

## I. Personal and study details

Student's name: **Pon ák Adam** Personal ID number: **485175**  
Faculty / Institute: **Faculty of Electrical Engineering**  
Department / Institute: **Department of Computer Graphics and Interaction**  
Study program: **Open Informatics**  
Specialisation: **Computer Graphics**

## II. Master's thesis details

Master's thesis title in English:

**Interactive Editor of Road Network Blocks**

Master's thesis title in Czech:

**Interaktivní editor blok silni ní sít**

Guidelines:

Review methods used to represent road networks in navigation systems and tools for creating geometric models of the road network and its surroundings.

Design an editor to create and edit the basic road network blocks (straight section, curved section, crossroads). The definition of the block will be based on the skeleton of the road network, where the nodes will correspond to the points in the centers of the lanes and the edges to the connectors between them. The editor will allow to define attributes for nodes and edges of the road network (speed limit, possibility of overtaking, etc.). Based on these definitions, the tool will interactively create a geometric model of the road surface, including a texture with traffic lanes, as well as a model of the immediate surroundings (roadside, drainage, ditch). Consider also the procedural generation of details along the roads (bushes, trees, grass).

The editor will be implemented inside a suitable framework such as Blender, Unity, or Virtual Reality Universal Toolkit (VRUT). The resulting roadblocks will be used and tested in the VRUT system. The functionality of the editor will be tested by creating at least ten different roadblocks that will be successfully used for the already implemented traffic simulation inside VRUT.

Bibliography / sources:

- [1] Daniel Aschermann. Modulární editor silni ní sít . Master's thesis, CTU FEE, 2023.
- [2] Vojt ch Kolínský. Editor silni ní sít v systému Virtual Reality Universal Toolkit, Bachelor's thesis, CTU FEE, 2020.
- [3] Jaroslav Mina ík. Simulace okolních dopravních d j . Master's thesis, CTU FEE, 2014.
- [4] Paden, B., áp, M., Yong, S. Z., Yershov, D., & Frazzoli, E. A survey of motion planning and control techniques for self-driving urban vehicles. IEEE Transactions on Intelligent Vehicles, 1(1), 33-55, 2016.
- [5] Projekt OpenDrive. <http://www.opendrive.org/>
- [6] Václav Kyba. Modulární 3D prohlíže . Master's thesis, CTU FEE, 2008.
- [7] Alena Mikushina. Tvorba modulárních 3D komponent pro videohry, Bachelor's thesis, CTU FEE, 2020.

Name and workplace of master's thesis supervisor:

**doc. Ing. Ji í Bittner, Ph.D. Department of Computer Graphics and Interaction**

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **15.02.2024** Deadline for master's thesis submission: **08.01.2025**

Assignment valid until: **21.09.2025**

doc. Ing. Ji í Bittner, Ph.D.  
Supervisor's signature

Head of department's signature

prof. Mgr. Petr Páta, Ph.D.  
Dean's signature

### III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

\_\_\_\_\_  
Date of assignment receipt

\_\_\_\_\_  
Student's signature

Master Thesis



Czech  
Technical  
University  
in Prague

**F3**

Faculty of Electrical Engineering  
Department of Computer Graphics and Interaction

## Interactive editor of road network blocks

**Bc. Adam Pončák**

Supervisor: doc. Ing. Jiří Bittner, Ph.D.

Field of study: Open Informatics

Subfield: Computer Graphics

January 2025





## Acknowledgements

I would like to thank my supervisor, doc. Ing. Jiří Bittner, Ph.D., for their patience, support, valuable feedback, and the opportunity to work on this project. I am especially grateful for their understanding and encouragement throughout the process.

Additionally, I would like to thank Czech Technical University in Prague for providing a supportive environment, valuable opportunities, and the necessary resources for my research.

I am also deeply thankful to my family and friends for their mental support, patience, and belief in me, which gave me the strength to complete this thesis.

## Declaration

I hereby declare that the submitted work is my own, completed independently. I have ensured that all sources and references used are properly cited. Additionally, I acknowledge the use of AI-based tools to support the development of this thesis. Specifically, I used Grammarly for spellchecking, formatting, and vocabulary enhancement, and ChatGPT for refining and improving the clarity and style of text originally prepared by me.

In Prague, January 2025

## Abstract

This thesis addresses the need for an interactive tool to create and edit road network blocks for virtual environments, particularly within the Virtual Reality Universal Toolkit (VRUT). The developed editor supports the design of straight sections, curves, and junctions using a skeleton-based representation, where nodes represent lane centers and edges of their connections. Procedural generation techniques enable the creation of detailed road surfaces, lane markings, and roadside features, such as vegetation and drainage. The editor, implemented in Blender, employs geometry nodes for efficient modeling and customization.

Junction modeling is a focal point featuring an algorithm for generating smooth geometries and customizable connections. Export functionality ensures compatibility with the VRUT system via its custom XML-based format. The editor's utility was validated by creating ten road scenarios tested within VRUT's simulation environment, demonstrating robust integration and versatility. This work contributes a flexible tool for road network modeling, advancing VRUT's capabilities in virtual environment design and traffic simulation.

**Keywords:** Road Network Modeling, Procedural Content Generation, Blender Geometry Nodes, VRUT System, Junction Generation, Traffic Simulation

**Supervisor:** doc. Ing. Jiří Bittner, Ph.D.

Karlovo náměstí 13,  
E-421,  
12000 Praha 2

## Abstrakt

Tato diplomová práce se zabývá potřebou interaktivního nástroje pro tvorbu a úpravu bloků silničních sítí ve virtuálním prostředí, konkrétně v rámci systému Virtual Reality Universal Toolkit (VRUT). Vyvinutý editor podporuje návrh přímých úseků, zatáček a křižovatek pomocí reprezentace založené na kostře, kde uzly představují středy jízdních pruhů a hrany jejich propojení. Procedurální generování umožňuje tvorbu detailních povrchů silnic, značení jízdních pruhů a okolních prvků, jako jsou vegetace a odvodňovací systémy. Editor implementovaný v prostředí Blender využívá geometry nodes pro efektivní modelování a přizpůsobení.

Hlavním bodem práce je modelování křižovatek, které obsahuje algoritmus pro generování plynulých geometrií a přizpůsobitelných spojení. Funkce exportu zajišťuje kompatibilitu se systémem VRUT prostřednictvím vlastního formátu XML. Užitečnost editoru byla ověřena vytvořením deseti scénářů silničních bloků testovaných ve VRUT simulátoru, čímž byla prokázána jeho robustní integrace a univerzálnost.

Tato práce přispívá flexibilním nástrojem pro modelování silničních sítí, který rozšiřuje možnosti VRUT systému v návrhu virtuálních prostředí a simulacích dopravy.

**Klíčová slova:** Modelování silniční sítě, Procedurální generování, Geometry Nodes v Blendru, Systém VRUT, Generování křižovatek, Simulace dopravy

**Překlad názvu:** Interaktivní editor bloků silniční sítě

# Contents

|                                                                    |           |                                                              |           |
|--------------------------------------------------------------------|-----------|--------------------------------------------------------------|-----------|
| <b>1 Introduction</b>                                              | <b>1</b>  | 3.2.1 Roads Section . . . . .                                | 29        |
|                                                                    |           | 3.2.2 Attributes Section . . . . .                           | 30        |
|                                                                    |           | 3.2.3 Connections Section . . . . .                          | 30        |
|                                                                    |           | 3.2.4 Junctions . . . . .                                    | 31        |
|                                                                    |           | 3.2.5 Summary of VRUT<br>representation . . . . .            | 32        |
| <b>2 Geometry Background</b>                                       | <b>5</b>  | <b>3.3 ASAM OpenDRIVE . . . . .</b>                          | <b>32</b> |
| 2.1 Representing Geometry . . . . .                                | 5         | 3.3.1 File Structure . . . . .                               | 33        |
| 2.1.1 Conclusion . . . . .                                         | 7         | 3.3.2 Road Reference Line . . . . .                          | 33        |
| 2.2 Definition & Properties of Curves                              | 7         | 3.3.3 Geometries . . . . .                                   | 33        |
| 2.2.1 Curves and Their Relationship<br>to Control Points . . . . . | 7         | 3.3.4 Roads . . . . .                                        | 34        |
| 2.2.2 Curve Representations . . . . .                              | 8         | 3.3.5 Lanes . . . . .                                        | 35        |
| 2.2.3 Piecewise Representations and<br>Splines . . . . .           | 8         | 3.3.6 Junctions . . . . .                                    | 36        |
| 2.2.4 Properties of Curves . . . . .                               | 8         | 3.3.7 Summary of ASAM<br>OpenDRIVE . . . . .                 | 37        |
| 2.3 Specific Curve Types . . . . .                                 | 9         | <b>3.4 Conclusion . . . . .</b>                              | <b>37</b> |
| 2.3.1 B-splines . . . . .                                          | 10        |                                                              |           |
| 2.3.2 Bézier curves . . . . .                                      | 12        | <b>4 Implementation Suitable<br/>Frameworks</b>              | <b>39</b> |
| 2.3.3 Catmull-Rom splines . . . . .                                | 12        | 4.1 VRUT . . . . .                                           | 39        |
| 2.3.4 Euler Spiral . . . . .                                       | 14        | 4.2 Game Engines . . . . .                                   | 40        |
| 2.3.5 Summary of Curves in Road<br>Design . . . . .                | 15        | 4.2.1 Unreal Engine . . . . .                                | 40        |
| 2.4 Polylines . . . . .                                            | 16        | 4.2.2 Unity . . . . .                                        | 41        |
| 2.4.1 Approximating Curves with<br>Polylines . . . . .             | 16        | 4.3 3D modeling software . . . . .                           | 42        |
| 2.4.2 Conclusion . . . . .                                         | 17        | 4.3.1 Blender . . . . .                                      | 42        |
| 2.5 Polygonal Meshes . . . . .                                     | 17        | 4.3.2 Houdini . . . . .                                      | 43        |
| 2.5.1 Mesh Triangulation . . . . .                                 | 19        | 4.3.3 Maya . . . . .                                         | 44        |
| 2.5.2 Textures . . . . .                                           | 19        | <b>4.4 Conclusion . . . . .</b>                              | <b>45</b> |
| 2.5.3 Export Formats . . . . .                                     | 20        |                                                              |           |
| 2.6 Procedural Content Generation<br>(PCG) . . . . .               | 22        | <b>5 Virtual Reality Universal Toolkit<br/>system (VRUT)</b> | <b>47</b> |
| 2.6.1 What is Procedural Content<br>Generation? . . . . .          | 23        | 5.1 Overview . . . . .                                       | 47        |
| 2.6.2 Usecases of PCG . . . . .                                    | 23        | 5.2 Core of VRUT . . . . .                                   | 48        |
| 2.6.3 Advantages of PCG . . . . .                                  | 23        | 5.3 Relevant Modules . . . . .                               | 48        |
| 2.6.4 Challenges of PCG . . . . .                                  | 24        | 5.3.1 Road Editor . . . . .                                  | 49        |
| 2.6.5 Generating Meshes using<br>Curves . . . . .                  | 24        | 5.3.2 Road Network Editor . . . . .                          | 49        |
| 2.6.6 Conclusion . . . . .                                         | 25        | 5.3.3 Traffic Module . . . . .                               | 50        |
|                                                                    |           | <b>Conclusion . . . . .</b>                                  | <b>51</b> |
| <b>3 Road Graph</b>                                                | <b>27</b> |                                                              |           |
| 3.1 Standards of Data Representation                               | 27        | <b>Part II<br/>Implementation</b>                            |           |
| 3.1.1 OpenStreetMap (OSM) . . . . .                                | 27        | <b>6 Road Geometry</b>                                       | <b>55</b> |
| 3.1.2 RoadXML . . . . .                                            | 28        | 6.1 Road-base Representation Types                           | 55        |
| 3.1.3 Vector Map (VMap) . . . . .                                  | 28        | 6.1.1 Catmull-Rom curves . . . . .                           | 55        |
| 3.1.4 Simulation-specific Formats . . . . .                        | 28        | 6.1.2 Bézier curves . . . . .                                | 56        |
| 3.2 VRUT representation . . . . .                                  | 29        | 6.1.3 NURBS curves . . . . .                                 | 56        |
|                                                                    |           | 6.1.4 Hair curves . . . . .                                  | 57        |
|                                                                    |           | 6.1.5 Mesh representation . . . . .                          | 57        |

|                                                         |           |
|---------------------------------------------------------|-----------|
| 6.1.6 Conclusion .....                                  | 58        |
| 6.2 Road splines .....                                  | 58        |
| 6.2.1 Central Spline .....                              | 59        |
| 6.2.2 Spline Classes .....                              | 60        |
| 6.2.3 Secondary Splines .....                           | 60        |
| 6.2.4 Output of spline creation ...                     | 60        |
| 6.3 Road mesh generation .....                          | 61        |
| 6.3.1 T-Vertices Problem .....                          | 61        |
| 6.3.2 Proposed Solutions .....                          | 61        |
| 6.4 Textures & Materials .....                          | 63        |
| 6.4.1 Baking Textures .....                             | 63        |
| <b>7 Junction Geometry</b>                              | <b>67</b> |
| 7.1 Internal data representation ....                   | 67        |
| 7.2 Junction creation algorithm ....                    | 67        |
| 7.2.1 Outline Intersections .....                       | 68        |
| 7.2.2 Marking Intersection Pairs ..                     | 68        |
| 7.2.3 Junction Geometry Generation                      | 69        |
| 7.3 Junction Modifier .....                             | 71        |
| 7.4 Alternative Tested Approaches .                     | 72        |
| 7.4.1 Initial Mesh-Based Approach                       | 72        |
| 7.4.2 Curve-Based Approach with<br>Geometry Nodes ..... | 73        |
| 7.4.3 Influence on Final<br>Implementation .....        | 73        |
| <b>8 Features</b>                                       | <b>75</b> |
| 8.1 Modifier Stack .....                                | 75        |
| 8.2 Profile Along the Road .....                        | 75        |
| Preset and Custom Profiles .....                        | 76        |
| Modifier Interface .....                                | 76        |
| Profile Stacking and Layering ....                      | 77        |
| Geometry Nodes Implementation .                         | 77        |
| Adding Custom Profiles .....                            | 78        |
| 8.3 Object Along the Road .....                         | 78        |
| Curve Sampling and Side Selection                       | 79        |
| Material Handling and UV Mapping                        | 79        |
| Integration and Limitations .....                       | 80        |
| Surroundings .....                                      | 80        |
| Graphical User Interface .....                          | 81        |
| Conclusion .....                                        | 81        |
| <b>9 Road Network</b>                                   | <b>83</b> |
| 9.1 Internal Representation .....                       | 83        |
| 9.2 Junctions .....                                     | 84        |
| 9.3 Export Formats .....                                | 84        |
| VRUT Format .....                                       | 84        |
| OpenDRIVE Format .....                                  | 85        |

|                                     |    |
|-------------------------------------|----|
| 9.3.1 Visualization of Road Network | 85 |
| Conclusion .....                    | 86 |

|                                               |           |
|-----------------------------------------------|-----------|
| <b>10 Testing in VRUT</b>                     | <b>87</b> |
| 10.1 Compilation of VRUT .....                | 87        |
| 10.2 Exporting Geometry for VRUT              | 87        |
| 10.3 Importing Road Blocks into<br>VRUT ..... | 89        |
| 10.4 Testing .....                            | 89        |
| <b>11 Conclusion</b>                          | <b>91</b> |

## Appendices

|                                                   |            |
|---------------------------------------------------|------------|
| <b>A Simple VRUT Road Network XML<br/>example</b> | <b>95</b>  |
| <b>B Simple ASAM OpenDRIVE XML<br/>example</b>    | <b>97</b>  |
| <b>C Ten Road Blocks Tested in<br/>VRUT</b>       | <b>99</b>  |
| <b>D Bibliography</b>                             | <b>105</b> |

## Figures

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |    |                                                                                                                                                                                                                                                                                                                                             |    |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 2.1 Raw data representation types, <i>point cloud</i> on the left (Source: [49]) and <i>polygon soup</i> on the right (Source: [28]) . . . . .                                                                                                                                                                                                                                                                                                                                                             | 6  | 2.11 Curve approximation. Different sampling methods. a) Uniform sampling, b) Adaptive sampling based on curvature . . . . .                                                                                                                                                                                                                | 18 |
| 2.2 Solid (volumetric) representation types, <i>CSG</i> on the left (Source: [89]) and <i>vertex grid</i> on the right (Source: [49]) . . . . .                                                                                                                                                                                                                                                                                                                                                            | 6  | 2.12 Polygonal mesh representation (Source: [21]) . . . . .                                                                                                                                                                                                                                                                                 | 18 |
| 2.3 Surface (boundary) representation types, <i>polygonal mesh</i> on the left (Source: [49]), <i>implicit surface</i> in the center (Source: [1]) and <i>parametric surface</i> on the right (Source: [32]) . . . . .                                                                                                                                                                                                                                                                                     | 6  | 2.13 Different triangulation methods. a) Ear cutting; b) Randomized incremental c) Constrained Delaunay triangulation; d) Delaunay refinement (Source: [60]) . . . . .                                                                                                                                                                      | 20 |
| 2.4 This image illustrates curve continuities under the assumption that the parameter $t$ represents time: $C^0$ and $G^0$ ensure positional continuity, $C^1$ and $G^1$ enforce tangent continuity, with $C^1$ also ensuring matching velocity. Similarly, $C^2$ and $G^2$ ensure curvature continuity, with $C^2$ also enforcing matching acceleration. Geometric continuity focuses on directional alignment, while parametric continuity requires exact derivative magnitudes (Source: [69]) . . . . . | 10 | 2.14 Different texture types visualized with a final result at the bottom (Source: [54]) . . . . .                                                                                                                                                                                                                                          | 21 |
| 2.5 a) Nonrational B-spline curves with different degrees, b) Comparison of nonrational and rational B-spline curves (Source: [52]) . . . . .                                                                                                                                                                                                                                                                                                                                                              | 11 | 3.1 OpenDRIVE reference line with other features attached to it [5] . . . . .                                                                                                                                                                                                                                                               | 33 |
| 2.6 Different degree Bézier curves (Source: [87]) . . . . .                                                                                                                                                                                                                                                                                                                                                                                                                                                | 12 | 3.2 ASAM OpenDRIVE: Road geometry representation [4] . . . . .                                                                                                                                                                                                                                                                              | 34 |
| 2.7 Different Catmull-Rom parameterizations: a) Control polygon, b) Uniform, c) Chordal, d) Centripetal (Source: [93]) . . . . .                                                                                                                                                                                                                                                                                                                                                                           | 14 | 3.3 ASAM OpenDRIVE: Junction example [5] . . . . .                                                                                                                                                                                                                                                                                          | 37 |
| 2.8 Euler spiral (Source: [23]) . . . . .                                                                                                                                                                                                                                                                                                                                                                                                                                                                  | 15 | 5.1 Examples of RoadEditor module's blocks [6] . . . . .                                                                                                                                                                                                                                                                                    | 49 |
| 2.9 A curve composed of Euler spirals, line and circular-arc segments (Source: [63]) . . . . .                                                                                                                                                                                                                                                                                                                                                                                                             | 15 | 5.2 A preview of Traffic editor GUI. . . . .                                                                                                                                                                                                                                                                                                | 50 |
| 2.10 A polyline composed of connected line segments (Source: [78]) . . . . .                                                                                                                                                                                                                                                                                                                                                                                                                               | 17 | 5.3 Traffic simulation module in action . . . . .                                                                                                                                                                                                                                                                                           | 50 |
|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |    | 6.1 Visualization of hair curves in the Blender viewport. The blue line represents the actual curve, yellow line indicate the control points of NURBS curves, red lines represent the control points of Bézier curves, and the green line illustrates the hair curve representation of these curves visible in Blender's viewport . . . . . | 57 |
|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |    | 6.2 Difference between a 'curve' and a 'spline' . . . . .                                                                                                                                                                                                                                                                                   | 59 |
|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |    | 6.3 Central spline creation . . . . .                                                                                                                                                                                                                                                                                                       | 59 |
|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |    | 6.4 Spline classes - visualized . . . . .                                                                                                                                                                                                                                                                                                   | 60 |
|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |    | 6.5 Function for profile on curve generation and UV mapping . . . . .                                                                                                                                                                                                                                                                       | 61 |
|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |    | 6.6 Demo of road objects generated geometry . . . . .                                                                                                                                                                                                                                                                                       | 62 |
|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |    | 6.7 Gaps between two lanes caused by different curve sampling . . . . .                                                                                                                                                                                                                                                                     | 63 |

|                                                                                                                                                        |    |                                                                                                  |     |
|--------------------------------------------------------------------------------------------------------------------------------------------------------|----|--------------------------------------------------------------------------------------------------|-----|
| 6.8 Gaps between lanes fixed using <i>Merge by Distance</i> , but corrupting UV mapping . . . . .                                                      | 64 | C.1 Road Block 1: Simple straight road . . . . .                                                 | 99  |
| 6.9 Gaps between lanes fixed by overlaying neighbouring lanes, maintaining UV mapping . . . . .                                                        | 65 | C.2 Road Block 2: Simple curved road . . . . .                                                   | 100 |
| 6.10 Concept of optimal lane gap fixing: blue vertices are created from red vertices, with blue lines indicating new edges splitting existing faces. . | 65 | C.3 Road Block 3: Straight road with features along the sides . . . . .                          | 100 |
| 6.11 Demo of UV mapping on road object . . . . .                                                                                                       | 66 | C.4 Road Block 4: Straight road with features between lanes as well as along the sides . . . . . | 101 |
| 7.1 Intersection chains forming junction outlines . . . . .                                                                                            | 71 | C.5 Road Block 5: Straight multilane road with multiple features . . . . .                       | 101 |
| 7.2 Example of mesh based approach for junction generation . . . . .                                                                                   | 73 | C.6 Road Block 6: X-shaped junction with features . . . . .                                      | 102 |
| 7.3 Intersecting splines using raycast method, showing intersection points in red with lines connecting them .                                         | 74 | C.7 Road Block 7: T-shaped junction with features . . . . .                                      | 102 |
| 8.1 Interface of the <i>profile-along-road</i> feature modifier (options for custom profiles and general settings only) .                              | 76 | C.8 Road Block 8: Junction with connecting lanes and features. . . .                             | 103 |
| 8.2 Demo of road features generated geometry and corresponding modifier stack . . . . .                                                                | 77 | C.9 Road Block 9: Looped track with features along the sides . . . . .                           | 103 |
| 8.3 Example of a custom profile curve and the resulting geometry . . . . .                                                                             | 78 | C.10 Road Block 10: Complex road network with multiple junctions and features . . . . .          | 104 |
| 8.4 Interface of the road objects modifier with red dots representing the sampled curve control points and the curve itself connecting them . .        | 79 |                                                                                                  |     |
| 8.5 Example of the surroundings feature with terrain and vegetation                                                                                    | 81 |                                                                                                  |     |
| 8.6 Graphical User Interface of the editor in Blender's side-panel . . . . .                                                                           | 82 |                                                                                                  |     |
| 9.1 Example of object properties in Blender used for connector pairing                                                                                 | 84 |                                                                                                  |     |
| 9.2 Visualization of the road network in Blender . . . . .                                                                                             | 86 |                                                                                                  |     |
| 10.1 A screenshot from CMake capturing some of the available modules . . . . .                                                                         | 88 |                                                                                                  |     |

## Tables







# Chapter 1

## Introduction

The automotive industry is undergoing a transformation driven by advances in vehicle design, propulsion systems, and connectivity. These refinements require tools to test and refine new functionalities, especially for autonomous driving. One such initiative is the Virtual Reality Universal Toolkit (VRUT), a collaborative project by students and engineers at Škoda Auto a.s. VRUT provides a flexible simulation environment for creating and testing realistic traffic scenarios, with a modular architecture enabling diverse applications.

Despite its capabilities, VRUT lacks an intuitive tool for creating realistic and customizable traffic environments. Its current Road Editor module relies on predefined geometric elements imported externally, making scene creation labor-intensive and inflexible. This thesis aims to design and implement an editor for generating and customizing road network elements to address this limitation. These elements include configurable segments such as straight lines, curved sections, and junctions, represented using a skeletal model where nodes correspond to lane centers and edges define their connections. This approach allows for scalable, flexible modeling of traffic environments without fixed-size constraints.

The proposed editor will integrate features for editing road networks and generating geometric surface models, including lane textures and roadside features like ditches, pavements, and vegetation. Procedural generation will enhance realism by populating scenes with features such as lamps, trees, and grass. By enabling the creation of targeted traffic scenarios, the tool will address challenges in simulating rare or complex situations that are difficult to capture using real-world data.

This thesis will develop the editor within a framework compatible with VRUT and suitable for generating road network elements. The planned editor will provide an intuitive interface for interactive design, efficient configuration of road properties, and generation of complex geometries. The contributions of this work include a comprehensive review of methods for representing road networks in navigation systems and geometric modeling tools. Additionally, it involves designing and implementing an editor for customizable road network elements, which incorporates procedural generation for surrounding details. Finally, the editor will be validated by creating diverse road network elements and testing them within the VRUT system.

By addressing VRUT's current limitations, this work aims to provide a robust tool for efficient traffic scene creation, contributing to advancements in simulation technologies and autonomous driving research, with potential secondary benefits, such as supporting simulated driver training environments or aiding in the study of traffic flow and scenario planning in educational contexts.



## **Part I**

### **Foundations & Analysis**



## Chapter 2

### Geometry Background

This chapter focuses on the geometric representation of objects, specifically the road blocks and related structures generated by the add-on developed as part of this thesis.

To build a foundation for understanding the design and implementation of the add-on, we begin by discussing various formats and types of geometric representations and analyzing their relevance to the modeling of road geometry. For each representation, key concepts and algorithms are explored in relation to their utility for this work. After identifying the most suitable representations for roads, we delve into the process of creating such geometry procedurally, relying on relevant user input to drive the generation.

This chapter then addresses additional geometry-related considerations, such as enhancing the visual appeal of the generated objects through the application of materials and textures. Finally, we explore the tools available for geometry creation, highlighting their strengths and their applicability to the goals of this thesis.

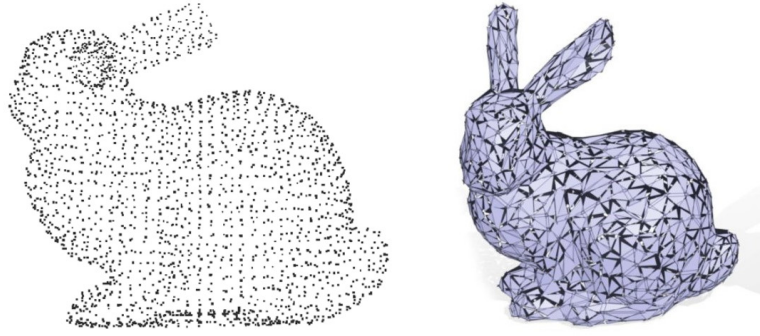
#### 2.1 Representing Geometry

To begin, we examine the topic of geometric representation. In computer graphics, geometric representations are often categorized into raw data, solids, curves, and surfaces, each tailored to specific applications and workflows [33, 38, 69].

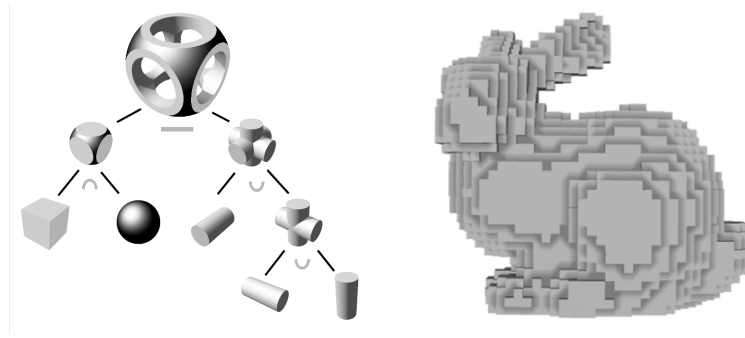
**Raw Data** refers to unprocessed geometric information, such as point clouds or polygonal meshes, directly from scanning devices or sensors. These representations capture the external structure of objects but lack connectivity or higher-level organization, making them insufficient for operations such as collision detection without further processing [38, 51, 53, 88].

**Solids** represent both the interior and exterior of objects, allowing calculations of physical properties such as volume and mass. Techniques like Constructive Solid Geometry (CSG) and voxel grids are common in this category. However, their computational intensity and focus on volumetric properties make them less suitable for rendering-focused applications [35, 69].

**Surfaces**, in contrast, focus solely on the visible boundaries of objects, offering a computationally efficient way to represent shapes for rendering

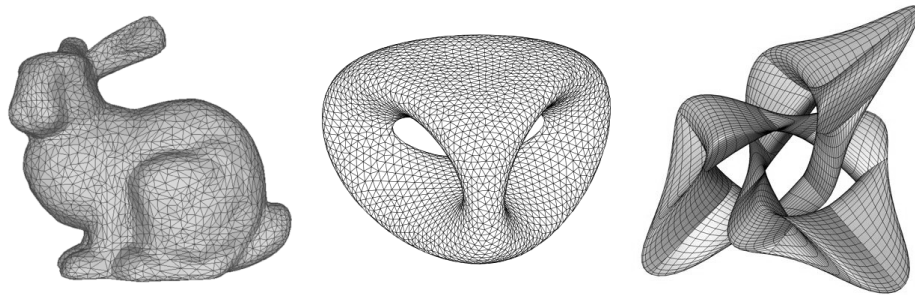


**Figure 2.1:** Raw data representation types, *point cloud* on the left (Source: [49]) and *polygon soup* on the right (Source: [28])



**Figure 2.2:** Solid (volumetric) representation types, *CSG* on the left (Source: [89]) and *vertex grid* on the right (Source: [49])

and visualization. Surface models, such as polygonal meshes and parametric surfaces, are widely used due to their compatibility with graphics hardware and their ability to support smooth shading and texture mapping [53, 69].



**Figure 2.3:** Surface (boundary) representation types, *polygonal mesh* on the left (Source: [49]), *implicit surface* in the center (Source: [1]) and *parametric surface* on the right (Source: [32])

**Curves** play a fundamental role in defining contours and guiding shapes in both 2D and 3D geometry. Representations like Bézier curves, B-splines, and Catmull-Rom splines allow for precise control and are particularly useful for generating smooth geometries, such as roads or guiding surfaces [45, 29,

53, 63].

### ■ 2.1.1 Conclusion

This thesis emphasizes the use of **surfaces** and **curves**, as they offer the flexibility and computational efficiency required for applications such as the generation of road geometry in simulation tools [29, 45]. While raw data and solids have their merits in specific contexts, they are less suitable for the rendering and geometry generation tasks central to this work. Among these geometric constructs, curves stand out as foundational elements in procedural workflows, serving as the basis for defining complex shapes like roads.

## 2.2 Definition & Properties of Curves

The following discussion on the theory of curves is primarily based on the book *Fundamentals of Computer Graphics* [69].

Curves are foundational geometric constructs that can be intuitively understood as the path traced by a moving point over time, much like the continuous stroke of a pen. Mathematically, a curve is a continuous path represented as the image of an interval in  $n$ -dimensional space, where the interval serves as a parameter, such as time or arc length, guiding the progression of the curve [69].

They serve as the basis for modeling and designing various real-world structures, including roads. In road design, curves are essential for defining smooth, navigable paths and ensuring transitions that prioritize safety and efficiency [79].

### 2.2.1 Curves and Their Relationship to Control Points

Curves can be classified into two types based on their relationship with control points—points that define their shape:

- **Interpolating Curves:** These curves pass directly through all their control points, ensuring precise adherence to specified positions. However, this exactness can lead to undesirable properties, such as reduced continuity and a tendency to extend outside the convex hull of the control points, especially in smooth interpolating curves such as Catmull-Rom splines [53].
- **Approximating Curves:** These curves do not necessarily pass through their control points but are influenced by them to shape the curve. This approach provides better smoothness, local control, and continuity, with Bézier and B-spline curves being prominent examples [69].

### 2.2.2 Curve Representations

Curves can be mathematically specified through implicit, parametric, or generative (procedural) representations:

- **Implicit Representation:** Defines a curve as the set of points that satisfy the equation  $f(x, y) = 0$ .
- **Parametric Representation:** Expresses the coordinates as functions of a parameter  $t$ , making it particularly favored in computer graphics for ease of manipulation and rendering.
- **Generative Representation:** Constructs curves through procedural methods such as fractals [69].

### Reparametrization

Reparameterization allows for tailoring parameterizations to specific applications, such as achieving uniform speed along the curve. Arc-length parameterization, a special case, ensures equal parameter intervals represent equal lengths, which is especially beneficial for uniform motion along the curve. However, computing arc-length parameterization often requires numerical methods due to its complexity [69].

### 2.2.3 Piecewise Representations and Splines

Defining a parametric function for simple shapes such as lines, circles, or ellipses is straightforward, but complex curves often require dividing them into smaller, simpler segments. These **piecewise representations**, or **splines**, use components such as line segments or arcs to approximate the overall shape. Ensuring continuity between pieces is critical to avoid disruptions in the flow of the curve. Although simpler pieces may not perfectly replicate a curve, increasing their number improves accuracy. This trade-off between accuracy, complexity, and simplicity makes piecewise representations highly adaptable in computer graphics [69].

### 2.2.4 Properties of Curves

The properties of a curve describe its shape and behaviour, offering a framework for analysis and representation. These properties can be categorized as **local** or **global**. Local properties, such as *continuity*, *position*, *direction*, and *curvature*, describe specific points on the curve without requiring knowledge of the entire shape. In contrast, global properties, such as *length*, *closedness* or *self-intersection*, involve the curve as a whole and provide a comprehensive view of its structure [69].

The following discussion will focus on the most relevant properties of curves in the creation of roads, particularly those that are less intuitive and require deeper exploration.



## ■ Continuity

Continuity ensures smooth transitions between segments of a curve and can be classified into two main types:

- **Parametric Continuity ( $C^n$ ):** Requires that the curve's derivatives up to order  $n$  match at the segment boundaries. It is dependent on the parameterization, so differences in segment speeds can disrupt continuity.
- **Geometric Continuity ( $G^n$ ):** Focuses on the shape of the curve, ensuring that derivatives align in direction but not necessarily in magnitude. For instance,  $G^1$  continuity requires tangents to point in the same direction, even if their lengths differ [69].

The degrees of continuity provide further classification:

- **$C^0$  Continuity:** Ensures continuity in position, meaning that the endpoints of adjacent segments meet.
- **$C^1$  Continuity:** Ensures the first derivative (tangent) is continuous, resulting in no abrupt changes in direction.
- **$C^2$  Continuity:** Ensures continuity in the second derivative (curvature), producing smoother transitions [69].

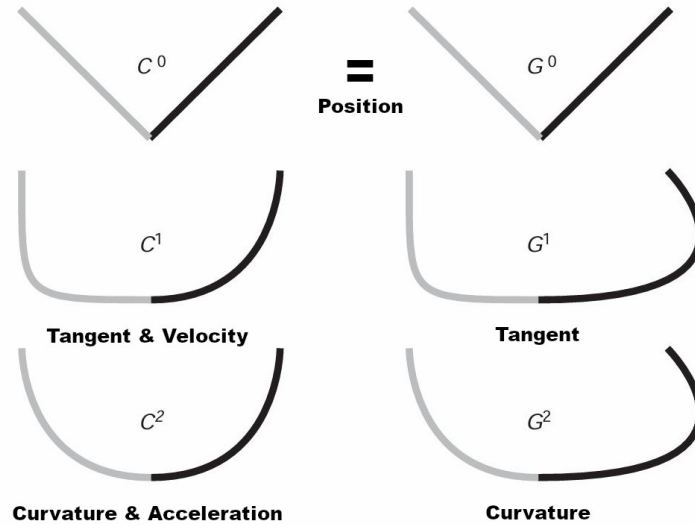
The choice of parametric or geometric continuity, as well as the degree of continuity, depends on the specific requirements of the curve's application [69]. Figure 2.4 illustrates the distinctions between different parametric and geometric continuities.

## ■ Curvature

Is defined as the rate at which the direction of the curve changes. Higher degrees of continuity, such as  $C^2$ , are often crucial in applications requiring smooth motion or fluid dynamics, where disruptions in curvature or higher derivatives can lead to instability or turbulence [69]. Therefore, achieving at least  $C^2$  continuity (matching second derivatives) is essential to ensure gradual changes in curvature, which is crucial for vehicle dynamics and safety [26]. For roads, the curvature is crucial to determine how quickly vehicles need to turn, which affects safety, efficiency, and comfort [86].

## ■ 2.3 Specific Curve Types

In this section, we will explore specific types of curves, such as B-splines, Bézier curves, Catmull-Rom curves, and Euler spirals (clothoids), examining their properties and applications. These curves play a critical role in the construction of roads, balancing functionality, efficiency, and safety in the design of roads and infrastructure.



**Figure 2.4:** This image illustrates curve continuities under the assumption that the parameter  $t$  represents time:  $C^0$  and  $G^0$  ensure positional continuity,  $C^1$  and  $G^1$  enforce tangent continuity, with  $C^1$  also ensuring matching velocity. Similarly,  $C^2$  and  $G^2$  ensure curvature continuity, with  $C^2$  also enforcing matching acceleration. Geometric continuity focuses on directional alignment, while parametric continuity requires exact derivative magnitudes (Source: [69])

### ■ 2.3.1 B-splines

B-splines (Basis splines) are a foundational tool for constructing smooth, piecewise polynomial curves in computer graphics and geometric modeling. Defined as a linear combination of basis functions, they are highly versatile and exhibit  $C^{d-1}$  continuity for degree  $d$ , making them suitable for applications demanding smooth transitions. B-splines are approximating curves, which means that they generally do not pass through their control points, but instead, these points influence the shape of the curve [69, 53].

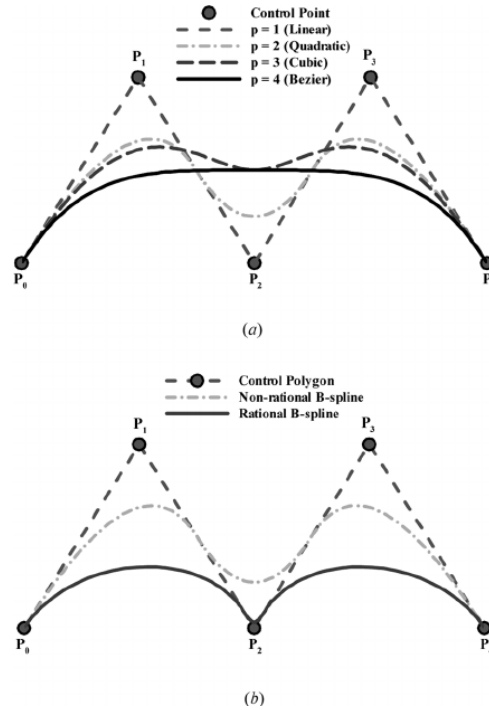
#### ■ Variants of B-Splines

- **Uniform B-Splines:** These curves have evenly spaced knots, ensuring consistent behavior and straightforward computations. However, their uniformity limits adaptability in applications with unevenly distributed control points or variable curve features [69].
- **Nonuniform B-Splines:** By allowing non-uniform knot spacing, these curves provide more localized control, allowing for sharper transitions or variable curve density where needed. This feature is particularly valuable in animations and complex modeling scenarios [53].
- **Cubic B-Splines:** A specific subset of B-splines with degree 3, cubic B-splines offer  $C^2$  continuity and are frequently used in computer graphics

due to their balance of smoothness and computational efficiency [69].

- **Non-Uniform Rational B-Splines (NURBS):** These curves enhance standard B-splines by associating scalar weights with each control point, enabling precise representation of conic sections such as circles and ellipses, which standard B-splines cannot accurately model. The non-uniform spacing of their knots provides greater control over localized curve behavior, making NURBS highly adaptable for complex and precise applications. Their ability to combine local control, high continuity, and accurate geometric representation makes them essential in CAD systems and other tools where both flexibility and precision are required [53].

NURBS are the most general form of B-splines, combining the strengths of high continuity, local control, and adaptability with the ability to accurately model complex and standard geometries. This makes them a preferred choice in high-precision industries [53].



**Figure 2.5:** a) Nonrational B-spline curves with different degrees, b) Comparison of nonrational and rational B-spline curves (Source: [52])

B-splines are a versatile framework for creating smooth curves and surfaces, offering flexibility through their control points and knot vectors. Within this framework, a special case arises when the degree of the curve equals the number of control points minus one, and the knot vector is clamped at both ends. Under these conditions, the B-spline curve simplifies to what is known as a Bézier curve [68].

### 2.3.2 Bézier curves

Bézier curves are widely used in computer graphics to represent free-form shapes due to their simplicity, flexibility, and efficiency. A Bézier curve is a polynomial curve defined by a set of control points, where the curve interpolates the first and last control points and is influenced by the intermediate points [69]. For cubic Bézier curves, the curve starts at the first control point ( $P_0$ ) and ends at the last ( $P_3$ ), with its initial and final tangents determined by the vectors  $3(P_1 - P_0)$  and  $3(P_3 - P_2)$ , respectively [53].

These curves are particularly useful because they are bounded by the convex hull of their control points, making them predictable and easy to manipulate. Bézier curves are  $C^1$ -continuous, ensuring smooth tangential transitions, and can achieve  $C^2$ -continuity with properly placed control points when forming splines. Furthermore, their parametric form allows for efficient rendering, subdivision, and approximation techniques, making them a cornerstone of modern design tools and applications such as Adobe Illustrator and font rendering [69].

For example, in the work of Darwiche and Nyström [29], Bézier curves are used to represent roads due to their smoothness and ease of manipulation. The convex hull property of Bézier curves ensures that the curve remains within the bounds defined by its control points, facilitating efficient collision detection, a crucial aspect in junction generation proposed in their work. Additionally, Bézier curves allow intuitive control over road shapes, enabling the creation of complex road networks with smooth transitions and intersections.

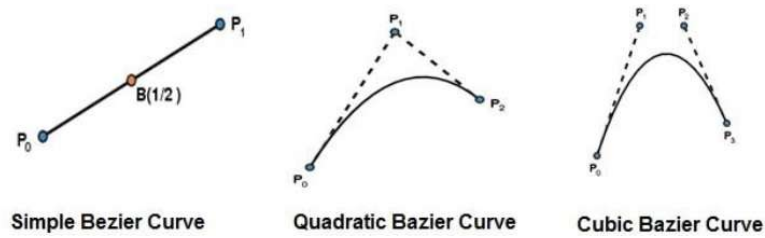


Figure 2.6: Different degree Bézier curves (Source: [87])

### 2.3.3 Catmull-Rom splines

Catmull-Rom splines are a type of  $C^1$  **interpolating curve** designed to pass through a series of control points while ensuring smoothness between segments. Each segment of the spline is a cubic polynomial determined by four control points, with adjacent segments sharing three points, which provides local control over the curve's shape. At each control point, the tangent vector is computed as a scaled difference between the neighboring control points, ensuring  $C^1$  continuity. For example, the tangent at a point  $P_i$  is computed as  $v_i = \frac{1}{3}(P_{i+1} - P_{i-1})$ , allowing the curve to smoothly interpolate the control points [53].

Catmull-Rom splines are a special case of cardinal splines where the tension parameter  $t$  is set to 0, producing a balance between smoothness and responsiveness to control points. The result is a curve that interpolates the control points with minimal overshooting and retains local control, meaning changes to one control point affect only the neighboring segments. This locality makes them computationally efficient and suitable for animation paths or motion trajectories, as only a fixed number of computations are required for any point on the curve, regardless of the total number of control points [69].

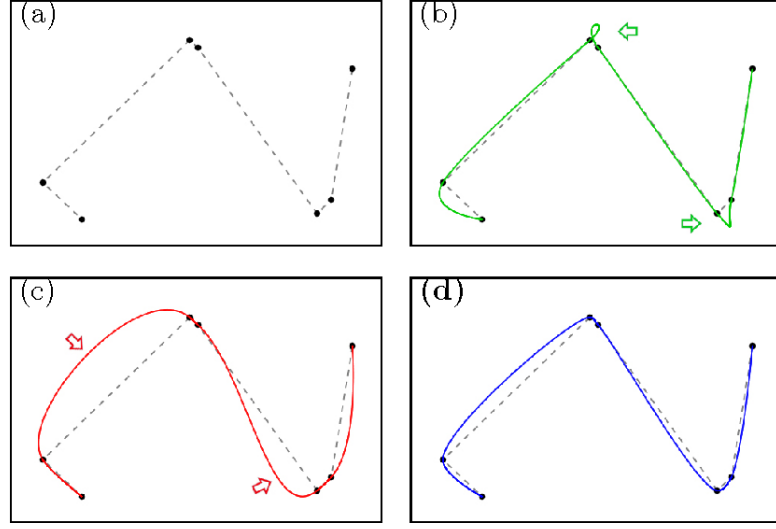
Although Catmull-Rom splines provide smooth interpolations, their continuity is limited to  $C^1$ , which means that abrupt changes in curvature can occur at control points. This property makes them ideal for animations where exact interpolation is required but less suitable for applications such as physically realistic motion paths, where higher degrees of continuity, such as  $C^2$ , are preferred [53].

### ■ Parametrization types of Catmull-Rom splines

The behavior of Catmull-Rom splines is significantly influenced by the parameterization method used, with three primary types being uniform, chordal, and centripetal parameterizations.

- **Uniform Parameterization:** In this method, the parameter intervals between control points are equal, regardless of the actual distances between the points. While simple to implement, uniform parameterization can lead to undesirable artifacts such as loops or self-intersections, especially when control points are unevenly spaced.
- **Chordal Parameterization:** This approach sets parameter intervals in proportion to the Euclidean distance between consecutive control points. By accounting for the spacing between points, chordal parameterization often produces curves that more accurately reflect the intended shape compared to uniform parameterization. However, it may still result in issues like overshooting or deviations from the control polygon.
- **Centripetal Parameterization:** In centripetal parameterization, the parameter intervals are proportional to the square root of the distance between control points. This method balances the influence of each control point more effectively, minimizing abrupt changes in curvature and reducing the likelihood of cusps and self-intersections within curve segments. Consequently, centripetal parameterization often produces smoother curves with gradual curvature transitions, resulting in visually appealing shapes that closely follow the control polygon [93].

For road design, centripetal parameterization is the most suitable parameterization for Catmull-Rom splines, as it minimizes abrupt changes in curvature, ensuring smoother transitions that align with safety and comfort requirements. A visual comparison of these parametrization types can be seen in Figure 2.7.



**Figure 2.7:** Different Catmull-Rom parameterizations: a) Control polygon, b) Uniform, c) Chordal, d) Centripetal (Source: [93])

### 2.3.4 Euler Spiral

An Euler spiral, also known as a clothoid or Cornu spiral, is a planar curve characterized by a curvature that increases linearly with its arc length. This unique property makes it particularly useful in applications that require smooth transitions between straight and curved paths [61].

Mathematically, the curvature  $k$  at a distance  $s$  from the origin is expressed as  $k = \frac{s}{A^2}$ , where  $A$  is a constant that determines the rate of curvature increase [50]. The Euler spiral is defined parametrically using the Fresnel integrals:

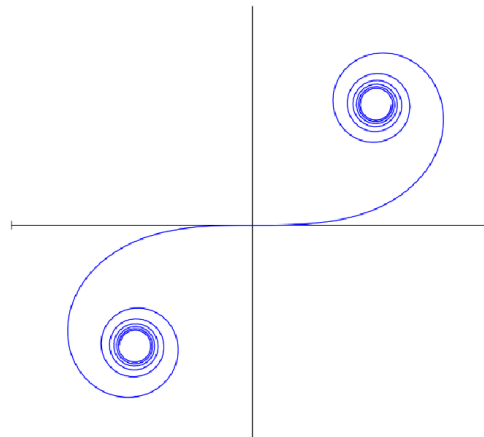
$$\blacksquare x(t) = \int_0^t \cos\left(\frac{\pi}{2}u^2\right) du$$

$$\blacksquare y(t) = \int_0^t \sin\left(\frac{\pi}{2}u^2\right) du$$

These integrals describe the curve coordinates as functions of the parameter  $t$ , illustrating how the curvature of the curve evolves with its length [92].

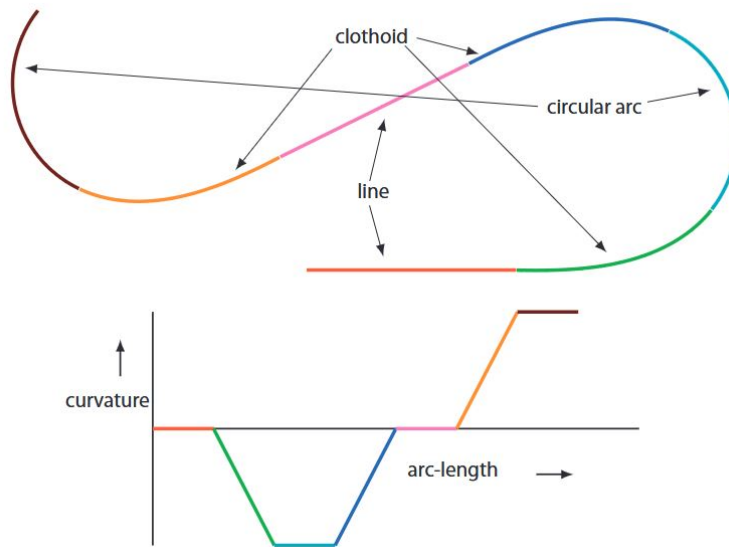
Euler spirals have significant applications in various fields. In road and rail engineering, they serve as transition curves between straight sections and circular arcs, following a principle similar to piecewise representations of splines (2.2.3). This approach allows for smooth connections between segments by utilizing their defining property: a linear change in curvature, which enhances safety and comfort by reducing sudden lateral acceleration [61].

Computationally, evaluating the Fresnel integrals required for plotting Euler spirals can be complex, as they do not have closed-form solutions. However, efficient numerical methods and approximations have been developed to compute these integrals with high precision, making the practical implementation of Euler spirals feasible in engineering applications [61].



**Figure 2.8:** Euler spiral (Source: [23])

In summary, the Euler spiral's mathematical properties and computational techniques ensure its utility in applications requiring smooth transitions, particularly in engineering contexts such as transportation infrastructure.



**Figure 2.9:** A curve composed of Euler spirals, line and circular-arc segments (Source: [63])

### ■ 2.3.5 Summary of Curves in Road Design

The paper *Spline Curves for Geometric Modelling of Highway Design* [26] reviews spline curves used in road design, focusing on their geometric properties and effectiveness in shaping highway alignments.

- **Bézier curves** are easy to control and computationally simple but may lack the flexibility and smoothness needed for complex designs.

- **B-Splines** offer better adaptability, enabling smooth transitions and accommodating geometric constraints.
- **NURBS** excel in precision and versatility, making them ideal for modeling complex road alignments and transitions.
- **Catmull-Rom splines**, while straightforward and interpolatory, lack the precision and flexibility of B-Splines and NURBS in road applications.

Overall, the paper highlights B-Splines and NURBS as the most effective tools for highway design due to their flexibility, precision, and capacity to handle complex geometries [26]. However, it does not address approaches that use piecewise combinations of Euler spirals, circular arcs, and straight segments (as seen in 2.9).

- **Euler spirals (Clothoids)** are the state-of-the-art for railway design [24], where smooth curvature transitions are critical for passenger comfort and safety. In road design, they are similarly effective. Nevertheless, their computational demands may limit their adoption in certain scenarios in favor of simpler alternatives like splines.

While curves like B-Splines, NURBS, and Euler spirals are essential for designing smooth and precise alignments in road infrastructure, their mathematical representations must eventually be translated into geometric models. This leads us to our next topics: polylines, which can approximate curves or define geometric outlines, and polygonal meshes, which represent objects and their surfaces in three-dimensional space.

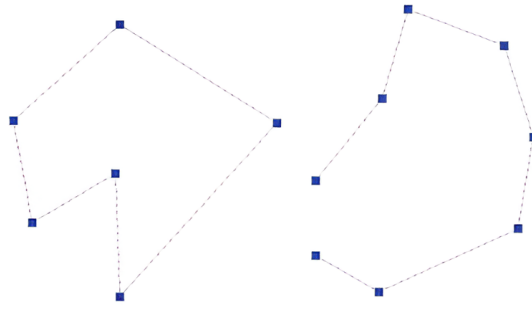
## 2.4 Polyline

A polyline (also referred to as a *path*, *piecewise linear curve* or *polygonal segment*) is a connected sequence of line segments, treated as a single geometric object (see Figure 2.10). Polylines are widely used to represent paths, boundaries, and other linear features in computer-aided design (CAD), computer graphics, geography information system (GIS), and other fields. They provide flexibility by allowing users to modify individual segments or the entire polyline as a whole, making them efficient for modeling and editing complex geometric shapes [14].

### 2.4.1 Approximating Curves with Polylines

In computational geometry and computer graphics, curve sampling is used for visualizing continuous curves on discrete digital screens. By sampling a curve at discrete points and connecting these points with line segments, a polyline approximation is created. This approach not only enables efficient rendering and processing of complex shapes but also simplifies computational algorithms by abstracting the underlying curve into a sequence of line segments. This abstraction allows for operations independent of the curve’s type [36].





**Figure 2.10:** A polyline composed of connected line segments (Source: [78])

Curve approximation techniques aim to balance visual accuracy and computational efficiency while maintaining the shape of the original curve. There are numerous ways to sample points along the curve. The most common way, uniform sampling, captures points evenly along the length. Alternatively, adaptive sampling using curvature captures more points in areas of higher curvature (see Figure 2.11) [36].

### ■ Douglas-Peucker Algorithm

The Douglas-Peucker algorithm simplifies a polyline by reducing the number of points. It works by finding the farthest point from the line connecting a segment's start and end points. If this point lies outside a set tolerance, the algorithm splits the segment at that point and continues recursively. If the point is within the tolerance, all intermediate points are removed. This method keeps only the points that significantly deviate from the start-end line, preserving the polyline's shape while reducing number of points of the polyline [31].

### ■ 2.4.2 Conclusion

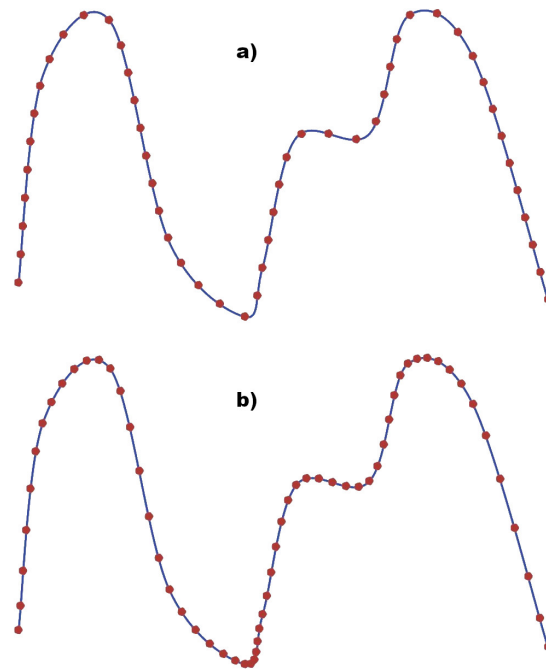
These approximation methods, integral to applications in computer graphics strike a balance between precision and performance, making them versatile tools for digital curve processing [36].

Although polylines effectively approximate curves in two dimensions, extending this concept into three dimensions allows for the creation of polygonal meshes that represent surfaces and volumes through interconnected polygons.

## ■ 2.5 Polygonal Meshes

A polygonal mesh is a representation of a 3D object using interconnected polygons, typically triangles, quads, or n-gons, to form its surface. Each mesh consists of three basic components: vertices, edges, and faces.

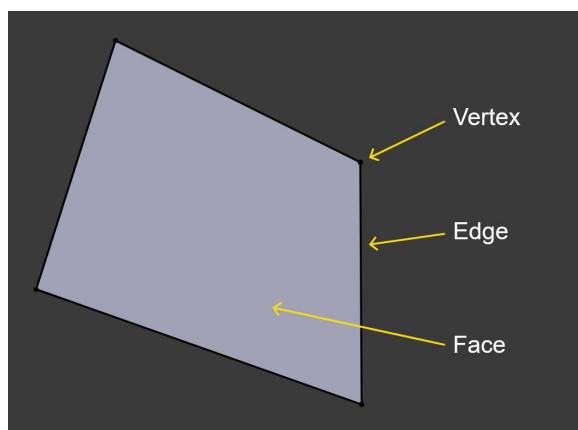
- **Vertices** are points in 3D space that define the shape's structure.



**Figure 2.11:** Curve approximation. Different sampling methods. a) Uniform sampling, b) Adaptive sampling based on curvature

- **Edges** are straight lines connecting two vertices.
- **Faces** are enclosed loops of edges that create the visible surface of the mesh.

These elements work together to form a network of polygons that represent the geometry of an object (see Figure 2.3) [53][69][21].



**Figure 2.12:** Polygonal mesh representation (Source: [21])

Triangle meshes, composed of triangular faces, are the most common 3D representation in computer graphics due to their simplicity, reliability, and

inherent planarity. Triangles provide reliable rendering and are easy to deform, which makes them ideal for tasks such as subdivision and simplifying models. While quads are sometimes used for their deformation properties in animation, they are typically converted to triangles for rendering, as most hardware is optimized for triangle processing. Their efficiency and compatibility make triangle meshes the standard for handling complex geometry in 3D applications [53][69].

### ■ 2.5.1 Mesh Triangulation

Triangulation refers to dividing a polygon into triangles such that no two triangles overlap, and the union of these triangles exactly covers the polygon. This creates a mesh useful for representing complex shapes and facilitating computations [30].

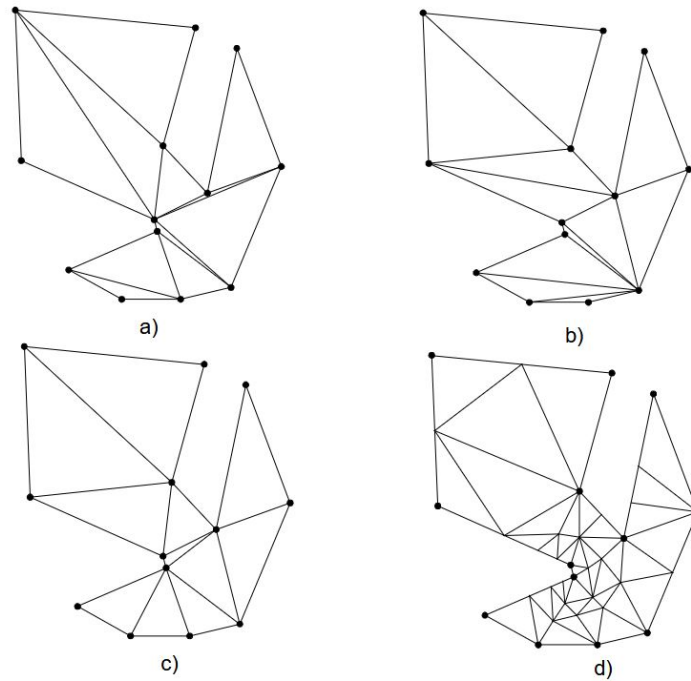
#### ■ Different methods for triangulation

Several methods exist for constructing triangulations, each tailored to different types of polygons or computational constraints:

- **Ear Clipping** is a straightforward approach that iteratively removes "ears"—triangles formed by three consecutive vertices—from the polygon until it is completely decomposed.
- **Incremental** algorithm dynamically adds triangles by introducing new vertices and updating the triangulation.
- **Divide-and-Conquer** method splits the polygon into smaller sub-polygons, triangulates each individually, and merges the results.
- **Sweep Line** uses a moving line to systematically identify diagonals and generate triangles, making it efficient for monotone polygons.
- **Delaunay Triangulation** creates triangles that maximize the minimum angle, avoiding skinny or elongated shapes. This makes it ideal for applications like finite element analysis, terrain modeling, and mesh generation, where stability and quality of the triangles are critical [30].

### ■ 2.5.2 Textures

Textures offer an efficient way to add visual detail to 3D models without increasing geometric complexity. At its core, a texture is a 2D image or pattern used to modify the appearance of a 3D surface. By mapping textures onto the polygons of a mesh, details such as colors, patterns, and even perceived depth can be applied, enhancing the realism of a model while keeping computational demands manageable [53].



**Figure 2.13:** Different triangulation methods. a) Ear cutting; b) Randomized incremental c) Constrained Delaunay triangulation; d) Delaunay refinement (Source: [60])

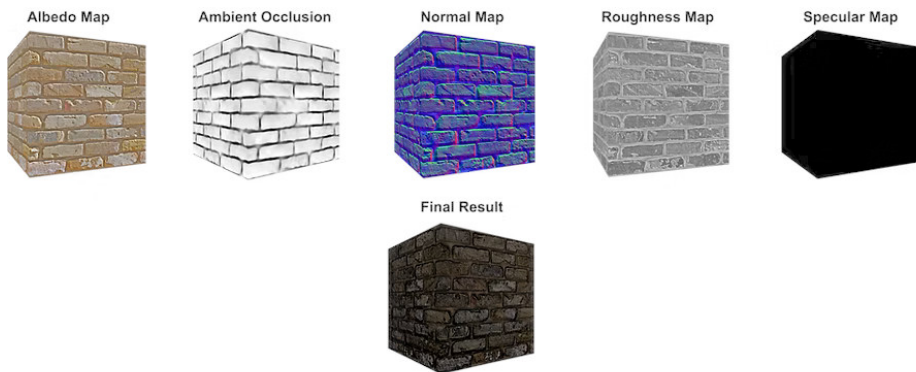
## ■ Texture Mapping and UV Coordinates

Texture mapping is the process of projecting a 2D texture onto a 3D surface. This requires establishing a correspondence between points on the texture (the 2D image) and points on the mesh's surface. This correspondence is typically defined using UV coordinates, a coordinate system where U and V represent the axes of the 2D texture space. Each vertex in a polygonal mesh is assigned UV coordinates, dictating how the texture is stretched or tiled across the surface. Usually, the UV coordinates are clipped between values in the range 0 and 1, and the texture is normalized to a unit square [53].

UV mapping can range from simple to highly complex. For example, a flat plane can use straightforward, evenly spaced UV coordinates, while intricate models like characters or vehicles often require careful UV unwrapping to avoid distortions. The quality of the UV map significantly impacts the effectiveness of the texture and its visual integration with the model [53].

### ■ 2.5.3 Export Formats

To transfer 3D models between different software applications, they must be saved in a compatible file format. Each of these formats has its own specifications, advantages, and limitations, making them suitable for different purposes. Some common 3D file formats include:



**Figure 2.14:** Different texture types visualized with a final result at the bottom (Source: [54])

- **Wavefront OBJ**, introduced by Wavefront Technologies, supports a variety of geometry types, including lines, polygons, and free-form curves, and is commonly represented in ASCII text format with the `.obj` extension. Materials are handled using an accompanying Material Template Library (`.mtl`) file, which defines surface properties like diffuse, ambient, and specular colors, along with transparency and texture mappings [65].

The OBJ format is now supported across almost all modern 3D software platforms, including Blender, Maya, and Unity, making it one of the flagship standards in 3D modeling and exchange workflows. Additionally, the format remains relevant despite its age due to its human-readable structure and ease of integration in pipeline scripts or software tools.

- **COLLADA** (Collaborative Design Activity) is an open-standard XML-based format developed by the Khronos Group. It supports geometry, materials, shaders, animations, and physics. COLLADA files use the `.dae` (Digital Asset Exchange) extension and store data in a structured, human-readable XML format [55].

The format is now widely supported in modern tools such as Blender and Unreal Engine, though its adoption has lessened with the rise of newer formats.

- **STL (Stereolithography)** The STL (Stereolithography) file format, developed by 3D Systems in 1987, is one of the earliest and most widely used file formats in 3D modeling and additive manufacturing. It is designed to represent the surface geometry of 3D objects using a tessellated structure of triangles, without including any color, texture, or material properties. This lack of additional information has limited STL's role to geometry-focused tasks. STL files can be stored in two formats: ASCII and binary, with binary being more compact and commonly used due to its smaller file size [62].

In its current state, STL remains a standard in 3D printing and prototyping workflows, favored for its simplicity and near-universal support

across 3D printing software and hardware [2].

- **FBX (Filmbox)** file format, originally developed by Kaydara in 1996 and later acquired by Autodesk in 2006, is a proprietary format widely used in the 3D industry for the exchange and storage of complex 3D assets. FBX supports a broad range of features, including geometry, textures, animations, skeletal rigs, and even scene hierarchies, making it a robust option for pipelines in game development, animation, and visual effects. It is compatible with both binary and ASCII encodings, with binary being more compact and commonly used for file storage and exchange [7].

FBX files are maintained and extended by Autodesk, with the most recent updates available through the FBX Software Development Kit (SDK), which provides developers with tools to parse and manipulate FBX data programmatically [7]. FBX remains a standard in production pipelines due to its widespread support in tools like Maya, Blender, Unity, and Unreal Engine.

- **GL Transmission Format (glTF)**, developed by the Khronos Group and introduced in 2015, is an open-standard file format designed for efficient transmission and rendering of 3D assets. The format is optimized for modern workflows, supporting compact storage through binary `.glb` files and extensibility with human-readable `.gltf` files. glTF 2.0, released in 2017, introduced support for Physically Based Rendering (PBR), animations, and scene hierarchies, making it ideal for real-time applications [56].

glTF minimizes runtime processing requirements by incorporating efficient compression and embedding data such as textures and animations directly into a single file or referencing external resources [56]. Its adoption has grown significantly due to its focus on modern rendering needs and its compatibility with tools like Blender, Unity, and Babylon.js.

## 2.6 Procedural Content Generation (PCG)

In the preceding sections, we established that defining a road using a curve or a polyline is a highly effective method for capturing their trajectory. This approach provides flexibility and precision, allowing user-defined input to guide the final geometry. We also discussed meshes and their role in representing surfaces within 3D modeling systems.

This chapter focuses on procedural content generation, particularly the techniques used to transform curves into mesh representations. Examining various approaches provides a more comprehensive understanding of how trajectory curves and procedural techniques work together to produce the final 3D geometry.

### ■ 2.6.1 What is Procedural Content Generation?

Procedural Content Generation (PCG) is defined as the automated creation of digital assets for applications such as games, simulations, and movies. This process relies on predefined algorithms and patterns, requiring minimal input from users to produce large-scale or complex outputs. Unlike manual modeling, PCG uses rules or constraints to generate assets such as textures, terrains, road networks, and even narratives [44].

PCG methods include **noise-based** algorithms for natural features like terrain, **grammar-based** systems for structured content such as cities or plant growth, and **agent-based** systems for dynamic or organic pattern creation. Additionally, **search-based** approaches use optimization techniques to adapt generated content to specific design requirements [44].

### ■ 2.6.2 Usecases of PCG

PCG is utilized in various domains to address challenges such as scalability, diversity, and budget constraints. Key applications mentioned by J. Freiknecht and W. Effelsberg [44] include:

- **Video Games:** For instance, PCG is integral in generating levels, terrains, and environmental details. These techniques enhance replayability by creating unique content for each session.
- **Simulations and Virtual Reality:** Generated virtual environments can support educational purposes, such as virtual driving schools, where randomized road networks help train users in different scenarios.
- **Urban and Natural Modeling:** PCG is also applied in urban planning, terrain generation, and vegetation modeling, for automated and large-scale creation of realistic environments.

### ■ 2.6.3 Advantages of PCG

PCG offers several benefits that make it indispensable in modern content creation:

- **Scalability:** It enables the generation of vast amounts of content efficiently, reducing the need for extensive manual work.
- **Customization:** Parameters allow users to control the generated output, tailoring it to specific requirements.
- **Diversity and Replayability:** PCG provides endless variations, ensuring unique user experiences and reducing repetition in content.
- **Efficiency:** By automating the creation process, PCG saves time and resources, allowing developers to focus on high-level design [44].

### ■ 2.6.4 Challenges of PCG

PCG also presents challenges, such as:

- **Control vs. Randomness:** Striking the right balance between procedural randomness and user-defined control can be difficult. Excessive randomness may lead to incoherent or unusable content, while rigid control may reduce diversity.
- **Integration Complexity:** Incorporating PCG seamlessly into workflows and game engines requires specialized tools and expertise.
- **Validation and Testing:** Ensuring the quality and functionality of procedurally generated content can be labor-intensive, as procedural systems may occasionally produce errors or undesired results [44].

### ■ 2.6.5 Generating Meshes using Curves

There are many different techniques that fall under the category of procedural geometry generation. This section, however, will focus only on methods that convert curves into detailed mesh representations. These techniques are particularly relevant for road design, as they allow for the transformation of road trajectory curves into realistic 3D models. These methods are:

- **Sweep Surfaces:** A sweep surface is generated by moving a curve, referred to as the profile  $C(u)$ , along another trajectory or applying a transformation rule  $T(w)$ . The transformation can involve translation, rotation, scaling, or shearing. The surface is mathematically expressed as  $P(u, w) = C(u) \cdot T(w)$ . Sweep surfaces are commonly used to create tubes, roads, or other extruded geometries [67].
- **Surfaces of Revolution:** A surface of revolution is a specific type of sweep surface formed by rotating a profile curve  $P(u)$  around an axis. The surface is defined by  $P(u, \theta) = P(u)T(\theta)$ , where  $T(\theta)$  is a rotation matrix applying the rotation transformation. This method is widely used for creating symmetric objects such as spheres, vases, and mechanical components [67].
- **Skinned Surfaces:** A skinned surface is created by interpolating or approximating a set of predefined curves, known as cross-sections or profiles. These profiles represent the shape of the surface at specific intervals. The resulting surface  $P(u, w)$  is computed as an interpolation between these curves. This technique is particularly useful when the surface needs to follow a complex path or vary in cross-section, such as in designing pipes or other intricate structures [67].



### 2.6.6 Conclusion

Procedural content generation offers a set of tools for creating complex and diverse 3D assets, including roads and their surroundings. By combining procedural techniques with curve-based representations, developers can efficiently generate detailed meshes that accurately represent road trajectories.

Multiple procedural methods, such as sweep surfaces, surfaces of revolution, and skinned surfaces, were discussed in this section. The most suitable approach for road design from these methods is sweep surfaces, as they allow for the extrusion of road trajectories along a path. This technique uses a profile curve to define the road's cross-section, enabling detailed and realistic road meshes to be created.

Now, with a solid understanding of geometry and its applications, we can explore techniques for representation and simulation of road networks, focusing on the road graph as a fundamental structure for modeling road infrastructure.



## Chapter 3

### Road Graph

A *road graph*, also called a *road network*, is a fundamental representation of road infrastructure, capturing the connectivity and topology of roads and junctions. This chapter explores the road graph's structure and different representations in road design, traffic simulation, autonomous driving, and navigation systems. After a concise overview of varying representation methods, the two most relevant road graph representations are discussed in more detail: the representation currently used in the VRUT system and the ASAM OpenDRIVE standard, which is to replace it inside VRUT in the future.

### 3.1 Standards of Data Representation

In the automotive industry, particularly within the domains of autonomous driving and advanced driver-assistance systems (ADAS), ensuring a standardized representation of road networks and environments plays important role for promoting effective communication and collaboration among diverse stakeholders. Standards, which are embraced as guidelines or specifications by industry organizations or corporations, establish a common language and structure. These standards facilitate the seamless exchange of data, promoting interoperability and accelerating innovation within the industry. [34]

We will now introduce some prominent standards and methods used for representing road networks and environments in the context of autonomous driving and navigation systems. The aim is to provide an overview of the existing standards that are outside this project's scope. The following sections will delve into the specifics of the VRUT system's road graph representation and the ASAM OpenDRIVE standard, which are most relevant to this project.

#### 3.1.1 OpenStreetMap (OSM)

OpenStreetMap is a collaborative project that creates a free, editable map of the world, built by volunteers through manual survey, GPS devices, aerial imagery, and other free sources.

OSM represents physical features on the ground—such as roads, buildings, and natural elements—using a system of tags attached to its basic data structures: nodes, ways, and relations. Each tag consists of a key-value

pair that describes a specific attribute of a feature, enabling detailed and customizable mapping. This flexible tagging system allows contributors to define an unlimited number of attributes for each feature, facilitating a comprehensive and nuanced representation of real-world objects. The community collaboratively agrees on certain key and value combinations for commonly used tags, which act as informal standards; however, users are encouraged to create new tags to capture previously unmapped attributes, enhancing the map's richness and utility [66].

### ■ 3.1.2 RoadXML

RoadXML is an XML-based file format (extension `.rnd`) designed for driving simulation applications. Developed through years of collaboration between universities and industry, it supports traffic simulation, scenario control, vehicle dynamics, and 3D road network generation [3].

Road networks in RoadXML consist of connected tracks and intersections, enhanced with additional layers such as road profiles, traffic descriptions, and environmental elements like road signs. It uses a graph-based structure to organize these features and supports user-defined data for customization. RoadXML's extensibility and structured design make it a valuable resource for simulating realistic road networks in various scenarios [3].

### ■ 3.1.3 Vector Map (VMap)

Vector Map (VMAP), also known as Vector Smart Map, is a geospatial data format developed by the National Geospatial-Intelligence Agency (NGA) to provide detailed vector-based representations of geographic features. It includes road networks, terrain features, hydrological systems, and points of interest. VMAP data is structured using the Vector Product Format (VPF) [91].

VMAP is available in multiple resolution levels: VMAP Level 0 (VMAP0) provides global coverage at a scale of 1:1,000,000 and is in the public domain, while higher-resolution levels such as VMAP Level 1 (1:250,000 scale) and VMAP Level 2 offer more localized and detailed data. This format supports applications in mapping, navigation, and geospatial analysis, offering comprehensive environmental context for a variety of use cases [91].

### ■ 3.1.4 Simulation-specific Formats

In addition to the above standards, there are simulation-specific formats used in autonomous driving research and development. Examples include the Unified System for Automation and Robot Simulation (USARSim) and the Simulation Open Framework Architecture (SOFA). USARSim is a high-fidelity simulation platform based on the Unreal Tournament game engine, designed for research in robotics and automation. It provides a realistic virtual environment for testing autonomous vehicle algorithms and systems under various scenarios [27].

SOFA, on the other hand, is an open-source framework primarily targeted at real-time physical simulation, with a focus on medical and robotics applications. It is widely used in research for prototyping and developing new simulation algorithms, as well as serving as a robust physics engine [90].

These simulation frameworks help researchers and developers test and improve autonomous driving systems in realistic virtual environments.

## ■ 3.2 VRUT representation

The VRUT system currently uses a custom representation for road networks, which uses a polyline-based structure to model the road network, with nodes representing road segments and edges representing connections between them. The system also incorporates additional attributes to capture lane information, road attributes, and traffic regulations. A visualization of the VRUT road graph representation is shown in Figure 5.2. Information provided in this chapter is gathered from examples of road networks and from a code of a VRUT format parser, defined in VRUT's *traffic module*. An example of a VRUT road graph XML representation is shown in Appendix A.

### ■ 3.2.1 Roads Section

At the base level, this road graph representation uses a hierarchical structure to organize road elements, with roads containing lanes and lanes containing nodes.

- **Roads** are the highest level of the hierarchy, representing the entire road segment. Roads are defined by an identifier and a type (e.g., "motorway"=0, "first-class road"=2, "second-class road"=3, etc.).

```
<road id="5" type="0">
```

- **Lanes** represent, as the name suggests, the lanes of the road. Each lane has a unique identifier, node count, a type (e.g., highway lane, highway exit, turning lane) and some optional attributes.

```
<lane id="0" nodes="3053" type="0" level="3" o="1">
```

- **Nodes** represent points along each lane that guide the path of vehicles. Edges later connect these nodes to form a continuous path. To define the lane geometry, they have multiple attributes, such as position, curvature, and lane width.

```
<node id="1" x="-7.67172" y="-21.0457" z="1" sl="120"
    w="3.7" r="10880.5" r2="10000" vmax="120" vdop="120" />
```

### ■ 3.2.2 Attributes Section

After all the road elements are defined with all the nodes' attributes set, the attributes section allows for the bulk setting of attributes for nodes. This section is handy for simplified setting of optional attributes across multiple nodes and is defined between the `<attributes>` tags. An example of how this section may look is:

```
<attributes>
  <level from="1" to="154" value="4" />
  <overtakable from="1" to="154" value="1" />
  <speedlimit from="1" to="154" value="130" />
  <vmax from="1" to="154" value="130" />
  <vdop from="1" to="154" value="130" />
  <navevt from="1" to="154" value="2" />
</attributes>
```

### ■ 3.2.3 Connections Section

This section, defines all the connections between different nodes, as well as defining junctions. The nodes already defined in the *Roads section* are not automatically interconnected between each other, even though they are a part of the same lane and the same road. That is partly because the nodes defined in each lane and the lanes defined in each nodes is not strictly ordered in a sorted manner, and even if that is most of the times the case, it's not a rule.

There are three different ways to define connections between nodes of roads and before they were defined, all the connection has to be defined manually, much like nodes are:

- **Sequence:** Defines, as the name suggests, a sequence of connections given a start and end index of nodes. Nodes are expected to be indexed in order, in which they appear in the road.

```
<sequence from="1" to="3053" dir="2" closed="1" />
```

- **Interconnection:** Characterizes connections between parallel lanes, to allow changing lanes and overtaking. Takes in start indices of nodes on both lanes, count of nodes that should be interconnected and direction in which the connections should be added.

```
<interconnection from="1" to="3054" count="3053" dir="2" />
```

- **Connection Group:** Used to group multiple connections into one structure and define default connection type for all of them (left, right, backwards, forwards). Connections inside this group connect two different nodes together and the default connection type for this group can be overridden for each connection.

```

<connectiongroup type="0">
  <connection from="6296" to="6297" type="0"/>
  <connection from="6297" to="6296" type="1"/>
</connectiongroup>

```

### ■ 3.2.4 Junctions

Each junction is encapsulated by the `<junction>` tag, which includes details about its auxiliary paths, interconnections between lanes, and priority rules. This tag is defined within the connections section. Below, the key components of a junction definition are explained:

Each junction is identified by its name and a unique ID, as shown in the following example:

```
<junction name="T-crossing" id="1">
```

#### ■ Auxiliary Paths

Auxiliary paths within junctions are defined using the `<path>` tag. Each path connects specific nodes representing points in the junction, with attributes describing the path's geometry and operational constraints:

```

<path id="0" nodes="2" type="11">
  <node id="0" x="5659.1785" y="-16528.178" z="29.743277"
    sl="30" w="3.6" vmax="30" vdop="30"/>
  <node id="1" x="5663.2515" y="-16522.115" z="29.743277"
    sl="30" w="3.6" vmax="30" vdop="30"/>
</path>

```

Attributes of the `<path>` tag include the path's ID, the number of nodes it contains, and its type (general connection, left & right turn, etc.). Each node within the path is defined by its ID, position (x, y, z), speed limit (sl), width (w), maximum speed (vmax), and design speed (vdop).

#### ■ Lane Interconnections

The `<laneLink>` tag specifies how lanes from different roads are interconnected within the junction:

```

<laneLink road="3" lane="1" node="11572">
  <successor road="3" lane="0" node="11576" path="0" />
</laneLink>

```

Attributes of the `<laneLink>` tag include the road and lane IDs, as well as the node ID where the connection occurs. The `<successor>` tag defines the connected road, lane, and node, along with the path ID which the connection follows.

### ■ Priority Rules

Priority rules are critical for managing vehicle precedence at junctions. They are defined using the `<priority>` tag, with `<check>` elements specifying lanes and nodes with higher priority:

```
<priority road="3" lane="0" node="11589">
  <check road="3" lane="0" node="11557" />
  <check road="3" lane="1" node="11572" />
</priority>
```

Attributes of the `<priority>` tag include the road, lane, and node IDs where the priority rule applies. The `<check>` elements define lanes and nodes that have priority over the specified lane. In the example above, nodes 11557 and 11572 have priority over node 11589 during traversal.

### ■ 3.2.5 Summary of VRUT representation

The VRUT road graph representation uses polyline-based structures to define road network. It uses hierarchical organization to model roads, lanes, and nodes. Attributes for nodes such as overtakability, level or speed-limit can be set for each node separately and overridden in bulk in attributes section. Connections section defines edges between nodes, representing the connectivity of the road network, as well as junctions with all their attributes.

This representation is intuitive and efficient for modeling simple road networks, but it may seem sporadic when dealing with complex road structures, such as intersections with multiple lanes and traffic rules. Some of the necessary attributes may feel redundant (such as a necessity manually adding connections between nodes that are already ordered and grouped inside lanes).

To address these limitations, the VRUT system (as well as this text) is transitioning to the ASAM OpenDRIVE standard, which provides a more complete and standardized representation of road networks and environments.

## ■ 3.3 ASAM OpenDRIVE

All the information provided in this section is based on the latest ASAM OpenDRIVE standard documentation [4].

ASAM OpenDRIVE, developed by the Association for Standardization of Automation and Measuring Systems (ASAM e.V.), is an XML-based standard for describing static road networks in driving simulation applications. It provides a framework to represent roads, lanes, signals, junctions, and other roadside objects such as traffic islands. By focusing on static elements, ASAM OpenDRIVE facilitates simulation scenarios while excluding dynamic elements like vehicles and pedestrians.

The specification promotes compatibility and standardization, enabling seamless data exchange between various simulation tools.



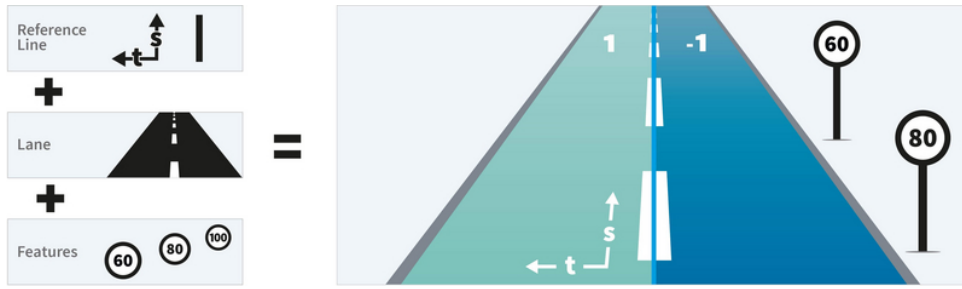


Figure 3.1: OpenDRIVE reference line with other features attached to it [5]

### 3.3.1 File Structure

ASAM OpenDRIVE files are structured with the root element `<OpenDRIVE>` encapsulating all data. The mandatory `<header>` element contains metadata such as the ASAM OpenDRIVE version and default regulations.

Key elements include:

- `<license>`: Stores licensing information about the file.
- `<defaultRegulations>`: Specifies default road parameters like speed limits or overtaking rules, applicable unless overridden by specific definitions.

Files are stored using the `.xodr` extension for uncompressed XML files, while compressed files use the `.xodrz` extension, following the `gzip` format. The standard also supports extensibility through the `<userData>` element for custom data and the `<include>` element for integrating external files.

An example of an ASAM OpenDRIVE file is provided in Appendix B. This example demonstrates a road composed of a straight section, an Euler spiral, and a curved segment. The combination of these elements highlights the flexibility and detail achievable with the OpenDRIVE format.

### 3.3.2 Road Reference Line

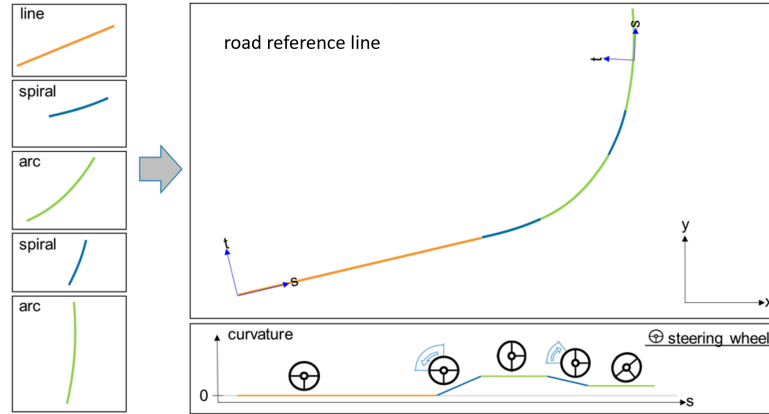
The road reference line is a central element in ASAM OpenDRIVE, defining the road's geometry and alignment. It runs along the *s*-axis, with lateral deviations in the *t*-direction. Elevation changes are represented along the *h*-axis. The reference line serves as the foundation for defining road segments and features, ensuring consistency and alignment across the network.

Furthermore, the road reference line forms the foundation for individual road definitions, serving as the central axis along which road segments are described.

The reference line is constructed using a series of geometric elements, which will be discussed in the next section.

### 3.3.3 Geometries

ASAM OpenDRIVE defines road layouts using following geometry elements:



**Figure 3.2:** ASAM OpenDRIVE: Road geometry representation [4]

- **Straight Lines:** Straight road segments are represented by the `<line>` element with attributes specifying the start position, heading angle, and length.
- **Spirals (Euler Spirals/Clothoids):** Spirals transition smoothly between curves of varying curvature. They are defined using attributes for start and end curvature.
- **Arcs:** Arcs represent road segments with constant curvature.
- **Parametric Cubic Curves:** Parametric cubic curves provide flexibility for modeling complex road shapes using coefficients for lateral and longitudinal splines. These curves allow precise modeling of continuous curves, making them suitable for approximating smooth polylines.

This representation is similar to the one discussed in section 2.3.4, where Euler spirals are used to model smooth transitions between straight and circular road segments. We can compare the OpenDRIVE representation (see Figure 3.2) with the one discussed in that section (see Figure 2.9) and see the similarity. The addition of this representation is parametric cubic curves, which provide a more detailed and flexible approach to modeling complex road shapes.

### ■ 3.3.4 Roads

Roads are the primary elements in ASAM OpenDRIVE, serving as the structural backbone of any road network. Defined along the road reference line, roads can represent continuous stretches, smaller segments between junctions, or even multiple distinct roads. Each road is represented by a `<road>` element and includes a unique set of properties and structural details.

Road linkage ensures continuity by connecting roads either directly or via junctions. This linkage is defined through `<link>` within `<road>` elements, specifying how roads connect and ensuring logical navigation across the network. Roads must be explicitly linked to maintain network consistency.

Each road type specifies its primary function and traffic rules, such as motorways or rural roads. A road type remains valid until another type is explicitly defined or the road ends.

Elevation changes along a road are handled by defining the elevation profile along the reference line. This allows roads to model realistic terrain, including hills and slopes, while ensuring a smooth transition in elevation. Additionally, cross-sections divide the road into segments that describe properties like superelevation and lane-specific details across the road's width.

Key attributes of a road include:

- an **ID**, which uniquely identifies the road;
- a **junction** identifier (-1 if the road does not belong to a junction);
- the **length** of the road;
- an optional **name** for descriptive purposes; and
- the **traffic rule**, specifying driving direction (e.g., Left-Hand Side or Right-Hand Side).

By integrating all these aspects, roads in ASAM OpenDRIVE form a coherent and robust framework for representing diverse and complex road networks, providing the foundation for lanes, junctions, and additional road-related elements.

### ■ 3.3.5 Lanes

Lanes are essential components of roads and are defined relative to the road reference line. Each road must have a center lane, which is a reference for lane numbering. The centre lane (ID 0) has no width and aligns with the road reference line by default. Additional lanes are numbered relative to the center lane: lanes on the left (positive  $t$ -direction) are numbered with positive integers, while lanes on the right (negative  $t$ -direction) are assigned negative integers.

For ease of organization, lanes are grouped into left, center, and right categories based on their position relative to the center lane. Lanes are further divided into lane sections, which segment the road whenever the number or characteristics of lanes change. Lane sections ensure consistency in lane definitions over the length of a road.

Lane geometry incorporates parameters such as width, lateral offset, and elevation, which can vary along the road. A lane offset can shift the center lane laterally, allowing for more complex road designs, such as turn lanes. Lane borders and widths define the boundaries of individual lanes, with widths adjustable along the road's length.

Linkage information for lanes, defined using `<predecessor>/<successor>` elements, connects lanes across road segments and ensures smooth navigation. Lane linkage is independent of driving direction, enabling flexible connections in diverse scenarios.

Additional properties include lane type (e.g., driving, parking, shoulder), speed limits, access restrictions, and surface material. Road markings, defined with `<roadMark>`, further enhance lane definitions by specifying line styles, colors, and visibility.

### ■ 3.3.6 Junctions

Junctions in ASAM OpenDRIVE provide a structured way to connect multiple roads, enabling complex traffic scenarios. They can be categorized based on their functionality:

- **Common Junctions:** The default type, these allow traffic to cross using overlapping drivable lanes, such as ordinary intersections or junctions with traffic lights.
- **Direct Junctions:** Used for entries and exits without crossing traffic, reducing the complexity of connecting roads. These are ideal for slip roads and certain highway interchanges.
- **Virtual Junctions:** Model connections within uninterrupted roads, such as driveways or parking lot entries, without altering the main road geometry.
- **Crossings:** Allow roads to intersect at the same level without enabling vehicles to change roads, e.g., railway or pedestrian crossings.

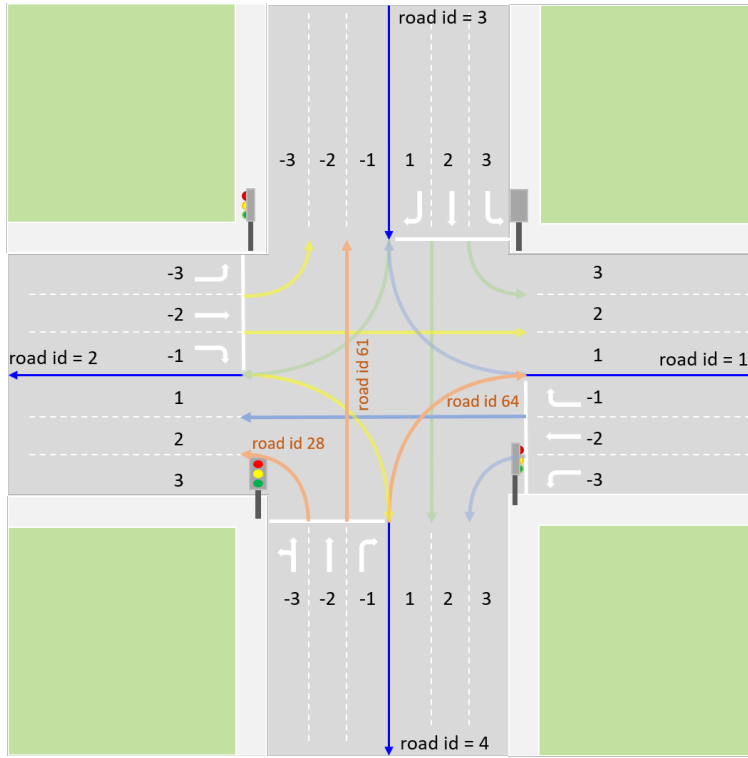
#### ■ Junction Components:

Each junction consists of several key components. **Incoming roads** lead into the junction and may also function as outgoing roads. **Connecting roads** define paths through the junction, linking lanes of incoming roads to those of outgoing roads, and are modeled similarly to standard roads. Additionally, some junctions may include **cross paths**, which add direct connections between specific lanes, enabling more precise traffic flow management.

#### ■ Additional Features:

Junctions incorporate additional features to handle more complex scenarios. **Junction boundaries** define the extent of the junction, including sidewalks and similar areas, forming a closed boundary. For cases where roads with different elevations meet, **elevation grids** provide smooth transitions by interpolating height data within the junction area. Junctions can also be grouped into **junction groups** to represent constructs such as roundabouts or complex interchanges. To improve traffic management, **signal synchronization groups** enable the coordination of multiple traffic signals within a junction.

By combining these elements, ASAM OpenDRIVE ensures that junctions can model a wide variety of real-world traffic scenarios, from simple intersections to highly complex interchanges.



**Figure 3.3:** ASAM OpenDRIVE: Junction example [5]

### 3.3.7 Summary of ASAM OpenDRIVE

ASAM OpenDRIVE uses a detailed geometric representation to define road layouts, including Euler spirals for smooth curvature transitions. This method is more precise than polyline-based approaches, such as the one used in VRUT, and better suited for representing precise road graph in simulation environments.

The standard allows for the representation of complex road networks, including roads, lanes, junctions, and various roadside features. Its flexibility and structured format make it ideal for creating realistic simulations and ensuring compatibility between different systems. ASAM OpenDRIVE effectively handles everything from simple roads to complex interchanges, making it a reliable foundation for modern driving simulations.

## 3.4 Conclusion

The main difference between VRUT format and OpenDRIVE standard is the approach to storing geometry of the road network. Where VRUT stores nodes and edges between them, much like polyline representation of a curve, OpenDRIVE stores parametric curve sections with three main forms: line segment, spiral, circular section, meaning that OpenDRIVE representation is much more precise where VRUT representation is more of an approximation,

but is easier and more intuitive to implement and model.

The way of storing attributes also differs. VRUT stores custom attributes directly within nodes, offering simplicity but limited flexibility for complex scenarios. OpenDRIVE uses a hierarchical structure, associating attributes with specific road elements like lanes or junctions, allowing for greater precision and modularity. While VRUT is easier to implement and suitable for straightforward networks, OpenDRIVE excels in representing complex geometries and advanced configurations.

## Chapter 4

### Implementation Suitable Frameworks

This chapter examines various frameworks considered during the selection process for implementing the RoadBlocks editor. The editor’s functionality focuses on two primary outputs: road geometry and the road graph. The ideal framework should seamlessly handle geometry and mesh operations while also supporting the definition of custom data structures and variables required for the road graph.

In addition to VRUT, which emerged as the most convenient choice for future developers and users, two other categories of applications were evaluated: Game Engines and Geometry Modeling Software. Both categories are inherently suited for working with geometries, satisfying the primary requirement. Game Engines are also highly programmable, and Geometry Modeling Software typically provides interfaces that enable developers to create custom add-ons.

An additional factor influencing the final framework selection was proficiency and prior knowledge of the framework, as familiarity can significantly streamline the development process and reduce onboarding efforts.

#### 4.1 VRUT

Given that the output of RoadBlocks editor must be compatible with VRUT, a detailed description of VRUT will be provided in Chapter 5, regardless of whether it is ultimately selected as the implementation framework. Please refer to that chapter for a comprehensive explanation of VRUT’s internal mechanisms. This section will focus solely on aspects pertinent to the framework selection process.

Since the final results are intended to integrate seamlessly with VRUT, and developers working within VRUT are the primary users of the editor, VRUT represents the most convenient option from a user perspective. Implementing the editor within VRUT would eliminate users needing to install or familiarize themselves with additional software. Furthermore, this approach would streamline workflows by removing the necessity of exporting results from external tools and subsequently importing them into VRUT.

However, VRUT also presents significant challenges as an implementation framework. It is the most complex option to work with due to the lack

of prior development experience, the absence of recent documentation, and the scarcity of individuals available to provide active support during the development process. Additionally, VRUT offers limited built-in tools for handling geometries compared to alternative frameworks. Consequently, most, if not all, of the required geometric methods would need to be implemented from scratch.

## ■ 4.2 Game Engines

Game engines provide a balance between precise geometry programming and the convenience of predefined functions found in modeling software. They offer a steeper learning curve but compensate with extensive documentation, active community support, and a wealth of online resources, which can simplify the development process.

These frameworks are highly programmable, supporting custom features like road geometry generation and road graph construction through scripting capabilities. Game engines also excel in rendering and interaction, enabling real-time visualization of geometries and graphs. Additionally, built-in tools, such as physics engines or pathfinding systems, may be repurposed to enhance the editor's functionality.

However, their broad functionality and abstraction can add complexity for geometry-focused tasks. Moreover, integration with VRUT would still require export-import workflows, potentially complicating the user experience. Despite these challenges, game engines offer a versatile platform that balances flexibility and capability for developing the RoadBlocks editor.

### ■ 4.2.1 Unreal Engine

Unreal Engine is a real-time 3D creation tool widely used for simulation, game development, and visualization. Its framework provides tools for representing, creating, and exporting geometric models, supporting procedural content generation [47].

#### ■ Representation of Objects

Unreal Engine supports splines and static meshes for representing objects. The *Spline Component* allows the definition and manipulation of paths or shapes in 3D space, useful as the skeleton for road geometry [48]. Static and procedural meshes can be used to represent road surfaces and nearby objects, enabling detailed modeling [47].

#### ■ Mesh Creation and Procedural Content Generation

Mesh creation in Unreal Engine includes manual tools and dynamic tools like the *Procedural Mesh Component*, which generates meshes at runtime [46]. This capability is suited for generating road surfaces and procedurally placing



roadside features such as trees and grass based on spline data or custom algorithms [47].

### ■ Storing Custom Properties

Unreal Engine allows storing custom metadata through *Actor Components* and Blueprint variables. Properties like speed limits and overtaking permissions can be added to splines or meshes, making road attributes accessible for further simulation [47].

### ■ Exporting Geometry and Meshes

Unreal Engine supports exporting static and skeletal meshes, as well as animations [47]. Procedural geometry generated at runtime can also be exported using custom scripts or plugins, ensuring compatibility with external systems.

## ■ 4.2.2 Unity

Unity is a widely used game engine that supports the creation of real-time 3D content for games, simulations, and visualizations. It offers robust tools for representing, creating, and exporting geometric models, as well as procedural content generation [85].

### ■ Representation of Objects

Unity uses Splines and Meshes to represent objects. The *Spline Package* in Unity allows for the definition of curves that can act as the skeleton for roads [84]. Static meshes are supported through Unity's *MeshFilter* and *MeshRenderer* components, enabling the visualization of geometric shapes [85].

### ■ Mesh Creation and Procedural Content Generation

Unity provides tools for both manual and automated mesh creation. Meshes can be created using the *ProBuilder* package [82], which supports manual modeling and editing. Procedural mesh generation can be achieved through scripting via the *Mesh* class, enabling runtime generation of road surfaces and procedural placement of roadside details such as trees and grass [81].

### ■ Storing Custom Properties

*ScriptableObjects* allow for storing custom metadata, and can be attached to objects to define attributes like speed limits, overtaking permissions, and lane-specific details. These properties can also be serialized for use in simulations and further processing [83].

## ■ Exporting Geometry and Meshes

Unity supports the export of models through various formats, such as FBX, using the *FBX Exporter* package [80]. Additionally, procedurally generated meshes can be serialized and exported via scripting, ensuring compatibility with external tools and platforms.

## ■ 4.3 3D modeling software

Modeling software is highly specialized for geometric operations, making these frameworks particularly effective for tasks centered on precise geometry manipulation. Since geometry is the most complex output of the editor, these tools offer an advantage with their robust feature sets designed for creating, editing, and exporting geometric models. Additionally, they streamline the geometry export process, a core functionality for such software.

A significant strength of modeling software lies in its procedural and visual programming capabilities, which enable the development of complex geometries through intuitive interfaces or scripting. This allows for greater control and efficiency when generating intricate road geometries. Moreover, these frameworks often support defining custom data structures and exporting them alongside geometry, facilitating the integration of road graphs into the workflow.

The active communities and comprehensive documentation associated with modeling software provide further benefits, offering ample support and resources for development. Their built-in tools and extensibility through scripting make them versatile for addressing specific requirements, such as defining road networks. Combined with their focus on geometry and ease of exporting, modeling software presents a compelling option for developing the RoadBlocks editor.

### ■ 4.3.1 Blender

Blender is a versatile open-source 3D creation project widely used for modeling, animation, and simulation. It offers robust tools for representing, creating, and exporting geometric models, supporting procedural content generation, and allows for extensive customization through its Python API [15].

## ■ Geometry Representation and Procedural Modeling

Blender utilizes splines and meshes to represent objects. Meshes, consisting of vertices, edges, and faces, form the basis for most 3D objects and can be extensively edited for fine-tuned geometry. Curves and surfaces, such as NURBS and Bézier curves, provide flexible representations for smooth and flowing forms, ideal for defining paths or skeletons for procedural modeling. These structures are enhanced by modifiers and the *Geometry Nodes* system, which supports procedural manipulation and complex object generation [19, 42].

### ■ Mesh Creation and Custom Attributes

Blender's *Geometry Nodes* system enables procedural mesh generation through a visual node-based programming approach [42]. For advanced geometry manipulation, Blender's Python API provides the `bmesh` module, offering direct access to Blender's internal mesh editing capabilities [41]. Additionally, Blender supports custom attributes on geometry through the *Geometry Nodes* system or via Python scripting. The `bpy` module facilitates the creation and manipulation of user-defined properties for various elements [40].

### ■ Add-on Development and Exporting

Blender's Python API supports the development of add-ons with custom graphical interfaces, enabling developers to create panels, menus, and operators to extend functionality. Comprehensive tutorials guide users through the process of developing tools with integrated GUIs [39]. Blender also supports exporting geometry and meshes in various formats, such as OBJ, FBX, and STL, which can be accessed via the user interface or automated through Python scripting [43]. These features ensure seamless integration with external workflows and applications.

### ■ Strengths and Accessibility

Blender's open-source nature and free license make it highly accessible for personal and professional use. Its combination of procedural generation capabilities, extensive scripting support, and robust export options make it a powerful tool for implementing road network modeling tasks. However, its steep learning curve for advanced procedural workflows may pose challenges for new users, particularly those unfamiliar with its node-based and scripting environments [19].

#### ■ 4.3.2 Houdini

Houdini is a high-end 3D animation and visual effects software renowned for its node-based procedural workflow. Widely used in film, gaming, and simulation industries, it provides a robust toolset for road network modeling, including procedural generation, attribute handling, and export capabilities [75].

### ■ Geometry Representation and Modeling

Houdini uses splines (referred to as curves) and geometry nodes to represent objects. The *Curve SOP* facilitates the creation and manipulation of curves, which can act as skeletons for road systems [71]. Geometry in Houdini is represented through procedural networks of nodes, enabling detailed and customizable modeling of road surfaces and related structures [74].

### ■ Procedural Mesh Creation and Custom Attributes

Houdini excels in procedural content generation through its node-based architecture. Tools like the *PolyExtrude SOP* and *Sweep SOP* allow developers to dynamically create meshes from curves [76]. Custom attributes, such as speed limits, overtaking permissions, and lane-specific details, can be added and manipulated through nodes or scripting, ensuring compatibility with simulations and external tools [70]. The *VEX* and *HScript* scripting systems further enhance procedural workflows, enabling the automated placement of roadside elements such as trees, grass, and ditches [77].

### ■ Tool Development and Exporting

Houdini provides extensive support for custom tool development through *Digital Assets* (HDA), allowing developers to package procedural systems into reusable tools with custom graphical interfaces [72]. Exporting capabilities include industry-standard formats like FBX, Alembic, and USD, ensuring seamless integration with external platforms such as VRUT. Additionally, procedural geometry can be baked into static models or exported dynamically [73].

### ■ Strengths and Limitations

Houdini's node-based procedural workflow offers unparalleled flexibility and scalability, making it a powerful tool for road network modeling tasks. However, the software's commercial nature poses accessibility challenges, as its full functionality requires a commercial license. While a free version, Houdini Apprentice, is available, it comes with significant feature limitations, making it less suitable for widespread use in accessible frameworks. Moreover, Houdini's steep learning curve and complexity may present additional challenges for developers unfamiliar with its workflow [75].

#### ■ 4.3.3 Maya

Autodesk Maya is a professional 3D modeling and animation software widely used in the film, gaming, and visualization industries. It provides a robust toolset for modeling, scripting, and exporting geometry, making it a potential candidate for road network modeling tasks [8].

### ■ Geometry Representation and Modeling

Maya supports object representation through NURBS curves and polygon meshes, which can be combined and manipulated with precision to model road networks [11]. The Modeling Toolkit offers features like extrusion, beveling, and bridging, providing flexible tools for creating and refining geometry. Advanced tools like *Bifrost* enhance procedural workflows, enabling the dynamic creation of road surfaces and roadside elements [12].

### ■ Custom Attributes and Procedural Tools

Custom attributes can be added to objects using the Attribute Editor or through scripting, enabling the storage of metadata such as speed limits, lane types, and overtaking permissions [9]. Maya integrates with Maya Embedded Language (MEL) and Python scripting, supporting automated workflows and procedural content generation. These tools allow for the efficient creation and customization of road network elements [8].

### ■ Add-on Development and Exporting

Maya supports the creation of custom tools and graphical interfaces through its Python API and Qt framework. This functionality enables the development of user-friendly workflows tailored to road modeling tasks [13]. Exporting capabilities include formats like FBX, OBJ, and USD, ensuring compatibility with simulation frameworks and rendering engines. Automated export workflows for procedurally generated content are also supported [10].

### ■ Strengths and Limitations

While Maya offers powerful tools and extensive scripting capabilities, it has several limitations that affect its suitability for this project. Its commercial nature and high price point may limit accessibility for many users, despite Autodesk's free educational licenses. Additionally, while Maya provides robust geometric tools, its learning curve and complexity may present challenges for those without prior experience with the software.

## ■ 4.4 Conclusion

In summary, the framework selection process highlighted several key considerations. While VRUT offers the greatest convenience for end-users by minimizing external dependencies and streamlining integration, it also presents significant implementation challenges. Its complexity, lack of documentation, and limited tools for geometric operations make it the most demanding framework to work with.

Although versatile and capable of handling various tasks, game engines are not inherently specialized for working with geometries or managing the efficient export and import of geometric data. These capabilities are critical for the RoadBlocks editor, and the learning curve associated with game engines would have required significant time and effort, given my limited prior experience with such frameworks.

In contrast, modeling software is more suitable for tasks involving complex geometry manipulation and data export. Its specialized focus on geometric operations, support for custom structures, and intuitive tools for add-on development align closely with the requirements of the RoadBlocks editor. Among the available options, Blender emerged as the optimal choice, thanks

to its free accessibility, extensive feature set, compatibility with the project's needs, and developers prior experience with the platform.

For these reasons, Blender was selected as the framework for implementing the editor.

## Chapter 5

# Virtual Reality Universal Toolkit system (VRUT)

Starting with a disclaimer, the data presented in this chapter is mostly based on information obtained from VRUT documentation (*Dokumentace aplikace VRUT*, 2010, [59]). It is essential to acknowledge that this documentation is outdated, and no updated version has been identified during the course of this research. Consequently, there is a possibility that the presented information may not accurately reflect the current state of the subject matter due to the ever-evolving nature of the application. The accuracy of all information relevant to the implementation of this work has been verified. Having said that, readers are advised to verify the data independently and consider seeking more recent sources for a comprehensive understanding of the topic, given that VRUT has mostly been used as "*output checker*" in the implementation chain.

### 5.1 Overview

The VRUT (Virtual Reality Universal Toolkit) application is a collaborative project between ŠKODA AUTO a.s. and the Czech Technical University in Prague, specifically the Faculty of Electrical Engineering, the Computer Graphics and Interaction Department. It aims to visualize and edit 3D data while incorporating new technologies and providing special functionality. The primary goal of VRUT is to support Škoda Auto systems and formats, ensuring high-speed performance and usability for both educational and experimental purposes [59].

The VRUT focuses on graphical data visualization and modular support. The modular approach allows for the expansion of core application functionality by developing compatible modules or plugins. These modules can address various tasks related to data visualization and manipulation, enabling developers to concentrate on their specific problems within their own modules while benefiting from the overall VRUT framework [59].

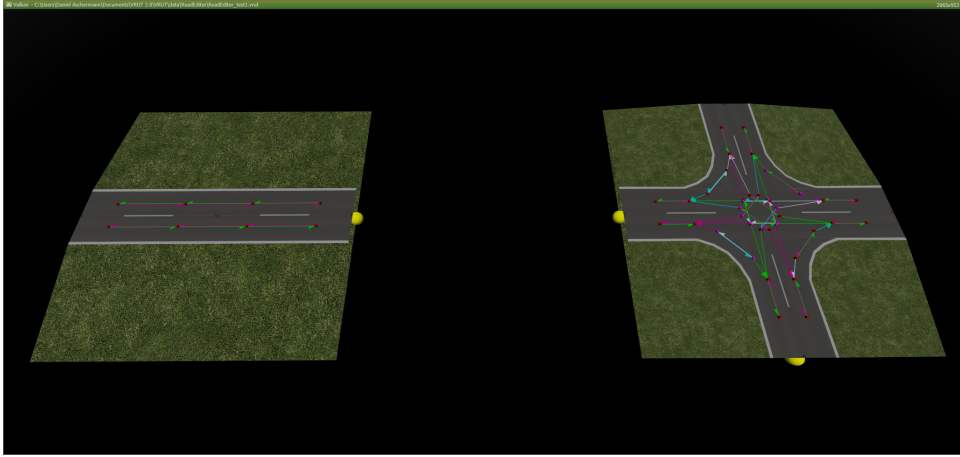
While the university employs VRUT as an auxiliary development tool, Škoda Auto utilizes it to implement diverse modules created by students and other individuals. By utilizing VRUT, these module developers can bypass

As noted in the introductory disclaimer, the documentation accompanying VRUT hasn't evolved over time and therefore contains outdated sections due to the application's dynamic nature. The main portion of the documentation originated from Václav Kyba's diploma thesis (*Modulární 3D prohlížeč*, 2008, [58]). However, an unified documentation system for individual modules has been established and appended to the original documentation.

## 5.2 Core of VRUT

### 5.3 Relevant Modules





**Figure 5.1:** Examples of RoadEditor module's blocks [6]

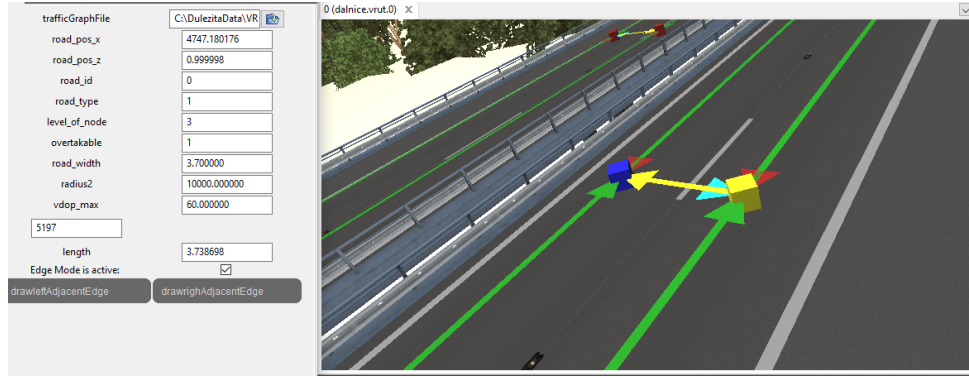
### ■ 5.3.1 Road Editor

The Road Editor module was initially envisioned as a foundational component for the editor developed in this thesis to build upon its functionality to support the creation and interconnection of various roadblocks. This module, created as a master thesis by D. Aschermann [6], designed to interconnect predefined roadblocks to form a network and construct scenes within VRUT, introduced a drag-and-drop mechanism for adding blocks to a predefined plane, enabling their seamless connection and deformation as needed (see Figure 5.1) [6].

As the implementation of this thesis progressed, the Road Editor module was ultimately excluded from the workflow. The editor developed in this thesis evolved to autonomously generate its roadblocks while creating new structures and merging them to form the whole traffic scene. Nonetheless, the concept of the Road Editor remains an interesting approach to modular scene creation and served as a reference point in the early stages of this work.

### ■ 5.3.2 Road Network Editor

The Road Network Editor module, developed as a bachelor's thesis of V. Kolin-sky [57], provides users with an intuitive interface to modify the road graph structure without manually editing the XML file or restarting the system, significantly improving the workflow for road graph manipulation. The editor developed as part of this thesis is specifically designed to facilitate the creation of connections and ensure uniform spacing between nodes, streamlining the initial generation of road networks. Additionally, this tool can be used to fine-tune auto-generated nodes that do not meet the specific requirements of developers, offering greater flexibility and precision in road network adjustments. Figure 5.2 [57] shows an example of this module's GUI.



**Figure 5.2:** A preview of Traffic editor GUI



**Figure 5.3:** Traffic simulation module in action

### 5.3.3 Traffic Module

The Traffic Module, developed as a component of the VRUT project by J. Minařík [64], enables realistic autonomous traffic simulation in virtual environments. It employs a road graph structure to model traffic behavior, including lane following, overtaking, and intersection handling, while ensuring adherence to traffic rules and preventing collisions.

One of the module's key features is the "simulation window," which optimizes computational resources by limiting simulation efforts to vehicles within a specific range of the user-driven vehicle. This approach ensures high-fidelity simulation for nearby traffic while maintaining overall efficiency.

The module dynamically generates and manages vehicles in the scene, adjusting their behavior to traffic conditions and scenarios defined in an XML-configured road graph. This design ensures that pre-defined traffic scenarios, described in an XML-configured road graph, are efficiently simulated during runtime. An example of a vehicle overtaking within the system is illustrated in Figure 5.3 [64].

The editor developed in this thesis is expected to produce output compatible with the Traffic Module, enabling seamless simulation of traffic on the roadblocks generated by this editor.

## ■ Conclusion

The detailed exploration of VRUT, its core functionalities, and relevant modules provided a foundation for ensuring the compatibility and seamless integration of the editor developed in this thesis. By analyzing the modular design of VRUT and its capabilities, this study identified key aspects that influenced the design and functionality of the editor.

With this foundational understanding of VRUT, we now transition to the implementation phase, where the design and development of the editor of roadblocks are detailed. This second part includes the algorithms, user interface, geometry generation, and testing processes necessary for creating an efficient and interactive tool for roadblock creation.





## **Part II**

## **Implementation**



## Chapter 6

### Road Geometry

In this chapter, the system that generates the geometry of the roads in the editor is described. The way of representing roads before the generation and the way the geometry is generated is discussed.

Throughout this project, multiple approaches were explored to implement road generation. From the outset, geometry nodes were regarded as the most promising method for geometry generation in Blender, due to their real-time capabilities and ease of implementing complex concepts. The key difference between the tested approaches is the type of input used for the geometry nodes. The input data should first and foremost be intuitive for users to input, while also being compatible with geometry nodes.

#### 6.1 Road-base Representation Types

In this section, Blender-supported curve types will be analyzed as road-base representation types. Since the goal is to represent roads as curves internally inside geometry nodes, inputting a curve directly and generating geometry based on it is an intuitive solution. There are multiple curve types in Blender, each with its own advantages and disadvantages. These curve types, as well as some alternatives, will be analyzed in the context of road generation in the following sections.

##### 6.1.1 Catmull-Rom curves

Catmull-Rom curves are a type of curve that is  $C^1$  continuous, meaning that the curve is smooth, but the curvature is not continuous. Based on the analysis made in the previous chapter, for a more optimal road representation, especially in context of highways, the curve should be at least  $C^2$  continuous. On top of the  $C^1$  continuity, the Blender's implementation of Catmull-Rom curves uses uniform parameterization, which can lead to unevenly distributed control points. However, Catmull-Rom curves are a good starting point for road generation, as they are easy to manipulate and visually pleasant for simple road shapes.

Blender does not natively support the Catmull-Rom curve type outside of geometry node modifiers, therefore an alternative input representation is

required for Catmull-Rom curves. These curves can be generated from other geometry types by transforming them into Catmull-Rom curves inside the geometry nodes modifier. Choosing the most fitting geometry representation for this conversion depends on multiple factors, however a representation like polylines has many similarities with control points of Catmull-Rom curves. Now, polylines are also not natively supported in Blender outside of geometry nodes, but they can be easily simulated using meshes without faces. So in order to generate Catmull-Rom curves from polylines, the input representation should be a mesh with vertices and edges, where the edges represent the connections between vertices.

### 6.1.2 Bézier curves

Using the most common curve type present in Blender, the Bézier curve was the first tested option. As discussed in the chapter about curves, these curves have  $C^2$  continuity, which is a requirement for more optimal road design in real world scenarios. However, since the conversion to any other representation only considers control points and ignores handles, the handles become redundant and may confuse the user. Therefore it's not a good representation in case a different curve is to be generated from this curve. There is also an issue with storing custom attributes since this type doesn't provide such capabilities.

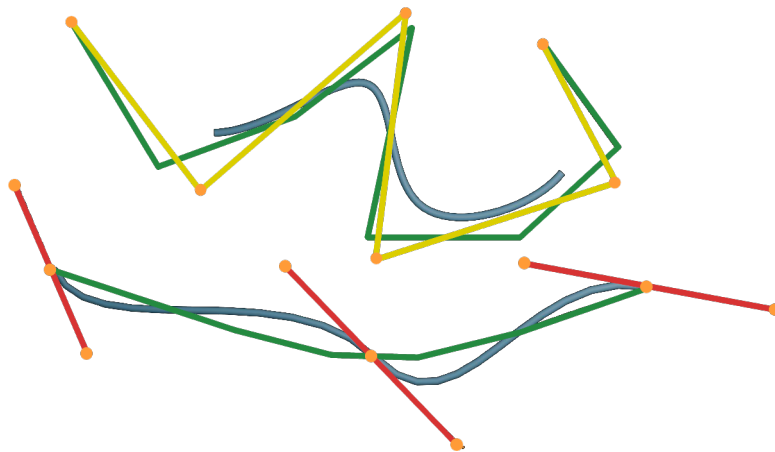
But a Bézier curve can be generated as a spline inside a *Hair Curves* object in Blender, which is supported outside of geometry nodes, and supports custom attributes. This means that Bézier curves have the potential to be a good representation for road generation, but only if created as a part of a *Curves* object, not as a *Bézier curve* object. We will discuss this in more detail in a section about hair curves.

### 6.1.3 NURBS curves

Alternatively, the NURBS input representation avoids using handles, relying solely on control points. NURBS curves meet this requirement as they are supported outside of geometry nodes and do not require handles. They are also  $C^2$  continuous, however, a significant drawback is the absence of the intrinsic ‘attributes’ property within the data, meaning custom attributes cannot be easily added from outside geometry nodes. This limitation severely restricts the add-on’s functionality and makes it nearly impossible to implement many desired features without a workaround. This makes NURBS curves a similar choice to Bézier curves.

As well as Bézier curves, NURBS curves can be generated as splines inside a *Hair Curves* object in Blender. This means that NURBS curves have the potential to be a good representation for road generation, but only if created as a part of a *Curves* object, not as a *NURBS curve* object. We will discuss this in more detail in a section about hair curves.





**Figure 6.1:** Visualization of hair curves in the Blender viewport. The blue line represents the actual curve, yellow line indicate the control points of NURBS curves, red lines represent the control points of Bézier curves, and the green line illustrates the hair curve representation of these curves visible in Blender's viewport

#### 6.1.4 Hair curves

As of Blender 4.2.0, the only curve type capable of storing custom attributes per control point is the hair curve type [37, 16]. The standard Curve data block, which stores standard curves, splines, and NURBS, is represented by the *bpy.types.Curve* data type [17]. In contrast, the hair data block, which stores hair curves, is represented by the *bpy.types.Curves* data type, with an "s" at the end [18].

The naming convention can be confusing here, and so the distinction, along with the capabilities of hair curves, was discovered later in the development process after the current implementation had already been completed.

Hair curves provide the capability to store all Blender-supported curve types, including Bézier, NURBS, and polyline representations, while also allowing for the inclusion of custom attributes. This combination positions hair curves as a particularly versatile and possibly optimal choice for curve representation. However, their visual representation in Blender's viewport interpolates control points with a large sampling step, rendering the curves as polylines (see Figure 6.1). This visualization can potentially cause confusion for users unfamiliar with the underlying data structure.

#### 6.1.5 Mesh representation

The mesh representation is similar to a polyline, as it consists of vertices and edges. This representation is the most versatile, as it can be easily converted to any curve type, including Catmull-Rom, Bézier, and NURBS and it can also store attributes for each vertex, edge or face.

However, the absence of handles for a Bézier type and a fact that NURBS type needs at least four control points to create a curve and such constraint

can't be enforced in the mesh representation, the mesh representation is considered less intuitive for users for these types.

On the other hand, the visual representation of the curves inside of mesh type object rendered in Blender's viewport is the most accurate, as it represents the actual curve and not a simplified polyline representation. This representation type is the best suited for either Catmull-Rom curves or NURBS curves, as they can be easily generated from the mesh representation.

Also, already approximated curves by polylines can be easily represented using mesh representation, which gives the user the most control over the curve, since he can first create the approximation in any way he wants and then input it as the mesh representation of the road. This allows user to generate scripts that can use other curves, that might not be supported by Blender, like the optimal Euler spiral curve, and then convert them to the mesh representation as use it as the road representation. Since this representation is the most versatile, it is the contender for the most suitable representation for the road generation.

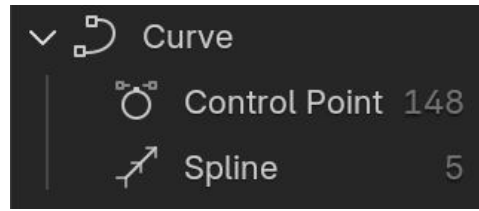
### 6.1.6 Conclusion

The chosen input representation, mesh, relies solely on vertices as control points. While this method has drawbacks, these limitations primarily effect the user experience and not the functionality of the add-on. The mesh representation is the most versatile. As a result, the mesh representation will serve as the base input for the geometry nodes, allowing the generation of the remaining road geometry.

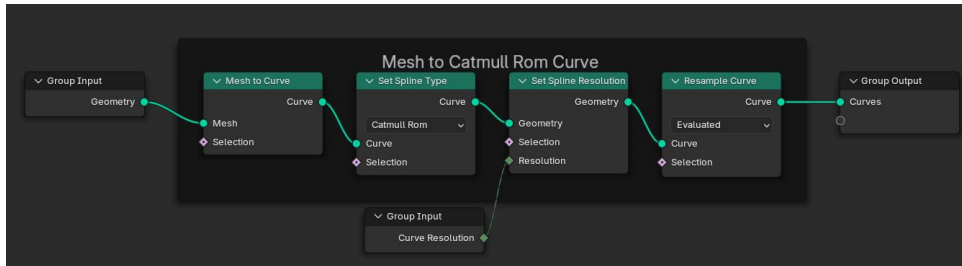
An even more optimal solution would be to use the hair curve representation and mesh representation in conjunction. The mesh representation would be used for polylines, Catmull-Rom and any other unsupported curve types, while the hair curve representation would be used for Bézier and NURBS curves. This would provide the most intuitive and versatile input for the user, as well as the most accurate visual representation of the curves in the viewport. However, this approach was not implemented in this work since it would reequire a significant rewrite of the current implementation and would not bring significant improvements.

## 6.2 Road splines

First, it is important to clarify the difference between a spline and a curve in context of Blender. In general, a curve is a general term for any smooth, continuous path, while a spline is a specific type of curve made of piecewise polynomials joined at control points. In short all splines are curves, but not all curves are splines. That holds true for Blender as well, where a curve is an object type, similar to mesh or text, that can contain multiple splines and their associated control points, which means that a single curve object can be composed of multiple splines that are not connected (see 6.2).



**Figure 6.2:** Difference between a 'curve' and a 'spline'

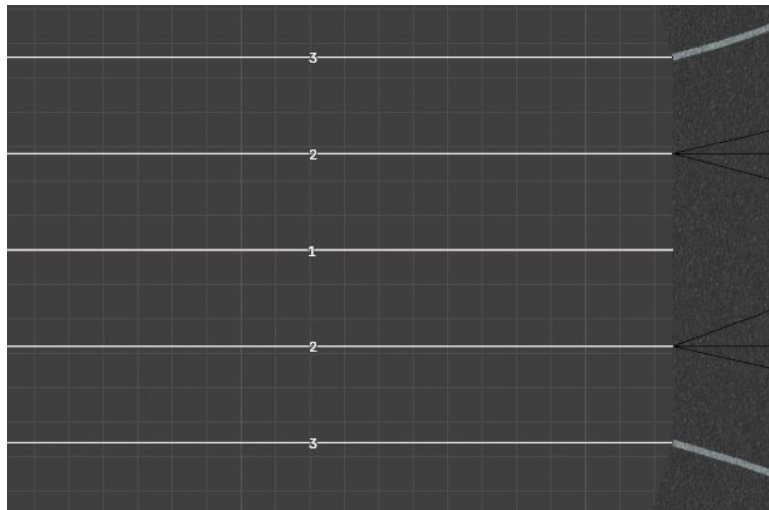


**Figure 6.3:** Central spline creation

The road is represented from multiple splines that all serve a different purpose. At the core of the road is the central spline, which represents the center of the road and is used as a base for generating any other spline used in geometry generation. This approach of having the geometry of the curve defined by a central spline is similar to ASAM OpenDRIVE representation, where the road is also defined by a central line.

### 6.2.1 Central Spline

A road is constructed from a central spline generated from the input data discussed in the previous section. In case of Catmull-Rom representation, the process begins by converting the mesh into a curve and setting the spline type to Catmull-Rom. The spline resolution is then set to a predefined value, after which the curve is resampled, as shown in 6.3. Custom attributes are subsequently stored either in the control points of the spline or bound to the spline instance. These attributes include lane count, lane width, roadside width, road width, factor, and spline class. These values are stored as attributes already before the geometry nodes start generating the geometry by a script, in order to pass them inside the geometry nodes. But for some attributes, like the factor, the value is calculated inside the geometry nodes, and for some, the default value is set inside the geometry nodes since no value is provided by the user or the initialization script. The central spline can also be trimmed on the basis of the factor (or length) of the curve to which it belongs, which is later used in the junction generation process to trim parts that would overlay the junction.



**Figure 6.4:** Spline classes - visualized

### 6.2.2 Spline Classes

Multiple additional splines are created from this central spline: one spline for each lane of the road, one spline for each outline, left and right and one spline for each lane marking line. Each of these splines is marked with a custom spline class attribute according to its purpose, which can be found inside `constants.py` file under the `SplineClass` enum.

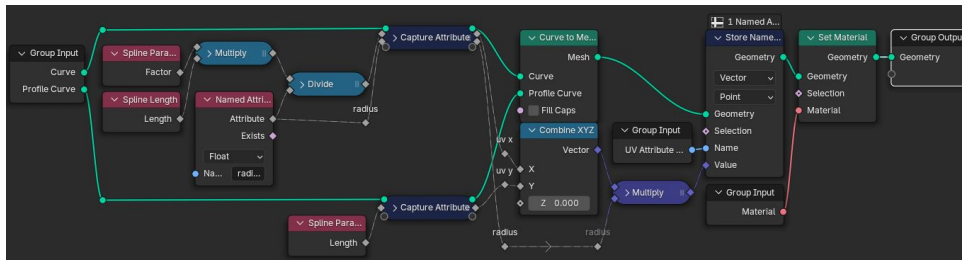
By default, the spline class value is set to 0. The central spline's class is set to 1, lane splines are set to 2, outlines are set to 3, etc. These class values are used to differentiate between the splines, as shown in 6.4.

### 6.2.3 Secondary Splines

Outline splines and lane splines are generated by duplicating the central spline and shifting its control points in the direction of or opposite to its normal vector by a specified distance. As a clean-up step, these splines are converted to meshes, and a *merge-by-distance* operation is applied to remove control points that are too close together, which might create unwanted shapes. After this, the geometry is converted back to a curve and resampled. The effects of this clean-up process are most noticeable for curved splines. Custom attributes are updated, stored, or removed as necessary after the clean-up. For example, some attributes of the central spline, such as width, may need adjustments. The width attribute, for instance, represents road width for the central spline, lane width for lane splines, and roadside width for outline splines.

### 6.2.4 Output of spline creation

These splines are then joined into a single curve object, forming the output of the initial part of the geometry nodes generation process. This output



**Figure 6.5:** Function for profile on curve generation and UV mapping

serves as input for the geometry generation and junction creation processes. These parts will be discussed in greater detail in later sections, with the next section focusing on geometry generation.

## 6.3 Road mesh generation

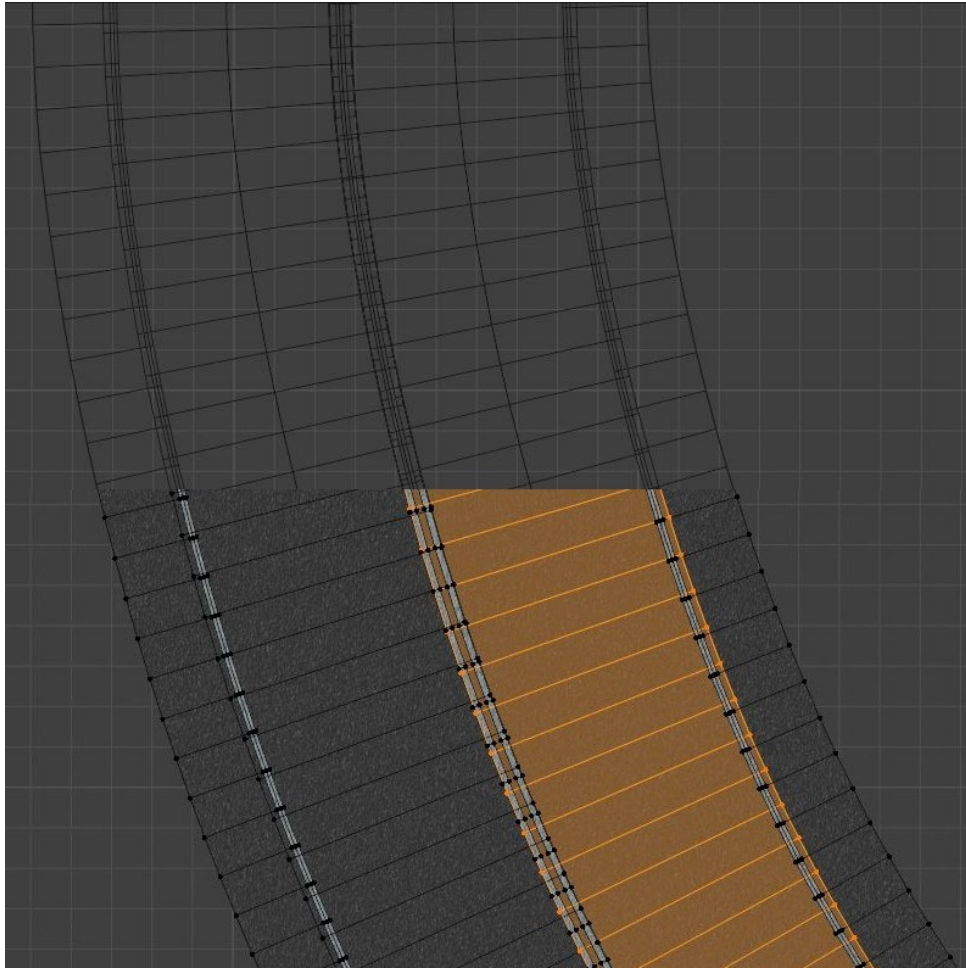
The process of generating road geometry can be divided into three main parts: lane geometry, roadside geometry, and lane markings. All three parts rely on the same fundamental principle: creating a mesh by extruding a subdivided plane along a curve (see *sweep surfaces* in section 2.6.5). The plane's width is defined using the *Set Curve Radius* node, after which a custom function is applied. This function generates the geometry based on a provided profile curve, calculates and stores UV coordinates, and assigns a material to the geometry. These topics are discussed in greater detail in the following section. The implementation of the nodes of the functions is illustrated in 6.5. Roadside geometry is generated from the outlines in the inward direction, while lane geometries and separating lines are created along the center of the curve. The results can be seen in 6.6.

### 6.3.1 T-Vertices Problem

Each curve is resampled separately to ensure a consistent spacing between points. For example, the outer outline of a curved road maintains the same distance between points as the inner outline. Lane geometries are resampled similarly. However, this approach introduces a problem: gaps can appear where the sampling of two neighbouring lanes does not align. This issue is demonstrated in 6.7. Although it resembles the T-vertex triangulation problem, it differs because no vertices are shared between lanes, as they are distinct mesh instances.

### 6.3.2 Proposed Solutions

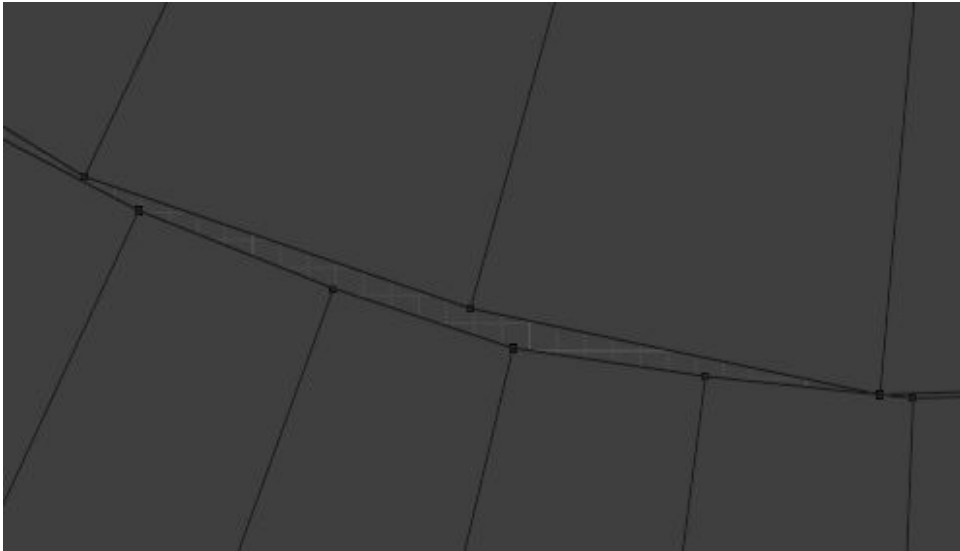
1. **Merge by Distance:** The first solution uses Blender's Merge by Distance tool. Although this method can close gaps, it is heavily dependent on the merge distance parameter. It also destructively alters the road geometry by merging all mesh instances into a single mesh, overwriting



**Figure 6.6:** Demo of road objects generated geometry

the previously generated UV mapping. The effects of this approach are shown in 6.8.

2. **Overlaying Lane Geometries:** The second solution overlays the neighbouring lane geometries. This approach preserves the UV coordinates, as shown in 6.9. However, the drawback is the appearance of z-fighting in overlapping areas. In this graphical artefact, two or more surfaces occupy the same space, causing flickering or other rendering issues.
3. **Optimal Solution:** An optimal solution would preserve UV coordinates while connecting lane geometries into a single mesh. It involves adding vertices along edges of mismatched lanes, merging only affected vertices, and maintaining UV mapping. While it may introduce minor inaccuracies, its impact is less significant than the *Merge by Distance* approach. This concept, shown in 6.10, was not implemented due to time constraints and shifting priorities, as the existing solutions were sufficient.



**Figure 6.7:** Gaps between two lanes caused by different curve sampling

## 6.4 Textures & Materials

The previous section of this document described the creation of *UV coordinates* for road geometry. They are visualized in 6.11. A custom node group called *RB\_get\_UV\_coords* has been created to load these coordinates in Blender's *Shader Editor*. This group can be used for any custom texture or material that a user wants to map onto junctions or roads.

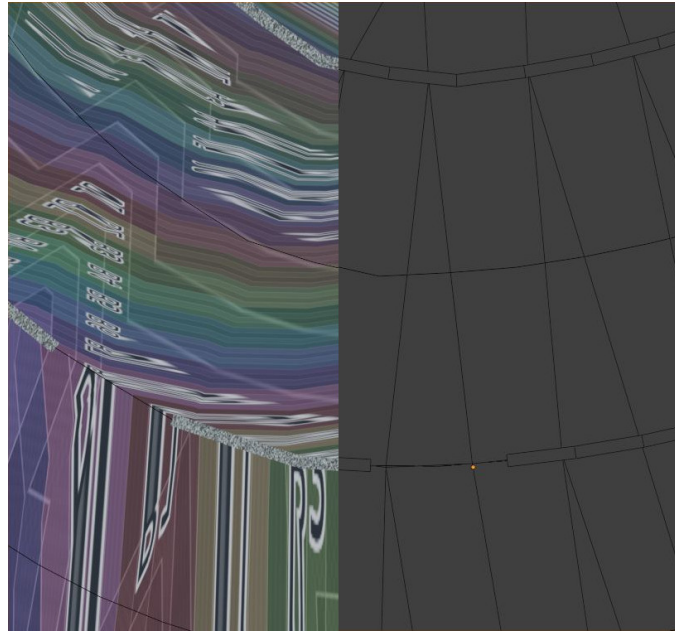
Default procedurally generated materials for junctions, roads, and separation lines are defined in the *"geometry\_nodes.blend"* file. These materials need to be baked onto the geometry, as described in the following section. In addition, pre-baked versions of these materials are available and ready for export. This allows the user to choose from existing presets, and if none of them fits their requirements, they can use Blender's capabilities to create custom material textures or effects as desired.

### 6.4.1 Baking Textures

The process of baking textures in Blender is relatively straightforward but is primarily useful for procedurally generated textures. It is important to clarify the distinction between *materials* and *textures*. In Blender, material refers to what is assigned to an object, while textures are optional components of that material.

To bake a texture, start by assigning a procedural material to the mesh of the object. In the *Shader Editor* for the material, create a *Image Texture* node. Using the *New* button in this node, create an image that will serve as the base for the baked texture. Here, you can specify parameters such as file name, image resolution, and color depth. Ensure that the *Image Texture* node is selected during the baking process to indicate where Blender should



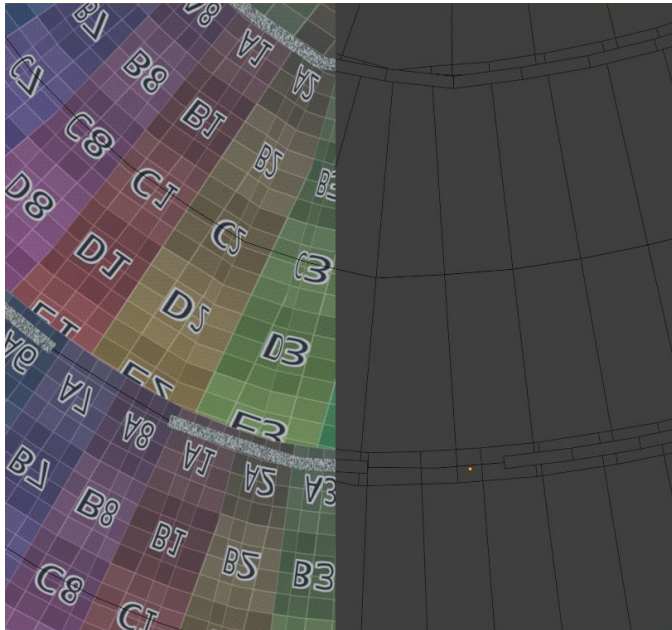


**Figure 6.8:** Gaps between lanes fixed using *Merge by Distance*, but corrupting UV mapping

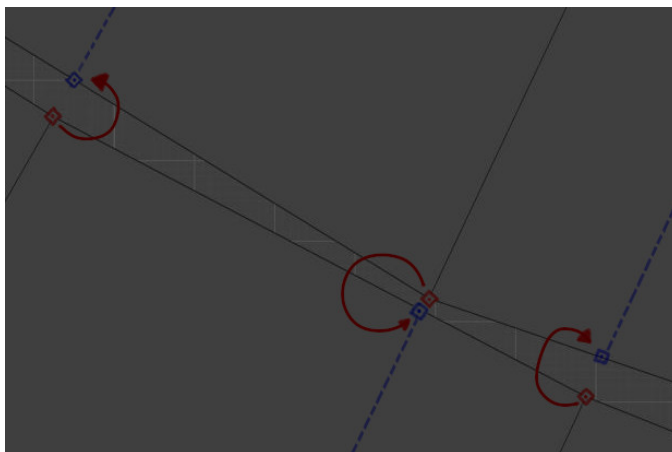
store the baked data.

Next, in the *Properties* window, navigate to the *Renderer* tab. Set the *Render Engine* to *Cycles*, and under the *Bake* section, adjust the settings according to the type of texture you are generating. For instance, select *Diffuse* to bake color textures. Once configured, press the *Bake* button. After the process is completed, the baked texture will be available [22].

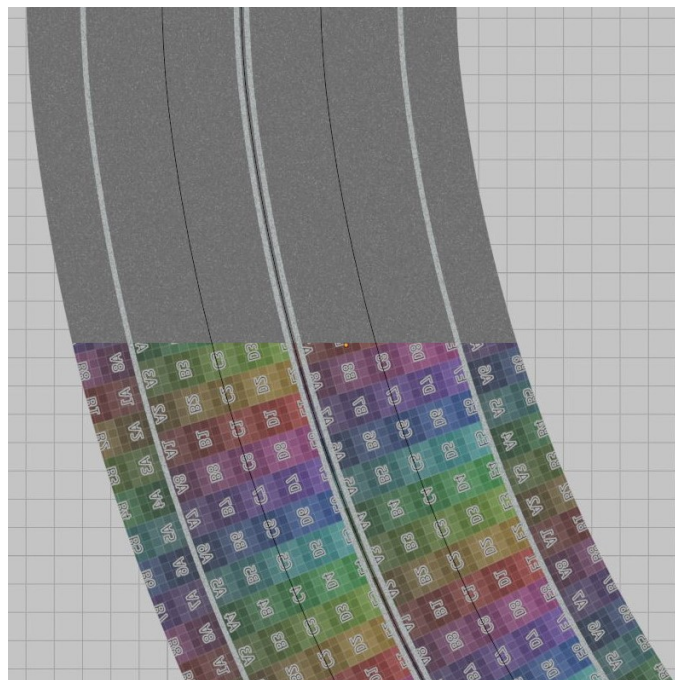




**Figure 6.9:** Gaps between lanes fixed by overlaying neighbouring lanes, maintaining UV mapping



**Figure 6.10:** Concept of optimal lane gap fixing: blue vertices are created from red vertices, with blue lines indicating new edges splitting existing faces.



**Figure 6.11:** Demo of UV mapping on road object

## Chapter 7

### Junction Geometry

In this chapter, the process of generating junction geometry is discussed. The geometry is generated based on road objects already present in the scene. The exact process of creating the junctions and generating the geometry is described in better detail in the following sections. A simphasis is put on customization options for the user and a generalized approach to junction creation.

The road network connections, the way they are generated and the data stored inside is also be discussed briefly, but the Chapter 3 will provide a more detailed explanation.

#### 7.1 Internal data representation

We discuss what kind of data is used to generate the junctions and how it is stored in Blender. At the beginning of the junction generation process, the original road objects are duplicated into a temporary collection as they are. The junctions are generated based on the outlines of the roads, therefore we first need to extract these outlines. To discard the rest of the geometry, we convert the road objects to a *"bpy.types.Curves"* objects, which Blender typically uses to represent hair curves. As discussed previously, hair curves allows for storing custom attributes, and can store multiple curves in one object. It's shortcomings are that it simplifies the curves on render, but since we are not rendering the objects, this is not a problem. This conversion works on mesh objects, as well as on curve objects, since curve object and curves object are different types in Blender (which was discussed in 6.1.4). And so, the input data for junction generation is a collection of *"bpy.types.Curves"* objects, each containing multiple curves, each representing a different aspect of the road geometry (as discussed in 6.2).

#### 7.2 Junction creation algorithm

The junction creation algorithm is structured around several key steps. First, the outlines of the roads are processed to calculate all potential intersection points. These intersections are then grouped into distinct junctions and orga-

nized in a clockwise order. Subsequently, geometric structures are generated based on these intersections, and the roads are trimmed accordingly to ensure coherence with the junction geometries. The following sections provide a detailed examination of each stage of the algorithm.

### 7.2.1 Outline Intersections

At the initial stage of junction creation, all roads are duplicated and converted into *Hair Curves*. This process eliminates the original geometry, retaining only the curve representations generated within the geometry nodes. Only the outlines are preserved from these curves. The curves are sampled at a high density, effectively resulting in a polyline representation.

Following the extraction of road outlines, the curves undergo simplification. The Douglas-Peucker algorithm is employed to remove redundant points from the curve representation. A brute-force intersection algorithm is run on these simplified curves to determine intersection points, but only on the outlines of the roads whose bounding boxes intersect. This approach reduces the number of intersection checks, improving the algorithm's efficiency. Once intersections are identified within the simplified polyline representation, the indices of the vertices of the two intersecting simplified line segments are retrieved. The precise intersection points are then calculated using bilinear interpolation on the non-simplified versions of these line segments.

While the intersection points derived from the simplified and non-simplified versions are nearly identical, the attribute values computed from the neighboring points around the intersection are significantly more accurate. This improvement in precision is particularly evident in straight road segments, where simplification reduces the entire curve to a single line segment, and interpolating attribute values from only the endpoints would result in a substantial loss of accuracy.

Finally, each intersection is appended to a list of intersections, along with all relevant attributes, such as the curve factor and the length along the curve at the intersection point.

### 7.2.2 Marking Intersection Pairs

An essential step in generating junctions is grouping intersections into pairs and junctions to facilitate the subsequent generation of junction geometry. This process pairs intersections logically, preparing the data required for constructing coherent junction structures.

To achieve this, the algorithm processes all road outlines, iterating through their intersections. The outlines are sorted by the roads they belong to, so the two outlines of the same road are adjacent in the list. This arrangement ensures efficient pairing of intersections.

For each outline, the algorithm loops through all intersections. For each intersection, it identifies the intersecting outline of the other road, reads the stored road width parameter for later use, and checks if a junction ID is

already assigned. If an ID exists for this intersection, it updates the pairing. Otherwise, it assigns a new junction ID.

If a pair for the intersection is not found or if this is the first mention of the road intersecting the current outline, a half-empty pair containing only this intersection is created and added to a list of pairs that are still awaiting the other intersection. When a pair for some intersection is found, the algorithm read lengths along the outlines from pair intersections, applies padding to extend the interval between the two intersections, and derives spline factors from the adjusted lengths. This information is stored as part of a new interval, which is then added to the road. Overlapping intervals on the same road are merged into larger intervals, ensuring that all intersections within an interval share the smallest junction ID.

Unpaired intersections are handled separately. These occur when there is an odd number of intersections between two outlines of two different roads, typically when a road starts or ends within another road. Therefore, for unpaired intersections, the algorithm creates endpoint intervals. If the intersection is in the first half of the outline, a *start* interval is created; for those in the second half, an *end* interval is generated. These are assigned the junction ID and added to the road similarly to paired intersections.

Additionally, if the unpaired intersection is surrounded by other intersections on the crossing outline, no further action is needed. However, if the unpaired intersection is an endpoint for the junction and outline, it is paired with the endpoint intersection of the opposite outline. If no such endpoint exists, the intersection is paired with a placeholder value of *-1*. This placeholder is used in subsequent steps to copy and process the entire section of the outline.

By systematically pairing intersections and managing unpaired cases, this step ensures that all intersections are grouped into junctions and prepared for the generation of junction geometry.

### 7.2.3 Junction Geometry Generation

To generate junction geometry, the algorithm follows a multi-step process designed to create smooth and accurate transitions between intersecting roads. This process begins with the calculation of intersection chains, which define the core structure of the junction, and proceeds through the generation of outlines, lane paths, and trimming lines, ultimately resulting in a complete mesh representation of the junction.

#### Intersection Chains

Before the junction geometry can be generated, the algorithm calculates what are referred to as *intersection chains*. An intersection chain is a polyline formed by a sequence of intersections that begins on one outline of a road entering the junction and ends on an outline of a different road exiting the junction. In most cases, this chain consists of a single intersection, but in more complex scenarios, it can include multiple intersections. Conceptually,

these chains can be thought of as forming one of the outline curves of the junction.

The algorithm calculates intersection chains by iterating through all intersections in a junction. For each intersection, it checks if the intersection has already been visited. If not, it marks the intersection as visited and traverses the outlines in both directions, collecting intersections until reaching the end of an outline, an intersection belonging to a different junction, or looping back to another intersection in the same junction. This process continues until all intersections have been processed, forming chains of intersections that represent junction outlines.

### ■ Junction Geometry Generation

Once intersection chains are defined, the algorithm generates the junction geometry. For each junction, it processes the connected roads. Using the interval calculated during the intersection pairing step, the algorithm determines the portion of the road's central spline that lies within the junction. If no intersections on a road's outline are part of the junction, the algorithm copies the entire segment of the outline. If intersections exist, it calculates the start and end vertices for both outlines and determines handle vertices extending away from the junction. These handles and vertices are used to form polylines, which are converted into curves and resampled to create smooth junction outlines. A custom attribute, `is_outline`, is saved for these points to distinguish them in Blender, and copied points are tagged with `is_copied`.

### ■ Trimming Lines and Final Geometry

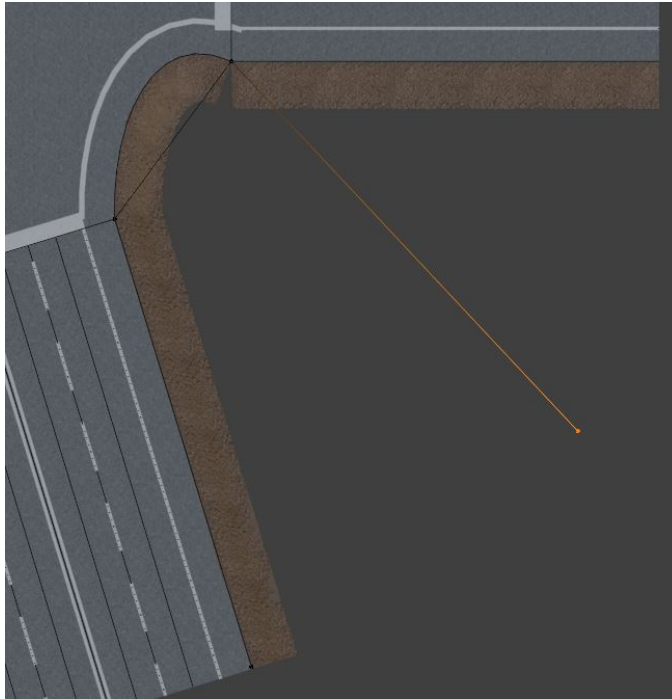
At the junction-road interface, a trimming line is generated to separate the road geometry from the junction geometry. This trimming line connects the final vertices of the road before the junction and is merged with the junction outlines to form the complete geometry.

The final junction geometry consists of copied outlines, trimmed lines, and smoothed outlines generated from handles. These components are combined into a new mesh object, stored in a dedicated `junctions` collection in Blender. The algorithm assigns custom attributes and applies a geometry nodes modifier to manage the visual aspects of the junction.

Example of handle vertices influencing final junction geometry is shown in Figures 7.1.

### ■ Trimmed Roads and User Interaction

Additionally, the road objects are duplicated and trimmed according to the junction intervals. The original road objects are hidden for user convenience but remain accessible in a `Roads` collection, while the trimmed copies are stored in a `Roads Trimmed` collection. A reset button in the interface allows users to revert to the original state by removing all junctions and trimmed



**Figure 7.1:** Intersection chains forming junction outlines

roads. However, this action deletes any user customizations, so users are advised to finalize the road layout before generating junctions. A more sophisticated solution for selectively regenerating junctions was not implemented due to time constraints.

### ■ Road Network Connections

A similar process is used to generate vertices and handles for lane splines as for outlines. These vertices and handles define smooth vehicle paths through the junction, which are then used as the foundation for road network connections. By linking lane paths across the junction, these connections represent the flow of vehicles and ensure logical transitions between roads. Further details on road network connections are provided in the section on road network implementation.

## ■ 7.3 Junction Modifier

The junction modifier is responsible for enhancing the visual appearance of the junction geometry generated by the algorithm. Implemented using Blender’s geometry nodes, the modifier creates complex procedural effects, enabling customization and refinement of the junction’s visual details.

To ensure consistent geometry across different junctions, the copied outlines of the junction are converted into curves and resampled to a uniform one-meter sampling distance. Outlines with handles are transformed into



Catmull-Rom splines and resampled at a higher density to produce smooth curves. A node setup trims these curves to remove handle vertices from the visible representation. These two curve types are then combined to form the junction's outline, with the *spline\_class* attribute set to a value designated for outlines. This setup allows junction outlines to inherit features assigned to road outlines. When merged with trimming lines, this outline creates the complete border of the junction geometry. A **Fill Curve** node is used to generate a flat, triangulated mesh surface, with a UVMap calculated based on vertex positions. The material passed in as a parameter is then assigned to the geometry. By modifying the handles of the junction outlines, users can customize the appearance of the junction.

Additional visual details enhance the realism of the junction. Thin lane marking lines are offset into the junction to simulate roadside markings while trimming lines are overlaid with thicker lines to depict the separation between the road and the junction. These features contribute to a more realistic and visually appealing junction representation.

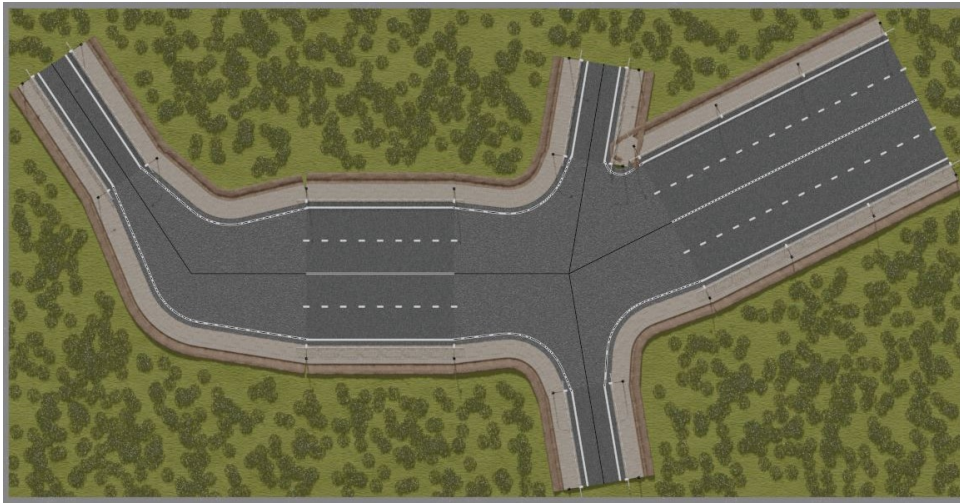
## 7.4 Alternative Tested Approaches

Several alternative methods were explored before adopting the current approach of using Blender API scripting in conjunction with geometry nodes. These alternatives, particularly the attempt to implement junction generation exclusively with geometry nodes, consumed a significant portion of the development timeline. Although these approaches ultimately proved infeasible, they were instrumental in shaping the final solution and provided valuable insights.

### 7.4.1 Initial Mesh-Based Approach

The first approach attempted to represent the road network as a singular mesh object designed to act as the skeleton of the road geometry. This method showed initial promise but encountered several critical challenges. Splitting the mesh at intersections disrupted road continuity, causing roads to appear fragmented. Additionally, managing large structures within geometry nodes became unwieldy due to complexity and performance limitations. Junction generation in this approach relied on creating 3D tubes from the road skeleton and calculating boolean unions. While the results were visually satisfactory, they failed to meet road design requirements. Sharp curvature changes in roads and junctions and sections of roads shrinking excessively into junction centers resulted in unrealistic geometry unsuitable for vehicle navigation. Furthermore, the computational cost was prohibitive, leading to significant performance degradation and an unworkable user experience. Despite its limitations, this approach demonstrated potential for specific use cases, as shown in Figure `effig:editor-alpha-version`, where early features of the road editor were evident. However, its reliance on a rigid skeletal representation rendered it less adaptable than the current method.





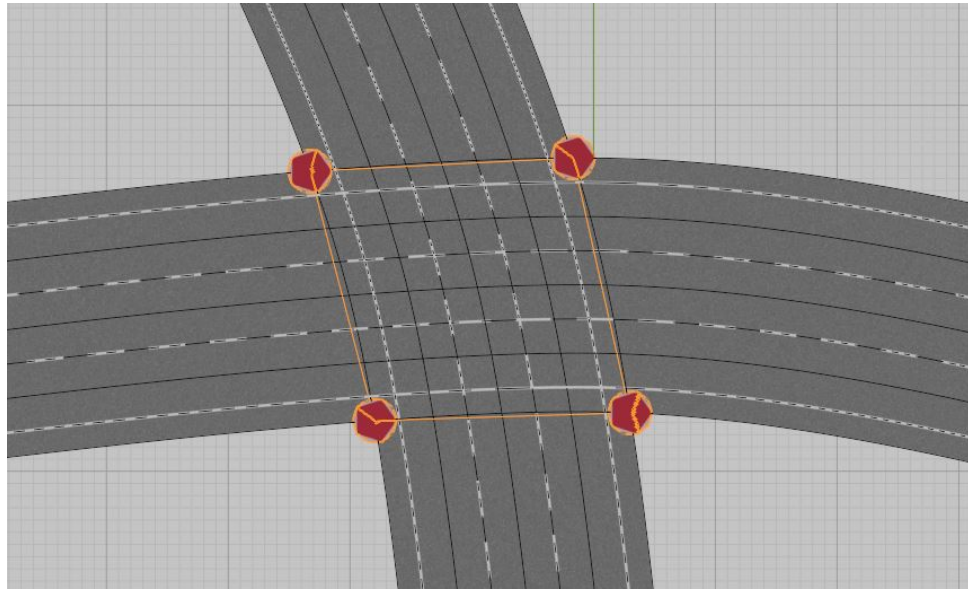
**Figure 7.2:** Example of mesh based approach for junction generation

#### ■ 7.4.2 Curve-Based Approach with Geometry Nodes

The second approach also attempted to leverage geometry nodes exclusively but shifted to representing roads as individual curve objects. Roads were organized in a collection; each assigned a geometry node modifier to generate the necessary curves. Junction generation was handled by a separate object that manipulated these splines to create intersections. Early attempts to find intersections involved sampling curves at regular intervals and merging points within a defined distance. While functional, this method suffered from severe performance issues. A subsequent method, inspired by *CG Build Up*[25], utilized a raycasting technique to detect intersections more efficiently (see Figure 7.3). Despite improved performance, the complexity of generating junctions in geometry nodes remained a significant obstacle. Additionally, junctions were created as singular objects, limiting user control and customization, which conflicted with the project's evolving requirements.

#### ■ 7.4.3 Influence on Final Implementation

Reflecting on these attempts underscores how the current approach emerged as the most optimal solution. While these alternative methods led to several dead ends, they contributed to a deeper understanding of the challenges involved and informed key design decisions.



**Figure 7.3:** Intersecting splines using raycast method, showing intersection points in red with lines connecting them

## Chapter 8

### Features

Achieving realism in geometric model generation involves efficiently populating the surroundings of roads with diverse elements, such as barriers, vegetation, and pavements. These features enhance both the visual fidelity and immersion of the virtual environment. This chapter examines the procedural generation of these elements, focusing on the implementation of roadside features and customization options.

Creating geometric features along the road relies on two primary methods: profiles and objects. Both approaches utilize Blender's geometry nodes modifier stack, enabling users to combine multiple modifiers to construct complex and customizable features.

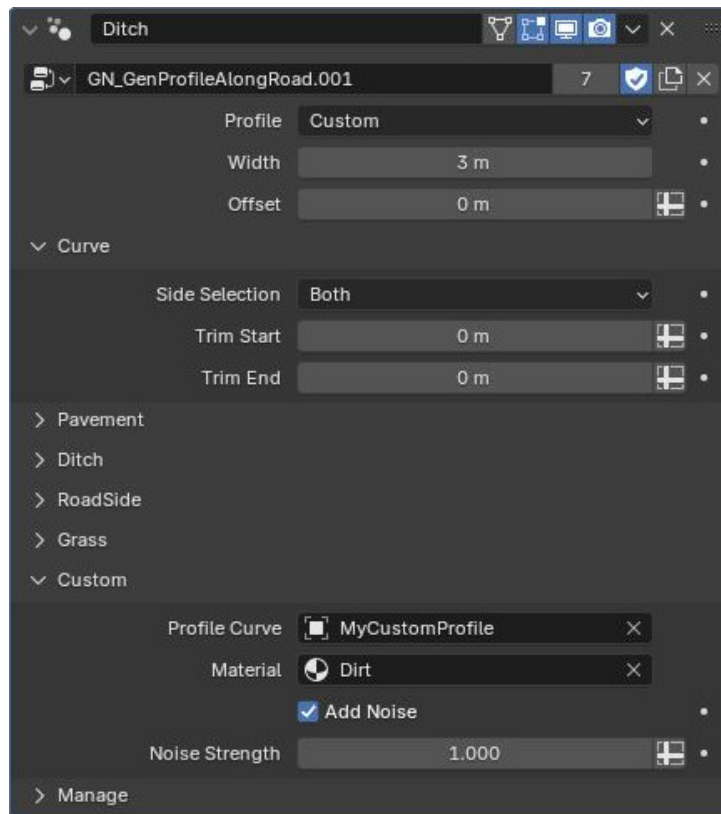
#### 8.1 Modifier Stack

The modifier stack is a core feature in Blender that enables non-destructive, automated operations on an object's geometry. Modifiers alter the appearance and rendering of an object without affecting its base geometry. This non-destructive approach allows users to efficiently perform tasks such as smoothing surfaces or generating intricate patterns [20].

A modifier stack consists of multiple modifiers applied sequentially, with the order determining the final output. Users can rearrange the stack to adjust the interaction between modifiers, ensuring flexibility in creating the desired geometry. In the road editor, the modifier stack is essential for combining and layering multiple features, allowing users to define a structured workflow. This concept supports the implementation of roadside profiles and other elements, as detailed in the following sections.

#### 8.2 Profile Along the Road

The profile method generates features along the length of the road by combining a curve with a profile curve to define the geometry. This approach mirrors the technique used for generating road geometry and provides extensive customization options.



**Figure 8.1:** Interface of the *profile-along-road* feature modifier (options for custom profiles and general settings only)

## ■ Preset and Custom Profiles

The editor includes several preset profile curves, such as pavements, grass, and ditches, which simplify the creation of standard roadside elements. Additionally, users can define custom profiles by specifying their geometry and materials. Customization options extend to applying noise displacement to create more natural-looking features, such as uneven dirt or grass surfaces. These settings allow users to balance functionality with ease of use.

## ■ Modifier Interface

The interface for the profile modifier, shown in Figure 8.1, is divided into several panels, including **General Settings**, **Curve**, and feature-specific panels such as **Pavement**, **Ditch**, **Roadside**, **Grass**, and **Custom**. These panels allow users to customize geometry generation with precision, whether using preset profiles or defining unique features.

The **General Settings** panel includes parameters such as width and offset, which control the size and position of the profile along the selected curve. The **Curve** panel provides options for trimming the start and end points of the curve and selecting the sides (left, right, or both) for profile generation. This functionality, however, is only available when generating profiles from



**Figure 8.2:** Demo of road features generated geometry and corresponding modifier stack

shifted outlines, not from the central spline.

For custom profiles, the **Custom Profile** panel allows users to select a custom curve object, assign materials, and enable or adjust noise displacement to add surface irregularities. Other panels, such as **Pavement** and **Grass**, provide predefined options for standard roadside features, while still allowing for further customization.

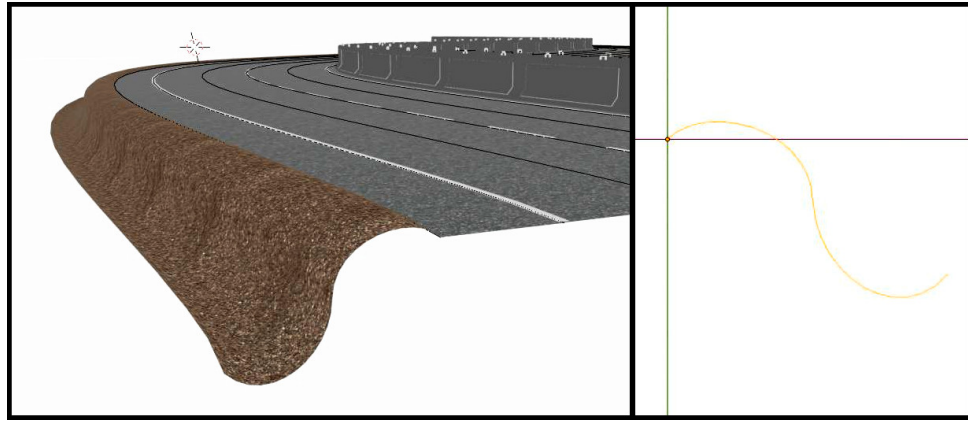
## Profile Stacking and Layering

Profiles are layered using the modifier stack, starting with the road outline and incrementally adding features outward. Each profile modifies the padding parameter for subsequent profiles, ensuring proper alignment and avoiding overlap. This structured approach facilitates the addition of multiple roadside features while maintaining the integrity of the overall geometry.

Central profiles, such as pavements or grass areas between lanes, are added to the central spline before the road geometry is generated. This process ensures the seamless integration of these features into the road layout. Users can control the order of operations within the modifier stack, disable specific modifiers, or rearrange them to experiment with different configurations. An example of the generated geometry and corresponding modifier stack is shown in Figure 8.2.

## Geometry Nodes Implementation

The geometry nodes implementation begins with a function `GetFeatureCurve`, which extracts either the central spline or the shifted outlines of the road, depending on the arrangement of the modifier stack. Geometry is generated outward from the selected curve, even for profiles originating from the central spline. The user selects the profile type via a dropdown menu in the modifier interface, and the profile is then extruded along the curve to create the geometry.



**Figure 8.3:** Example of a custom profile curve and the resulting geometry

Padding adjustments are made to the original road outlines to reflect the width of the selected profile, as the shifted outlines are used solely for geometry generation. Additional implementation details, such as ditch profile logic and curve selection mechanisms, are omitted for brevity but are part of the overall process.

### ■ Adding Custom Profiles

Custom profiles can be incorporated into the editor by creating a new curve object in Blender. For clarity and easier editing, it is recommended to position the curve at the world origin, with one end at the origin and the other extending in the positive  $x$  and  $y$  directions. While this positioning is not strictly required, it helps users visualize the resulting geometry more intuitively. Maintaining a unit width for the profile curve is also advised, as the modifier automatically scales the curve to unit width for compatibility with its settings.

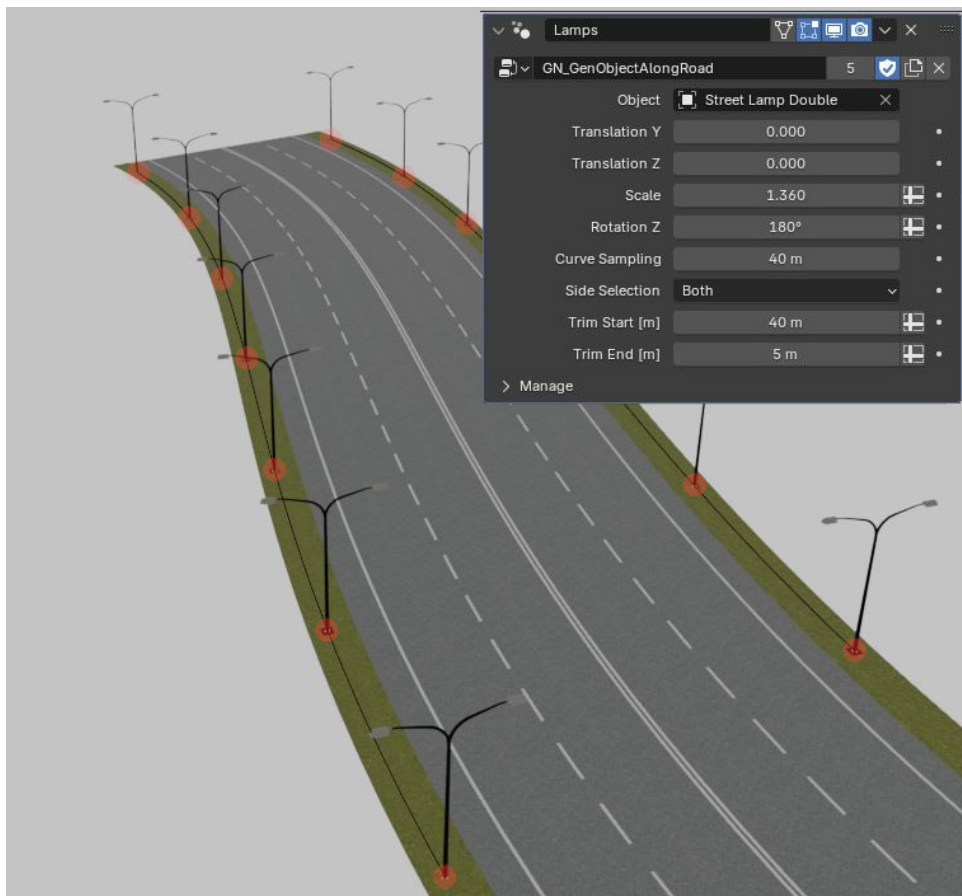
This workflow provides users with flexibility and precision when designing custom roadside features. An example of a custom profile curve and its resulting geometry is illustrated in Figure 8.3.

## ■ 8.3 Object Along the Road

Another method to enhance the visual detail of the road model is through the placement of discrete objects along the road using the object-based method. This approach enables the user to select a mesh object, such as a streetlamp, and generate instances of this object along the road curve. Similar to the profile method, the order of modifiers in the modifier stack influences the starting point of these instances, ensuring compatibility with other features.

The object-based method provides additional customization options to fine-tune the placement of instances. Users can adjust the distance between instances, as well as apply offsets in the latitudinal, longitudinal, and altitu-





**Figure 8.4:** Interface of the road objects modifier with red dots representing the sampled curve control points and the curve itself connecting them

dinal directions. Further adjustments include scaling the objects and rotating them around the vertical axis (z-axis). These settings provide flexibility in defining the arrangement of objects, allowing for a diverse range of roadside elements. An example of streetlamps generated by this modifier and the corresponding interface is shown in Figure 8.4.

### ■ Curve Sampling and Side Selection

The object-based method uses the same principle as the profile method to extract the curve. The curve is resampled based on length using the *Curve Sampling* parameter available in the interface. However, this parameter does not correspond directly to the exact distance between instances, leading to *discrete jumps* in behavior, where gradual adjustments to the parameter cause no change until a threshold is crossed. A more precise approach for controlling instance placement would improve usability and accuracy. To address edge cases, the curve can be trimmed from both endpoints, and users can specify the side of the road where objects should be placed.

## ■ Material Handling and UV Mapping

When instances are realized as actual mesh geometry, they automatically retain their original materials. However, the UV map of the geometry is overwritten during the realization process. To preserve the UV map and enable the use of complex textures or materials, the *UVMap* parameter from the original geometry is captured using the **Capture Attribute** node.

This captured attribute is then reassigned to each instance using the **Store Named Attribute** node for face corners. As long as the original geometry has a UV map named *UVMap*, the instances will inherit it. This workflow allows users to apply intricate materials without the need to manually unwrap the UVs for each instance.

## ■ Integration and Limitations

The object-based method provides robust functionality for enhancing roadside features through various customization options. Users can adjust the placement of instances with transformations such as scaling, rotation around the vertical axis, and latitudinal or altitudinal offsets. However, the longitudinal shift parameter exhibits *step-like* behavior, where changes only take effect after crossing specific thresholds, potentially complicating precise adjustments. Despite this limitation, the overall flexibility of this method, combined with automatic material handling and UVMap preservation, makes it an efficient tool for adding detailed features to the road environment.

## ■ Surroundings

An additional feature of the road editor is the ability to generate surroundings around the road. This feature is implemented as a separate modifier applied to a separate object, stored in a collection called **Surroundings**. It takes on input a collection of objects and creates an uneven surface under the road that represents the terrain. The terrain is generated by taking a bounding box of the input collection and placing a grid of vertices on the bottom. The vertices are then displaced using a noise texture to create an uneven surface. Users can expand the terrain around the objects, set the influence of the noise texture, and set the minimal distance from the collection objects at which the distortion starts taking effect. The terrain is then triangulated, and a UV map is generated based on the vertex positions. The material passed in as a parameter is then assigned to the geometry.

Finally, this surroundings object allows for adding vegetation using a new modifier group assigned directly to the surroundings object. The vegetation modifier generates instances of the objects of the collection passed as a parameter, placing them randomly on the terrain. The user can set the vegetation's density, the objects' scale, and the random seed for the placement. Minimal and maximal distances from road elements can be set to control the placement of the vegetation. This feature is illustrated in Figure 8.5.





**Figure 8.5:** Example of the surroundings feature with terrain and vegetation

## Graphical User Interface

In addition to the modifier-specific customization options, the graphical user interface (GUI) simplifies the workflow with dedicated buttons for each major feature. By selecting a road object (or multiple objects) and clicking a corresponding button, the appropriate modifier is automatically loaded into the Blender file from a pre-packaged .blend file within the add-on directory. The modifier is then applied to the selected road objects and can be further customized or reordered in the Modifiers tab of the Properties Editor.

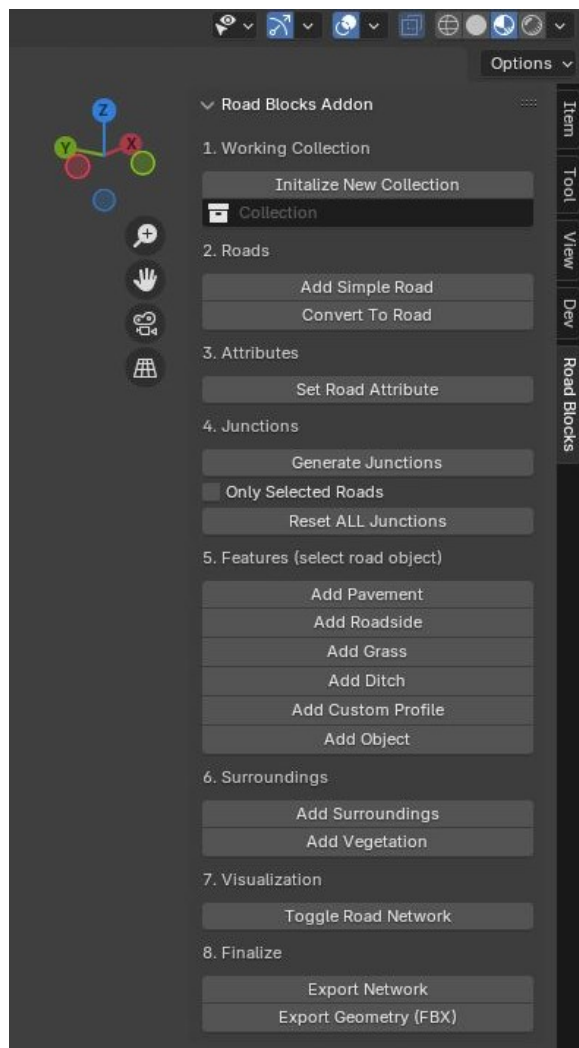
This streamlined interface bridges the gap between automation and flexibility, enabling users to quickly implement features while retaining full control over the modifier stack. An example of the GUI layout and functionality is shown in Figure 8.6.

## Conclusion

This chapter explored the key features implemented for enhancing the realism and functionality of road models, focusing on the two primary methods: profile-based and object-based generation.

The profile-based method enables the creation of continuous roadside features, such as pavements, ditches, and grass, along the length of the road. By leveraging the geometry nodes modifier stack, users can layer profiles sequentially, adjust parameters like width, offset, and noise displacement, and even define custom profiles for unique geometry. The structured stacking process ensures compatibility between features and provides flexibility for customization.

The object-based method complements this by allowing discrete objects, such as streetlamps or signs, to be positioned along the road. With transformation controls for scaling, rotation, and offsets, users can fine-tune object placement to suit their needs. While the longitudinal shift parameter exhibits



**Figure 8.6:** Graphical User Interface of the editor in Blender's side-panel

step-like behavior, the method remains a powerful tool for adding detail, with seamless material and UVMap handling.

Together, these features provide a robust framework for generating and customizing road environments.

## Chapter 9

### Road Network

The road network serves as the foundational framework for any traffic simulation system, providing the structure and connections required for vehicles to navigate their environment. This chapter outlines the implementation of the road network within the road editor, focusing on the internal representation of road network data, the generation of network structures, and the export process for integration with external simulation systems, such as VRUT.

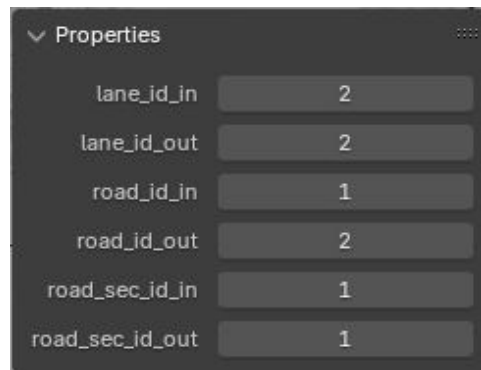
#### 9.1 Internal Representation

The road network forms the skeleton of road geometry, consisting of nodes and edges, where nodes represent points at the center of lanes, and edges define the connections between them. In Blender, the internal representation of road network data is modeled using curves generated from lane splines, with each node corresponding to a control point on the lane curve. These lane splines are generated as part of the road geometry creation process.

A geometry nodes modifier is applied to the road geometry to extract the lane splines and prepare the network structure. This modifier also provides a visualization of the road network, enabling users to preview the generated structure before export.

Extracted lane splines are resampled uniformly based on a user-defined sampling distance, which can be set either in the export window or within the modifier for visualization purposes. Any existing road network modifiers are removed prior to export to ensure the network structure is generated consistently and with uniform sampling.

Most attributes, such as lane IDs for VRUT and OpenDRIVE formats, are calculated during the lane spline generation process. The modifier allows for additional attributes to be computed before export, including the *curvature* attribute and a unique *node\_index* used for global node ID calculations in VRUT format export. Users can also define additional parameters, such as lane width or speed limit, through the **Set Road Attribute** button in the GUI. Available attribute options can be extended in future updates.



**Figure 9.1:** Example of object properties in Blender used for connector pairing

## 9.2 Junctions

The junction generation process also incorporates road network data, primarily through the creation of connectors between lanes entering and exiting the junction. For each lane, a vertex with a handle is generated, similar to the approach used for junction outlines. Connections between these vertices are then created based on predefined rules.

To keep the junction geometry clean, connectors are exported as separate objects, parented to the junction geometry. This approach facilitates simpler editing and allows unwanted connectors to be removed easily. Users can also manually add connectors if required. Future extensions could include options to generate all possible connectors, restrict generation to rule-based connectors, or define custom rules for connector creation.

Additional data, such as junction IDs, junction in/out IDs for trimmed road segments, and lane/road segment in/out IDs for connectors, are added during the junction generation process. These values, stored as object properties, are used for pairing objects in the road network export process (see Figure 9.1). It is recommended not to modify these properties manually, as this could disrupt the add-on's functionality.

## 9.3 Export Formats

As analyzed in the discussion on road graph representations, this project examines two road network formats: the VRUT format and the OpenDRIVE format. Given the requirement to validate the implementation within the VRUT system, the main focus is on the VRUT format. This chapter begins with the implementation details of the VRUT format and subsequently addresses the OpenDRIVE format.

## ■ VRUT Format

The VRUT format is a custom XML-based representation used by the VRUT system to define road networks. It describes the network as a skeleton structure comprising nodes and edges, where nodes correspond to control points on lane splines and edges represent the connections between them.

The export process involves the geometry nodes modifier and a separate export operator. The modifier filters relevant data, such as lane splines and junction connectors, and calculates additional attributes required for export. The export operator reads this data using the Blender API and transforms it into the final VRUT format.

Each node in a road object is assigned a unique ID for each control point on the curve. At the start of the export algorithm, global node offsets are calculated for each road, ensuring unique global node IDs across all roads as required by the VRUT format.

Control points from resampled splines are exported as nodes, with splines representing lanes and curve objects representing roads. Junction connectors are matched to roads using junction IDs and to lanes using lane IDs. Connector control points, excluding endpoints, are used to generate node paths for junctions. These paths link lanes by matching road IDs, lane IDs, and node IDs to establish connections.

For each lane, sequences and interconnections are defined to represent connections between nodes. By default, lanes traveling in the same direction are interconnected, ensuring overtaking is always possible within a single direction. This approach reflects the current Blender interface capabilities, which do not support this specific scenario of detailed lane marking.

The final data is formatted as XML and saved to a file, ready for import into the VRUT system.

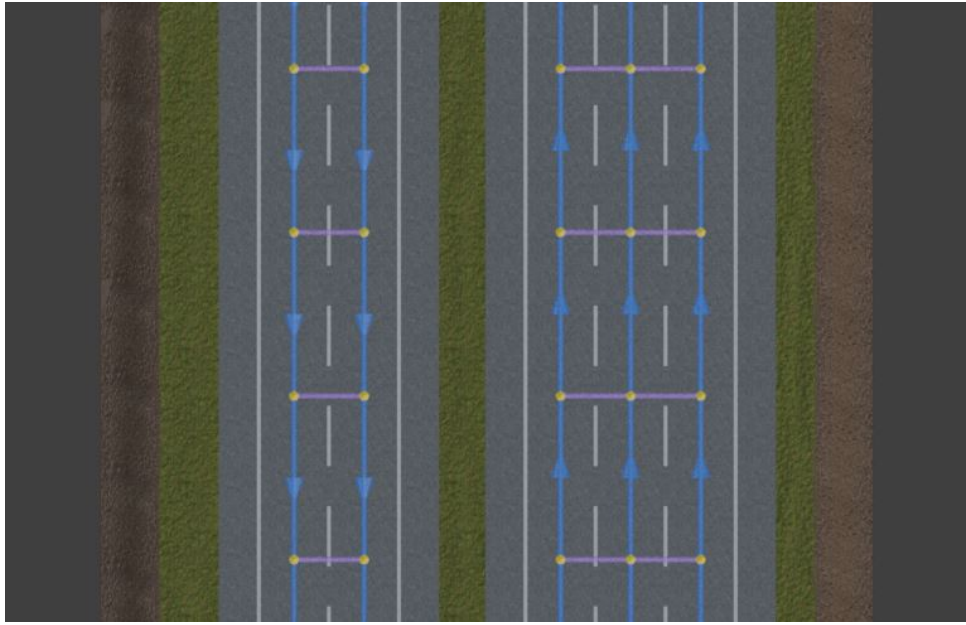
## ■ OpenDRIVE Format

The OpenDRIVE format represents road networks as a series of connected road segments defined by geometric primitives, such as lines, arcs, and spirals. This format differs fundamentally from VRUT and requires a distinct approach for generating network structures.

The OpenDRIVE format has not been implemented in this project; however, groundwork has been laid for its future development. Attributes such as OpenDRIVE-style lane indices and curvature per node are calculated in the road network modifier and assigned to the geometry. The implementation of the OpenDRIVE format is beyond the current scope of this thesis but remains a potential avenue for future work.

### ■ 9.3.1 Visualization of Road Network

The road network is visualized in Blender using the geometry nodes modifier, the same that prepares the network for export. Lane curves and interconnections are visualized with directional arrows to indicate vehicle movement,



**Figure 9.2:** Visualization of the road network in Blender

and nodes are represented as yellow spheres. This visualization, inspired by the VRUT system and based on the work of V. Kolínský [57], allows users to preview the road network structure before export.

Users can test different sampling distances and observe the resulting network. Interconnections are displayed to show overtaking possibilities between lanes. An example of the visualization is shown in Figure 9.2.

## Conclusion

This chapter outlined the implementation of the road network within the road editor, focusing on its internal representation, junction integration, export formats, and visualization. The road network is modeled as a skeleton of nodes and edges extracted from lane splines, using a geometry nodes modifier to prepare the network structure for export. Junctions are integrated into the network with connectors, enabling seamless transitions between lanes. The VRUT format, as the primary focus of this project, is a custom XML-based format specific to the VRUT system. The OpenDRIVE format remains a future avenue for development. Visualization tools within Blender allow users to inspect and test the road network structure, providing valuable insights before final export. This workflow ensures the road network is both functional for simulation and adaptable for future enhancements.

## Chapter 10

### Testing in VRUT

The editor's compatibility and functionality within the VRUT system were evaluated as stated in the requirements of this project. This chapter details the testing process, including the compilation and execution of VRUT, the import of road network data, and the validation of road network geometry and behavior within VRUT.

#### 10.1 Compilation of VRUT

This section outlines the steps required to compile the VRUT system on a Windows 10 environment.

The process begins by cloning the VRUT project from Škoda's remote repository. CMake is then utilized to configure the project and generate the required binaries. It is recommended to build the CMake project within a "build" folder located in the root directory of the VRUT project. During configuration, CMake allows users to enable or disable specific modules. Ensuring that all necessary modules are activated is essential, as some may be disabled by default. Key modules relevant to this thesis, such as `ROAD_EDITOR`, `IO_IOVRUT`, and `JSSCRIPTING`, are illustrated in the module selection interface shown in Figure 10.1.

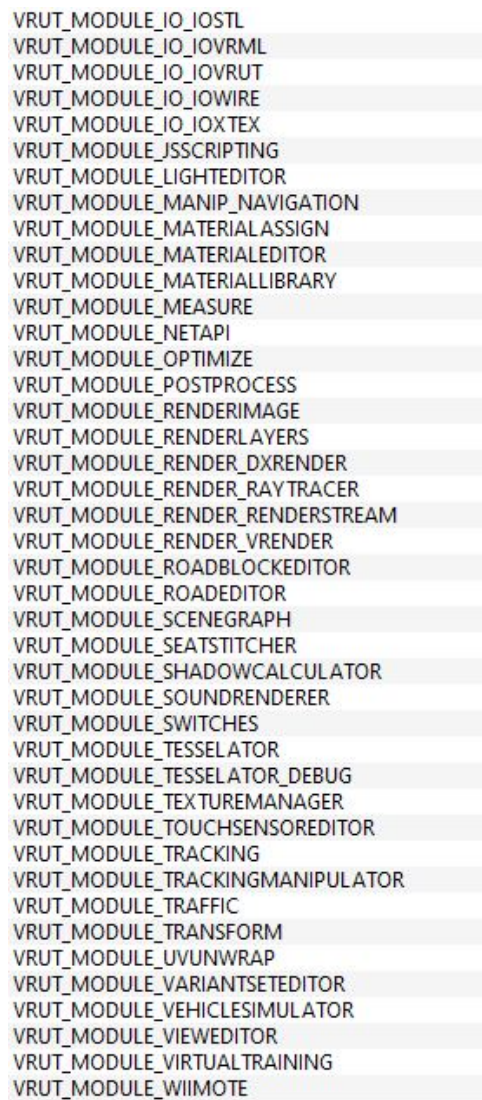
Once configuration is complete and project binaries are generated, the project is opened in Visual Studio for compilation. Initial compilation may encounter issues related to library paths or linking. Common problems include incomplete configuration parameters in CMake, such as missing paths to external libraries, or the need to manually delete specific auto-generated files in Visual Studio. These issues can typically be resolved by carefully reviewing CMake settings and the Visual Studio output log.

The compilation process, which may take several minutes depending on the device's performance, concludes with a fully built VRUT system.

#### 10.2 Exporting Geometry for VRUT

Blender offers various geometry export functionalities that can be utilized for exporting models. The VRUT system supports multiple input formats, the





```

VRUT_MODULE_IO_IOSL
VRUT_MODULE_IO_IOVRML
VRUT_MODULE_IO_IOVRUT
VRUT_MODULE_IO_IOWIRE
VRUT_MODULE_IO_IOTEX
VRUT_MODULE_JSSCRIPTING
VRUT_MODULE_LIGHTEDITOR
VRUT_MODULE_MANIP_NAVIGATION
VRUT_MODULE_MATERIALASSIGN
VRUT_MODULE_MATERIAEDITOR
VRUT_MODULE_MATERIALLIBRARY
VRUT_MODULE_MEASURE
VRUT_MODULE_NETAPI
VRUT_MODULE_OPTIMIZE
VRUT_MODULE_POSTPROCESS
VRUT_MODULE_RENDERIMAGE
VRUT_MODULE_RENDERLAYERS
VRUT_MODULE_RENDER_DXRENDER
VRUT_MODULE_RENDER_RAYTRACER
VRUT_MODULE_RENDER_RENDERSTREAM
VRUT_MODULE_RENDER_VRENDER
VRUT_MODULE_ROADBLOCKEDITOR
VRUT_MODULE_ROADEEDITOR
VRUT_MODULE_SCENEGRAPH
VRUT_MODULE_SEATSTITCHER
VRUT_MODULE_SHADOWCALCULATOR
VRUT_MODULE_SOUNDRENDERER
VRUT_MODULE_SWITCHES
VRUT_MODULE_TESSELATOR
VRUT_MODULE_TESSELATOR_DEBUG
VRUT_MODULE_TEXTUREMANAGER
VRUT_MODULE_TOUCHSENSOREEDITOR
VRUT_MODULE_TRACKING
VRUT_MODULE_TRACKINGMANIPULATOR
VRUT_MODULE_TRAFFIC
VRUT_MODULE_TRANSFORM
VRUT_MODULE_UVUNWRAP
VRUT_MODULE_VARIANTSETEDITOR
VRUT_MODULE_VEHICLESIMULATOR
VRUT_MODULE_VIEWEDITOR
VRUT_MODULE_VIRTUALTRAINING
VRUT_MODULE_WIIMOTE

```

**Figure 10.1:** A screenshot from CMake capturing some of the available modules

most optimal being the `.vrut` format. An export option was integrated into the road editor to enhance user convenience, enabling geometry preparation for VRUT and its export in the `.fbx` format—a format supported by both Blender and VRUT. The export process is straightforward and can be completed with a single click. A typical Blender export window appears, pre-configured with options tailored for VRUT compatibility.

The export functionality duplicates all objects under a new *root* empty object, which serves as the main element for collision detection in VRUT. Each object is assigned an additional layer, and all objects are split into separate parts based on their respective materials. This approach addresses issues encountered with single objects containing multiple materials in VRUT. Finally, the prepared objects are exported as `.fbx` files, ready for import into the VRUT system.



## 10.3 Importing Road Blocks into VRUT

The process of exporting the road network and roadblock geometry has been described previously. This section focuses on importing these files into VRUT.

Before importing roadblocks into the traffic simulation interface, it is recommended to export the geometry into the `.vrut` format using VRUT's export tool. To open the scene in VRUT, simply drag and drop the `.fbx` file into the main VRUT window. The Scene Graph module, accessible via the side panel, allows users to verify the correctness of the scene structure. Additional lighting can be configured in the Light Editor module within the same panel. To save the scene in the `.vrut` format, use the command `exportscene "scene-number" "relative-save-path"`.

With the scene optimized, the traffic simulation module can be started. The simplest way is to launch VRUT using the pre-configured `traffic.cfg` file available in the repository. This configuration opens VRUT alongside a window for selecting scenes and starting simulations. To add a custom scene, modifications to the `traffic.js` file are required. This file defines available scenes, vehicle models, and key parameters. Adding a new scene involves creating a new `case` in the switch statement, specifying the geometry file (`.fbx` or `.vrut`) as `trackFile`, and the road network file (`.xml`) as `trafficGraphFile`. Other parameters include `trackCFGFile`, which is required but can be reused from other scenes, and the `worldNodeName` which should be set to the root object name of the scene structure. The editor's exporter automatically sets this to `root`. Adjustable parameters include `CAR_COUNT` for the number of cars in the simulation and `carPositions` for their starting positions.

Once the simulation is started, the behavior of vehicles and the road network's structure can be observed. The Traffic module interface includes checkboxes for visualizing nodes and connections of the road network, providing a means to validate its structure.

## 10.4 Testing

Now that the whole process of creating the roadblock, exporting it into VRUT, and running the simulation is described, it is time to test the roadblock editor. The testing process was divided into several stages, each focusing on the editor's functionality.

The primary objectives were to validate the roadblock's geometry, ensure the correct generation of road features, and verify the compatibility of the exported data with the VRUT system. This process included checking the generated scene structure in the scene manager of VRUT, where the correct hierarchy of the scene was verified. The geometry was also visually inspected in the VRUT interface to ensure the correct scaling and positioning of the roadblock and to check for any potential defects not present in Blender. Finally, the materials were inspected for proper assignment.

The road network was tested for accuracy using a combination of visual inspection, thanks to the traffic module's road network visualization capabilities, and observation of the behavior of the simulated vehicles. Ten different road blocks, or road scenarios, were created using the editor, exported and tested in VRUT. The scenarios included various road geometries, junctions, and road features to ensure the editor's versatility and compatibility with the VRUT system. The complete list of tested road blocks and their appearance in Blender and VRUT is provided in the appendix B.

Although the testing was conducted in a limited time frame, the results were positive, with all tested road blocks successfully imported into VRUT and displayed correctly. The road network structure was accurately represented, and the road features were visible and functional within the simulation. There were some minor visual defects, but they stem from the geometry generation process and not from exporting and importing process. The editor's compatibility with VRUT was confirmed, demonstrating its potential as a robust tool for traffic simulation and virtual environment modeling.

# Chapter 11

## Conclusion

This thesis presented the design and implementation of an interactive editor for road network blocks, addressing the need for flexible and efficient tools to create and customize road geometries and networks within virtual environments. The project focused on enhancing the VRUT system by offering a robust editor that integrates seamlessly with existing modules while expanding its capabilities.

The contributions of this work are multifaceted. A comprehensive review of existing methods for representing road networks and geometric modeling techniques was conducted, highlighting the strengths and weaknesses of standards like ASAM OpenDRIVE and VRUT's custom format. These insights informed the design choices for the editor's implementation. Blender was selected as the development framework due to its procedural modeling capabilities, extensibility, and integration potential with VRUT.

The editor supports the creation of essential road elements such as straight sections, curves, and junctions, using a skeleton representation where nodes represent lane centers and edges define their connections. Procedural generation techniques were employed to produce detailed road surfaces, lane markings, and surrounding features like ditches and vegetation. These workflows were implemented using Blender's geometry nodes, ensuring efficiency and customization in geometry generation. Challenges such as the T-vertices problem were addressed with practical solutions to maintain visual quality.

Junction modeling was a critical focus, resulting in the development of an algorithm for generating smooth junction geometries. This algorithm incorporates intersection chains and customizable handle placements, enabling flexible and precise junction design. A modular approach to junction creation ensures compatibility with the broader road network structure and supports various junction types.

The editor includes tools for procedural roadside feature generation and the placement of objects such as lamps and signs, enhancing the realism and functionality of the created road environments. Export functionality was implemented to ensure compatibility with the VRUT system, focusing on its XML-based road network format. While support for the ASAM OpenDRIVE format remains incomplete, the groundwork has been laid for future integration.

Testing in the VRUT system validated the editor's outputs by simulating at least ten distinct road scenarios. These tests confirmed the editor's effectiveness and compatibility with the VRUT framework, demonstrating its potential as a robust tool for traffic simulation and virtual environment modeling.

Despite its accomplishments, this work identifies several areas for future exploration. Completing support for the OpenDRIVE format would enhance interoperability with other platforms. Addressing the elevation limitations of the current implementation and integrating additional junction types, such as lane merges and diverges or roundabouts, would further enhance the editor's utility. Finally, enhancing the graphical user interface could make the tool more accessible and intuitive for users.

In conclusion, this thesis contributes a significant interactive roadblocks modeling tool, bridging the gap between geometric precision and procedural flexibility. Combining this editor with the VRUT system's capabilities supports advanced traffic simulations and fosters innovation in virtual environments for automotive research and beyond.



## Appendices



## Appendix A

### Simple VRUT Road Network XML example

*Disclaimer: The following VRUT file is a fictional example created solely for illustrative purposes and does not represent a real-world road graph or functional data.*

```
<?xml version="1.0" encoding="UTF-8"?>
<xml>
  <roadgraph version="1.2" />

  <!-- Roads Definition -->
  <road id="1" type="0">
    <lane id="0" nodes="2" type="0" level="3">
      <node id="1" x="0.0" y="0.0" z="0.0" sl="120" w="3.7" />
      <node id="2" x="50.0" y="0.0" z="0.0" sl="120" w="3.7" />
    </lane>
    <lane id="1" nodes="2" type="0" level="3">
      <node id="3" x="0.0" y="3.7" z="0.0" sl="120" w="3.7" />
      <node id="4" x="50.0" y="3.7" z="0.0" sl="120" w="3.7" />
    </lane>
    <lane id="2" nodes="2" type="0" level="3">
      <node id="5" x="0.0" y="7.4" z="0.0" sl="120" w="3.7" />
      <node id="6" x="50.0" y="7.4" z="0.0" sl="120" w="3.7" />
    </lane>
  </road>

  <road id="2" type="0">
    <lane id="0" nodes="2" type="0" level="3">
      <node id="7" x="50.0" y="0.0" z="0.0" sl="80" w="3.7" />
      <node id="8" x="100.0" y="0.0" z="0.0" sl="80" w="3.7" />
    </lane>
    <lane id="1" nodes="2" type="0" level="3">
      <node id="9" x="50.0" y="3.7" z="0.0" sl="80" w="3.7" />
      <node id="10" x="100.0" y="3.7" z="0.0" sl="80" w="3.7" />
    </lane>
  </road>
  % more roads ...

  <!-- Attributes -->
  <attributes>
    <speedlimit from="1" to="2" value="80" />
    <overtakable from="1" to="2" value="1" />
  </attributes>
```

```

<!-- Connections -->
<connections>
  <sequence from="1" to="2" dir="2" closed="0" />
  <sequence from="3" to="4" dir="2" closed="0" />
  <interconnection from="1" to="3" count="2" dir="2" />
  % ...

  <!-- Junction Definition -->
  <junction name="Simple_Intersection" id="1">
    <path id="0" nodes="2" type="0">
      <node id="11" x="50.0" y="0.0" z="0.0" />
      <node id="12" x="50.0" y="3.7" z="0.0" />
    </path>
    <laneLink road="1" lane="0" node="2">
      <successor road="2" lane="0" node="7" path="0" />
    </laneLink>
    <laneLink road="1" lane="1" node="4">
      <successor road="2" lane="1" node="9" path="0" />
    </laneLink>
    <priority road="1" lane="0" node="2">
      <check road="2" lane="1" node="9" />
    </priority>
    <priority road="1" lane="1" node="4">
      <check road="2" lane="0" node="7" />
    </priority>
  </junction>
</connections>
</xml>

```



## Appendix B

### Simple ASAM OpenDRIVE XML example

```
<?xml version="1.0" encoding="UTF-8"?>
<OpenDRIVE>
  <header revMajor="1" revMinor="8" name="SimpleRoadExample"
    version="1.8" date="2024-12-28">
    <geoReference>WGS84</geoReference>
  </header>
  <!-- Road Definition -->
  <road name="StraightToCurve" length="300.0" id="1" junction="-1">
    <!-- Road Geometry -->
    <planView>
      <!-- Straight Section -->
      <geometry s="0.0" x="0.0" y="0.0" hdg="0.0" length="100.0">
        <line/>
      </geometry>
      <!-- Euler Spiral Section -->
      <geometry s="100.0" x="100.0" y="0.0" hdg="0.0" length="50.0">
        <spiral curvStart="0.0" curvEnd="0.02"/>
      </geometry>
      <!-- Curve Section -->
      <geometry s="150.0" x="150.0" y="0.5" hdg="0.5" length="150.0">
        <arc curvature="0.02"/>
      </geometry>
    </planView>

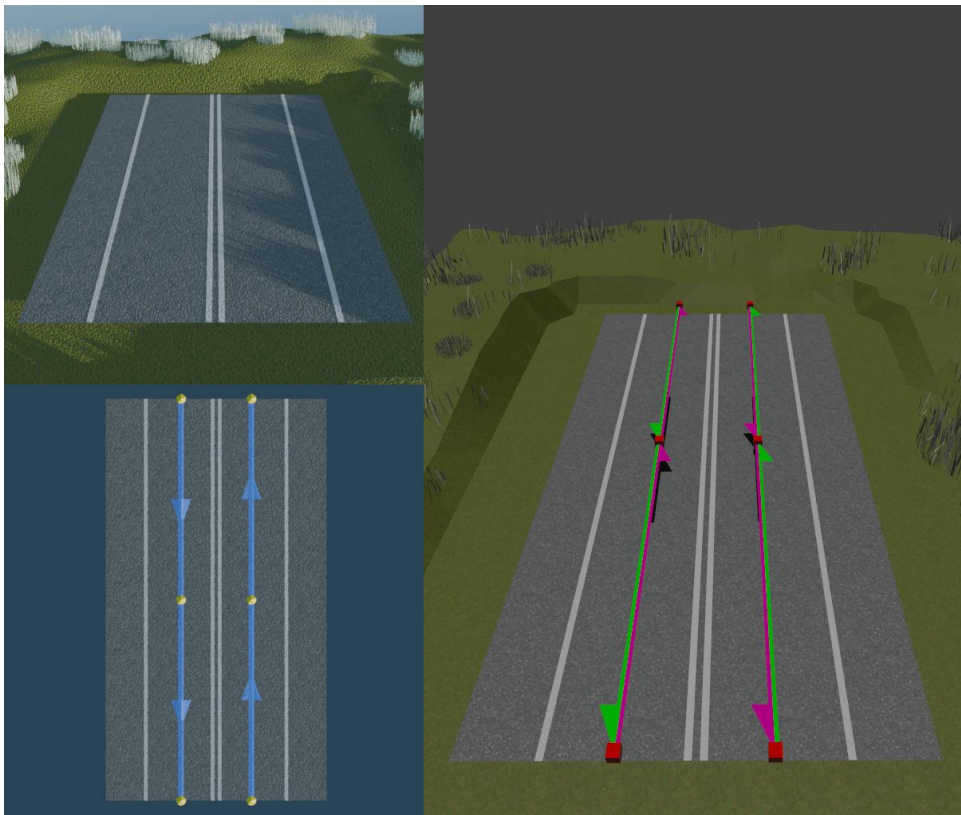
    <!-- Lanes -->
    <lanes>
      <laneSection s="0.0">
        <center> <lane id="0" type="driving" level="false"/> </center>
        <left>
          <lane id="1" type="driving" level="false">
            <width sOffset="0.0" a="3.5" b="0.0" c="0.0" d="0.0"/>
          </lane>
        </left>
        <right>
          <lane id="-1" type="driving" level="false">
            <width sOffset="0.0" a="3.5" b="0.0" c="0.0" d="0.0"/>
          </lane>
        </right>
      </laneSection>
    </lanes>
  </road>
</OpenDRIVE>
```



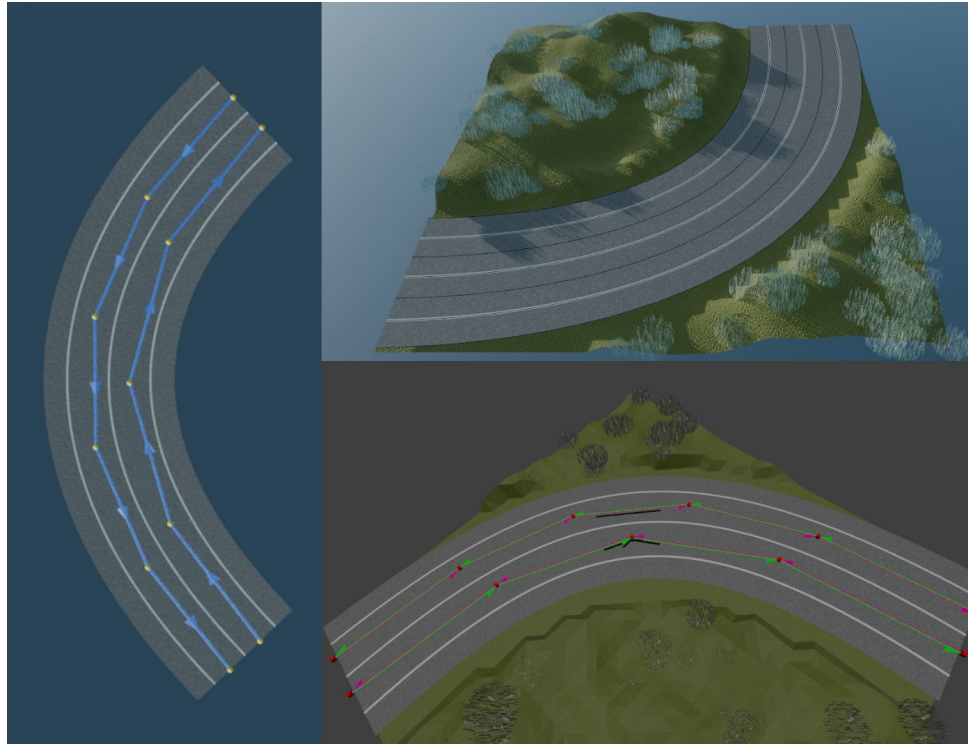
## Appendix C

### Ten Road Blocks Tested in VRUT

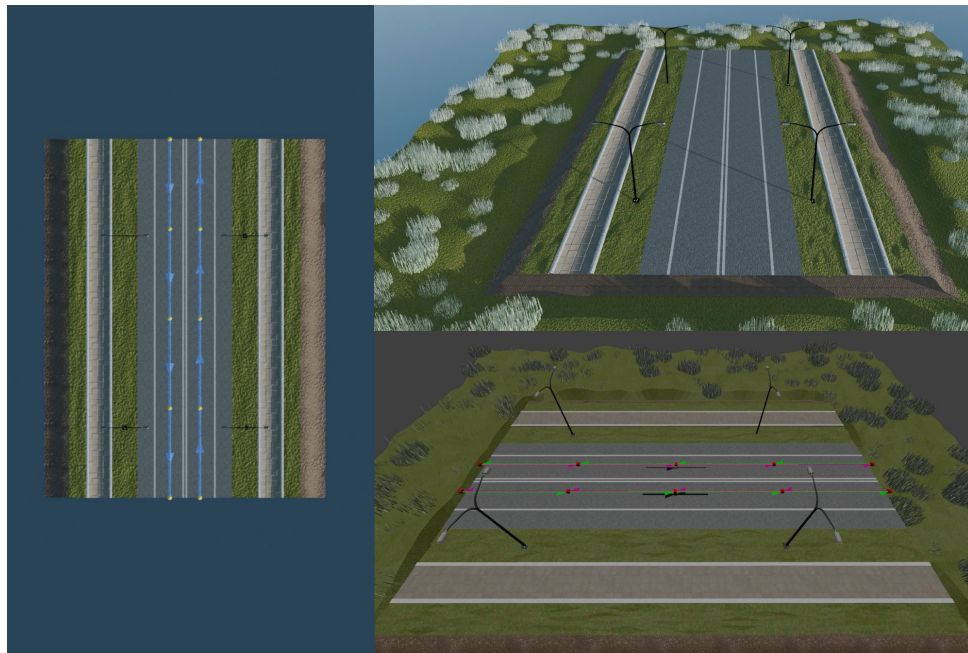
*Disclaimer: Images with dark blue background are top-down views of the roadblocks in Blender with the road network visualized. Images with a light blue background are screenshots from Blender showing the roadblocks geometry. Images with a dark gray background are screenshots from VRUT showing the roadblocks in the traffic simulation environment with the road network visualized.*



**Figure C.1:** Road Block 1: Simple straight road

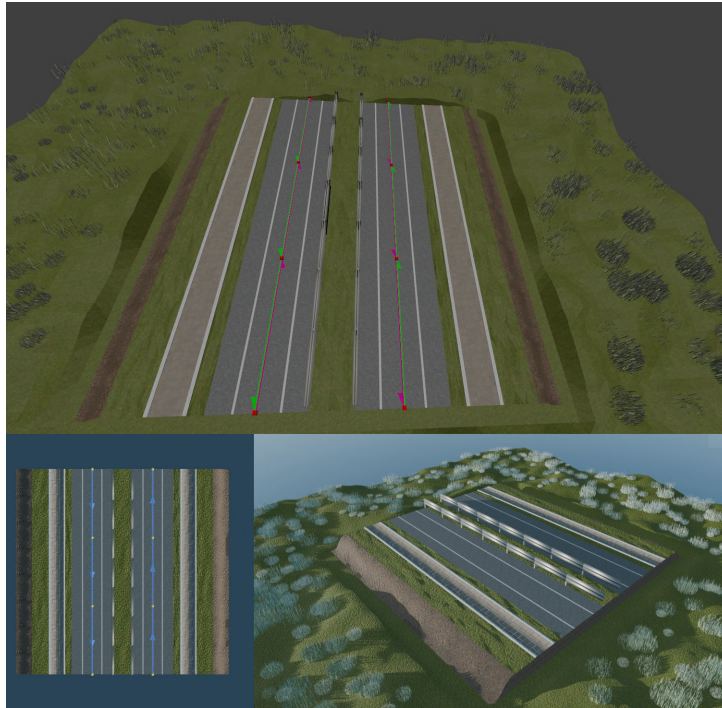


**Figure C.2:** Road Block 2: Simple curved road



**Figure C.3:** Road Block 3: Straight road with features along the sides

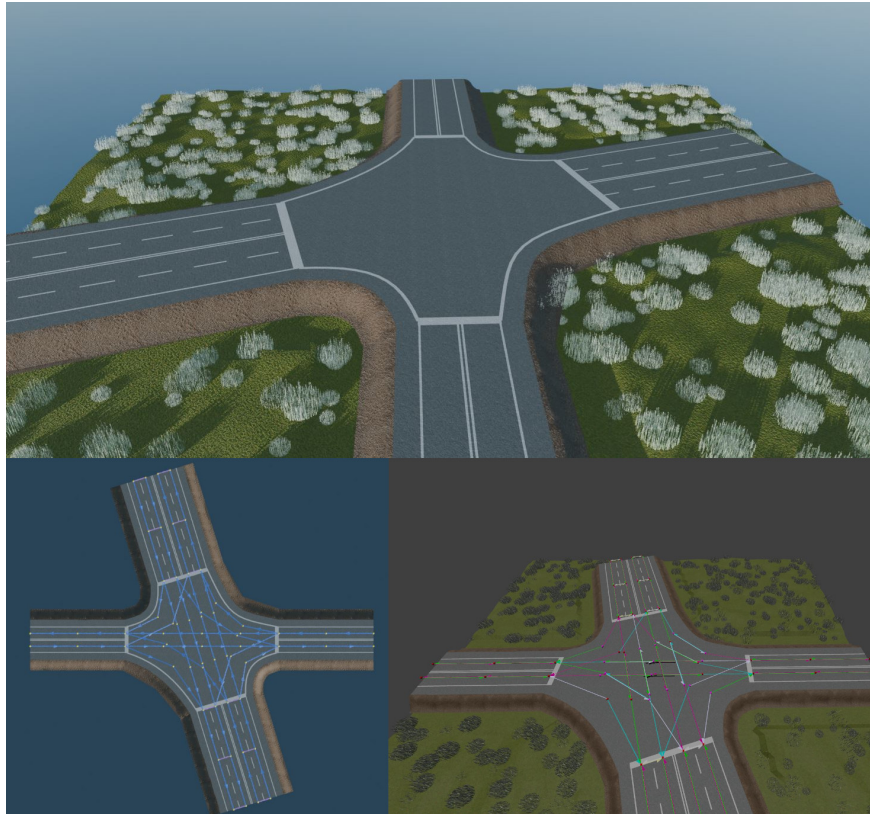




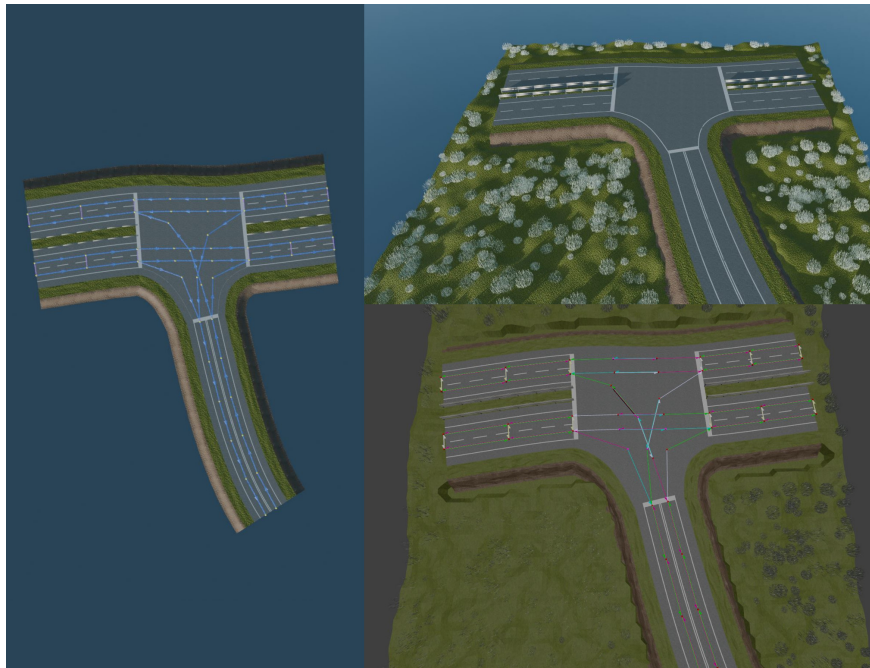
**Figure C.4:** Road Block 4: Straight road with features between lanes as well as along the sides



**Figure C.5:** Road Block 5: Straight multilane road with multiple features



**Figure C.6:** Road Block 6: X-shaped junction with features

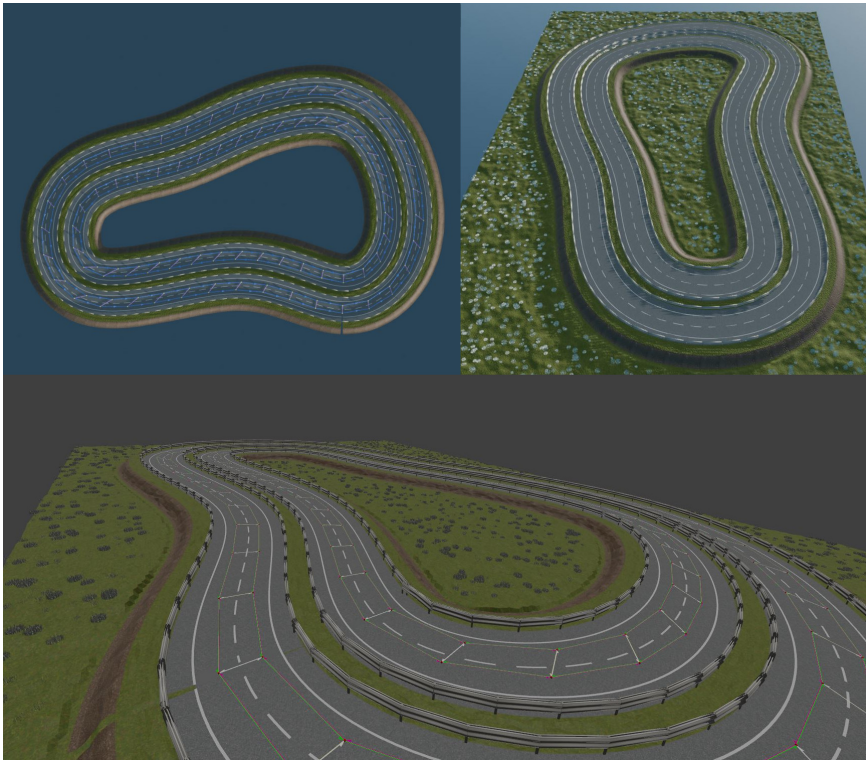


**Figure C.7:** Road Block 7: T-shaped junction with features

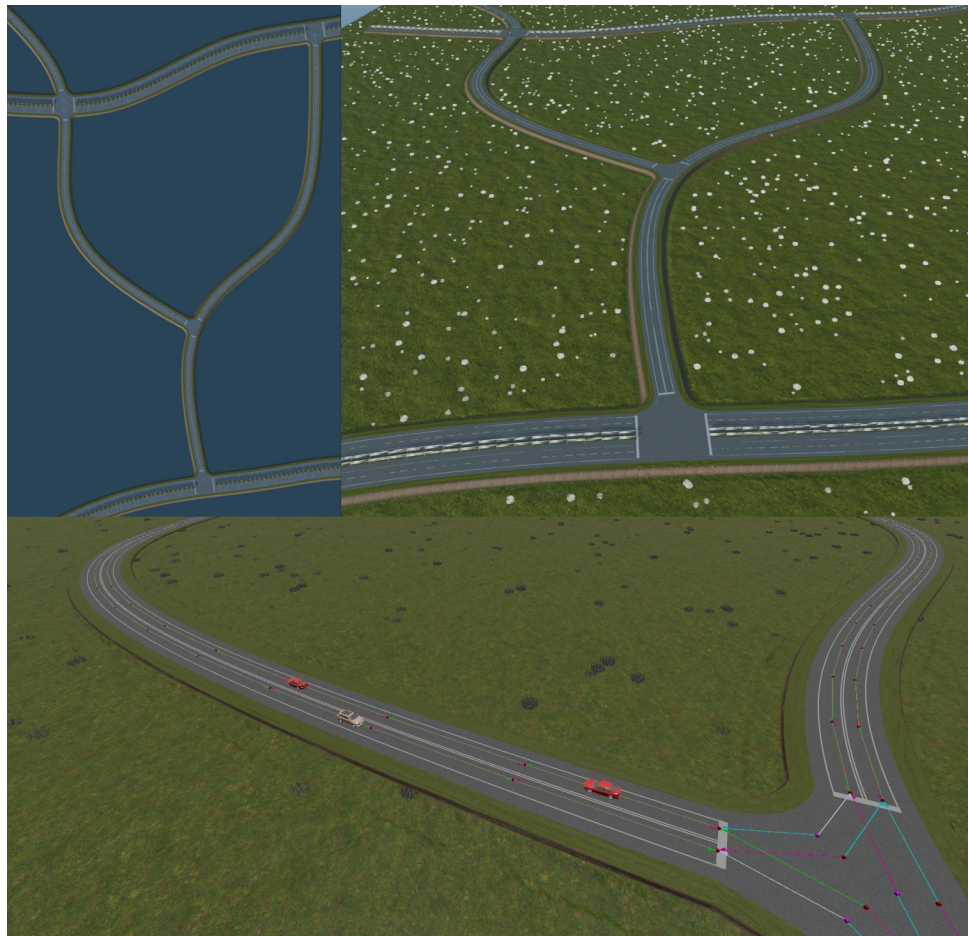




**Figure C.8:** Road Block 8: Junction with connecting lanes and features



**Figure C.9:** Road Block 9: Looped track with features along the sides



**Figure C.10:** Road Block 10: Complex road network with multiple junctions and features



## Appendix D

### Bibliography

- [1] Implicit surface, 2024. Accessed: December 17, 2024.
- [2] 3D Systems. What is an stl file?, 2024. Accessed: December 17, 2024.
- [3] David Aimsun and Contributors. Roadxml repository, 2024. Accessed: 2024-12-23.
- [4] ASAM e.V. *ASAM OpenDRIVE 1.8.1 Specification*, November 21 2024.
- [5] ASAM e.V. ASAM OpenDRIVE standard, 2024.
- [6] Daniel Aschermann. Modulární editor silniční sítě pro vrut. Master’s thesis, Czech Technical University in Prague, Faculty of Electrical Engineering, 2023.
- [7] Autodesk. What is autodesk fbx technology?, 2020.
- [8] Autodesk. Autodesk maya 2023, 2023. Accessed: 2024-12-30.
- [9] Autodesk. Custom attributes in maya, 2023. Accessed: 2024-12-30.
- [10] Autodesk. Export formats in maya, 2023. Accessed: 2024-12-30.
- [11] Autodesk. Nurbs modeling in maya, 2023. Accessed: 2024-12-30.
- [12] Autodesk. Procedural workflows with bifrost in maya, 2023. Accessed: 2024-12-30.
- [13] Autodesk. Python api documentation for maya, 2023. Accessed: 2024-12-30.
- [14] Autodesk. Creating and editing splines, 2025.
- [15] Blender Foundation. About blender, 2024. Accessed: 2024-12-23.
- [16] Blender Foundation. *Blender API: Attributes*, 2024.
- [17] Blender Foundation. *Blender API: bpy.types.Curve(ID)*, 2024.
- [18] Blender Foundation. *Blender API: bpy.types.Curves(ID)*, 2024.

- [19] Blender Foundation. *Blender Python API Documentation*, 2024.
- [20] Blender Foundation. *Introduction to Modifiers*, 2024. Accessed: 2024-12-23.
- [21] Blender Foundation. *Mesh Structure*. Blender Foundation, 2024. Accessed: December 17, 2024.
- [22] Blender Foundation. *Render Baking*, 2024.
- [23] Tanita Brustad. Preliminary studies on transition curve geometry: Reality and virtual reality. *Emerging Science Journal*, 4:1–10, 02 2020.
- [24] Tanita Fossli Brustad and Rune Dalmo. Railway transition curves: A review of the state-of-the-art and future research. *Infrastructures*, 5(5), 2020.
- [25] CG build up. Creating intersection points along the intersecting parts of two splines in blender geometry nodes, 2022. Accessed: 2024-08-23.
- [26] Giuseppe Cantisani, Davide Dondi, Giuseppe Loprencipe, and Alessandro Ranzo. Spline curves for geometric modelling of highway design. In *Proceedings of the SIIV Conference*, 2003.
- [27] Stefano Carpin, Mike Lewis, Jijun Wang, Stephen Balakirsky, and Chris Scrapper. Usarsim: a robot simulator for research and education. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 1400–1405, 2007.
- [28] Thomas Collet, Luce Morin, Stéphane Pateux, and C. Labit. Floating polygon soup. pages 1–8, 07 2011.
- [29] Danny Darwiche and Isak Nyström. Finding junctions in spline-based road generation, 2022.
- [30] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, Berlin, Heidelberg, 3rd edition, 2008.
- [31] DAVID H DOUGLAS and THOMAS K PEUCKER. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica*, 10(2):112–122, 1973.
- [32] Dr. Huang’s Mathematics and Science Resources. Parametric surface, 2024.
- [33] Ridhima Dutta and Poonam Dhand. 3d object representations, 2024. Accessed: December 17, 2024.
- [34] Encyclopedia.com. Information technology standards, 2024. Accessed: 2024-12-23.

- [35] Engineering.com. What is solid modeling?, 2024.
- [36] Gerald Farin, Josef Hoschek, and Myung-Soo Kim. *Interactive Curve Modeling: With Applications to Computer Graphics, Vision and Image Processing*. Springer, London, 2008.
- [37] Dalai Felinto et al. New curves object type, 2019.
- [38] Adam Finkelstein. Overview of 3d object representations. COS 426 Lecture Slides, Princeton University, 2005. Accessed: December 17, 2024.
- [39] Blender Foundation. Add-on tutorial for blender, 2024. Accessed: 2024-12-30.
- [40] Blender Foundation. Attributes in blender, 2024. Accessed: 2024-12-30.
- [41] Blender Foundation. Bmesh module documentation, 2024. Accessed: 2024-12-30.
- [42] Blender Foundation. Geometry nodes documentation, 2024. Accessed: 2024-12-30.
- [43] Blender Foundation. Importing & exporting files, 2024. Accessed: 2024-12-30.
- [44] Jonas Freiknecht and Wolfgang Effelsberg. A survey on the procedural generation of virtual worlds. *Multimodal Technologies and Interaction*, 1(4):27, 2018.
- [45] Christina Gackstatter, Sven Thomas, Patrick Heinemann, and Gudrun Klinker. Stable road lane model based on clothoids. In *Proceedings of the 10th International Conference on Advanced Microsystems for Automotive Applications (AMAA)*, 2010.
- [46] Epic Games. Procedural mesh component, 2024. Accessed: 2024-12-30.
- [47] Epic Games. Unreal documentation, 2024. Accessed: 2024-12-30.
- [48] Epic Games. Uspline component, 2024. Accessed: 2024-12-30.
- [49] Mengran Gao, Ningjun Ruan, Junpeng Shi, and Wanli Zhou. Deep neural network for 3d shape classification based on mesh feature. *Sensors*, 22(18), 2022.
- [50] GeoGebra. Curves and their properties, 2024.
- [51] Andy Green. Polygon soup – for 3d artists, 2024. Accessed: December 17, 2024.
- [52] Seyed Masoud Hosseini Sarvari. Optimal geometry design of radiative enclosures using the genetic algorithm. *Numerical Heat Transfer, Part A: Applications*:127–143, 07 2007.

- [53] James F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, 3rd edition, 2014.
- [54] Gary Keen. Texture maps, 2024.
- [55] Khronos Group. *COLLADA - Digital Asset Exchange Schema: Version 1.5 Specification*, 2008.
- [56] Khronos Group. *glTF 2.0 Specification*, 2020.
- [57] Vojtěch Kolínský. Editor silniční sítě v systému virtual reality universal toolkit (road network editor for virtual reality universal toolkit), 2024.
- [58] Václav Kyba. Modulární 3d prohlížeč. Master's thesis, Czech Technical University in Prague, Faculty of Electrical Engineering, November 2008.
- [59] Vít Kyba, Aleš Míšek, et al. Documentation of the vrut application. Accessed: 2024-12-23.
- [60] Marko Lamot and Borut Žalik. An overview of triangulation algorithms for simple polygons. *1999 IEEE International Conference on Information Visualization (Cat. No. PR00210)*, pages 153–158, 1999.
- [61] Raph Levien. The euler spiral: A mathematical history. Technical Report EECS-2008-111, University of California, Berkeley, 2008.
- [62] Library of Congress. Stl (stereolithography) file format family, 2024.
- [63] J. McCrae and K. Singh. Sketching piecewise clothoid curves. In C. Alvarado and M.-P. Cani, editors, *EUROGRAPHICS Workshop on Sketch-Based Interfaces and Modeling*, 2008.
- [64] Jaroslav Minařík. Simulace okolních dopravních dějů (simulation of surrounding traffic). Master's thesis, Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Computer Graphics and Interaction, 2014.
- [65] James D. Murray and William VanRyper. *Graphics File Formats*. O'Reilly Media, 2nd edition, 1996.
- [66] OpenStreetMap Contributors. Map features, 2024. Accessed: 2024-12-23.
- [67] David Salomon. *The Computer Graphics Manual*. Springer, 1st edition, 2011. An ebook version is available for subscribers.
- [68] Shene, C.-K. B-spline: Special case, 2024.
- [69] Peter Shirley, Michael Ashikhmin, and Steve Marschner. *Fundamentals of Computer Graphics*. A K Peters/CRC Press, 3rd edition, 2009.
- [70] SideFX. Attributes in houdini documentation, 2024. Accessed: 2024-12-30.

- [71] SideFX. Curve sop documentation, 2024. Accessed: 2024-12-30.
- [72] SideFX. Digital assets documentation, 2024. Accessed: 2024-12-30.
- [73] SideFX. Exporting documentation, 2024. Accessed: 2024-12-30.
- [74] SideFX. Geometry nodes documentation, 2024. Accessed: 2024-12-30.
- [75] SideFX. Houdini documentation, 2024. Accessed: 2024-12-30.
- [76] SideFX. Polyextrude sop documentation, 2024. Accessed: 2024-12-30.
- [77] SideFX. Vex language documentation, 2024. Accessed: 2024-12-30.
- [78] Signum Ops. Polylines help documentation, 2024.
- [79] Gregory J. Taylor. Roadway horizontal alignment. Technical Report C04-034, Continuing Education and Development, Inc., 2023. Accessed: 2024-12-19.
- [80] Unity Technologies. Fbx exporter package in unity, 2024. Accessed: 2024-12-30.
- [81] Unity Technologies. Mesh class documentation in unity, 2024. Accessed: 2024-12-30.
- [82] Unity Technologies. Probuilder package in unity, 2024. Accessed: 2024-12-30.
- [83] Unity Technologies. Scriptableobjects in unity, 2024. Accessed: 2024-12-30.
- [84] Unity Technologies. Spline package in unity, 2024. Accessed: 2024-12-30.
- [85] Unity Technologies. Unity 6 user manual, 2024. Accessed: 2024-12-30.
- [86] True Geometry. Radius of curvature in context of road radius calculation, 2024.
- [87] TutorialsPoint. Computer graphics - curves, 2024.
- [88] Zhengren Wang. 3d representation methods: A survey. *arXiv preprint arXiv:2410.06475*, 2024. Accessed: December 17, 2024.
- [89] Wikimedia Commons contributors. Csg tree, 2005.
- [90] Wikipedia Contributors. Simulation open framework architecture, 2024. Accessed: 2024-12-23.
- [91] Wikipedia Contributors. Vector map, 2024. Accessed: 2024-12-23.
- [92] Wolfram MathWorld. Cornu spiral, 2024.
- [93] Cem Yuksel, Scott Schaefer, and John Keyser. Parameterization and applications of catmull-rom curves. *Comput. Aided Des.*, 43:747–755, 2011.