



DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Randomized Generation of Road Networks and 3D Visualization of Traffic Scenarios in Gazebo

Hans Kirchner





DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Randomized Generation of Road Networks and 3D Visualization of Traffic Scenarios in Gazebo

Zufallsgenerierung von Straßennetzen und 3D-Visualisierung von Verkehrsszenarien in Gazebo

Author:	Hans Kirchner
Supervisor:	Prof. Matthias Althoff
Advisor:	Markus Koschi, Stefanie Manzinger
Submission Date:	April 18th 2017



I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, April 18th 2017

Hans Kirchner

Acknowledgments

I want to thank Steffi and Markus for advising and guiding me in the right direction. Also, I want to thank Kathrin and Philipp for proof-reading and giving me helpful tips.

Abstract

Testing software for autonomous vehicles is difficult because input data from sensors and output to actuators is needed to do full integration tests. Simulating the vehicle in a virtual environment solves this problem by feeding the software with generated sensor data and applying the output on the vehicle model. To do this, a virtual environment has to be created first. A declarative approach to automate scenario generation, called Template XML, is proposed and developed specifically for the Carolo-Cup, a student's competition for autonomous driving. Also, a 3D visualization for the scenario was implemented for Gazebo, that is able to render typical real-world and Carolo-Cup specific traffic scenarios. Using this approach, it is possible to generate and render several similar but different scenarios from a single template description within seconds.

Contents

Acknowledgments	iii
Abstract	iv
1. Introduction	1
2. Related Work	4
3. Randomized Road Generation	5
3.1. Primitives	5
3.1.1. Straight Road	6
3.1.2. Arc	7
3.1.3. Quadratic Bézier curve	8
3.1.4. Cubic Bézier curve	11
3.1.5. Static Obstacle	12
3.1.6. Intersection	13
3.1.7. Zebra crossing	15
3.1.8. Blocked area	15
3.1.9. Traffic sign	16
3.1.10. Parking Lot	16
3.1.11. Parking obstacle	17
3.2. Control structures	18
3.2.1. Sequence	18
3.2.2. Optional	18
3.2.3. Select+Case	18
3.2.4. Repeat	19
3.2.5. Shuffle	19
3.3. Evaluation of Template XML	19
3.4. Concatenation of Primitives	20
4. CommonRoad XML Format	23
4.1. CommonRoad Basics	23
4.1.1. Coordinate System and Units	23

Contents

4.1.2. Lanelets	24
4.1.3. Obstacles	24
4.1.4. Ego Vehicle	25
4.2. CommonRoad Extensions	25
4.2.1. Intersections	25
4.2.2. Zebra crossings	28
4.2.3. Blocked Areas	28
4.2.4. Parking Lots	28
4.2.5. Traffic Signs	28
4.2.6. Stop line	28
5. Visualization	31
5.1. Groundplane	31
5.2. Line Markings	36
5.3. Traffic Signs	39
5.4. Obstacle	39
5.5. Ego Vehicle	40
6. Results	43
6.1. Driving without obstacles	43
6.2. Driving in extended mode	43
6.3. Parallel Parking	49
7. Future work	51
7.1. Road Generation	51
7.2. Visualization	51
7.3. Simulation	52
A. XML Schema	53
A.1. Template XML Schema	53
A.2. CommonRoad XML Schema	59
List of Figures	68
List of Tables	69
Bibliography	70

1. Introduction

The Carolo-Cup¹ is a student's competition with the goal of developing a cost-efficient and energy-efficient 1:10 autonomously-driving model car. The vehicles should drive as fast as possible and error-free in three different scenarios. In a so-called static discipline, the teams must explain their concepts and results to a jury in a presentation. The cup takes place annually in Braunschweig.

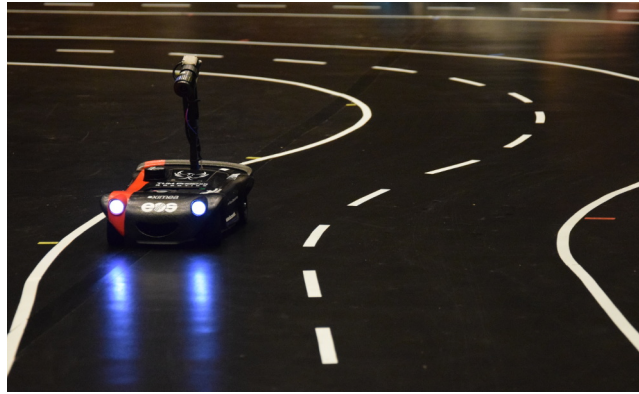


Figure 1.1.: Vehicle of TUM Phoenix Robotics at Carolo-Cup 2017

Source: Hans Kirchner, 2017-02-06

The three challenges are driving without obstacles, driving with obstacles and parallel parking. In the first challenge, the vehicle must follow a two-lane road in a round trip without leaving its lane for two minutes. The vehicle may encounter intersections, s-bend, long straights and sharp turns. In the challenge of driving with obstacles, static and dynamic obstacles are placed on the road. Static obstacles do not move and must be bypassed without hitting, while dynamic obstacles may move along a lane where they must be overtaken, or can appear at intersections with right of way. If the vehicle reaches an intersection, it must hold at the stop line and wait for the dynamic obstacle, if there is one, to cross the intersection. In parking challenge, the vehicle must find a

¹<https://wiki.ifr.ing.tu-bs.de/carolocup/>

suitable parking lot on the right side, do a parking maneuver and stop. There are three different sizes of parking lots. The smallest parking lot has no penalty whereas the other add a few seconds on the measured time.

The team's task is to design mechanical, electronical and software aspects to build up a real model car that is able to overcome all three explained challenges. This requires knowledge in CAD, steering models, controlling, sensors, image recognition, sensor fusion and hardware programming, to name a few.

The vehicles drive on a black, flat surface with white line markings. Each road lane is 400 mm in width. The vehicle can be up to 300 mm high and 300 mm wide [Bra16]. Obstacles are placed as white boxes.

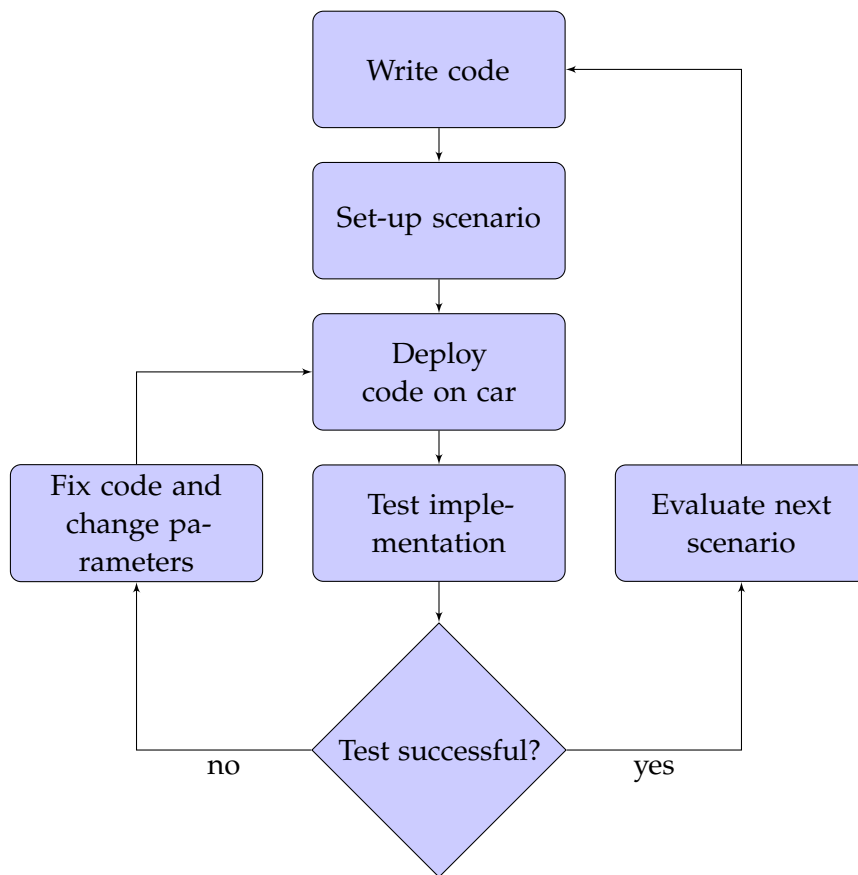


Figure 1.2.: Testing of vehicle software on the real track

As of 2017, the Carolo-Cup rules were changed to include an extended mode featuring additional challenges like zebra crossings with pedestrians, road signs, restricted areas and more complex intersection situations. To test all those scenarios on the real track, huge testing areas must be prepared and managed which costs lots of time and effort. Because the author of this thesis is member of TUM Phoenix Robotics, the Carolo-Cup team from Technical University Munich, he is interested to improve on the current, complex and error-prone software testing model.

Physical testing of the real vehicle at TUM Phoenix is usually done as shown in figure 1.2: Software is first written for a specific scenario that is currently built up. Exemplary, the scenario might be an obstacle in a sharp curve. The software is then iterated until the vehicle recognizes the obstacle correctly and behaves as expected. Then, a new scenario built up, e.g. an obstacle outside of the road that should just be ignored. Now the software is changed to satisfy the new requirement. The software may now also ignore the obstacle from the first scenario but is never tested again there. Obviously, this testing style can never lead to a fully-tested hardware-software concept. By changing model parameters for one scenario, another scenario might fail. The only viable solution to this problem is Software-In-The-Loop (SIL) testing. The vehicle's software is detached from its physical model and inserted into a simulation of the real scenario. Sensor data is generated from a virtual 2D or 3D world and fed into the software loop while actuator data is applied back onto the virtual car.

The simulation environment should enable rapid software development and reliable preliminary testing of the on-board software. It should be able to communicate with the existing software modules and simulate various sensor signals, e.g. accelerometer and camera data.

This bachelor thesis consists of three parts (see figure 1.3): The first part focuses on randomized road generation using a newly-proposed Template XML language. The road generator outputs CommonRoad XML. The second part explains the CommonRoad XML scenario description language and newly-proposed extensions. In the third part, a 3D visualization of the CommonRoad XML format in Gazebo is implemented.

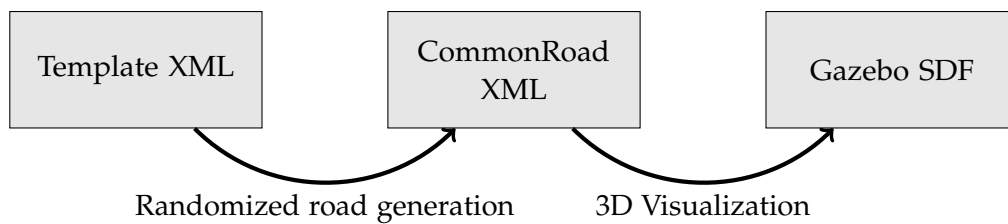


Figure 1.3.: Overview over this thesis

2. Related Work

There is currently no open-source solution available to simulate a vehicle in a Carolo-Cup like scenario. Though, there are two commercially available car simulators, “Virtual Test Drive” by VIRES and “CarMaker” by IGP. Their features will be explained in the next sections.

However, both frameworks do not suffice the financial and organizational constraints of a typical team of the Carolo-Cup. Software that is not freely available, let alone open-source, does not fit well with constantly changing team structures and yearly changing requirements on the competition and the vehicle. Also, all available solutions are optimized for real-world scenarios and would require major adjustments to fit the tight rule set of the Carolo-Cup.

CarMaker¹ by IGP Automotive is a framework for virtual test driving of automobiles. The framework includes several vehicle models that are simulated by a physics engine that allows multi-body systems. Vehicles can be configured to move on a predefined trajectory. The environment model allows multi-lane roads, intersections and highway exits. Additionally, traffic-specific objects like line markings, traffic signs and guide posts can be visualized in the simulation. Also, atmospheric objects like buildings and vegetation can be inserted into the virtual world. The framework includes a test manager that allows automated testing. CarMaker integrates with MATLAB and ADTF, the automotive framework developed by Elektrobit and mainly used by Audi.

Virtual Test Drive² by VIRES is mainly used as a simulator in driving studies with real drivers. However, Virtual Test Drive is able to simulate several sensors and traffic flow via SUMO³. Roads are described in the OpenDRIVE⁴ XML format.

¹<https://ipg-automotive.com/products-services/simulation-software/carmaker/>

²<https://www.vires.com/products.html>

³http://www.dlr.de/ts/en/desktopdefault.aspx/tabid-9883/16931_read-41000/

⁴<http://www.opendrive.org/>

3. Randomized Road Generation

The Carolo Cup routes are round trips in order to imitate an infinitely long track. For unit testing this is neither necessary nor target-aimed since a lot of different test cases should be run instead of a single long one. For a specific scenario, a lot of different test cases should be generated. For this purpose, this paper proposes *Template XML*, an XML-based language to define road primitives and their ordering. A road primitive can be a short road section or a more complex object like an intersection, as shown in figure 3.1. *Template XML* knows different concatenation strategies that can be nested at will.

During road generation, the template is evaluated top-down, resulting in a list of primitives. The primitives are then placed on a 2D coordinate system to build a single, connected road. This road is then exported as CommonRoad XML, which is explained in a later chapter. In the next section, all available primitives in *Template XML* are explained.

3.1. Primitives

A single primitive consists of one or more road lanes, traffic signs and/or obstacles. Each primitive lives in its own coordinate system. In this coordinate system, the primitive must have a begin and end point with a begin and end vector. The begin point is usually placed at beginning of the middle line with the begin vector aligned along the middle line's tangent looking *backwards* from the primitive. Similarly, the end point should be at the end of the middle line with the end vector aligned along the tangent looking *forwards* from the primitive.

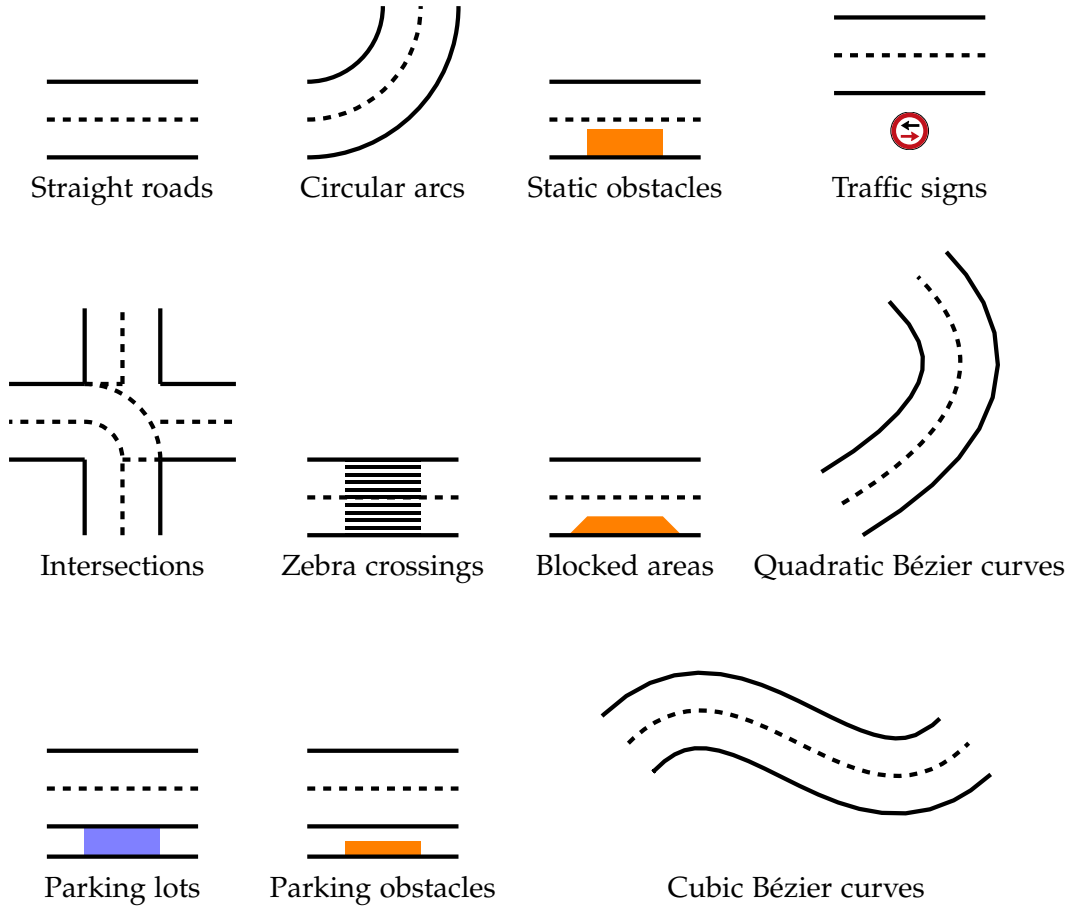


Figure 3.1.: Overview of all primitives

3.1.1. Straight Road

The straight line primitive is, as its name suggests, a straight line of arbitrary length. To account for special line markings, as defined in the Carolo Cup rules, the marking type can be set for each line, see table 3.1. The default markings are solid for the left and right line, and dashed for the middle line. Line markings can also be set to *missing*, meaning that the line will not be rendered in visualization. The straight road's begin point is at $(0,0)$ with the end vector rotated at π . The end point is at $(length,0)$ with the end vector pointing at angle 0, as shown in figure 3.2.

<line>		
length	<i>required</i>	float
leftLine	<i>optional</i>	string, default=solid
middleLine	<i>optional</i>	string, default=dashed
rightLine	<i>optional</i>	string, default=solid

Table 3.1.: Definition of the <line> element

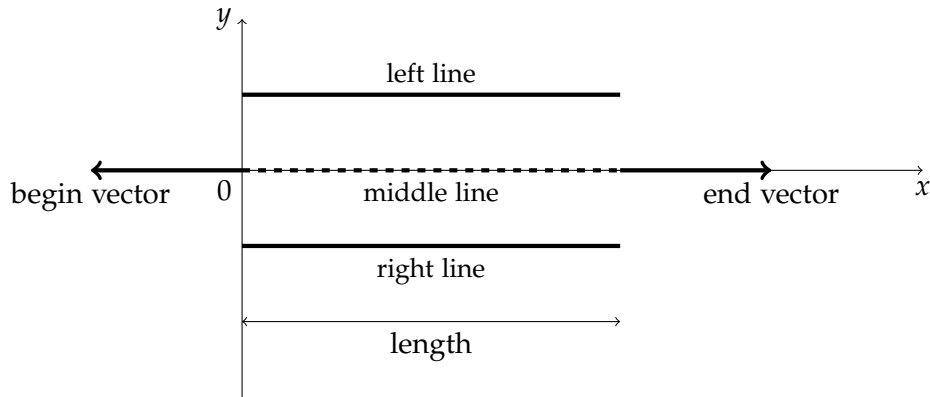


Figure 3.2.: Schematic presentation of the straight road primitive

3.1.2. Arc

The circular arc primitive bends with constant curvature, see figure 3.3. The primitive's parameters are the radius and the opening angle, as shown in table 3.2. The radius is measured from the curve's center point to the road's middle line. The angle is set in degrees. Arcs can be curved to the left or to the right. Similarly to straight lines, arcs can have modified line markings.

<leftArc> / <rightArc>		
radius	<i>required</i>	float
angle	<i>required</i>	float
leftLine	<i>optional</i>	string, default=solid
middleLine	<i>optional</i>	string, default=dashed
rightLine	<i>optional</i>	string, default=solid

Table 3.2.: Definition of the <leftArc> / <rightArc> element

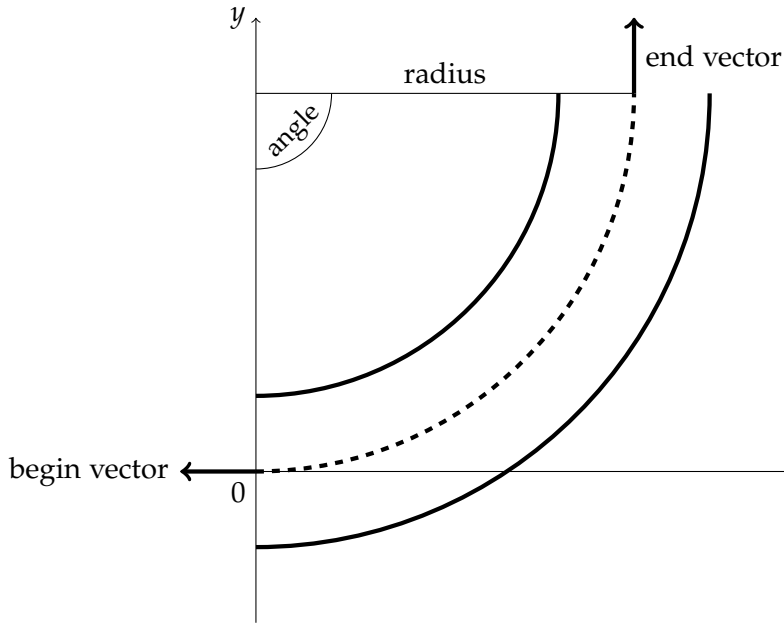


Figure 3.3.: Schematic presentation of the left arc primitive

3.1.3. Quadratic Bézier curve

The quadratic Bézier is based on the mathematical description of a curve that is defined by a start point P_0 , an end point P_2 and a single control point P_1 . A Bézier curve of grade n can be defined recursively as $C_0^n(t)$, $t \in [0, 1]$ using De-Casteljau's algorithm [She02]:

$$C_i^0(t) = P_i \quad (3.1)$$

$$C_i^j(t) = (1-t)C_i^{j-1}(t) + tC_{i+1}^{j-1}(t) \quad (3.2)$$

By iterating over t from 0 to 1 in arbitrary small steps, a polyline can be built that closely resembles the actual curve.

The quadratic Bézier $C_0^2(t)$ is then derived as follows:

$$C_0^2(t) = (1-t)C_0^1(t) + tC_1^1(t) \quad (3.3)$$

$$= (1-t)((1-t)C_0^0 + tC_1^0) + t((1-t)C_1^0 + tC_2^0) \quad (3.4)$$

$$= (1-t)((1-t)P_0 + tP_1) + t((1-t)P_1 + tP_2) \quad (3.5)$$

$$= (1-t)^2P_0 + 2t(1-t)P_1 + t^2P_2 \quad (3.6)$$

In order to get the curve's tangent at P_0 and P_2 , the curve equation is differentiated

by t :

$$\frac{d}{dt}C_0^2(t) = 2(1-t)(P_1 - P_0) + 2t(P_2 - P_1) \quad (3.7)$$

By evaluation of (3.7) for $t = 0$ and $t = 1$, the tangents can be computed:

$$\frac{d}{dt}C_0^2(0) = 2(P_1 - P_0) \quad (3.8)$$

$$\frac{d}{dt}C_0^2(1) = 2(P_2 - P_1) \quad (3.9)$$

The tangent at P_0 is the line between P_0 and P_1 (3.8) and the tangent at P_1 is the line between P_1 and P_2 (3.9), as shown in figure 3.4.

Since the primitive is moved and rotated during road generation, the exact position in the coordinate system does not matter, thus the curve's first point can be defined to be $P_0 = (0,0)$ without restricting its shape. All other points can be freely configured by the user, see table 3.3.

<quadBezier>		
p1x	<i>required</i>	float
p1y	<i>required</i>	float
p2x	<i>required</i>	float
p2y	<i>required</i>	float

Table 3.3.: Definition of the <quadBezier> element

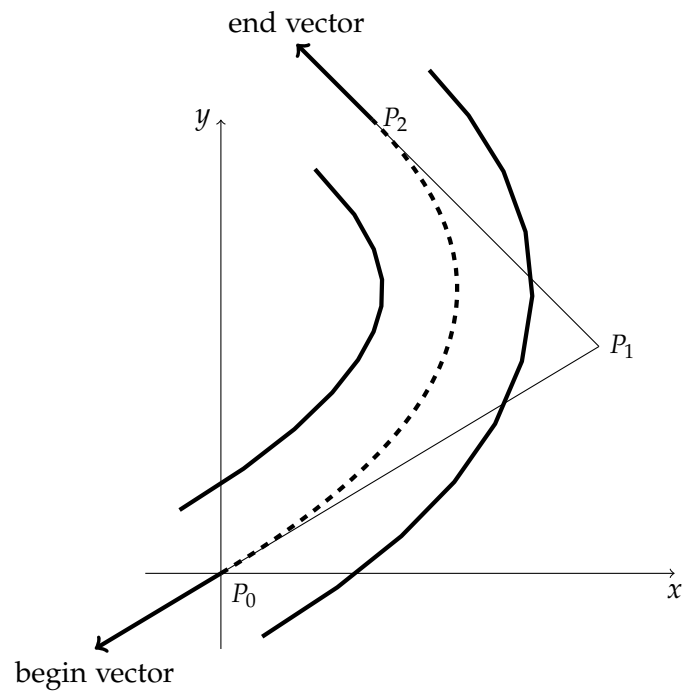


Figure 3.4.: Schematic presentation of the quadratic bézier primitive

3.1.4. Cubic Bézier curve

The cubic Bézier primitive is useful to define S-bends and even more complex curves than circular arcs and quadratic Béziers, see figure 3.5. A cubic bezier curve is unambiguously defined by a start point P_0 , an end point P_3 and two control points P_1 and P_2 . Reusing De-Casteljau's algorithm (3.1) and (3.2), the cubic Bézier $C_0^3(t)$ is defined as:

$$C_0^3(t) = (1-t) \left\{ (1-t) \left[(1-t)P_0 + tP_1 \right] + t \left[(1-t)P_1 + tP_2 \right] \right\} + \quad (3.10)$$

$$t \left\{ (1-t) \left[(1-t)P_1 + tP_2 \right] + t \left[(1-t)P_2 + tP_3 \right] \right\} \\ = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3 \quad (3.11)$$

The curve equation is then differentiated by t

$$\frac{d}{dt} C_0^3(t) = 3(1-t)^2 (P_1 - P_0) + 6(1-t)t (P_2 - P_1) + 3t^2 (P_3 - P_2) \quad (3.12)$$

to compute the tangent at P_0 and P_3 :

$$\frac{d}{dt} C_0^3(0) = 3(P_1 - P_0) \quad (3.13)$$

$$\frac{d}{dt} C_0^3(1) = 3(P_3 - P_2) \quad (3.14)$$

Similarly to the quadratic Bézier curve, the tangent at P_0 is line between P_0 and P_1 , and the tangent at P_3 is line between P_2 and P_3 .

P_0 is again set to $(0,0)$. All other points can be freely configured as long as the resulting road does not intersect with itself. The XML definition is shown in table 3.4.

<cubicBezier>		
p1x	required	float
p1y	required	float
p2x	required	float
p2y	required	float
p3x	required	float
p3y	required	float

Table 3.4.: Definition of the <cubicBezier> element

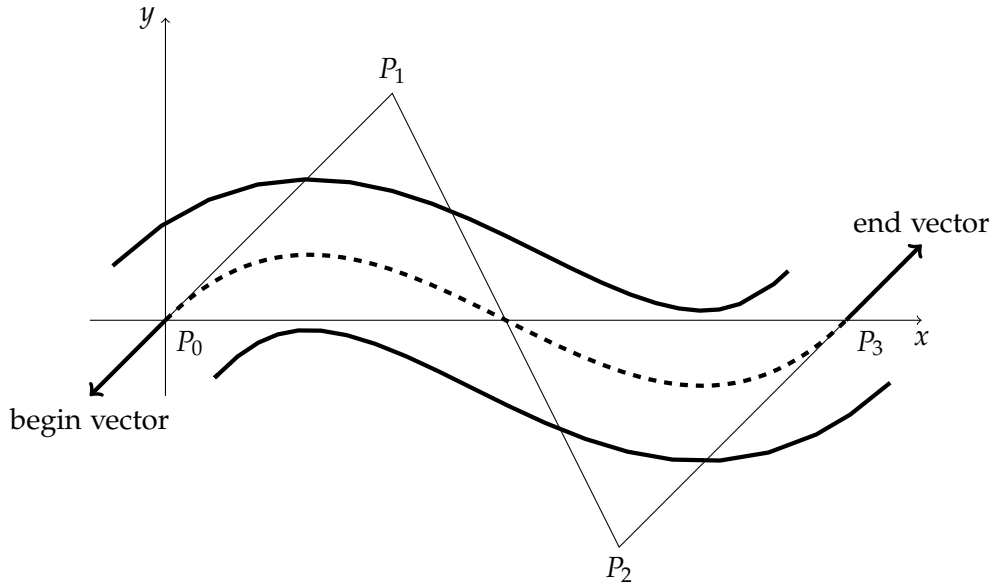


Figure 3.5.: Schematic presentation of the cubic Bézier primitive

3.1.5. Static Obstacle

The static obstacle primitive is a specialization of the straight road primitive. The length of the short straight road is also the length of the static obstacle. Additionally, the width and position of the obstacle can be specified, as shown in table 3.5. The position is relative to a given anchor point that can be the left or right edge or the center of the obstacle. The obstacle is then placed at the given position on the y-axis. Setting an obstacle's position to -0.5 with anchor at the center will always position the obstacle in the middle of the right lane regardless of the road's width and obstacle's width, as shown in figure 3.6.

<staticObstacle>		
width	<i>required</i>	float
length	<i>required</i>	float
position	<i>required</i>	float
anchor	<i>required</i>	enum(left, center, right)

Table 3.5.: Definition of the <staticObstacle> element

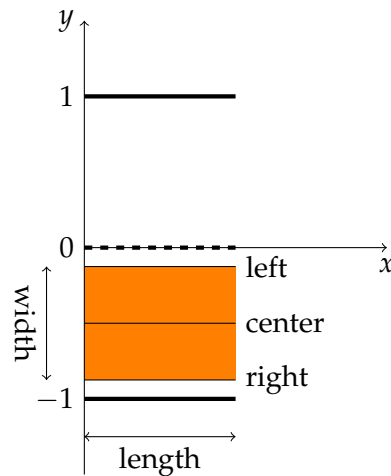


Figure 3.6.: Schematic presentation of the static obstacle primitive

3.1.6. Intersection

In the Carolo-Cup, only intersections of two roads are allowed. In figure 3.7, the vehicle is coming from the lower road. From the perspective of the arriving vehicle, there can be five different traffic rules: The vehicle can be on the priority road and the intersecting road has either yield or stop priority. Otherwise the vehicle itself may come from the yield or stop road. The intersection may also have the *left yields before right* rule. Depending on the traffic rules at the intersection, there may be traffic signs and stop lines at the tip of some lanes. For yield lanes, the stop line is dashed and the yield sign is placed. For stop lanes, the stop line is solid and a stop sign is placed beside the lane. On priority lanes there is no stop line and a priority sign shows the traffic precedence. Additionally, turning may be mandatory. The vehicle is then required to turn left or right at the intersection. In this case, dashed turn lanes and a direction sign indicate the turning direction. The XML definition is shown in table 3.6.

<intersection>		
rule	required	enum(priority-yield, priority-stop, yield, stop, equal)
turn	required	enum(left, straight, right)

Table 3.6.: Definition of the <intersection> element

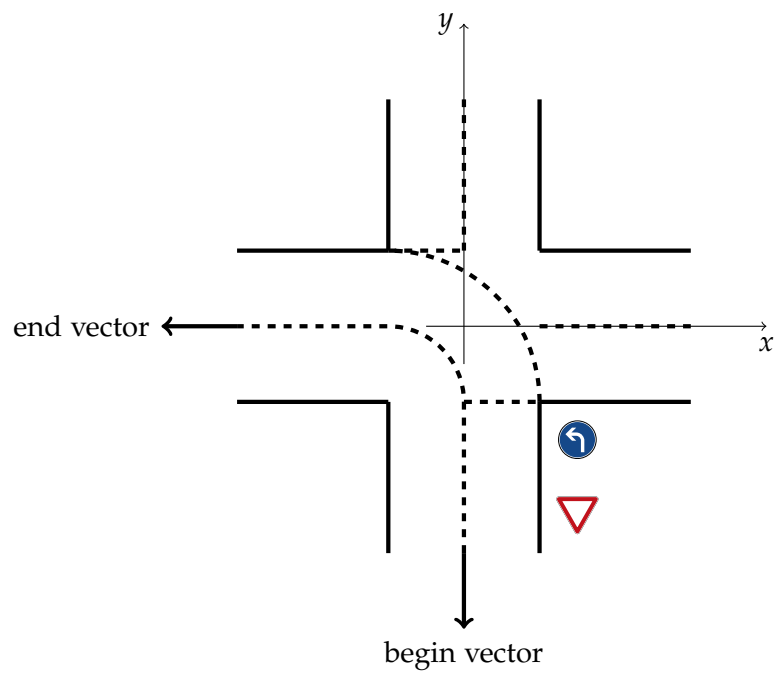


Figure 3.7.: Schematic presentation of an intersection with turn lane

3.1.7. Zebra crossing

The zebra crossing is a specialization of the straight road primitive. Only the length of this primitive can be specified and the actual zebra crossing is as long as the primitive, see figure 3.8. The exact placement of the stripes is up to the visualization and cannot be further defined. The zebra crossing has also no visible left, right or middle line. See table 3.7 for the definition in *Template XML*.

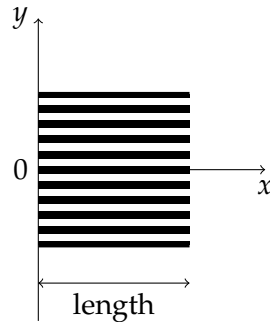


Figure 3.8.: Schematic presentation of the zebra crossing primitive

<zebraCrossing>		
length	required	float

Table 3.7.: Definition of the <zebraCrossing> element

3.1.8. Blocked area

The blocked area primitive is similar to the static obstacle primitive. It represents a straight road segment with a specially labeled area on the right lane. This area symbolizes a building site or fenced off area on a road that must not be entered or touched. In the Carolo-Cup, this area has a trapezoid shape that is striped diagonally. Only its length and width can be specified, see definition table 3.8. As shown in figure 3.9, the blocked always aligns with the right line.

<blockedArea>		
width	required	float
length	required	float

Table 3.8.: Definition of the <blockedArea> element

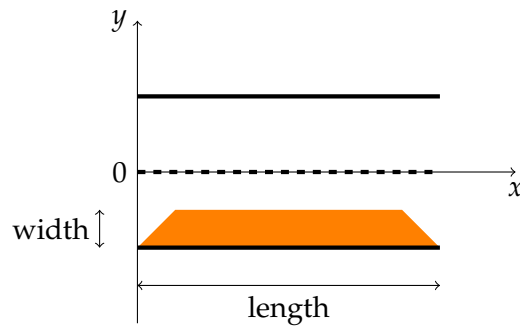


Figure 3.9.: Schematic presentation of the blocked area primitive

3.1.9. Traffic sign

The traffic sign primitive is special, because it does not contain a road element, instead it solely consists of a traffic sign, as shown in 3.9. Begin and end point of this primitive are at $(0,0)$. The begin vector is oriented at angle π and the end vector in the opposite direction at 0. The traffic sign is placed $15cm$ besides the right line. Valid sign types are defined in a later chapter in table 4.2.

<trafficSign>		
type	required	enum(stv-205, stvo-206, ...)

Table 3.9.: Definition of the <trafficSign> element

3.1.10. Parking Lot

The parking lot primitive is a specialization of the straight road primitive. The only addition is a line that marks the parking lot. The vehicle is expected to park parallel to the road. The lot is only defined by its length, as shown in figure 3.10. The width is $30cm$, as specified in the Carolo-Cup rules. See table 3.10 for the XML specification.

<parkingLot>		
length	required	float

Table 3.10.: Definition of the <parkingLot> element

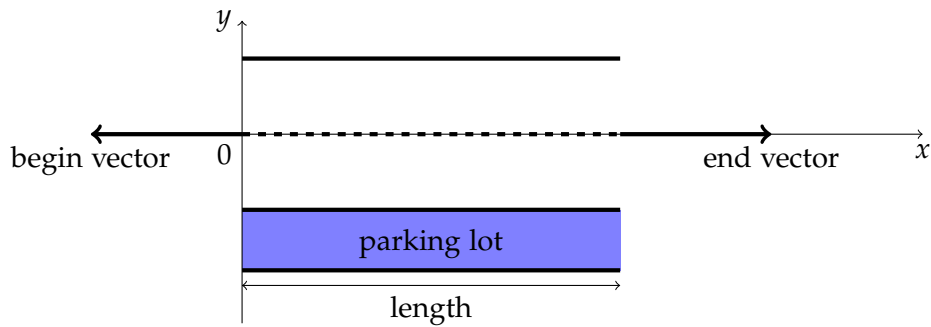


Figure 3.10.: Schematic presentation of the parking lot primitive

3.1.11. Parking obstacle

The parking obstacle is a specialization of the parking lot primitive. But for this primitive, the parking lot is occupied by an obstacle of a given width, as shown in figure 3.11. The obstacle is always aligned at the parking line. See table 3.11 for the definition in *Template XML*.

<parkingObstacle>		
length	<i>required</i>	float
width	<i>required</i>	float

Table 3.11.: Definition of the <parkingObstacle> element

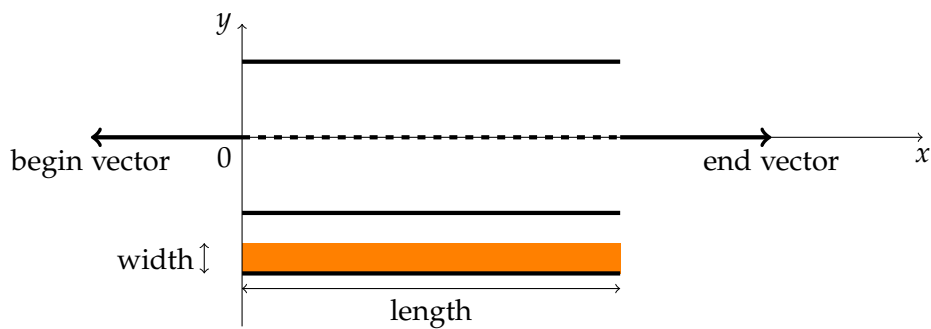


Figure 3.11.: Schematic presentation of the parking obstacle primitive

3.2. Control structures

In the previous section, different primitives were explained in detail. In this section, a declarative approach to define the order of several primitives is elaborated. Based on control structures found in imperative programming languages, like loops and if-statements, several declarative structures were invented. All of these structures allow one or more child elements and return them based on probabilistic decisions.

3.2.1. Sequence

The sequence element returns all children elements in the unchanged order and exactly once. It is useful to define a specific sequence of elements to act as one, e.g. a zebra crossing with the corresponding traffic sign in front of it, see listing 1.

```
<sequence>
  <trafficSign type="stvo-350-10" />
  <zebraCrossing length="0.4" />
</sequence>
```

Listing 1: Zebra crossing with traffic sign in a sequence

3.2.2. Optional

The `<optional>` element allows only one or more child elements. With probability p , all child elements are returned in unchanged order and with probability $(1 - p)$ an empty list is returned. For probability p applies $p \in [0, 1]$. The `<optional>` element is useful to optionally add a certain primitive.

3.2.3. Select+Case

The `<select>` element is a generalization of the `<optional>` element. It only allows `<case>` elements as children and returns one of them based on weighted probabilities. Each `<case>` element specifies a weight w that is proportionally weighted against the other `<case>` elements. For example, three `<case>` elements with $w_0 = 2, w_1 = 3, w_2 = 4$ result in the probability $p_0 = 2/(2 + 3 + 4) = 2/9$ for the first, $p_1 = 3/9$ for the second and $p_3 = 4/9$ for the third `<case>`. All child elements of the chosen `<case>` are then returned in unchanged order.

3.2.4. Repeat

The `<repeat>` element treats all child elements as sequence and repeats this sequence as often as specified. Either an exact number n or an interval for the number repetitions can be set. If an interval $[min, max]$ is given, the number of repetitions is determined by a uniform distribution over all interger values in the interval.

3.2.5. Shuffle

The `<shuffle>` element returns a random permutation of its children. The element has no attributes. It is useful to generate parking scenarios.

For generating a random permutation, the Fisher-Yates shuffle algorithm [FY38] should be used. All possible permutations are equally likely to be produced by this algorithm. Its optimized version has a time complexity of $\mathcal{O}(n)$ on the number of elements to shuffle and can be implemented to work in-place on the input array.

3.3. Evaluation of Template XML

The control structures can be nested and are evaluated top-down. That means that outer element is evaluated first, then its children, then their children and so on. Exemplary, the following template description is evaluated:

```
<sequence>
  <trafficSign type="stvo-350-10" />
  <zebraCrossing length="0.4" />
  <repeat n="3">
    <line length="2" />
    <optional p="0.33">
      <leftArc radius="2" angle="10" />
    </optional>
  </repeat>
</sequence>
```

First, the `<sequence>` is evaluated, since its the top most control structure. This will just return all child elements in unchanged order:

```
<trafficSign type="stvo-350-10" />
<zebraCrossing length="0.4" />
<repeat n="3">
  <line length="2" />
```

```
<optional p="0.33">
  <leftArc radius="2" angle="10" />
</optional>
</repeat>
```

Elements of primitives cannot be evaluated and are left as they are. Instead, the `<repeat>` element is now evaluated. In this case, it returns all children three times:

```
<trafficSign type="stvo-350-10" />
<zebraCrossing length="0.4" />
<line length="2" />
<optional p="0.33">
  <leftArc radius="2" angle="10" />
</optional>
<line length="2" />
<optional p="0.33">
  <leftArc radius="2" angle="10" />
</optional>
<line length="2" />
<optional p="0.33">
  <leftArc radius="2" angle="10" />
</optional>
```

Next, all `<optional>` elements are evaluated. Since `<optional>` elements do not have a deterministic result, the following evaluation step is only a possible outcome:

```
<trafficSign type="stvo-350-10" />
<zebraCrossing length="0.4" />
<line length="2" />
<leftArc radius="2" angle="10" />
<line length="2" />
<line length="2" />
```

Now, there are no control structures left to evaluate, so the evaluation is finished and the order of primitives is fixed. The next section shows how the primitives are placed in a global coordinate system to build a road network.

3.4. Concatenation of Primitives

After evaluating *Template XML*, a fixed order of primitives was determined. The primitives can now be placed in a common coordinate system by translating and

rotating their local coordinate system.

Using the begin/end point and vector, the primitives can be connected to build a connected road. The first primitive is translated and rotated so that its begin point is at $(0,0)$ and the begin vector is looking at orientation π . The second primitive is then translated so that its begin point covers the first primitive's end point and rotated so its begin vector looks in the opposite direction of the first primitive's end vector. The same is applied for all following primitives and their direct predecessor.

In figure 3.12, a left arc is the first primitive. The first primitive is already in place and does not need a translation or rotation. Its end point is at $(3,3)$ and the end vector looks at $\pi/2$. The second primitive, a right arc, is now translated so that its begin point is at $(3,3)$, covering the left arc's end point. The begin vector of a right looks at π . That means, the right arc must be rotated $\pi/2$ so that its begin vector looks in the opposite direction of the left arc's end vector. The third primitive is a straight road. It is translated by $(6,6)$ so that its begin point covers the right arc's end point. The straight road does not a rotation since the begin/end vectors already look in opposite directions.

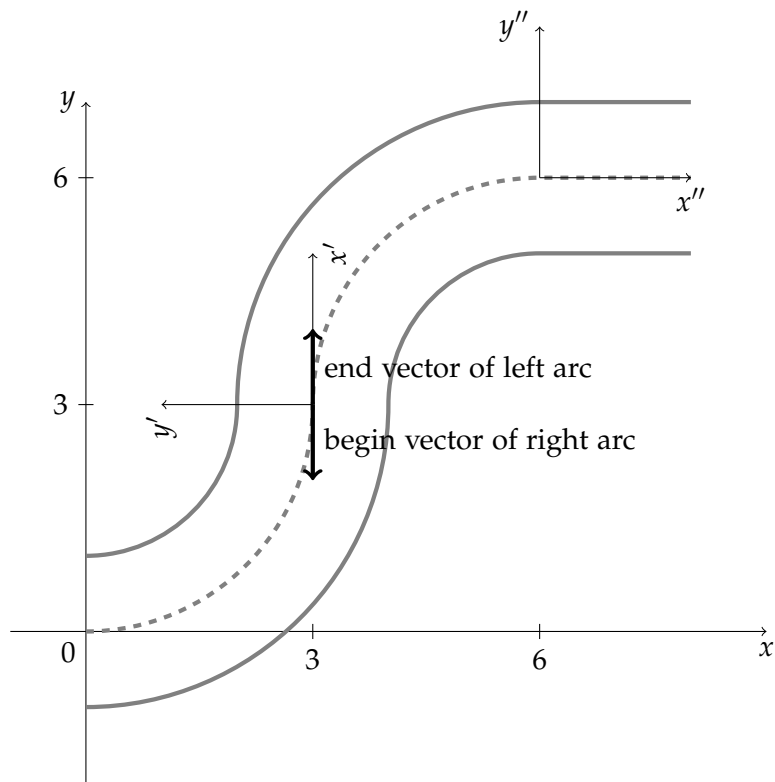


Figure 3.12.: Concatenation of primitives

4. CommonRoad XML Format

4.1. CommonRoad Basics

The CommonRoad XML specification [AKM17] allows to define an abstract map consisting of roads, static and dynamic obstacles. It is designed to be used for highway and country road scenarios with multi-lane roads and complex junctions. The specification [KA17] defines four root objects that will be shortly explained in the next sections.

4.1.1. Coordinate System and Units

All coordinates are defined in a cartesian coordinate system with the x-axis to the right and the y-axis to the top. For 3D coordinates, right-handed coordinates are used. This is an often used convention and particularly useful for the interaction with OpenGL¹ and Gazebo². Orientation is defined on the x-y-plane in mathematically positive (anti-clockwise) direction with orientation 0 starting from the x-axis, see figure 4.1.

Numeric values in the CommonRoad specification use SI units. Angle are defined in radian.

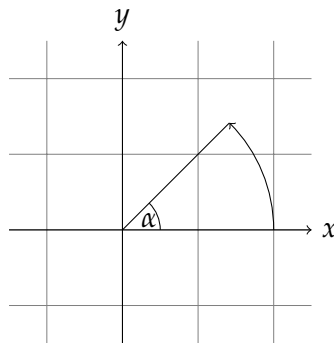


Figure 4.1.: Coordinate system and orientation α

¹<https://www.opengl.org/>

²<http://gazebo.org/>

4.1.2. Lanelets

“Lanelets compose the road network with lanes, roads and intersections.” [BZS14]. A lanelet is part of a roadway of any length that is wide enough to be used by a single line of vehicles. Lanelets can have different purposes, ranging from turn lanes to acceleration lanes that allow drivers to speed up before merging into a highway. Lanes are usually bordered by line markings to visually separate them from other lanes. Lanelets have a predestined traffic direction and do not allow vehicles to move in the opposite direction except for overtaking maneuvers and temporary restrictions like building sites.

Lanelets are an abstract model of real world lanes. Contrary to OpenDRIVE [Dup15], lanelets as defined by [BZS14] do not model lanes with complex geometrical objects like arcs, clothoids and polynomials but rather simple polylines. Each lanelet consists of two boundaries, defining the left and the right boundary of the street. Each boundary may have a dashed or solid line marking. The stop line is modeled as the connection between the last points of each boundary.

Since a single lanelet cannot represent a complex road graph, a lanelet has succeeding and predecesing lanelets that directly follow from or merge into the lanelet. The end points of the boundaries of one lanelet must be equal with the start points of the succeeding lanelet, and similarly for the predecesing lanelet.

Lanelets can also have adjacent lanelets that may have the same or the opposite driving direction. On a two-lane road, two lanelets have an adjacent lanelet on their left side with opposite driving direction. Lanelets may not have more than one adjacent lanelet on one side. If this is the case, the lanelet must be splitted so that the adjacence can be defined properly.

Lanelets may be also without any line marking to model special lanes like u-turns or turn lanes at junctions. At junctions, each turn lane must be explicitly defined.

See section A.2 for the full XML definition of lanelets.

4.1.3. Obstacles

To further enhance the environment, static and dynamic obstacles can be defined. Static obstacles can be e.g. walls, parked cars or building sites. Dynamic obstacles are other road users like cars, trucks, motorcycles or pedestrians. An obstacle is modeled as a shape, consisting of rectangles, circles or polygons.

Additionally, obstacles can have either a trajectory or an occupancy set. The trajectory is a list of states, where each state is a tuple of time, position, orientation, velocity and acceleration. The dynamic obstacle will adopt the state on each exact point in time and interpolates linearly between them.

An occupancy set is a list of shapes the vehicle will occupy at certain time points. This may be useful to model uncertainties from the real world [AM16].

4.1.4. Ego Vehicle

The ego vehicle shares some concepts with the obstacle definition but is inherently different in its behavior. The ego vehicle is a dynamic vehicle without a rigid trajectory, it is rather the object under simulation or being benchmarked. Therefore, the ego vehicle has a goal region, that are regions with a target orientation, velocity and acceleration. If a goal region's conditions are met, the test is successful.

In a Software-In-The-Loop simulation [Bra+14], the ego vehicle's sensor data will be fed into a software system.

4.2. CommonRoad Extensions

Additionally to the CommonRoad specification, this thesis proposes extensions useful for urban areas and the Carolo Cup extended competition. The extensions include new root elements like intersections and traffic signs, as well as small additions like stop lines and zebra crossings. These extensions will be explained in this section.

4.2.1. Intersections

When two or more roads cross there can be an intersection. Special traffic laws are applied to intersections: Some lanes can be in priority to other lanes which is designated by traffic signs, traffic lights or implicit rules like "left yields to right".

In CommonRoad, each possible straight crossing or turning possibility must be defined with its own lanelet. Exemplary, in figure 4.2, lanelet A is connected to lanelet C by lanelet B that represents *turning left* at the junction coming from road 1. Similarly, lanelet D connects roads 4 to 2 for vehicles going straight.

To model the priority relations, an additional `<intersection>` element is introduced to CommonRoad. The element declares all lanelets being part of the junction and 1-to-1 relations defining a weak ordering between them. Each `<priority>` is a relation between one major road and a minor road, as shown in listing 2.

For the junction of figure 4.2, two priority tables (table 4.1) were made to show the two cases for vehicles coming from the priority lane of road 1 and vehicles coming from road 2 and yielding. A vehicle coming from the priority road (road 1) and showing a left turn signal will use lanelet "B". The column " $1 \rightarrow 4$ " now shows the vehicle's behavior if there are vehicle on any of the other lanelets. For example a vehicle coming from road 3 and turning right into road 4 has priority over the one turning left.

Each priority sign means a priority for the ego vehicle whereas each yield sign implies the ego vehicle to wait for another vehicle. Lanes that do not cross have usually no relation to each other and are without any symbol in the table, e.g. two opposing vehicles both turning left will not obstruct each other, therefore no priority is necessary.

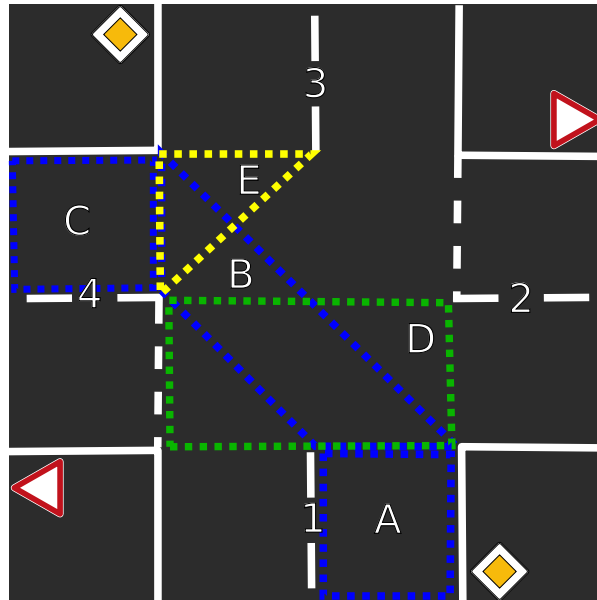


Figure 4.2.: Lanelets on intersection with priority and yield lanes





























Other \ Ego	1 → 2	1 → 3	1 → 4	Other \ Ego	2 → 3	2 → 4	2 → 1
2 → 1				1 → 2			
2 → 3				1 → 3			
2 → 4				1 → 4			
3 → 1				3 → 1			
3 → 2				3 → 2			
3 → 4				3 → 4			
4 → 1				4 → 1			
4 → 2				4 → 2			
4 → 3				4 → 3			

Table 4.1.: Priorities for vehicles coming from the priority lane (left) and the yield lane (right)

```

<intersection>
  <composition>
    <lanelet ref="-7" />
    ...
    <lanelet ref="-12" />
  </composition>
  <priority low="-8" high="-7" />
  <priority low="-9" high="-7" />
  <priority low="-12" high="-10" />
  ...
</intersection>

```

Listing 2: Example for an intersection

4.2.2. Zebra crossings

A zebra crossing is a crossing of pedestrian side walks and roads. Since sidewalks can also be modeled with lanelets, the zebra crossing is designed to be a connection between lanelets for pedestrians of opposite sides of a road. Lanelet for zebra crossings set their lanelet type to `zebraCrossing`, see section A.2 for the specification. The intersecting lanelets must be split at the borders of the zebra crossing to make new lanelets without line markings.

4.2.3. Blocked Areas

A blocked area is a building site on a road. Blocked areas are modeled as static obstacles in the XML format. For this purpose, the new obstacle type `blockedArea` is added.

4.2.4. Parking Lots

Parking lots are places to leave a vehicle. In parking scenarios, they are the goal regions that must be reached. In CommonRoad XML they are defined similarly to obstacles with a rectangular shape. Parking lots can be placed on a lanelet and may be surrounded and occupied by static obstacles or other parking lots.

4.2.5. Traffic Signs

Traffic signs are placed next to roads and give hints or rules for the upcoming part of the road. Priority, yield and stop signs control the traffic at intersections. Speed limit signs and *no overtaking* restrict the vehicles abilities in what is usually allowed. These traffic signs come in pairs, one to start a zone and one to end it. The zebra crossing and curve signs are hints to the driver to reduce the chance of accidents. Table 4.2 lists all fifteen traffic signs being used in the Carolo Cup[Bra16] that can also be found at German roads³.

In CommonRoad XML each traffic sign is a root element with a position and orientation. Its type is defined by the numbering used in the German traffic law (dt. "StVO" [Ver13]). Each sign has a position and orientation.

4.2.6. Stop line

Stop lines can be located at intersections and indicate a vehicle to stop or yield there. Every lanelet can have a stop line. Stop lines can be `solid` or `dashed`.

³Image sources: <https://de.wikipedia.org> and <https://commons.wikimedia.org>

Symbol	English name	German name	XML type
	Priority / Right of way	Vorfahrtsstraße	stvo-306
	Yield	Vorfahrt gewähren!	stvo-205
	Stop	Halt! Vorfahrt gewähren!	stvo-206
	Give way to oncoming vehi- cles	Dem Gegenverkehr Vorrang gewähren!	stvo-208
	No overtaking	Überholverbot für Kraft- fahrzeuge aller Art	stvo-276
	No overtaking revoked	Ende des Überholverbot für Kraftfahrzeuge aller Art	stvo-280
	Speed limit 30	Beginn der Zone mit zuläs- siger Höchstgeschwindigkeit 30 km/h	stvo-274.1
	Speed limit 30 revoked	Ende der Zone mit zuläs- siger Höchstgeschwindigkeit 30 km/h	stvo-274.2
	Zebra crossing	Fußgängerüberweg	stvo-350-10
	Turn left ahead	Vorgeschriebene Fahrtrich- tung - links	stvo-209-10
	Turn right ahead	Vorgeschriebene Fahrtrich- tung - rechts	stvo-209-20
	Curve sign to the left (small)	Richtungstafel in Links- Kurven (klein)	stvo-625-10
	Curve sign to the left (large)	Richtungstafel in Links- Kurven (groß)	stvo-625-11


	Curve to the (small)	sign right	Richtungstafel Kurven (klein)	in	Rechts-	stvo-625-20
	Curve to the (large)	sign right	Richtungstafel Kurven (groß)	in	Rechts-	stvo-625-21

Table 4.2.: Traffic signs

5. Visualization

The 3D visualization implemented for this thesis is based on the Gazebo framework. Gazebo is able to simulate a three-dimensional world with integrated physics. It consists of the OGRE 3D engine to render an octree-based scenegraph and different physics engines like ODE¹ and Bullet² to detect collisions between moving objects and to simulate gravity, friction and wind. Gazebo is also able to simulate sensor data from cameras, laser scanners or simple distance sensor, including sensor noise.

A Gazebo simulation was provided for teams of the DARPA challenge³ that could not afford to build their own robot. The teams could implement their robot in ROS and run it either on the real robot or the simulation environment using Gazebo's ROS integration. Gazebo is also able to load C++ plugins that can move an object and read its sensor values.

On start up, Gazebo loads a SDF file that defines the simulation world. SDF is an open XML format to model a visual and physical 3D world using predefined primitives and custom models. The SDF file also specifies light sources and physical properties of the world and the models.

To visualize a CommonRoad XML file, all elements must be translated to models in the Gazebo's SDF world file. First lanelets, blocked areas, intersections and zebra crossings are rendered as a flat groundplane. Second, static and dynamic obstacles are added as boxes at their initial pose in the world. Third, traffic signs are inserted as semi-transparent flat objects. Fourth, the ego vehicle is placed.

5.1. Groundplane

The groundplane could be rendered as a single SDF-<plane> primitive with a single large texture. However, since OGRE uses OpenGL, the maximum texture size is platform-dependent and can range from 1024x1024 px to 16384x16384 px or even higher. By testing on different hardware (Intel HD Graphics and Nvidia Geforce), a texture size of 2048x2048 seems to have good support on current hardware. This parameter is

¹<http://www.ode.org/>

²<http://www.bulletphysics.org/>

³<http://spectrum.ieee.org/automaton/robotics/robotics-software/darpa-robotics-challenge-simulation-software-open-source-robotics-foundation>

called the tile size t .

Another parameter to consider is the meter-to-pixel ratio r . There is a trade-off between long rendering times and huge GPU and disk memory usage for huge ratios and blurred textures for small ratios. In the Carolo-Cup scenario, the thinnest lines are 2cm wide. By experimentation, a ratio of $r = 500$ pixel per meter was determined, leading to at least 20 pixel wide lines.

Since all rendered objects should fit onto the groundplane, the bounding box of all objects must be computed. This is done by calculating the union over all bounding boxes of all lanelets, obstacles and traffic signs. A bounding box B is defined as a tuple as in (5.1). The union of two bounding boxes A and B is then computed as in (5.2).

$$B = (x_{min}, y_{min}, x_{max}, y_{max}) \quad (5.1)$$

$$\begin{aligned} \text{union}(A, B) &= \text{union}(x_{min}, y_{min}, x_{max}, y_{max}), (x'_{min}, y'_{min}, x'_{max}, y'_{max})) \\ &= (\min(x_{min}, x'_{min}), \min(y_{min}, y'_{min}), \max(x_{max}, x'_{max}), \max(y_{max}, y'_{max})) \end{aligned} \quad (5.2)$$

$$\text{padding}(B, p) = (x_{min} - p, y_{min} + p, x_{max} + p, y_{max} + p) \quad (5.3)$$

A lanelet's bounding box is computed as the minimum and maximum over all points of both boundaries. An obstacle's bounding box is the union of its shapes and a traffic sign's bounding box is just its center point.

Then a padding p is applied to the bounding box, as shown in (5.3). Padding is necessary to prevent the dynamic obstacles and ego vehicle to fall off the groundplane once they accidentally leave the road and to make sure that the ego vehicle's camera does not see a sharp edge at the groundplane's border. The padding is usually set to a few meters. A typical bounding box of $(0, 0, 10, 6)$ for a small world leads to a bounding box of $(-3, -3, 13, 9)$ when a padding of 3 meters is included. Therefore the bounding box is 13 meters wide and 10 long. A single groundplane texture would be 8000x6000 pixels large when a ratio of 500 pixel per meter is used. However, the texture is bigger than allowed by the platform, so it must be split into smaller textures. Assuming each tile to be of size $t = 2048$, the number of tiles needed (n_x, n_y) can be computed as in (5.4) and (5.5) for a world of width $w_x = 8000$ and length $w_y = 6000$.

$$n_x = \left\lceil \frac{w_x}{t} \right\rceil = \left\lceil \frac{8000}{2048} \right\rceil = 4 \quad (5.4) \quad n_y = \left\lceil \frac{w_y}{t} \right\rceil = \left\lceil \frac{6000}{2048} \right\rceil = 3 \quad (5.5)$$

It can be concluded that $n_x \cdot n_y = 12$ tiles are needed to cover the world in the example. Each tile can be rendered independently if the translated coordinate system is correctly computed for each.

In the reference implementation, the `cairocffi` package for Python 3 was used to

render the groundplane. For each tile, an `ImageSurface` of size (t, t) with type `RGB24` is created. The surface is then filled black and its y-axis is inverted to resolve the problem that most computer graphics libraries, like `cairocff`, use a coordinate system with an inverted y-axis and an origin in the upper left corner. Then the coordinates are scaled by the meter-to-pixel ratio. This makes it a lot easier to get the proportions right because lengths and coordinates no longer resemble pixels but meters. Thus, a line width of 0.02 units resolves to $0.02m = 2cm$ as expected. Now the tile's coordinate system must be translated to its position in the grid. This is done by first translating it to the lower left corner of the bounding box (x_{min}, y_{min}) and then to its position in the grid multiplied with adjusted tile size:

$$(x_{min} + g_x \cdot \frac{t}{r}, y_{min} + g_y \cdot \frac{t}{r}) \quad (5.6)$$

For the previous example, $t/r = 2048px/(500px/m) \approx 4.1m$. Thus, a single tile covers an area of roughly 4.1×4.1 meters.

Then, all objects can be rendered on the tile. Objects that do not intersect with the tile will be naturally ignored and but can be safely rendered nonetheless. For performance reasons, one could check intersection of an objects bounding box with the tile's bounding box before rendering. However, this is only necessary for large worlds with a lot of objects to render but in that case a quadtree or R-tree [Gut84] could be used to reduce intersection checks.

After rendering each tile is then saved in three different parts. The first part is to save the `ImageSurface` as a PNG image in the `materials/textures` folder. The file name is determined by the SHA-256 hash of the image's data. This is useful to remove duplicate textures. It is rather unlikely that two textures have the exact same hash except for one simple case: All textures that remain 100% black after rendering will have the same hash, thus they will be loaded only once by OGRE reducing the GPU memory usage. Second, the tile's material file is generated and saved in the `materials/scripts` folder. The material file is written in OGRE's material scripting language which allows to define complex texturing and lighting properties for 3D meshes. For the groundplane tile, only a simple `texture_unit` pass is needed, as shown in listing 3. The `PF_L8` defines the texture's pixel format to use 8 bits per pixel and only a single luminance channel, basically setting the texture to greyscale mode. This reduces the memory usage by 75% compared to the default pixel format `PF_A8R8G8B8` that uses four bytes per pixel. Assuming $d = 1GB$ of GPU memory and 8 bits per pixel $BBP = 8bits/px$, the maximum world area A can be computed as in equation 5.7.

$$A = \left(\frac{1}{r}\right)^2 \cdot \frac{d}{BBP} \quad (5.7)$$

$$= \left(\frac{1}{500\text{px/m}}\right)^2 \cdot \frac{8 \cdot 1024^3\text{bits}}{8\text{bits/px}} \approx 4295\text{m}^2 \quad (5.8)$$

Texture filtering is needed to improve the rendering of textures from steep view angles. Figure 5.1 compares different filtering methods for a short curved road. Bilinear and trilinear filtering show visible artifacts on the right line in the distance. Bilinear filtering introduces stairstepping effects. The *no filtering* image shows visible pixel steps even in the near distance because no anti-aliasing takes place. In the far distance, the left line even has gaps despite the line not having any gaps. Only anisotropic filtering shows satisfiable results with minor stairstepping in the far distance.

Third, a SDF model is generated which is appended to main world file. Every model consists of a visual component that declares how the model should look like and a collision component that declares how the model should interact with other objects. In case of the groundplane, the visual and collision component are both defined to be a plane with the normal vector along the z-axis, thus looking up. The plane's size is t/r and its center point can be computed from the tile grid. Additionally, the plane links to its material file which refers to its texture.

```
material Tile/8dc...9df {
    technique {
        pass {
            texture_unit {
                texture tile-8dc...9df.png PF_L8
                filtering anisotropic
                max_anisotropy 16
            }
        }
    }
}
```

Listing 3: Material script for groundplane tile

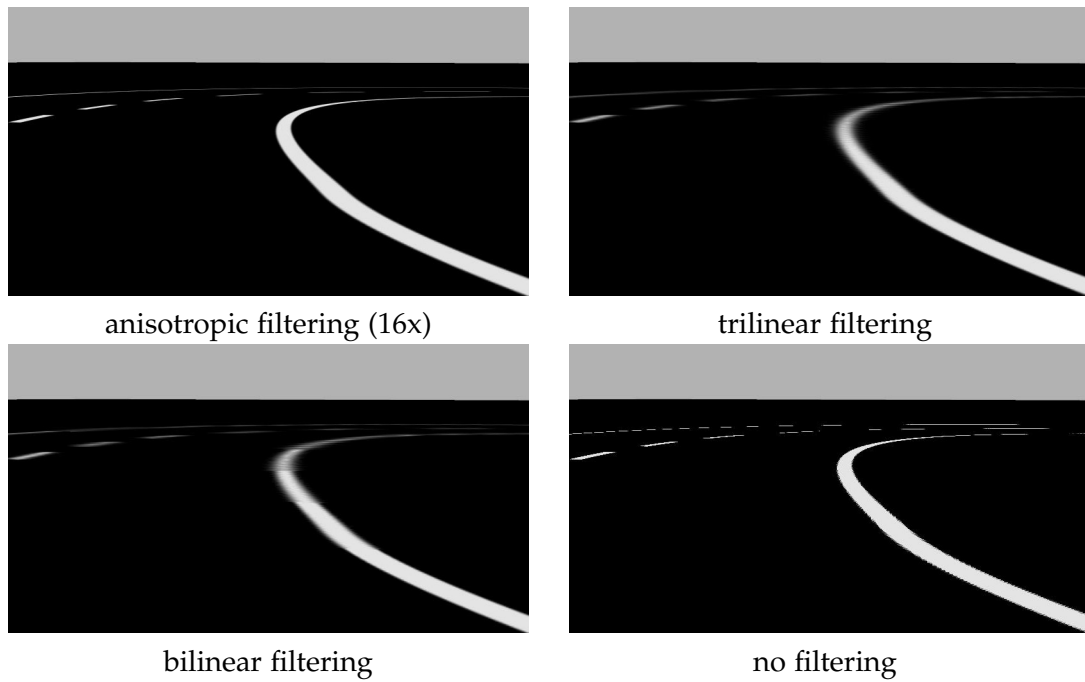


Figure 5.1.: Comparison of different filtering techniques for the groundplane

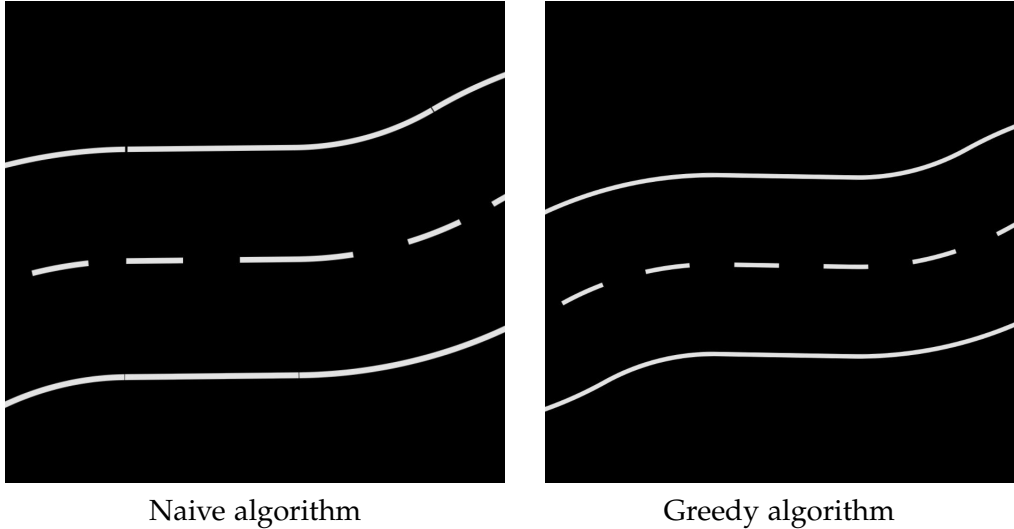


Figure 5.2.: Comparison of different line rendering algorithms

5.2. Line Markings

Polylines can be easily rendered in `cairocff` by first moving to the line's first point with `move_to` and then calling `line_to` for each point. For this section it is assumed that dashes should be $0.2m$ in length. If the line should be dashed, `set_dash([0.2, 0.2])` can be used to set a dash pattern of $20cm$ line, $20cm$ gap, and so on. Now every lanelet could be rendered by drawing both boundaries as a polyline. This algorithm will be called *naive algorithm*. However, this does not give the expected result, as can be seen in figure 5.2. The road was generated from a right arc, followed by a straight line followed by a left arc. Two artifacts are visible in the road rendered by the naive algorithm: Some dashes of the middle lane are longer, some gaps are shorter than others. Second, the left and the right line show small gaps even though there should be one solid line.

The problem of dash lengths originates from primitives with a length which is not a multiply of $0.4m$. In the example from figure 5.2, the straight line has a length of $0.6m$ leading to a dash pattern of (0.2 dash, 0.2 gap, 0.2 dash). The straight line ends with a full dash. The next primitive, the left arc, starts again with a dash, resulting in (0.2 dash, 0.2 gap, ...).

Drawing both primitives leads to (0.2 dash, 0.2 gap, 0.4 dash, 0.2 gap, ...).

Now there is a dash with double the length of other dashes.

The small gaps on the left and right line are the result of floating point inaccuracies when generating the road. The last point of on primitive's boundary may not exactly

match the first point of the succeeding primitive's boundary.

Both artifacts can be removed by using the *greedy algorithm*, that will be explained next. By using the successor and predecessor references from CommonRoad XML, a polyline from one boundary may be expanded to predecessoring or succeeding lanelets if the line marking is the same.

The greedy algorithm (see algorithm 1) takes a set of all lanelets as its input, called *pool* (line 1). The main algorithm is then called for the left and right boundaries separately. As long as there are still lanelets in the pool, a random lanelet is chosen (line 8) and two expansion steps take place, separately for the succeeding and the predecessoring lanelets (lines 9 and 10). In the expansion step, all directly succeeding or predecessoring lanelets (depending on specified *direction*) of the current lanelet are checked (line 22) whether they have the same line marking as the original given lanelet (line 23). If one of them has the same line marking, then it is appended to the local *chain* (line 24) and set it as the new *current* lanelet (line 25). The Append function takes an n -tuple and an arbitrary value as arguments and returns an $(n + 1)$ -tuple with the second argument as the last element in the tuple, like this:

$$\text{Append}((a_1, a_2, \dots, a_n), b) = (a_1, a_2, \dots, a_n, b) \quad (5.9)$$

$$\text{Append}(), b = (b) \quad (5.10)$$

After expansion in both directions, a single, long chain is created from the predecessoring, the chosen and the succeeding lanelets (line 11). It is important to note, that the predecessoring lanelets must be inserted in reversed order into the chain. Then, the left or right boundaries (depending on the *boundary*) of all lanelets in the chain can be drawn as a single polyline (line 12). The line marking is the same for all boundaries to draw due to the way the expansion step works. Last, all drawn lanelets are removed from the *pool* (line 13). The TupleToSet function takes a tuple and converts it to a set, like this:

$$\text{TupleToSet}((a_1, a_2, \dots, a_n)) = \{a_1, a_2, \dots, a_n\} \quad (5.11)$$

Algorithm 1 Greedy algorithm to draw line markings

```

1: procedure GREEDYALGORITHM(pool)
2:   GreedyAlgorithmForBoundary(pool, "leftBoundary")
3:   GreedyAlgorithmForBoundary(pool, "rightBoundary")
4: end procedure
5:
6: procedure GREEDYALGORITHMFORBOUNDARY(pool, boundary)
7:   while pool  $\neq \emptyset$  do
8:     Choose  $x \in \textit{pool}$ 
9:      $(s_1, s_2, \dots, s_n) \leftarrow \text{Expand}(x, \textit{boundary}, \text{"successor"})$ 
10:     $(p_1, p_2, \dots, p_k) \leftarrow \text{Expand}(x, \textit{boundary}, \text{"predecessor"})$ 
11:     $\textit{chain} \leftarrow (p_k, p_{k-1}, \dots, p_2, p_1, x, s_1, s_2, \dots, s_{n-1}, s_n)$ 
12:    DrawBoundary(chain, boundary)  $\triangleright$  Draw the lanelets in order
13:     $\textit{pool} \leftarrow \textit{pool} \setminus \text{TupleToSet}(\textit{chain})$ 
14:   end while
15: end procedure
16:
17: function EXPAND(current, boundary, direction)
18:   marking  $\leftarrow \text{GetLineMarking}(\textit{current}, \textit{boundary})$ 
19:   chain  $\leftarrow ()$   $\triangleright$  Initialize with empty tuple
20:   repeat
21:     found  $\leftarrow \text{false}$ 
22:     for next  $\in \text{GetNextInDirection}(\textit{current}, \textit{direction})$  do
23:       if  $\text{GetLineMarking}(\textit{next}, \textit{boundary}) = \textit{marking}$  then
24:          $\textit{chain} \leftarrow \text{Append}(\textit{chain}, \textit{next})$ 
25:          $\textit{current} \leftarrow \textit{next}$ 
26:          $\textit{found} \leftarrow \text{true}$ 
27:       end if
28:     end for
29:   until  $\neg \textit{found}$ 
30:   return chain
31: end function

```

5.3. Traffic Signs

Traffic signs can be rendered using a model with a plane primitive. The plane's size is determined from a look-up table based on its type. The plane is rolled by $\pi/2$ to make it upright and yawed by its orientation angle as defined in CommonRoad XML. The plane's (x, y) coordinate can be taken from its `<centerPoint>` and the z-coordinate is $0.15 + h/2$ with h as the plane height. The model is defined without collision component since a vehicle colliding with a traffic sign is no additional rule violation after leaving the track. For each type of traffic sign, a material file is generated as shown in listing 4. The texture images for all needed traffic signs are copied to the `materials/scripts` folder. Since most traffic signs are no perfect rectangles, the images should be created as PNG files to use the alpha channel for transparent regions. OGRE then loads the images as alpha-enabled textures but needs the `scene_blend` and `depth_write` options to correctly render them as shown in figure 5.4.

```
material Sign/stvo-625-10 {
    technique {
        pass {
            scene_blend alpha_blend
            depth_write off

            texture_unit {
                texture stvo-625-10.png
                filtering trilinear
            }
        }
    }
}
```

Listing 4: Material script for a traffic sign

5.4. Obstacle

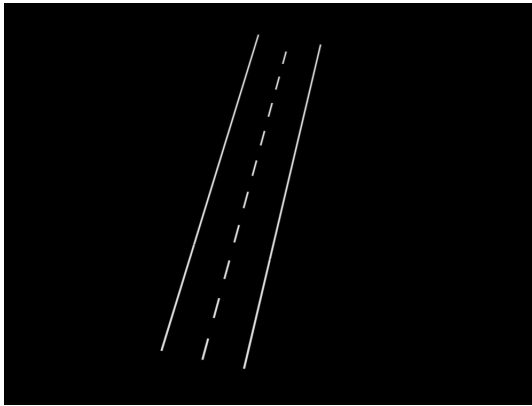
For each obstacle a model is created consisting of a box primitive for rectangular shaped obstacles. The model defines both the visual and the collision component to be the same box. The box' center point, width and length is equal to the rectangle definition in CommonRoad XML. Its height is a constant 0.2 as defined in the Carolo-Cup rules.

The yaw angle is derived from the orientation. The box uses the built-in `Gzbeo/White` material to get a simple white paint.

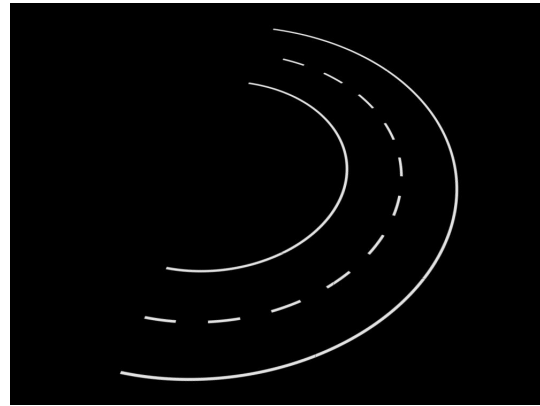
5.5. Ego Vehicle

The ego vehicle is similar to the static obstacle but uses a custom mesh based on the CAD model of TUM Phoenix Carolo-Cup 2017 car. The model was aligned along the x-axis and scaled to match world coordinates. Furthermore, the model's mesh complexity was reduced to save memory and shorten load times using MeshLab.

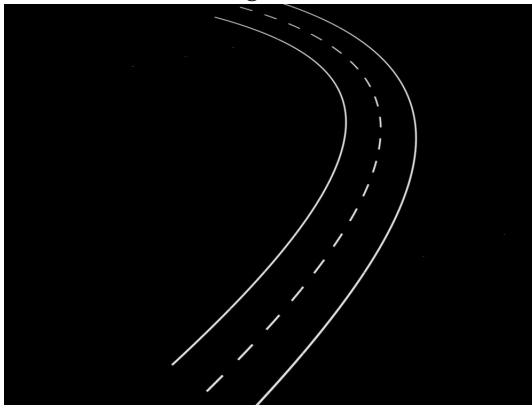
The model includes two sensor, a camera and a laser scanner that resemble the position in the original car.



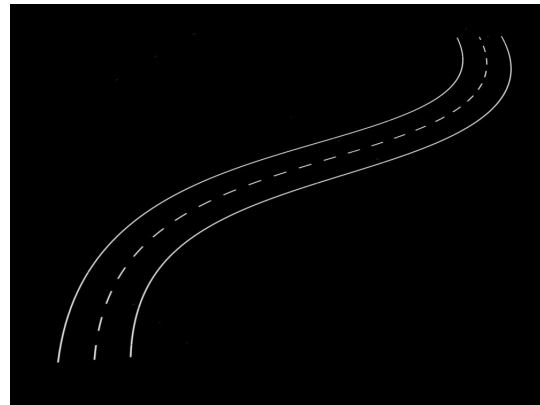
Straight road



Circular arc



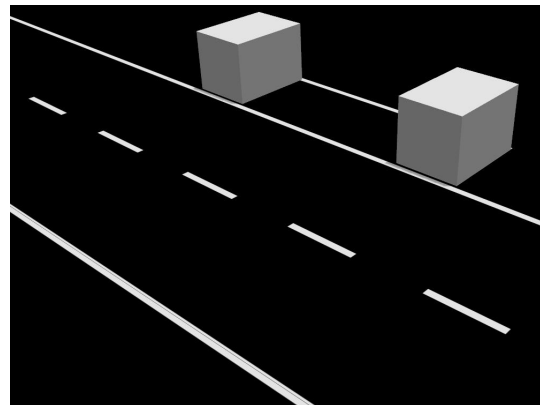
Quadratic Bézier curve



Cubic Bézier curve



Zebra crossing

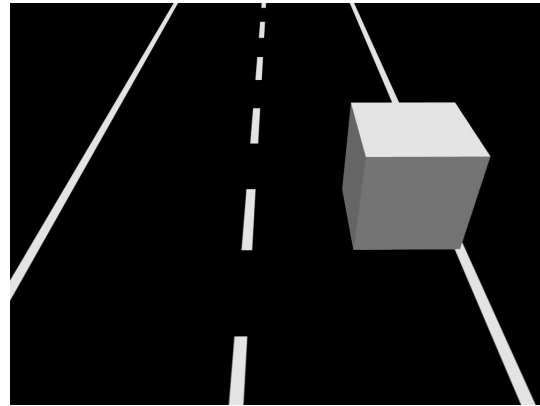


Parking lot

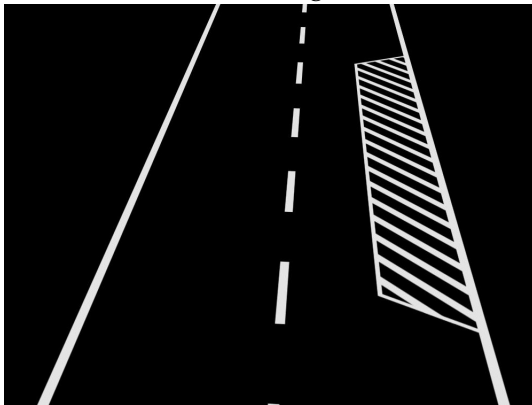
Figure 5.3.: Examples for different objects rendered in Gazebo (part 1)



Traffic sign



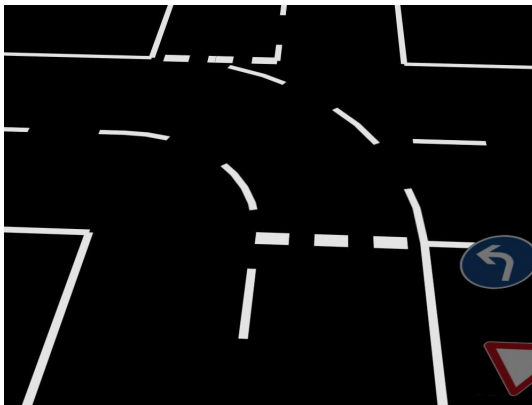
Static obstacle



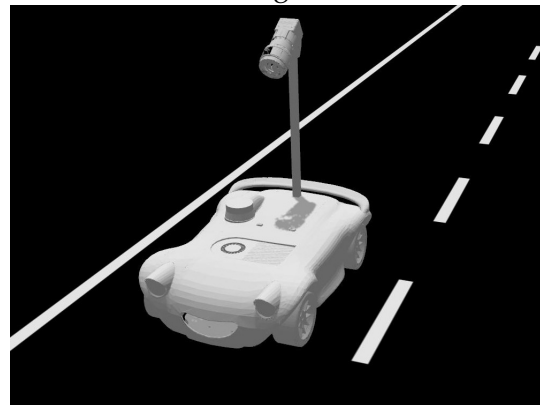
Blocked area



Missing lines



Intersection with turn lane



Ego vehicle

Figure 5.4.: Examples for different objects rendered in Gazebo (part 2)

6. Results

To show the capabilities of the road generation and visualization, three different challenges were created in *Template XML*. Each template was then evaluated several times to create a set of *CommonRoad XML* files. Each XML file was then rendered to a *Gazebo SDF* and a screenshots of the visualization were taken. All screenshots are shown with inverted groundplane colors to save ink in the printed version of this thesis.

6.1. Driving without obstacles

Listing 5 shows the template that was used to generate different *driving without obstacles* scenarios shown in figure 6.1. All roads begin with a straight road of 1m. Then, five to twenty elements may appear. This can be either a left arc, straight road or right arc. For the arcs, two different radien are used, 1.4m and 1.6m. To generate straight roads with different lengths and arcs with different opening angles, the `<repeat>` structure was used. The third case's weight was set to 2 to make straight roads equally likely to be generated as left arcs and right arcs.

6.2. Driving in extended mode

Listing 6 shows the template that was used to generate the scenarios shown in figures 6.2 and 6.3. All traffic signs were rotated and up-scaled to be visible from above. The template generates two to three features. A feature can be an urban area, a set of static obstacles, a curve with curve signs or an intersection. The urban area always begin with a zone 30 and can contain a zebra crossing or a blocked area. There can be up to five obstacles in a feature. Between two features is either a straight road or an arc of 45° .

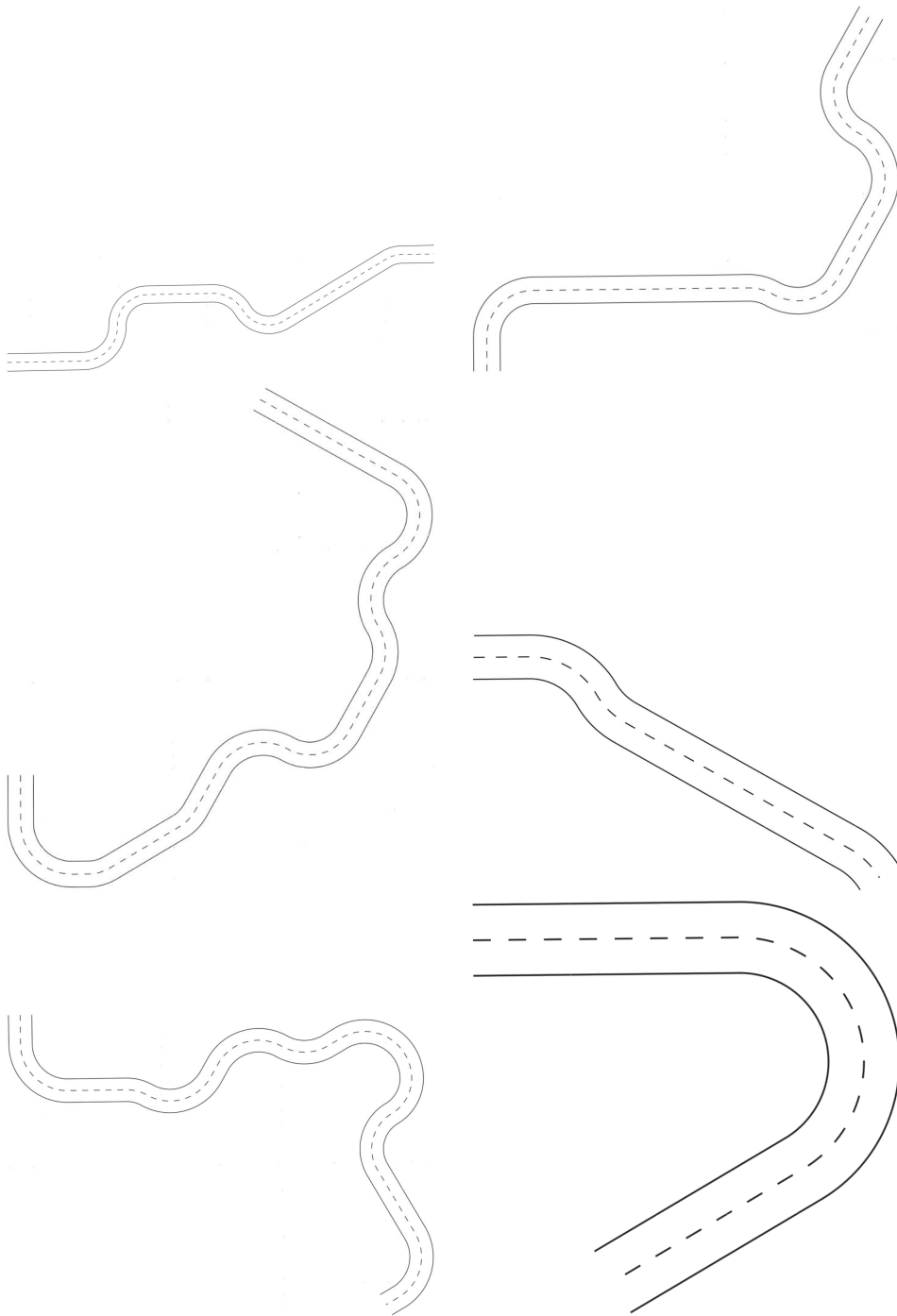


Figure 6.1.: Screenshots of six generated roads for driving without obstacles

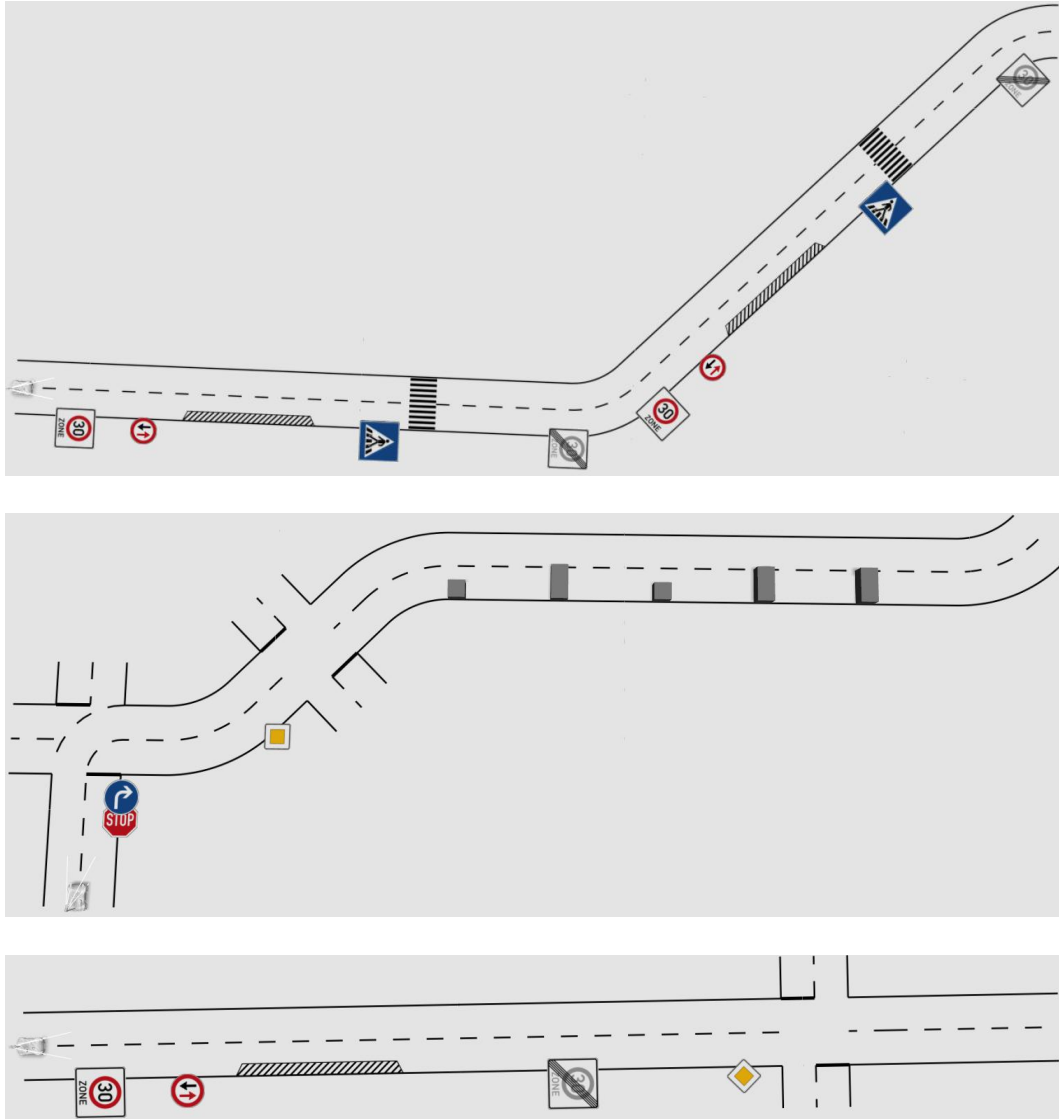


Figure 6.2.: Screenshots of six generated roads for driving in extended mode (part 1)

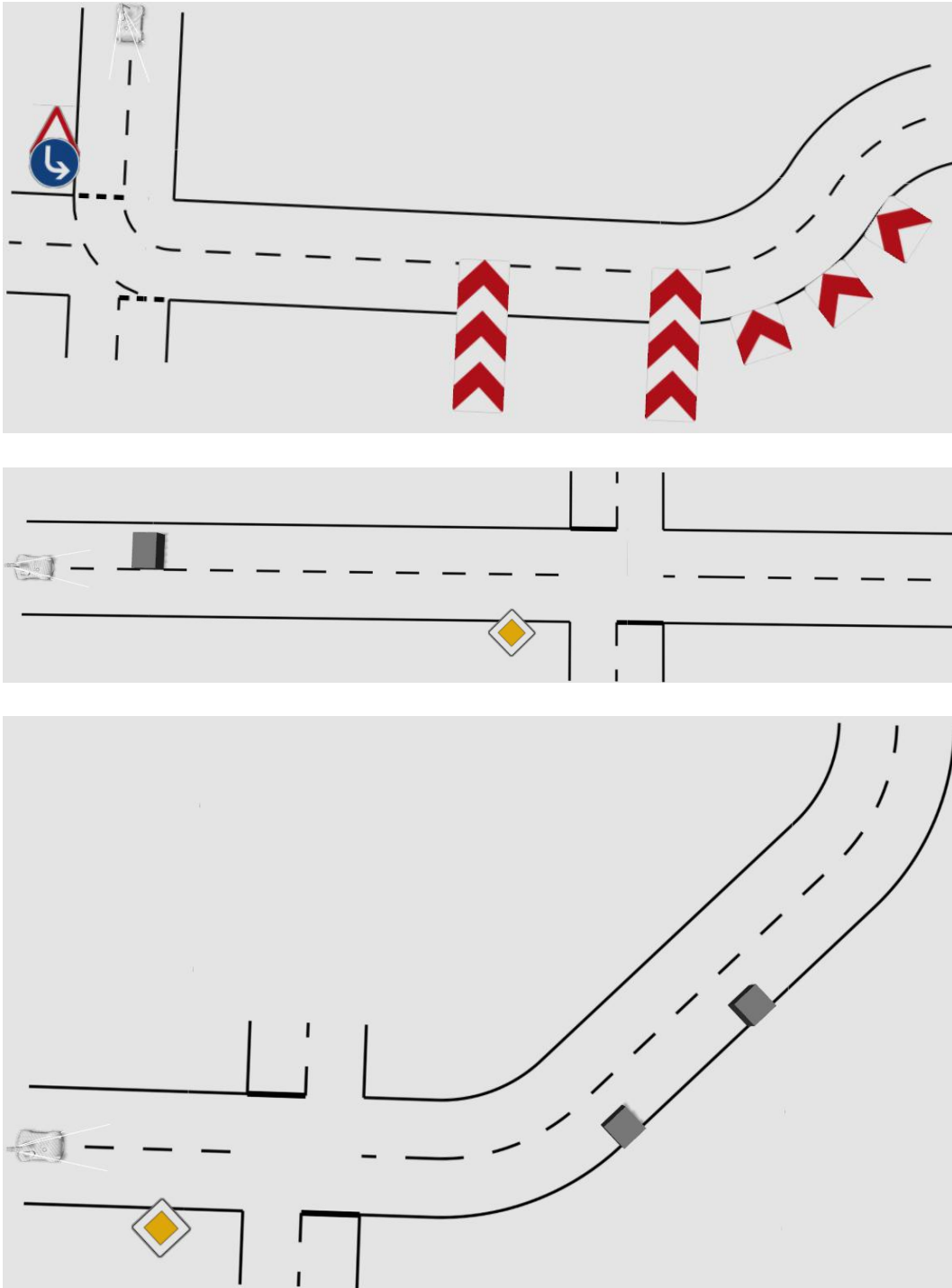


Figure 6.3.: Screenshots of six generated roads for driving in extended mode (part 2)

```
<template>
  <sequence>
    <line length="1" />
    <repeat min="5" max="20">
      <select>
        <case w="1">
          <repeat min="1" max="3">
            <leftArc radius="1.4" angle="30" />
          </repeat>
        </case>
        <case w="1">
          <repeat min="1" max="3">
            <leftArc radius="1.6" angle="30" />
          </repeat>
        </case>
        <case w="2">
          <repeat min="1" max="5">
            <line length="0.5" />
          </repeat>
        </case>
        <case w="1">
          <repeat min="1" max="3">
            <rightArc radius="1.4" angle="30" />
          </repeat>
        </case>
        <case w="1">
          <repeat min="1" max="3">
            <rightArc radius="1.6" angle="30" />
          </repeat>
        </case>
      </select>
    </repeat>
  </sequence>
</template>
```

Listing 5: Template for driving without obstacles

```
<template>
  <sequence>
    <line length="1" />
    <repeat min="2" max="3">
      <select>
        <case w="1">
          <!-- urban area -->
          <trafficSign type="stvo-274.1" />
          <line length="1" />
          <repeat min="1" max="3">
            <select>
              <case w="1">
                <trafficSign type="stvo-350-10" />
                <line length="0.4" />
                <zebraCrossing length="0.4" />
              </case>
              <case w="1">
                <trafficSign type="stvo-208" />
                <line length="0.5" />
                <blockedArea width="0.15" length="2" />
              </case>
            </select>
            <line length="1" />
          </repeat>
          <line length="1" />
          <trafficSign type="stvo-274.2" />
        </case>
        <case w="1">
          <repeat min="1" max="5">
            <!-- static obstacles -->
          </repeat>
        </case>
        <!-- ... -->
      </select>
      <!-- ... -->
    </repeat>
  </sequence>
</template>
```

Listing 6: Template for driving in extended mode

6.3. Parallel Parking

Using the template in listing 7, six different parking scenarios were generated, as shown in figure 6.4. Using the `<shuffle>` structure, the three parking lots of 55, 63 and 70cm and the obstacles are placed in an arbitrary order. A `<repeat>` structure is used to generate a series of obstacles with gaps up to 40cm between them.

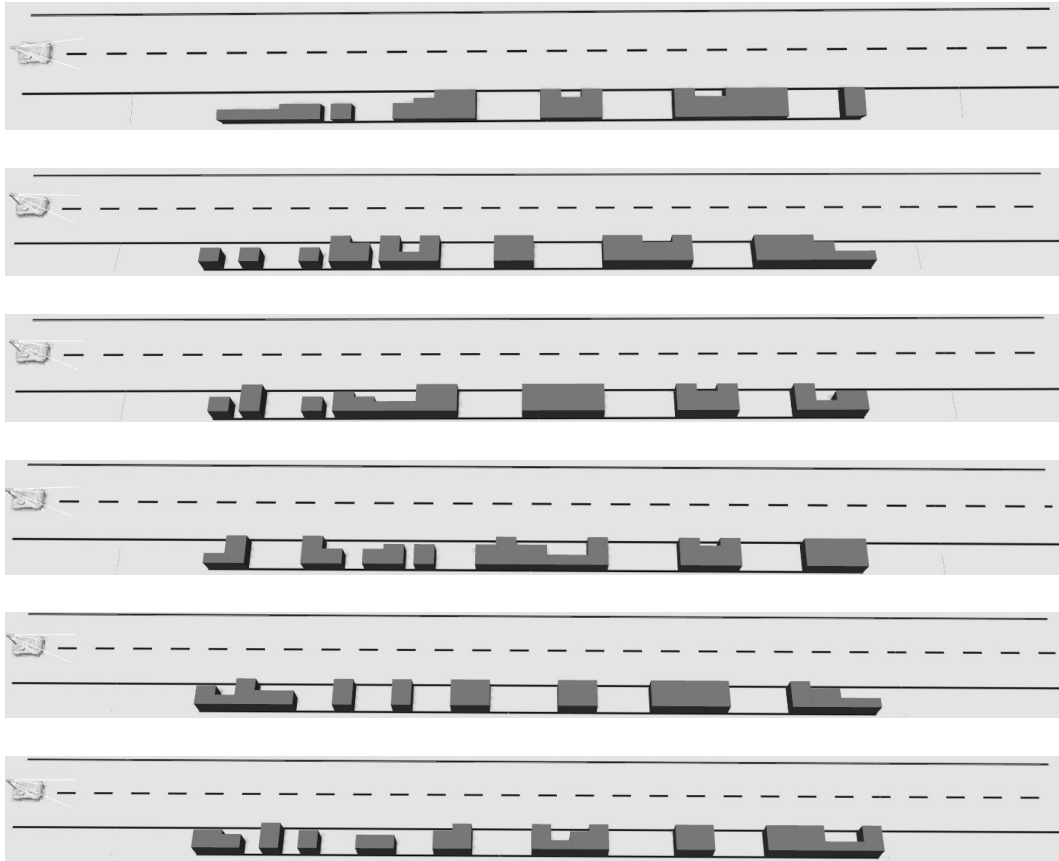


Figure 6.4.: Screenshots of six generated parallel parking scenarios


```
<template><sequence>
  <line length="2" />
  <shuffle>
    <sequence>
      <parkingObstacle width="0.28" length="0.2" />
      <parkingLot length="0.55" />
      <parkingObstacle width="0.28" length="0.2" />
    </sequence>
    <sequence>
      <parkingObstacle width="0.28" length="0.2" />
      <parkingLot length="0.63" />
      <parkingObstacle width="0.28" length="0.2" />
    </sequence>
    <sequence>
      <parkingObstacle width="0.28" length="0.2" />
      <parkingLot length="0.70" />
      <parkingObstacle width="0.28" length="0.2" />
    </sequence>
    <!-- more obstacles ... -->
    <sequence>
      <parkingObstacle width="0.15" length="0.2" />
      <repeat min="3" max="5">
        <optional p="0.8">
          <select>
            <case w="1"><parkingLot length="0.40" /></case>
            <!-- other gap sizes -->
          </select>
        </optional>
        <select>
          <case w="1"><parkingObstacle width="0.28"
            length="0.2" /></case>
          <!-- other parking obstacle sizes -->
        </select>
      </repeat>
    </sequence>
  </shuffle>
  <line length="2" />
</sequence></template>
```

Listing 7: Template for parallel parking

7. Future work

The concepts and implementations that were presented in this thesis allow to generate and render most traffic scenarios that can be found in the Carolo-Cup. However, not all subtleties were implemented and some parts could be optimized to be faster and to use less memory. Some of those ideas are explained in the next sections.

7.1. Road Generation

The current road generation algorithm is only able to generate a single, long road. That means, the ego vehicle will usually enter and exit every primitive of the road network only once. Though, for some test scenarios it might be useful to build round trips or roads that intersect with previously used roads. To close the loop between two ends of a road, a special primitive must be constructed.

Additional primitives are needed to also generate perpendicular parking lots on the left side of the road. However, with the current implementation there is no easy way to allow parallel and perpendicular parking lots at opposing sides of the road.

As an extension to zebra crossing, pedestrians could be generated. In the Carolo-Cup, the pedestrians are small white boxes with a stick figure drawn on to them. A pedestrian waits at the zebra crossing until the ego vehicle stops. He will then cross the road, clearing the way for the ego vehicle after him.

7.2. Visualization

Rendering of the groundplane could be made faster by using an R-Tree or Quadtree. For every object that is rendered on the groundplane, like lanelets and blocked areas, a bounding box must be computed. The object is then inserted into the tree structure. During rendering of a tile, objects that intersect with that tile can be queried performatly. Only intersecting objects should be drawn.

As an alternative to the tile-based groundplane, one could also generate a large, flat mesh. All lines and rectangles that would be usually drawn on the groundplane are then approximated using triangles. No textures are needed, instead every triangle has

a fixed color, either black or white. This approach may reduce GPU memory usage but is complex to implement.

7.3. Simulation

Using a static trajectory for dynamic obstacles is fine for small test scenarios but does not scale for larger ones. If the ego vehicle is really slow, some dynamic obstacles may already crossed an intersection when it arrives. To solve this problem, *trigger zones* could be used. If the ego vehicle enters a *zone*, a dynamic obstacle is *triggered* to start moving on its trajectory. A trigger zone can have an arbitrary shape and is usually placed on a lane. This is also useful for pedestrians waiting at zebra crossings.

For unit-testing, a method of validation is needed. This validator must be able to detect rule infringements and confirm successful tests. Infringements may be leaving the lane, colliding with an obstacle, turning wrongly at an intersection, ignoring priorities of dynamic obstacles at intersections and so on. Some of those infringements may just be a warning, others may fail the test case immediately. A test is successful, when the ego vehicle enters a specific goal region or holds at a parking lot without making any errors.

A. XML Schema

A.1. Template XML Schema

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:simpleType name="lineType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="solid" />
      <xs:enumeration value="dashed" />
      <xs:enumeration value="missing" />
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="anchorPosition">
    <xs:restriction base="xs:string">
      <xs:enumeration value="left" />
      <xs:enumeration value="center" />
      <xs:enumeration value="right" />
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="trafficSignType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="stvo-306" />
      <xs:enumeration value="stvo-205" />
      <xs:enumeration value="stvo-206" />
      <xs:enumeration value="stvo-208" />
      <xs:enumeration value="stvo-276" />
      <xs:enumeration value="stvo-280" />
      <xs:enumeration value="stvo-274.1" />
      <xs:enumeration value="stvo-274.2" />
      <xs:enumeration value="stvo-350-10" />
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

```
<xs:enumeration value="stvo-209-10" />
<xs:enumeration value="stvo-209-20" />
<xs:enumeration value="stvo-625-10" />
<xs:enumeration value="stvo-625-11" />
<xs:enumeration value="stvo-625-20" />
<xs:enumeration value="stvo-625-21" />
</xs:restriction>
</xs:simpleType>

<xs:complexType name="line">
  <xs:sequence />
  <xs:attribute name="length" type="xs:float" use="required" />
  <xs:attribute name="leftLine" type="lineType" use="optional"
    ↪ default="solid" />
  <xs:attribute name="middleLine" type="lineType" use="optional"
    ↪ default="dashed" />
  <xs:attribute name="rightLine" type="lineType" use="optional"
    ↪ default="solid" />
</xs:complexType>

<xs:complexType name="leftArc">
  <xs:sequence />
  <xs:attribute name="radius" type="xs:float" use="required" />
  <xs:attribute name="angle" type="xs:float" use="required" />
  <xs:attribute name="leftLine" type="lineType" use="optional"
    ↪ default="solid" />
  <xs:attribute name="middleLine" type="lineType" use="optional"
    ↪ default="dashed" />
  <xs:attribute name="rightLine" type="lineType" use="optional"
    ↪ default="solid" />
</xs:complexType>

<xs:complexType name="rightArc">
  <xs:sequence />
  <xs:attribute name="radius" type="xs:float" use="required" />
  <xs:attribute name="angle" type="xs:float" use="required" />
  <xs:attribute name="leftLine" type="lineType" use="optional"
    ↪ default="solid" />
```

```
<xs:attribute name="middleLine" type="lineType" use="optional"
  ↪ default="dashed" />
<xs:attribute name="rightLine" type="lineType" use="optional"
  ↪ default="solid" />
</xs:complexType>

<xs:complexType name="quadBezier">
  <xs:sequence />
  <xs:attribute name="p1x" type="xs:float" use="required" />
  <xs:attribute name="p1y" type="xs:float" use="required" />
  <xs:attribute name="p2x" type="xs:float" use="required" />
  <xs:attribute name="p2y" type="xs:float" use="required" />
  <xs:attribute name="leftLine" type="lineType" use="optional"
    ↪ default="solid" />
  <xs:attribute name="middleLine" type="lineType" use="optional"
    ↪ default="dashed" />
  <xs:attribute name="rightLine" type="lineType" use="optional"
    ↪ default="solid" />
</xs:complexType>

<xs:complexType name="cubicBezier">
  <xs:complexContent>
    <xs:extension base="quadBezier">
      <xs:attribute name="p3x" type="xs:float" use="required" />
      <xs:attribute name="p3y" type="xs:float" use="required" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="staticObstacle">
  <xs:attribute name="width" type="xs:float" use="required" />
  <xs:attribute name="length" type="xs:float" use="required" />
  <xs:attribute name="position" type="xs:float" use="required" />
  <xs:attribute name="anchor" type="anchorPosition" use="optional"
    ↪ default="center" />
</xs:complexType>

<xs:complexType name="zebraCrossing">
  <xs:attribute name="length" type="xs:float" use="required" />
```

```
</xs:complexType>

<xs:complexType name="blockedArea">
  <xs:attribute name="length" type="xs:float" use="required" />
  <xs:attribute name="width" type="xs:float" use="required" />
</xs:complexType>

<xs:complexType name="trafficSign">
  <xs:attribute name="type" type="trafficSignType" use="required" />
</xs:complexType>

<xs:complexType name="intersection">
  <xs:attribute name="rule" use="required">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="priority-yield" />
        <xs:enumeration value="priority-stop" />
        <xs:enumeration value="yield" />
        <xs:enumeration value="stop" />
        <xs:enumeration value="equal" />
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="turn">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="straight" />
        <xs:enumeration value="left" />
        <xs:enumeration value="right" />
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
</xs:complexType>

<xs:complexType name="parkingLot">
  <xs:attribute name="length" type="xs:float" use="required" />
</xs:complexType>

<xs:complexType name="parkingObstacle">
```

```
<xs:attribute name="length" type="xs:float" use="required" />
<xs:attribute name="width" type="xs:float" use="required" />
</xs:complexType>

<xs:complexType name="sequence">
  <xs:complexContent>
    <xs:extension base="anyUnbounded">
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

<xs:complexType name="optional">
  <xs:complexContent>
    <xs:extension base="anyUnbounded">
      <xs:attribute name="p" type="xs:float" use="required" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="select">
  <xs:sequence>
    <xs:element name="case" maxOccurs="unbounded">
      <xs:complexType>
        <xs:complexContent>
          <xs:extension base="anyUnbounded">
            <xs:attribute name="w" type="xs:float" use="
              ↪ required" />
          </xs:extension>
        </xs:complexContent>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="repeat">
  <xs:complexContent>
    <xs:extension base="anyUnbounded">
      <xs:attribute name="n" type="xs:integer" use="optional" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```



```
<xs:attribute name="min" type="xs:integer" use="optional"
  ↪ />
<xs:attribute name="max" type="xs:integer" use="optional"
  ↪ />
</xs:extension>
</xs:complexContent>
</xs:complexType>

<xs:complexType name="shuffle">
  <xs:complexContent>
    <xs:extension base="anyUnbounded">
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

<xs:complexType name="anyUnbounded">
  <xs:choice maxOccurs="unbounded">
    <xs:element name="line" type="line" maxOccurs="unbounded"/>
    <xs:element name="leftArc" type="leftArc" maxOccurs="unbounded"
      ↪ "/>
    <xs:element name="rightArc" type="rightArc" maxOccurs="
      ↪ unbounded"/>
    <xs:element name="quadBezier" type="quadBezier" maxOccurs="
      ↪ unbounded"/>
    <xs:element name="cubicBezier" type="cubicBezier" maxOccurs="
      ↪ unbounded"/>
    <xs:element name="zebraCrossing" type="zebraCrossing" maxOccurs
      ↪ ="unbounded"/>
    <xs:element name="blockedArea" type="blockedArea" maxOccurs="
      ↪ unbounded"/>
    <xs:element name="trafficSign" type="trafficSign" maxOccurs="
      ↪ unbounded"/>
    <xs:element name="intersection" type="intersection" maxOccurs="
      ↪ unbounded"/>
    <xs:element name="staticObstacle" type="staticObstacle"
      ↪ maxOccurs="unbounded"/>
    <xs:element name="parkingLot" type="parkingLot" maxOccurs="
      ↪ unbounded"/>
```

```
<xs:element name="parkingObstacle" type="parkingObstacle"
  ↪ maxOccurs="unbounded"/>
<xs:element name="sequence" type="sequence" maxOccurs="
  ↪ unbounded"/>
<xs:element name="optional" type="optional" maxOccurs="
  ↪ unbounded"/>
<xs:element name="select" type="select" maxOccurs="unbounded"/>
<xs:element name="repeat" type="repeat" maxOccurs="unbounded"/>
<xs:element name="shuffle" type="shuffle" maxOccurs="unbounded
  ↪ "/>
</xs:choice>
</xs:complexType>

<xs:element name="template">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="sequence" type="sequence" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

A.2. CommonRoad XML Schema

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:simpleType name="distance">
    <xs:restriction base="xs:float">
      <xs:minExclusive value="0.0"></xs:minExclusive>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="floatInterval">
    <xs:choice>
      <xs:element name="exact" type="xs:float" />
      <xs:sequence>
        <xs:element name="intervalStart" type="xs:float" />
        <xs:element name="intervalEnd" type="xs:float" />
      </xs:sequence>
    </xs:choice>
  </xs:complexType>
</xs:schema>
```

```

    </xs:choice>
</xs:complexType>

<xs:complexType name="point">
  <xs:sequence>
    <xs:element name="x" type="xs:float" />
    <xs:element name="y" type="xs:float" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="rectangle">
  <xs:sequence>
    <xs:element name="length" type="distance" />
    <xs:element name="width" type="distance" />
    <xs:element name="orientation" type="xs:float" />
    <xs:element name="centerPoint" type="point" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="circle">
  <xs:sequence>
    <xs:element name="radius" type="distance" />
    <xs:element name="centerPoint" type="point" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="polygon">
  <xs:sequence>
    <xs:element minOccurs="3" maxOccurs="unbounded" name="point"
      type="point" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="shape">
  <xs:choice maxOccurs="unbounded">
    <xs:element name="rectangle" type="rectangle" />
    <xs:element name="circle" type="circle" />
    <xs:element name="polygon" type="polygon" />
  </xs:choice>

```

```
</xs:complexType>

<xs:complexType name="occupancy">
  <xs:sequence>
    <xs:element name="shape" type="shape" />
    <xs:element name="time" type="floatInterval" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="state">
  <xs:sequence>
    <xs:element name="position">
      <xs:complexType>
        <xs:choice>
          <xs:element name="point" type="point" />
          <xs:choice maxOccurs="unbounded">
            <xs:element name="rectangle" type="rectangle" />
            <xs:element name="circle" type="circle" />
            <xs:element name="polygon" type="polygon" />
          </xs:choice>
        </xs:choice>
      </xs:complexType>
    </xs:element>
    <xs:element name="orientation" type="floatInterval" />
    <xs:element name="time" type="floatInterval" />
    <xs:element name="velocity" type="floatInterval" minOccurs="0"
      ↪ />
    <xs:element name="acceleration" type="floatInterval" minOccurs
      ↪ ="0" />
  </xs:sequence>
</xs:complexType>

<xs:simpleType name="obstacleRole">
  <xs:restriction base="xs:string">
    <xs:enumeration value="static" />
    <xs:enumeration value="dynamic" />
  </xs:restriction>
</xs:simpleType>
```



```
        </xs:complexType>
      </xs:element>
    </xs:choice>
  </xs:sequence>
  <xs:attribute name="id" type="xs:integer" use="required" />
</xs:complexType>

<xs:simpleType name="lineMarking">
  <xs:restriction base="xs:string">
    <xs:enumeration value="dashed" />
    <xs:enumeration value="solid" />
  </xs:restriction>
</xs:simpleType>

<xs:complexType name="boundary">
  <xs:sequence>
    <xs:element name="point" type="point"
      maxOccurs="unbounded" />
    <xs:element name="lineMarking" type="lineMarking"
      minOccurs="0" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="laneletRef">
  <xs:attribute name="ref" type="xs:integer"
    use="required"/>
</xs:complexType>

<xs:complexType name="laneletRefList">
  <xs:sequence>
    <xs:element name="lanelet" type="laneletRef" minOccurs="0"
      maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>

<xs:simpleType name="drivingDir">
  <xs:restriction base="xs:string">
    <xs:enumeration value="same" />
    <xs:enumeration value="opposite" />
  </xs:restriction>
</xs:simpleType>
```

```
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="laneletAdjacentRef">
    <xs:attribute name="ref" type="xs:integer" use="required" />
    <xs:attribute name="drivingDir" type="drivingDir"
      use="required" />
  </xs:complexType>

  <xs:simpleType name="laneletType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="road" />
      <xs:enumeration value="sidewalk" />
      <xs:enumeration value="zebraCrossing" />
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="lanelet">
    <xs:sequence>
      <xs:element name="type" type="laneletType" minOccurs="0"
        ↪ default="road" />
      <xs:element name="leftBoundary" type="boundary" />
      <xs:element name="rightBoundary" type="boundary" />
      <xs:element name="predecessor" type="laneletRefList"
        minOccurs="0" />
      <xs:element name="successor" type="laneletRefList"
        minOccurs="0" />
      <xs:element name="adjacentLeft" type="laneletAdjacentRef"
        minOccurs="0" />
      <xs:element name="adjacentRight" type="laneletAdjacentRef"
        minOccurs="0" />
      <!-- EXTENSION -->
      <xs:element name="stopLine" type="lineMarking" minOccurs="0" />
    </xs:sequence>
    <xs:attribute name="id" type="xs:integer" use="required" />
  </xs:complexType>

  <xs:complexType name="egoVehicle">
    <xs:sequence>
```

```
<xs:element name="type" type="obstacleType" />
<xs:element name="shape" type="shape" />
<xs:element name="initialState" type="state" />
<xs:element name="goalRegion">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="state" type="state"
        maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:sequence>
<xs:attribute name="id" type="xs:integer" use="required" />
</xs:complexType>

<!-- EXTENSION -->
<xs:complexType name="parkingLot">
  <xs:sequence>
    <xs:element name="shape" type="shape" />
  </xs:sequence>
  <xs:attribute name="id" type="xs:integer" use="required" />
</xs:complexType>

<!-- EXTENSION -->
<xs:complexType name="trafficSign">
  <xs:sequence>
    <xs:element name="type">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="stvo-306" />
          <xs:enumeration value="stvo-205" />
          <xs:enumeration value="stvo-206" />
          <xs:enumeration value="stvo-208" />
          <xs:enumeration value="stvo-276" />
          <xs:enumeration value="stvo-280" />
          <xs:enumeration value="stvo-274.1" />
          <xs:enumeration value="stvo-274.2" />
          <xs:enumeration value="stvo-350-10" />
          <xs:enumeration value="stvo-209-10" />
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```



```
        <xs:enumeration value="stvo-209-20" />
        <xs:enumeration value="stvo-625-10" />
        <xs:enumeration value="stvo-625-11" />
        <xs:enumeration value="stvo-625-20" />
        <xs:enumeration value="stvo-625-21" />
    </xs:restriction>
</xs:simpleType>
</xs:element>
<xs:element name="orientation" type="xs:float" />
<xs:element name="centerPoint" type="point" />
</xs:sequence>
<xs:attribute name="id" type="xs:integer" use="required" />
</xs:complexType>

<!-- EXTENSION -->
<xs:complexType name="intersection">
    <xs:sequence>
        <xs:element name="composition" type="laneletRefList" />
        <xs:element name="priority" maxOccurs="unbounded">
            <xs:complexType>
                <xs:attribute name="low" type="xs:integer" use="
                    ↪ required" />
                <xs:attribute name="high" type="xs:integer" use="
                    ↪ required" />
            </xs:complexType>
        </xs:element>
    </xs:sequence>
    <xs:attribute name="id" type="xs:integer" use="required" />
</xs:complexType>

<xs:complexType name="CommonRoad">
    <xs:choice maxOccurs="unbounded">
        <xs:element name="obstacle" type="obstacle"
            maxOccurs="unbounded" />
        <xs:element name="lanelet" type="lanelet"
            maxOccurs="unbounded" />
        <xs:element name="egoVehicle" type="egoVehicle"
            maxOccurs="unbounded" />
        <xs:element name="parkingLot" type="parkingLot" />
    </xs:choice>
</xs:complexType>
```

```
        maxOccurs="unbounded" />
    <xs:element name="trafficSign" type="trafficSign"
        maxOccurs="unbounded" />
    <xs:element name="intersection" type="intersection"
        maxOccurs="unbounded" />
</xs:choice>
<xs:attribute name="commonRoadVersion" use="required">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:enumeration value="1.0" />
        </xs:restriction>
    </xs:simpleType>
</xs:attribute>
<xs:attribute name="benchmarkID" type="xs:string" />
<xs:attribute name="date" type="xs:date" />
<xs:attribute name="timeStepSize" type="xs:float" />
</xs:complexType>

<!-- Root element -->
<xs:element name="commonRoad" type="CommonRoad">
    <xs:key name="id">
        <xs:selector xpath="/*" />
        <xs:field xpath="@id" />
    </xs:key>
    <xs:keyref name="idref" refer="id">
        <xs:selector xpath="/*" />
        <xs:field xpath="@ref" />
    </xs:keyref>
    <xs:keyref name="idhigh" refer="id">
        <xs:selector xpath="./intersection/priority" />
        <xs:field xpath="@high" />
    </xs:keyref>
    <xs:keyref name="idlow" refer="id">
        <xs:selector xpath="./intersection/priority" />
        <xs:field xpath="@low" />
    </xs:keyref>
</xs:element>
</xs:schema>
```

List of Figures

1.1. Vehicle of TUM Phoenix Robotics at Carolo-Cup 2017	1
1.2. Testing of vehicle software on the real track	2
1.3. Overview over this thesis	3
3.1. Overview of all primitives	6
3.2. Schematic presentation of the straight road primitive	7
3.3. Schematic presentation of the left arc primitive	8
3.4. Schematic presentation of the quadratic bézier primitive	10
3.5. Schematic presentation of the cubic bézier primitive	12
3.6. Schematic presentation of the static obstacle primitive	13
3.7. Schematic presentation of an intersection with turn lane	14
3.8. Schematic presentation of the zebra crossing primitive	15
3.9. Schematic presentation of the blocked area primitive	16
3.10. Schematic presentation of the parking lot primitive	17
3.11. Schematic presentation of the parking obstacle primitive	17
3.12. Concatenation of primitives	22
4.1. Coordinate system and orientation α	23
4.2. Lanelets on intersection with priority and yield lanes	26
5.1. Comparison of different filtering techniques for the groundplane	35
5.2. Comparison of different line rendering algorithms	36
5.3. Examples for different objects rendered in Gazebo (part 1)	41
5.4. Examples for different objects rendered in Gazebo (part 2)	42
6.1. Screenshots of six generated roads for driving without obstacles	44
6.2. Screenshots of six generated roads for driving in extended mode (part 1)	45
6.3. Screenshots of six generated roads for driving in extended mode (part 2)	46
6.4. Screenshots of six generated parallel parking scenarios	49

List of Tables

3.1. Definition of the <line> element	7
3.2. Definition of the <leftArc>/<rightArc> element	7
3.3. Definition of the <quadBezier> element	9
3.4. Definition of the <cubicBezier> element	11
3.5. Definition of the <staticObstacle> element	12
3.6. Definition of the <intersection> element	13
3.7. Definition of the <zebraCrossing> element	15
3.8. Definition of the <blockedArea> element	15
3.9. Definition of the <trafficSign> element	16
3.10. Definition of the <parkingLot> element	16
3.11. Definition of the <parkingObstacle> element	17
4.1. Priorities for vehicles coming from the priority lane (left) and the yield lane (right)	27
4.2. Traffic signs	30

Bibliography

- [AKM17] M. Althoff, M. Koschi, and S. Manzingner. “CommonRoad: Composable Benchmarks for Motion Planning on Roads.” In: *Proc. of the IEEE Intelligent Vehicles Symposium*. 2017.
- [AM16] M. Althoff and S. Magdici. “Set-Based Prediction of Traffic Participants on Arbitrary Road Networks.” In: *IEEE Transactions on Intelligent Vehicles* 1.2 (2016), pp. 187–202.
- [Bra+14] G. Brambilla, A. Grazioli, M. Picone, F. Zanichelli, and M. Amoretti. “A cost-effective approach to software-in-the-loop simulation of pervasive systems and applications.” In: *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2014 IEEE International Conference on*. IEEE. 2014, pp. 207–210.
- [Bra16] T. Braunschweig. *Carolo-Cup Regulations 2017*. https://wiki.ifr.ing.tu-bs.de/carolocup/system/files/Regelwerk_2017_20161016_en_0.pdf. [Online; accessed 31-March-2017]. 2016.
- [BZS14] P. Bender, J. Ziegler, and C. Stiller. “Lanelets: Efficient map representation for autonomous driving.” In: *Intelligent Vehicles Symposium Proceedings, 2014 IEEE*. IEEE. 2014, pp. 420–425.
- [Dup15] M. Dupuis. *OpenDRIVE - Format Specification, Rev. 1.4*. <http://www.opendrive.org/docs/OpenDRIVEFormatSpecRev1.4H.pdf>. [Online; accessed 8-March-2017]. 2015.
- [FY38] R. Fisher and F. Yates. *Statistical tables for biological, agricultural and medical research*. London:[sn], 1938.
- [Gut84] A. Guttman. *R-trees: a dynamic index structure for spatial searching*. Vol. 14. 2. ACM, 1984.
- [KA17] M. Koschi and M. Althoff. “CommonRoad - Scenario Documentation.” 2017.
- [She02] D. C.-K. Shene. *Derivatives of a Bézier Curve*. <http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/Bezier/bezier-der.html>. [Online; accessed 09-April-2017]. 2002.

- [Ver13] B. für Verkehr und digitale Infrastruktur. *Straßenverkehrsordnung*. http://www.gesetze-im-internet.de/stvo_2013/. [Online; accessed 17-April-2017]. 2013.