When the pedestrian is out of the way, the Codriver resumes driving.

In the second situation, the pedestrian has been set up to cross the road from right to left at a constant speed of 4 km/h. The timing is exactly as before; however, the pedestrian stops walking suddenly when arriving at the sidewalk (cf. Figure 19, image 3). At that precise moment, the Codriver was already going stops the vehicle, but after a while, the Codriver resumes driving and passes the waiting pedestrian with caution.

The third pedestrian has been set up to cross the road from left to right at a constant speed of 4 km/h without stopping. Since the pedestrian arrives too late at the intersection point (cf. Figure 19, image 4), the Codriver-controlled car goes on driving without reducing speed.

Table 18 provides the location of the corresponding road description file, the simulation file, and the log data which is produced when running the scenario.

Data	Location
Generation	./input/pedestrianApproachingTest1.xml
Simulation	./tools/OpenDS_4.9/assets/DrivingTasks/Projects/PedestrianApproachingTest1/ pedestrianApproachingTest1.xml
Results	./tools/OpenDS_4.9/log/PedestrianApproachingTest1/

Table 18: Pedestrian approaching example 1 – Location of description files and resulting data

3.2.5.2 Two-lane Road

The Codriver-controlled car is placed at the beginning of a straight two-lane road of 2 km length. The additional lane to the left provides space for lateral evasive manoeuvres. The car is set up to follow the right lane at 50 km/h. Along the road, four pedestrian-crossing events (one every 500 meters) can be found. Figure 20 (left) shows a sketch of the track including the Codriver-controlled vehicle (arrow) and the four pedestrians (circles). The other images of Figure 20 show the car view of the four pedestrians.

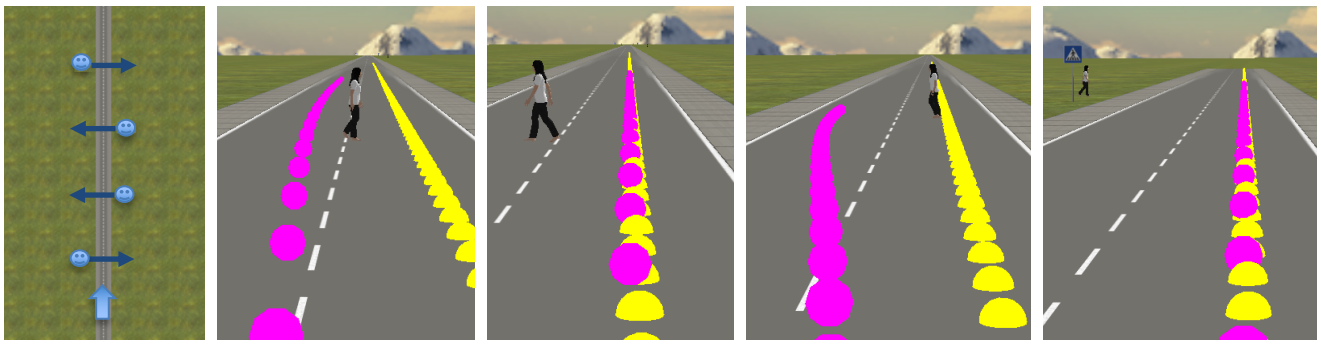


Figure 20: Pedestrian approaching on a two-lane road

The first pedestrian has been set up to cross the road from left to right at a constant speed of 4 km/h without stopping. She arrives at the centre of the right lane exactly at the same time the Codriver-controlled car would arrive (cf. Figure 20, image 2). The situation is an copy of the first pedestrian situation described in 3.2.5.1 except for the additional lane to the left. The reaction of the Codriver in this situation is reducing speed and using the left lane for collision avoidance – as one can see in the planned trajectory visualized by pink dots in Figure 20 (image 2). After passing the pedestrian, the Codriver changes back to the right lane and continues

driving. This situation, but with standing pedestrian on the lane side, has been reproduced with the real vehicle MIA car (see D5.4).

In the second situation, the pedestrian has been set up to cross the road from right to left at a constant speed of 4 km/h without stopping. Since the pedestrian is already leaving the lane centre when the Codriver-controlled car approaches to the intersection point, no further action of the Codriver is required (cf. Figure 20, image 3).

The setup of situation 3 is an copy of situation 2. However, the pedestrian suddenly stops walking in the centre of the right lane (cf. Figure 20, image 4). The Codriver, which intended to pass behind the pedestrian (as in situation 2), must adapt to the change and a lane change to the left is initiated – as indicated by the planned trajectory. After passing the pedestrian (with reduced speed), the Codriver changes back to the right lane and continues driving.

The fourth pedestrian has been set up to cross the road from left to right at a constant speed of 4 km/h without stopping. Since the pedestrian arrives too late at the intersection point (cf. Figure 20, image 5), the Codriver-controlled car goes on driving without reducing speed.

Table 19 provides the location of the corresponding road description file, the simulation file, and the log data which is produced when running the scenario.

Data	Location
Generation	./input/pedestrianApproachingTest2.xml
Simulation	./tools/OpenDS_4.9/assets/DrivingTasks/Projects/PedestrianApproachingTest2/ pedestrianApproachingTest2.xml
Results	./tools/OpenDS_4.9/log/PedestrianApproachingTest2/

Table 19: Pedestrian approaching example 2 – Location of description files and resulting data

3.2.5.3 Two-lane Road with Oncoming Traffic

The Codriver-controlled car is placed at the beginning of a straight two-lane road of 2 km length. The car is set up to follow the right lane at 50 km/h. After 500 meters of driving a pedestrian crossing situation including oncoming traffic is found.

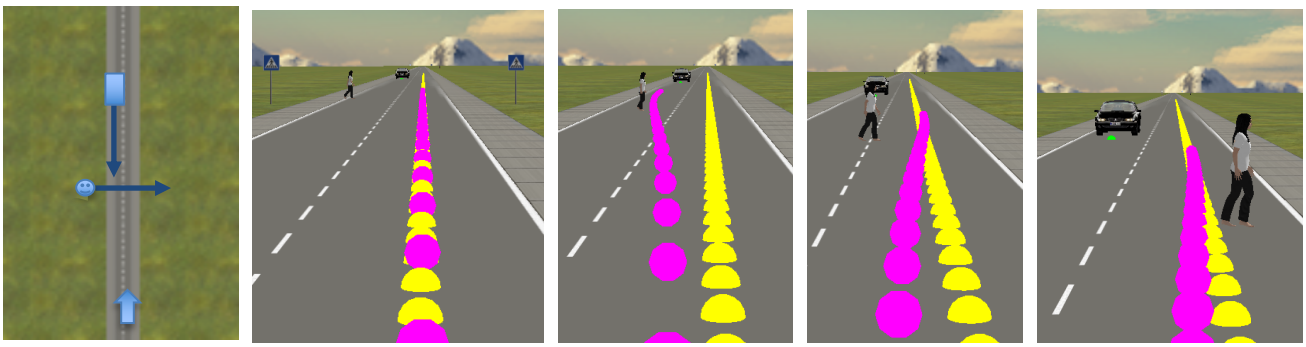


Figure 21: Pedestrian approaching on a two-lane road with oncoming traffic

The pedestrian has been set up to cross the road from left to right at a constant speed of 4 km/h without stopping. The pedestrian arrives at the centre of the right lane exactly at the same time as the Codriver-controlled car would arrive (cf. Figure 21, image 2). Since the situation is similar to the first pedestrian

situation described in 3.2.5.2, the Codriver plans to pass behind the pedestrian using the left lane (cf. planned trajectory in Figure 21, image 3). Exactly in this moment, the oncoming car (which is driving at 50 km/h) enters the detection range of the simulate sensors of the Codriver. Hence, changing the lane will avoid a collision with the pedestrian, but not with the car. Thus, the Codriver decides to stay in the lane and stops the vehicle (cf. planned trajectory in Figure 21, image 4). As soon as the pedestrian is out of the way, the Codriver resumes driving (cf. Figure 21, image 5).

Table 20 provides the location of the corresponding road description file, the simulation file, and the log data which is produced when running the scenario.

Data	Location
Generation	./input/pedestrianApproachingTest3.xml
Simulation	./tools/OpenDS_4.9/assets/DrivingTasks/Projects/PedestrianApproachingTest3/ pedestrianApproachingTest3.xml
Results	./tools/OpenDS_4.9/log/PedestrianApproachingTest3/

Table 20: Pedestrian approaching example 3 – Location of description files and resulting data

3.2.6 Lane Following

In this test the Codriver-controlled car drives on a curvy road. The main goal is to test the inverse model and low-level control for lateral control.

3.2.6.1 Road with Moderate Curves

The Codriver-controlled car is placed in the right lane at the beginning of a curvy 3-km-long two-lane road. The lane width is 3.0 meters and the curve radii are kept within reasonable limits like they can be found in real-world environments. There is no speed limit; however the Codriver will have to adjust the speed to the upcoming curvature [2], [3].

Figure 22 provides a sketch of the track (left image) and a selection of screenshots of some curves used in the test.

Figure 22: Lane following on a road with moderate curves

Table 21 gioves the location of the corresponding road description file, the simulation file, and the log data which is produced when running the scenario.

Data	Location
Generation	./input/laneFollowingTest1.xml
Simulation	./tools/OpenDS_4.9/assets/DrivingTasks/Projects/LaneFollowingTest1/ laneFollowingTest1.xml
Results	./tools/OpenDS_4.9/log/LaneFollowingTest1/

Table 21: Lane following example 1 – Location of description files and resulting data

3.2.6.2 Road with narrower Curves

The Codriver-controlled car is placed on the right lane at the beginning of a road with narrower curves. The track is 5 km long and consists of two lanes with width of 2.60 meters. There is no speed limit, however, the Codriver will have to adjust the speed to the upcoming curvature in order to stay in the lane.

Figure 23 provides a sketch of the track (left image) and a selection of screenshots of some curves used in the test (from the driver's perspective and from a bird's view).

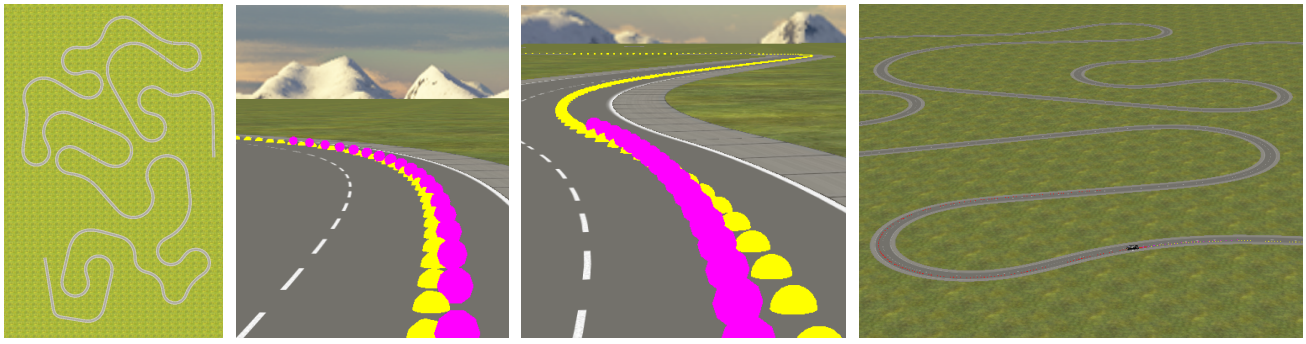


Figure 23: Lane following on a road with intense curves

Table 22 provides the location of the corresponding road description file, the simulation file, and the log data which is produced when running the scenario.

Data	Location
Generation	./input/laneFollowingTest2.xml
Simulation	./tools/OpenDS_4.9/assets/DrivingTasks/Projects/LaneFollowingTest2/ laneFollowingTest2.xml
Results	./tools/OpenDS_4.9/log/LaneFollowingTest2/

Table 22: Lane following example 2 – Location of description files and resulting data

3.2.7 Overtaking a Slow Vehicle

In this test, the Codriver-controlled car drives on a two-lane road. The goal is to demonstrate the lane change and overtake abilities at different speeds and different oncoming traffic situations. The test is carried out using the straight and the curvy road scenarios from 3.2.3.

3.2.7.1 Straight Road

The Codriver is set up to obey the speed limits while coping with four situations of slow vehicles ahead. Placing those vehicles at different positions allows to examine the Codriver behaviour at different speeds. Figure 24 provides a sketch of the track (left) and screenshots of possible overtaking situations: overtaking a slow vehicle, overtaking a slow vehicle while coping with oncoming traffic, and overtaking more than one slow vehicle at once. It should be reminded that these behaviours (as well as all the other examples) are not programmed but they are emergent behaviours produced by the agent sensorimotor architecture (D2.2, D2.3, D7.2 and D7.3):

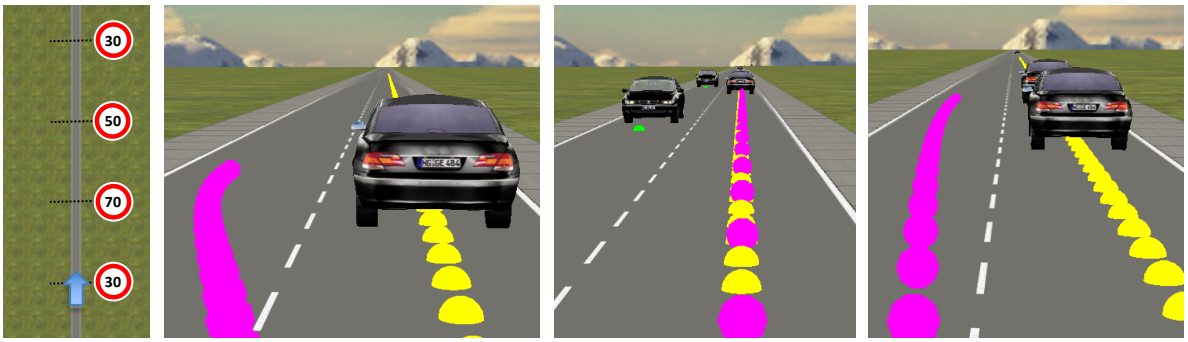


Figure 24: Overtaking on a straight road

The first encounter with a slow vehicle can be found in a speed limit zone of 70 km/h. Since the leading vehicle is driving at 20 km/h and there is no traffic in the opposite direction, the Codriver changes the lane to overtake (cf. Figure 24, image 2). After passing the vehicle, the Codriver goes back into the right lane and continues driving at 70 km/h.

The second encounter is located in the same speed limit zone (70 km/h) with a vehicle ahead driving at 25 km/h. Since the left lane is blocked by two oncoming vehicles (cf. Figure 24, image 3), the Codriver needs to decelerate in order to not collide with the vehicle ahead. Once the left lane is clear, the Codriver accelerates again and changes into the left lane to initiate the overtaking manoeuvre as shown in Figure 25.

Figure 25: Overtaking on a straight road with oncoming traffic

The third situation occurs at a speed of 50 km/h. The Codriver is supposed to overtake two vehicles at the same time (cf. Figure 24, image 4), as the headway between both vehicles is not sufficient to go back into the lane after overtaking the first vehicle. The vehicles to overtake travel at a speed of 30 km/h. The detailed overtaking manoeuvre is shown in Figure 26.

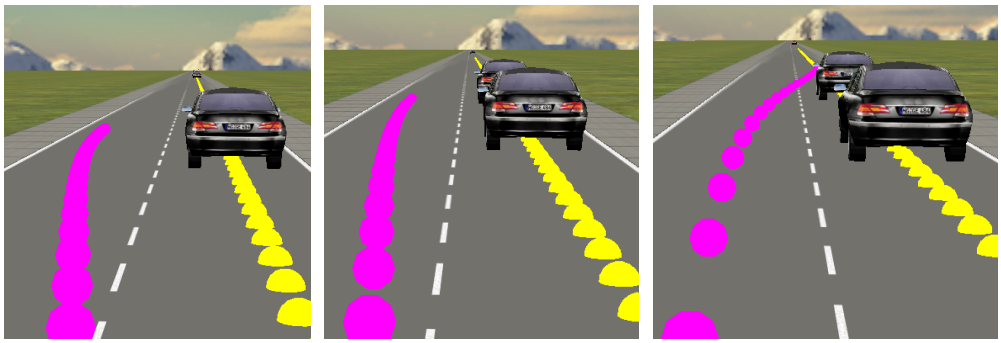


Figure 26: Overtaking more than one vehicle at once on a straight road

The fourth situation takes place in a speed limit zone of 30 km/h and the vehicle to overtake drives at 20 km/h. As there is no oncoming traffic, the Codriver changes lane in order to initiate the overtaking manoeuvre even with a minimum speed difference (cf. Figure 24, image 2).

Table 23 provides the location of the corresponding road description file, the simulation file, and the log data which is produced when running the scenario.

Data	Location
Generation	./input/overtakingTest1.xml
Simulation	./tools/OpenDS_4.9/assets/DrivingTasks/Projects/OvertakingTest1/overtakingTest1.xml
Results	./tools/OpenDS_4.9/log/OvertakingTest1/

Table 23: Overtaking example 1 – Location of description files and resulting data

3.2.7.2 Curvy Road

The Codriver has to cope with the same overtaking situations as in the previous version (cf. 3.2.7.1) but on a curvy road. Modified speed limits are passed in the following order: 70, 50, 80, 30, STOP. Screenshots are shown in Figure 27.

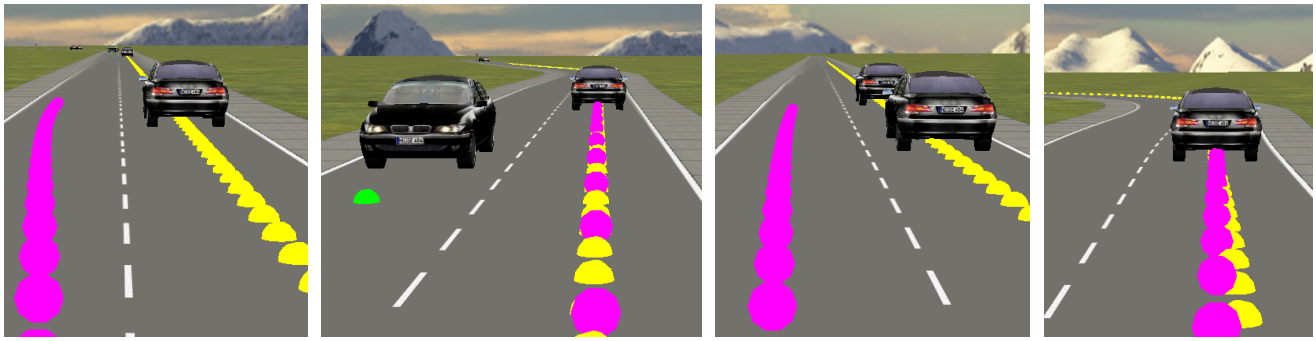


Figure 27: Overtaking on a curvy road

The first and second encounter with slow vehicles take place in a speed limit zone of 70 km/h under the same conditions, and results as described in 3.2.7.1. Notably, the Codriver selects rather straight segments to overtake vehicles (a higher-level biasing loop prevents lane change in curves, see D2.2, D2.3m, D/.2 and D7.3) . The third situation (overtaking two vehicles at the same time) occurs – this time – at a speed of 80 km/h and

the final situation again at 30 km/h. The conditions of situation 3 and 4 are equal to the ones described in 3.2.7.1. While the result of situation 3 is the same as in 3.2.7.1, the Codriver does not overtake the vehicle of situation 4 because approaching the curve (this is a notable example of biasing behaviours).

Table 24 provides the location of the corresponding road description file, the simulation file, and the log data which is produced when running the scenario.

Data	Location
Generation	./input/overtakingTest2.xml
Simulation	./tools/OpenDS_4.9/assets/DrivingTasks/Projects/OvertakingTest2/overtakingTest2.xml
Results	./tools/OpenDS_4.9/log/OvertakingTest2/

Table 24: Overtaking example 2 – Location of description files and resulting data

3.2.8 Lane Change with Stationary Vehicles

The goal of this test is to force a lane change in a safe and reproducible scenario. This can be achieved by e.g. putting two (or more) stationary vehicles in two lanes so that the Codriver is forced to perform a second lane change after the first stationary vehicle has been passed.

In this test, the Codriver-controlled car is placed on the right lane at the beginning of a straight 4-km-long road. Every 500 to 1000 meters the Codriver faces stationary vehicles, forcing lane change. In order to span different situations, the speed of the Codriver-controlled vehicle and the lane configuration differ in each situation. The Codriver is set up to obey the speed limits and to drive in the rightmost lane whenever possible. Speed limits are in the following order: 40, 70, 50, 80, 30, STOP.

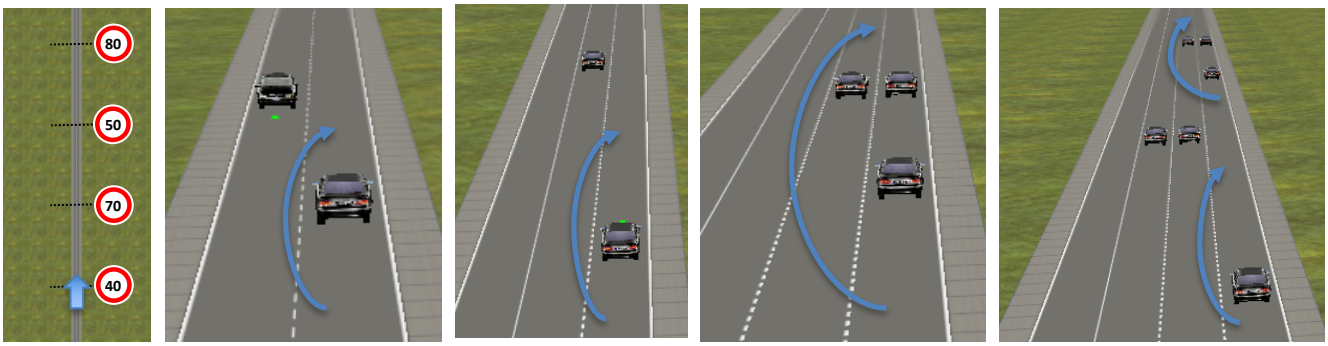


Figure 28: Lane change with stationary vehicles

Figure 28 gives a sketch of the track's speed limits and screenshots of four different situations including expeted paths to pass the stationary vehicles: 1) using the opposite direction lane, 2) using same direction lane, 3) performing a double lane change, and 4) performing several consecutive lane changes.

The first encounter with two stationary vehicles (Figure 29) occurs at the speed of 40 km/h. The first obstacle is located in the right lane forcing the Codriver to change to the left lane, which is heading in the opposite direction. At a headway of 30 meters, the second stationary obstacle appears in the left lane making the Codriver to change back into the right lane.

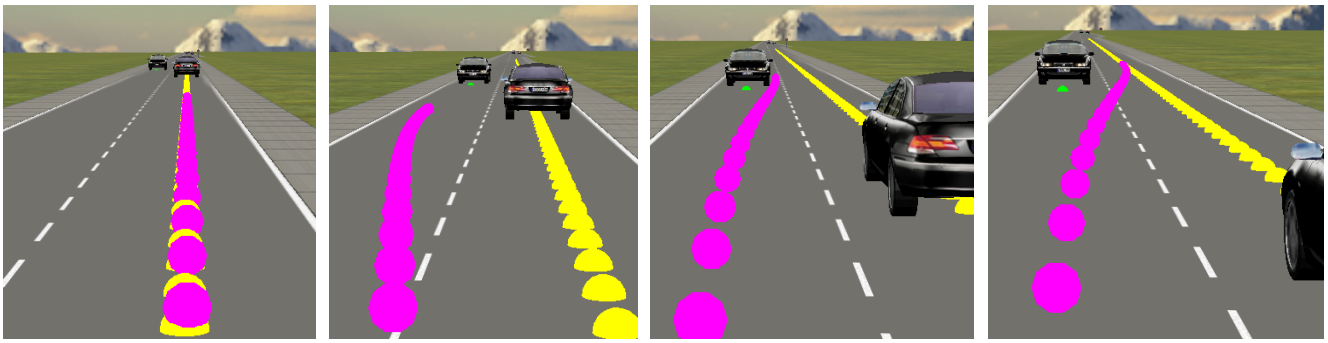


Figure 29: Bypassing two stationary vehicles using the opposite direction lane

For the second situation, the limit becomes 70 km/h and an additional lane appears to the right (same direction), resulting in a total of two lanes in driving direction and one in opposite direction. The Codriver-controlled car changes to the rightmost lane and approaches to the second situation (Figure 30).

In the second situation, the first obstacle is located in the right lane forcing the Codriver to change to the left lane, which is heading in the same direction. At a headway of 60 meters (higher distance due to higher speed), the second obstacle appears in the left lane. The Codriver must change back into the right lane in order not to collide or infringe the highway code by crossing a solid line.

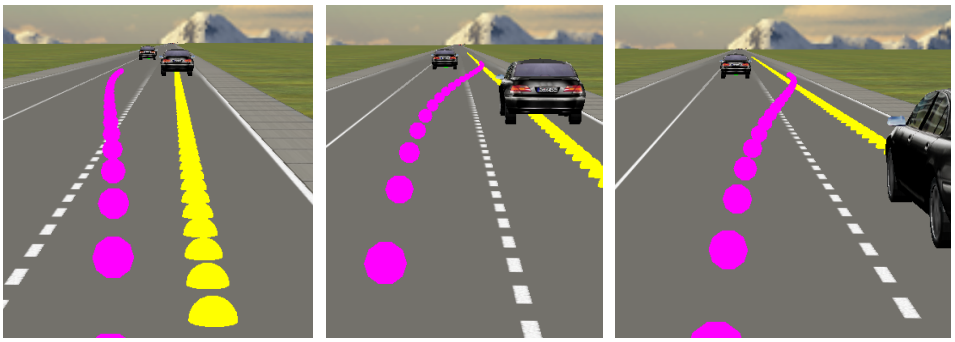


Figure 30: Bypassing two stationary vehicles using the left lane (same direction)

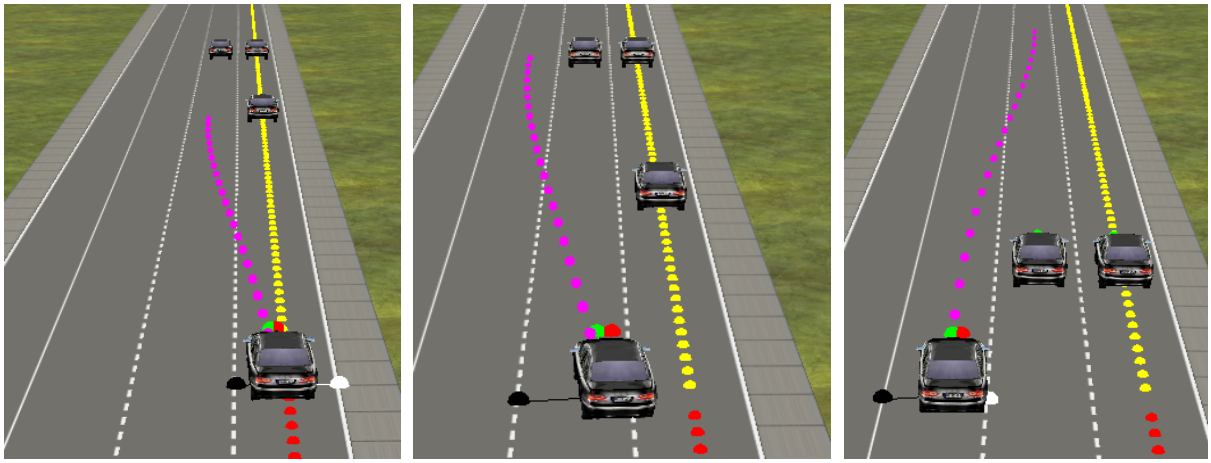


Figure 31: Bypassing three stationary vehicles by two consecutive lane changes (same direction)

In the third situation (Figure 31), the Codriver enters a speed limit zone of 50 km/h and another lane appears to the right (same direction), resulting in a total of three lanes in the driving direction and one in the opposite direction. The Codriver-controlled car changes to the rightmost lane and approaches to the third stationary

vehicles situation. Then the Codriver finds the right and centre lanes blocked, forcing it to perform two consecutive lane changes to the left (Figure 31).

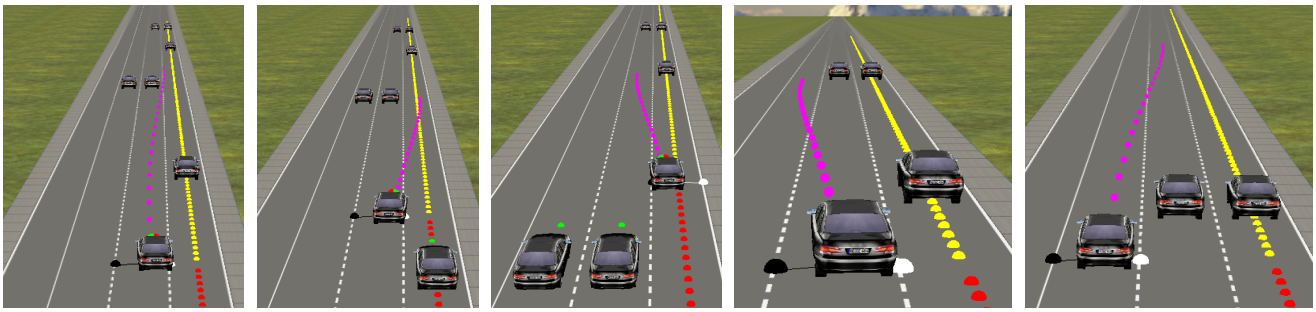


Figure 32: Bypassing six stationary vehicles by several consecutive lane changes (same direction)

The final situation occurs in a speed limit zone of 80 km/h and a lane configuration of one opposite and three same direction lanes. The sequence of obstacles is right→left→right, where the last obstacle blocks the right and centre lane (cf. Figure 32, image 1). While approaching, the Codriver changes to the centre lane in order to bypass the first obstacle in the right lane. After 50 meters, the next obstacle appears in the centre and left lane, forcing the Codriver-controlled vehicle to change back into to right lane (cf. Figure 32, image 2). Another 50 meters further ahead the road, and the final obstacle appears in the right and centre lane (cf. Figure 32, image 3 and 4) forcing the Codriver to perform two consecutive lane changes to the leftmost lane. After passing all obstacles, the Codriver returns back to the rightmost lane (cf. Figure 32, image 5).

Table 25 provides the location of the corresponding road description file, the simulation file, and the log data which is produced when running the scenario.

Data	Location
Generation	./input/obstacleTest.xml
Simulation	./tools/OpenDS_4.9/assets/DrivingTasks/Projects/ObstacleTest/obstacleTest.xml
Results	./tools/OpenDS_4.9/log/ObstacleTest/

Table 25: Obstacle example – Location of description files and resulting data

3.3 Open Software and Documentation

The latest version of the simulation environment (version 1.5, as of 1st October 2019) including the following main features is available in ZENODO[8] free for research, studies and benchmarks. It includes:

- the automatized terrain and road generation toolchain,
- the OpenDS driving simulation (version 4.9), based on the jMonkey Engine (version 3.1.0),
- the Codriver agent (version 9.6.1204), and
- the integration of the Chrono physics engine (version 4.0.0) into OpenDS.

The above components will be maintained and updated with new versions of the modules, when they will be ready and tested. Also, bug reports and new features may be requested.

Figure 33 shows the components of the simulation environment and the flow of data.

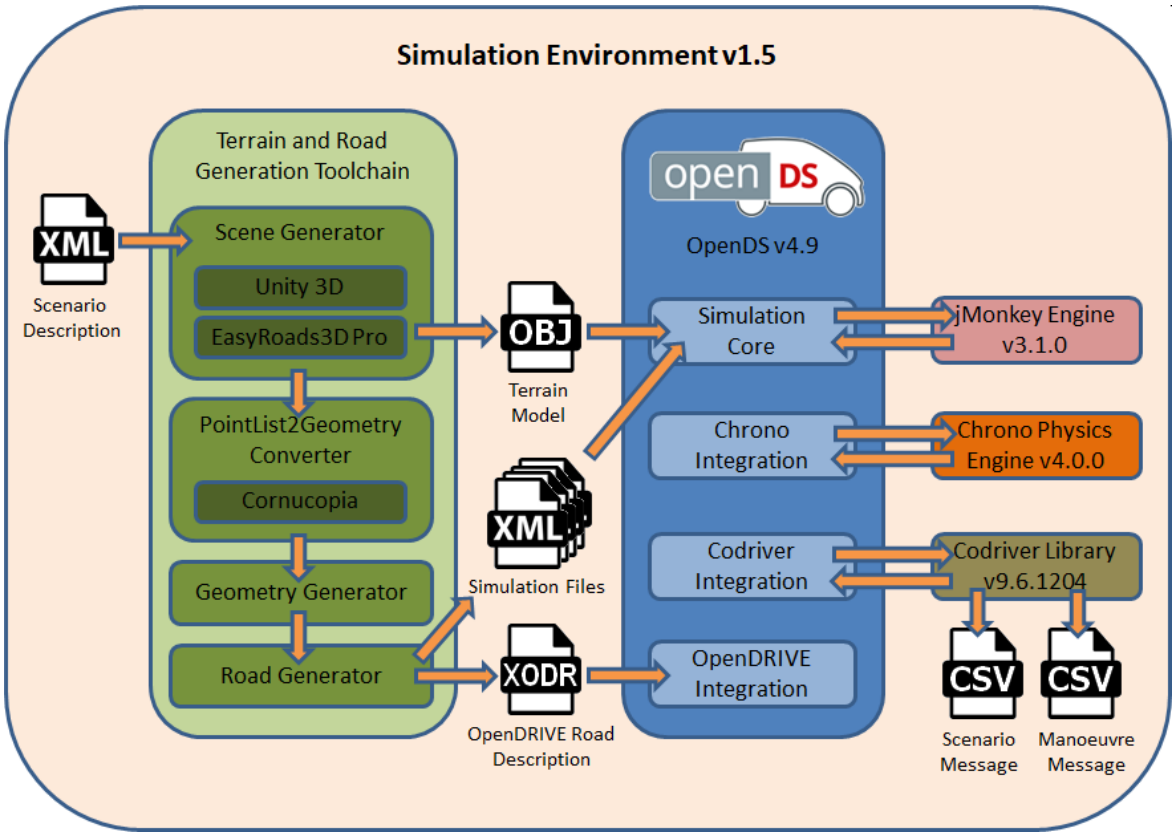


Figure 33: Building blocks of the simulation environment (final version)

In addition to the software components mentioned above, ready-to-use driving scenarios with videos (cf. Section 3.2) have been uploaded to ZENODO demonstrating the capabilities of the final Codriver implementation. For each driving scenario, the following data are available (cf. Figure 33, text files):

- one scenario description file,
- several simulation files (including a terrain model and an OpenDRIVE road description),
- two log files (representing the bidirectional communication between Codriver and OpenDS).
- ready to view videos (for some scenario)

[8] https://zenodo.org/communities/dreams4cars/ (DOI: 10.5281/zenodo.3582054).

Processing the scenario description of one of the examples by the terrain and road generation toolchain will re-generate the respective simulation files (including terrain and OpenDRIVE file), which have been included in the upload for the reader's convenience. Processing the simulation files of one of the examples by OpenDS will re-generate the respective logfiles, which, again, have been included in the upload for the reader's convenience.

Ready to view videos are also available for quick evaluation.

Table 26 gives an overview of all provided software components including the license and operating system compatibility (W = Windows, L = Linux, M = MacOS). Furthermore, the availability of the source code and the programming language/framework used to implement the software component is shown in the table.

Software	Component	Implementation	License	Source Code	Binaries		
					W	L	M
OpenDS	Simulation Core	Java	GNU GPL	✓	✓	✓	✓
	OpenDRIVE Integration	Java	GNU GPL	✓	✓	✓	✓
	Chrono Integration	Java	GNU GPL	✓	✓	✓	✗
	Codriver Integration	Java	GNU GPL	✓	✓	✓	✓
jMonkey Engine	Renderer, Bullet Physics	Java	BSD-3	✓	✓	✓	✓
Chrono Engine	OpenDS adaptation	C++	BSD-3	✓	✓	✓	✗
Automatized Terrain and Road Generation Toolchain	Scene Generator	Unity (C#)	GNU GPL	✓	✓	✓	✗
	PointList2Geometry Converter	C++	GNU GPL	✓	✓	✓	✗
	Geometry Generator	Java	GNU GPL	✓	✓	✓	✓
	Road Generator	Java	GNU GPL	✓	✓	✓	✓
Codriver	Server / Library	C++	[9]	✗	✓	✓	✓

Table 26: Overview of provided software components including license, source code availability and OS compatibility

As far as no restrictions of the underlying libraries apply, most software components have been published under the GNU GPL v3 (GNU General Public License Version 3) open-source license, which allows the user to copy, distribute and modify the software as long as changes/dates are tracked in source files. Any modifications to it or software including GPL-licensed code must also be made available under the GPL along with build & install instructions.

[9] The license for the Codriver library is given as a separate license file in the repository.

Except for the Codriver and some external resources (e.g. *EasyRoads 3D Pro*) used by the terrain and road generation toolchain, the respective source code is enclosed in the software submission. In order to build the terrain and road generation toolchain from sources, one needs to purchase a license for *Unity* and *EasyRoads 3D Pro*, first. The additional source code is open and contained in the upload. Documentation and further information about how to build the source code and run the software can be found next to the respective component.

4 Bibliographical References

[1] I. Baran, J. Lehtinen, J. Popović, «Sketching Clothoid Splines Using Shortest Paths», Computer Graphics Forum, 29: 655-664 (http://people.csail.mit.edu/ibaran/papers/2010-EG-Curves.pdf**).**

[2] P. Bosetti, M. Da Lio, and A. Saroldi, "On the Human Control of Vehicles: an Experimental Study of Acceleration," Eur. Transp. Res. Rev., 2013, doi: 10.1007/s12544-013-0120-2.

[3] P. Bosetti, M. Da Lio, and A. Saroldi, "On Curve Negotiation: From Driver Support to Automation," IEEE Trans. Intell. Transp. Syst., vol. 16, no. 4, pp. 2082–2093, 2015, doi: 10.1109/TITS.2015.2395819.