Construction of an autonomous driving model vehicle

# Procedural track generation for driving simulations

Callum Munro, Roman Wecker, Bünyamin Örümcek

June 2021

Hochschule Kempten
University of Applied Sciences
Fakultät Informatik

# Procedural track generation for driving simulations

Callum Munro, Roman Wecker, Bünyamin Örümcek

***Abstract*—This paper introduces a method for generating round courses based on the VDI or Carolo Specification and shows how to export them to the OpenDRIVE-Format. The method shown in this paper uses a graph and the A* algorithm to create tracks. These tracks are meant to be used for testing algorithms and training AI. Because of this, a great variety of tracks is needed. To achieve this, the weights for the edges are based on noise functions. The paper also presents a possible way to place traffic signs based on the track's geometry and road marking.**

***Index Terms*—noise functions, procedural generation, A* algorithm, pathfinding, VDI Cup, Carolo Cup, race track, driving simulation.**

## I. INTRODUCTION

Simulations have a lot of advantages compared to experiments, when it comes to driving: They often cost less, can be sped up, allow for simple data collection, allow for testing dangerous scenarios(E.g. a child runs on to the street) without actually endangering anyone and also allow for a greater variety of test cases. For driving simulations a track is often needed. Those tracks can be modelled by hand, using real world data or procedurally.

The simulations that are supposed to run, using these tracks, are mainly used to test algorithms for self-driving miniature cars, which compete in the Carolo Cup and VDI Cup and training neural networks for those cars. It was therefore important that the approach would generate tracks that are permissible for the Cups's specifications. In this paper, we present an approach for generating differentiating round courses for those Cups.

## II. METHOD

### A. Track Generator

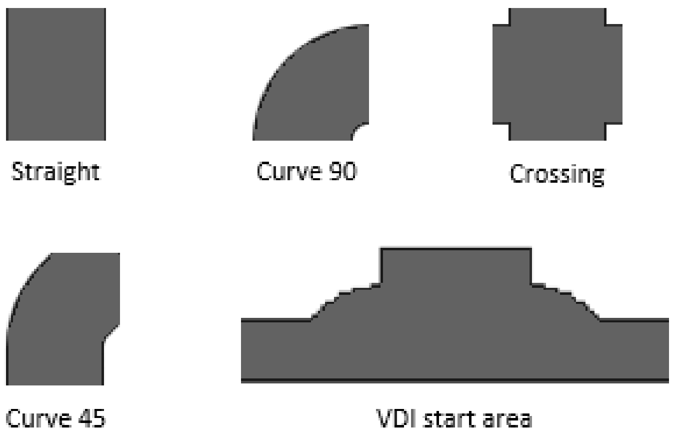

Figure 1: All possible segments for both cups

The tracks of the VDI and Carolo Cup can be divided into individual segments (see Fig. 1). With these segments the entire circuit can be built. Therefore, a grid-based method was chosen for the generation of the track. Both Cups have 90° and 45° curves and all segments exist in all 45° rotated variants. This means that the algorithm can search horizontally, vertically and diagonally in the grid. Both Cups have a special starting area. In terms of generation, the main difference between these two Cups is that the Carolo Cup has crossing segments.

The semester project "Construction of an autonomous driving model vehicle" needs a large amount of randomly generated tracks to create a dataset to train future AIs and test algorithms. This variety of different tracks are achieved with fractal noise functions.

**The Algorithm:** Track generation process

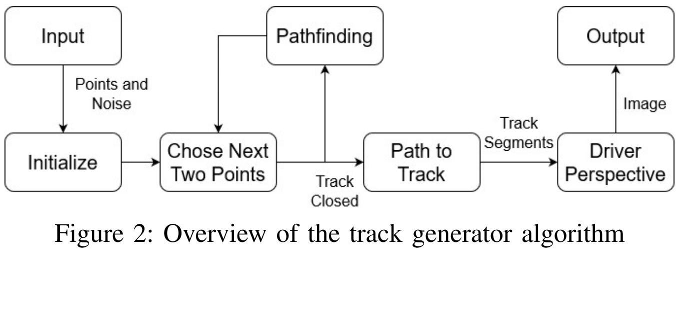

Figure 2: Overview of the track generator algorithm

This section presents the track generation method. The initial state of the generator is that at least 2 points and a fractal noise function are given. The order of the points is important, because it determines which points are connected. At the beginning, a path between the first two points is searched using the A* algorithm, where the noise function equals the cost of a grid cell. This process is repeated until the last point is reached. At the end, the last point is connected to the first and thus the track is closed.

As input paramaters, the track generator needs a list of points, the grid size, the fractal noise function, Cup specification, and a scaling variable, wich determines how much the noise affects the path finding.

Table 1: Parameters used in the following algorithms

| Parameter | Name | Value |
|---|---|---|
| P | current point | - |
| cs | crossing point | 3 |
| cc | crossing counter | 2 |
| $P_{last}$ | last point | - |
| $P_s$ | start point | - |
| $P_e$ | end point | - |

The parameter cs in Table 1 determines from which point a crossing should be generated and cc specifies how many crossings there should be. Start and endpoints define which points are to be connected by the A* algorithm.

**Algorithm 1:** Track generator Initialize

1) Reset all previous values.
2) Create an open-, closed-, distancePlusHeuristic- and a previous-list.
3) Depending on the cup specification, add an artificial point after the first point.
4) If VDI Cup, insert the start area between the first and the artificial point, otherwise add Carolo starting path.
5) Move all input points out of the area of the first and artificial point.

In the Initialize part, the track generator ensures that given points are valid and resets all previous values. In addition, for the VDI Cup, the start area is added to the path beginning from the first point to the right until it reaches the artificial point. For the Carolo Cup, a straight path is created instead, offset by its width.

**Algorithm 2:** Choose next two points

1) Close a 2x2 area starting from the first input point.
2) Close the already found path extended by one grid cell in all cardinal directions. Close all unused input points.
3) If $P \geq P_{last}$, then set $P_s = P_{last}$ and $P_e$ = first point
4) Otherwise if it is the Carolo Cup and $P \geq$ cs and cc $\geq 0$, perform a bounding box check only once. If the check failed, place a crossing. Set the next $P_s$ and $P_e$ as crossing connection points until cc $\leq 0$. Save all crossing indices temporarily.
5) Otherwise set $P_s = P$ and $P_e$ = next point.
6) Do pathfinding between $P_s$ and $P_e$ using A* algorithm and noise as cost.
7) Repeat steps 1 - 6 until path is closed, $P \geq P_{last}$.
8) Draw the noise, the given points and the found path into a new image.
9) Call Algorithm 3.
10) Call Algorithm 4.
11) Return final image.

Algorithm 2 is the core of the track generation. It ensures that the first point always remains reachable. It also adds all path cells found so far extended by one grid cell in all cardinal directions to the closed list. This way path segments have a minimum distance of one grid cell and cannot overlap. The generator always connects the current point with the next point, whereby there are two special cases. One special case is that the last point $P_{last}$ has been reached. When this point is reached, the last point $P_{last}$ is connected to the first point so that the track closes.
The other case applies only to the Carolo Cup. If the crossing point cs is reached, the algorithm tries to add a crossing at this point. For this purpose, it forms a vector $\vec{v}$ between the previous and current path segment. The vector $\vec{v}$ is used to check whether a crossing can be placed and does not intersect with any closed cells or input points. This is done from the current point with a bounding box check in the range of [0; $2{\cdot}\vec{v}$] to [$-2{\cdot}\vec{b}$; $2{\cdot}\vec{b}$], with $\vec{b}$ a perpendicular vector to $\vec{v}$. If the check failed, the track generator selects a perpendicular vector $\vec{s}$ for which the following two statements are true:

1) The euclidean distance between the west/east point of the crossing to the next input point is shorter.
2) A Raycast with width of two from the west/east point of the crossing to the next input point is not blocked.

When the perpendicular vector $\vec{s}$ is found, the algorithm adds four points in the direction $\vec{v}$ to the path. The first and the third become the crossing segments. Each crossing has

exactly three points that can be connected, since the north side is always connected with the main path.
At first, the south side of the second crossing connects to its west/east side according to the vector -$\vec{s}$. Then the west/east side of the second crossing is connected according to the vector $\vec{s}$ with the same cardinal direction as the first crossing. The crossing is completed by connecting the last remaining side of the first crossing with the next input point, or the first input point if the crossing point was the last.
Once the track generator has treated all input points, an image is created. In this image the noise, the input points and the found path are drawn. Then the segments are inserted with algorithm 3. After the segments have been determined, they are saved in a list in driver perspective with algorithm 4. Finally, the image for the UI is returned (see Fig. 3).

**Algorithm 3:** Path to Track

1) Starting from the current cell, compare the previous and next path cells with each other.
2) Determine the logical segment depending on step 1.
3) Draw found segments into the image.

With Algorithm 3: Path to Track the positions of the previous and next path cells are compared. From this comparison, the correct segment can be determined logically. An example would be that if a previous cell is left and the next cell is right then it can only be a straight path segment.

**Algorithm 4:** Driver Perspective

1) For every Straight segment add a Drive Straight segment into the driver perspective list.
2) For every Curve 90 segment add a Turn Left 90 or Turn Right 90 segment into the driver perspective list.
3) For every Curve 45 segment add a Turn Left 45 or Turn Right 45 segment into the driver perspective list.
4) For every Crossing segment add a Crossing segment into the driver perspective list.

The exporter works in the driver perspective. This means that the segments must be converted. From the driver's perspective, it does not matter if it is a 45 degree rotated straight segment or not, because for the driver it is always a straight path.

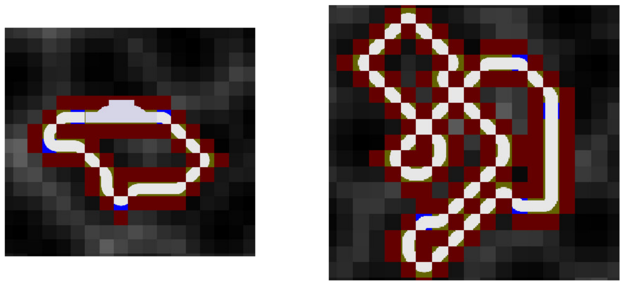
Figure 3: Generated VDI Cup (left) and Carolo Cup (right)

### B. *OpenDrive Exporter*

The OpenDrive Exporter creates a road network from the given route data and exports an .xodr file that can be used to import the route into simulation software such as CarMaker or Carla.

**OpenDrive:** Hierarchical tree structure

OpenDrive is an XML format where data is stored within nodes in a tree structure. In many cases, we need to reference nodes from other subtrees by their IDs. This makes object accesses inconvenient and slow. A better approach is an easily accessible C++ object structure of OpenDrive (see Fig. 4), where node data is stored inside objects, which can be modified while the tree is set up. In a final step, all children of the tree are traversed to generate the OpenDrive file.

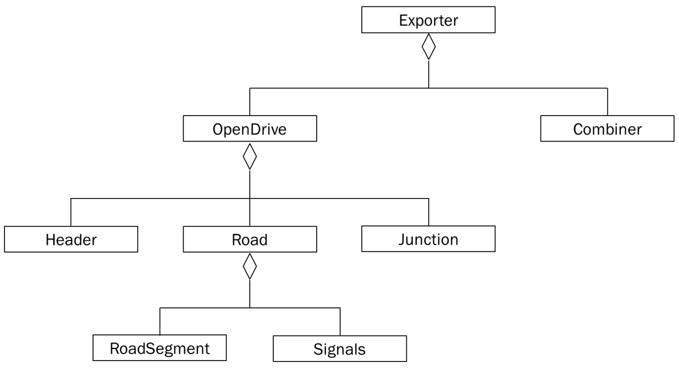

Figure 4: Easily accessible C++ object structure of OpenDRIVE

The Exporter class consists of an OpenDrive class that contains the entire OpenDrive tree and a Combiner class that is used to load additional information from other OpenDrive files and append it within the tree structure. The OpenDrive class consists of the Header, Road, and Junction classes. The Header class contains generic header information. The Road class consists of the RoadSegment and Signals classes. An OpenDrive document can contain any number of roads and a road can contain any number of road segments and signals. The RoadSegment class contains the actual geometric information of a road segment. The Signals class represents a traffic sign placed along a road. The Junction class adds an intersection between roads.

**Roads:** Geometric approach

Each RoadSegment represents a planView element in OpenDrive. A planView in OpenDrive (see Table 2) is defined by its x and y position, inertial heading, segment length and an attribute s, which specifies how far a car would need to travel in a local s-t coordinate system, to reach the start point of this segment. In other words the attribute s are the accumulated lengths of all previous road segments.

Table 2: Attributes of the planView element [3]

| Attribute | Description | Value |
|---|---|---|
| s | s-coordinate of start position | $[0;\infty[$ |
| x | Start position (x inertial) | $]-\infty;\infty[$ |
| y | Start position (y inertial) | $]-\infty;\infty[$ |
| hdg | Start orientation (inertial heading) | $]-\infty;\infty[$ |
| length | Length of the element's reference line | $[0;\infty[$ |

1) Straights:
There are two different types of straight lines that must be treated separately (see Fig. 5).

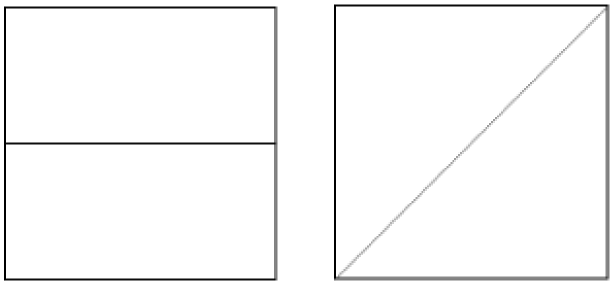
Figure 5: Straights

Formula for the directions WEST, SOUTH, EAST, NORTH

$$length = defaultLength$$

Formula for the directions NORTHEAST, SOUTHEAST, SOUTHWEST, NORTHWEST

$$length = \sqrt{2 \cdot defaultLength^2}$$

2) 90 Degree Curves:

There are two different types of 90 degree curves that must be treated separately (see Fig. 6). Both lengths can be determined as part of the circumference of a circle.

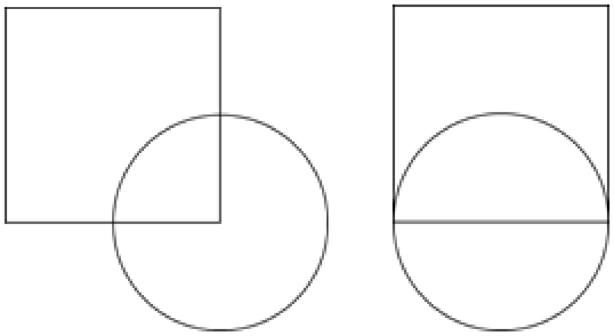
Figure 6: 90 Degree Curves

Formula for the directions WEST, SOUTH, EAST, NORTH

$$length = defaultLength \cdot \pi \cdot 0.25$$

Formula for the directions NORTHEAST, SOUTHEAST, SOUTHWEST, NORTHWEST

$$length = \sqrt{2 * defaultLength^2} \cdot \pi \cdot 0.25$$

90 degree curves can be described as an arc with a constant curvature. The curvature can be calculated using the two following formulas [4].

$$u_{local} = \frac{1}{curvature} \cdot \sin\left(\frac{length \cdot curvature \cdot 360°}{2 \cdot \pi}\right)$$
$$v_{local} = \frac{1}{curvature} \cdot \cos\left(\frac{length \cdot curvature \cdot 360°}{2 \cdot \pi}\right)$$

3) 45 Degree Curves:

There are two different types of 45 degree curves that must be treated separately (see Fig. 7). Cubic polynomials can be used to find a curve, which is steady at the start and end point.

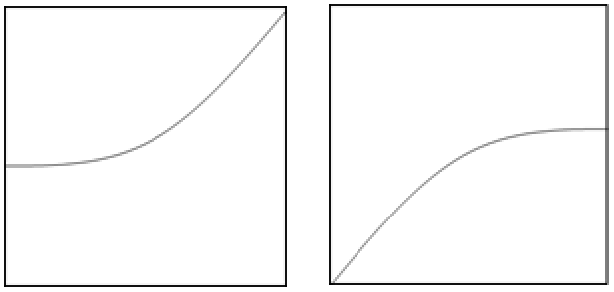
Figure 7: 45 Degree Curves

Formula for the directions WEST, SOUTH, EAST, NORTH
General polynomial of third degree

$$f(x) = ax^3 + bx^2 + cx + d$$
$$f'(x) = 3ax^2 + 2bx + c$$

Conditions:

$$\begin{aligned} f(0) &= 0 \\ f(defaultLength) &= defaultLength/2 \\ f'(0) &= 0 \\ f'(defaultLength) &= 1 \end{aligned}$$

The searched function:

$$\rightarrow f(x) = \frac{defaultLength/2}{defaultLength^2} \cdot x^2$$

$$f'(x) = 2 \cdot \frac{defaultLength/2}{defaultLength^2} \cdot x$$

The determination of the length:

$$length = \int_0^{defaultLength} \sqrt{1 + f'(x)}\, dx$$

The formula for the directions NORTHEAST, SOUTHEAST, SOUTHWEST, NORTHWEST can be determined, for example, using the Lagrange interpolation polynomial.

**Combiner:** Include external OpenDrive files

The Combiner class can be used to load additional information from other OpenDrive files and append it inside the tree structure. The OpenDrive specification offers a feature, which allows to combine files. However the feature is not available in CarMaker at the present time. Therefore, when the file is written for the first time, include tags are added to the XML structure. The Combiner class loads the written file and replaces all includes with the actual files from the drive. This makes it easy to load complex objects or reoccuring parts throughout the document. The Combiner class is used to load the corresponding start segments for Carolo and VDI and the associated lane definitions.

**Signals:** OpenDrive traffic signs

Traffic signs in OpenDrive are placed inside a local s-t coordinate system. As stated before, s is the sum of all lengths of previous road segments. The t axis is perpendicular to s and indicates how far a point is from the center of the road. Road signs are placed procedurally along the road, based on specific conditions.

1) No overtaking / No overtaking end
   These traffic signs are automatically placed when a road has a solid line.

2) Priority road / Give way
   These traffic signs are automatically placed 5 meters ahead of a crossing. The main crossing arm features a priority road sign, the oncoming road a give way sign.

3) Speed limit zone / Speed limit zone end
   A speed limit zone can occur at random positions, under the condition that no other zones are active.

4) Pedestrian crossing
   A pedestrian crossing has 50% chance to be spawned inside a speed limit zone.

5) Limited access road / Limited access road end
   For every segment inside a road the curvature is measured and accumulated. If the sum is low enough, which means the road is almost straight, a limited access road sign is placed. If the sum gets to high, which indicates a curvy road layout, the limited access road end sign is placed.

## III. Conclusion

The approach shown in this paper quickly generates appropriate tracks for the Cups. Because it is a tile based approach the variety of curves is limited. Therefore it would be best to choose another approach or improve it, if a greater variety of curves is needed.

**Future Work:** Known issues and suggestions

The OpenDrive format is currently only partially supported by CarMaker and still has many limitations [5]. Work based on this could consider implementing a Road5 exporter, since Road5 is the primary file format used within CarMaker.

Future work could expand the number of crossings, which is limited to exactly two or none. The decision of when to insert a crossing could also be determined with an algorithm. Furthermore poorly chosen points can cause the path to enclose itself. Therefore finding a method for placing points correctly could be interesting for future work.

## Acknowledgment

We would like to thank Prof. Dr. Göhner, Stuhr and Haselberger for supporting and helping us while working on this project.

## References

[1] J. "Auburn" Peck: FastNoiseLite. Available online at https://github.com/Auburn/FastNoiseLite, last checked on 30.06.2021.
[2] isaac computer science: A* search algorithm. Available online at https://isaaccomputerscience.org/concepts/dsa_search_a_star, last checked on 30.06.2021.
[3] ASAM e.V.: OpenDRIVE 1.6. Available online at https://releases.asam.net/OpenDRIVE/1.6.0/ASAM_OpenDRIVE_BS_V1-6-0.html#_geometry, last checked on 01.07.2021.
[4] ASAM e.V.: ASAM OpenDrive Webinar Part 1. Available online at https://www.youtube.com/watch?v=44Rm7hmqO2M&ab_channel=ASAMe.V., last checked on 01.07.2021.
[5] IPG Automotive: General Advice and Limitations of the OpenDRIVE Import. Available online at https://ipg-automotive.com/support/client-area/faq/ticket/general-advice-and-limitations-of-the-opendrive-import/, last checked on 01.07.2021.