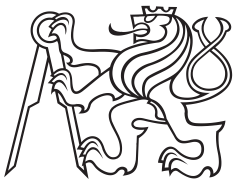
Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Modular road graph editor for VRUT
(Virtual Reality Universal Toolkit)

Daniel Aschermann

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
January 2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Aschermann** Jméno: **Daniel** Osobní číslo: **474486**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačová grafika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Modulární editor silniční sítě pro VRUT

Název diplomové práce anglicky:

Modular Road Network Editor for VRUT

Pokyny pro vypracování:

Prostudujte metody používané pro reprezentaci silniční sítě v mapových a navigačních systémech. Zmapujte používané datové struktury a související atributy potřebné pro popis umožňující efektivní dopravní simulaci s využitím těchto dat. Navrhněte interaktivní editor, který umožní komponovat silniční síť pomocí skládání předpřipravených bloků (modulů), které reprezentují důležité části silniční sítě. Tyto bloky budou reprezentovat úseky silnic o daném počtu jízdních pruhů, různé typy křižovatek, kruhové objezdy nebo dálniční sjezdy. Vytvořte základní geometrické modely těchto bloků a související komponenty silničního grafu. Navrhněte způsob aplikace deformačních transformací pro interaktivní přizpůsobení zakřivení jednotlivých úseků silniční sítě. Zmapujte možnost zvýšení realističnosti výsledných modelů pomocí procedurálního generovaní vizuálních detailů do předpřipravených bloků.
V systému Virtual Reality Universal Toolkit (VRUT) implementujte uživatelsky přívětivý editor, který umožní intuitivně komponovat silniční síť pomocí výše uvedených bloků, jejich snadné navazování a deformování. Otestujte funkčnost editoru vytvořením nejméně tří různých silničních sítí.

Seznam doporučené literatury:

[1] Paden, B., Čáp, 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.
[2] Projekt OpenDrive. http://www.opendrive.org/
[3] David Tichý. Simulátor jízdy městem. Diplomová práce ČVUT FEL, 2006.
[4] Václav Kyba. Modulární 3D prohlížeč. Diplomová práce ČVUT FEL, 2008.
[5] Jaroslav Minařík. Simulace okolních dopravních dějů. Diplomová práce ČVUT FEL, 2014.
[6] Alena Mikushina. Tvorba modulárních 3D komponent pro videohry, Bakalářská práce ČVUT FEL, 2020.
[7] Vojtěch Kolínský. Editor silniční sítě v systému Virtual Reality Universal Toolkit, Bakalářská práce ČVUT FEL, 2020.
[8] Klára Pudová. Automatická tvorba provozu ve scénářích pro vozidlové simulátory, Diplomová práce ČVUT FD, 2019.

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Jiří Bittner, Ph.D. Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **12.09.2022** Termín odevzdání diplomové práce: _____

Platnost zadání diplomové práce: **19.02.2024**

doc. Ing. Jiří Bittner, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

29/11/22
Datum převzetí zadání

Podpis studenta

Acknowledgements

Many thanks to Doc. Ing. Jiří Bittner, Ph.D. for patience, helpfulness and guidance in this work. Further many thanks to Mgr. Antonín Míšek, Ph.D. and Ing. Jaroslav Sloup for help with the development and to my family for the great support.

Declaration

I declare that I developed the master's thesis entitled Modular road graph editor for VRUT independently and have listed all the used literature.

In Prague, 10. January 2023

.....
signature

Abstract

Today's technology moves unstoppably forward and the autonomous car industries have a large share. The car industry is focused on traffic simulation's evolution as it is a safe way to test both the car and the driver. This master's thesis describes possible approaches in traffic simulations and the creation of road networks in order to make autonomous cars drive them. For road network creation in this thesis, we use a block solution that divides complex traffic problems in order to create a road network. The thesis focuses on studying methods for traffic network representation and implementation of such a module in the Virtual Reality Universal Toolkit engine. The main goal was to develop an interactive tool for road network creation and deformation. Problems like junctions and road graph definition have been resolved and implemented in the solution. Implementation is tested by creating six different sample road networks with parameters and using the procedure listed. All scenes differ in size, complexity and whether the deformation was used or not.

Keywords: VRUT (Virtual Reality Universal Toolkit), Road Graph, Traffic Simulation, GUI (Graphical User Interface), MTS (Mobile Traffic System), Deformation

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

Abstrakt

Dnešní technologie se stále rozvíjí a autonomní mobilní průmysl má na vývoji velký podíl. Automobilový průmysl se zaměřuje na vývoj simulace provozu, protože se jedná o bezpečný způsob testování jak vozu, tak i řidiče. Tato diplomová práce popisuje možné přístupy v dopravních simulacích a vytváření silničních sítí tak, aby je autonomní vozy mohly řídit. Pro tvorbu silniční sítě v této práci používáme blokové řešení, které rozmělňuje složité dopravní problémy za účelem vytvoření silniční sítě. Práce se zaměřuje na studii metod pro reprezentaci dopravní sítě a implementaci modulu v aplikaci Virtual Reality Universal Toolkit. Hlavním cílem bylo vyvinout interaktivní nástroj pro tvorbu a deformaci silniční sítě. Problémy jako křižovatky a definice silničního grafu byly vyřešeny a implementovány do řešení. Implementace byla testována vytvořením šesti různých vzorových silničních sítí s parametry a použitím uvedeného postupu. Všechny scény se liší velikostí, složitostí a tím, zda byla či nebyla použita deformace.

Klíčová slova: VRUT (Virtual Reality Universal Toolkit), Road Graph, Traffic Simulation, GUI (Graphical User Interface), MTS (Mobile Traffic System), Deformace

Překlad názvu: Modulární editor silniční sítě pro VRUT (Virtual Reality Universal Toolkit)

Contents

Figures

Tables

Chapter 1

Introduction

This chapter contains the concise introduction of the assignment and its required research and implementation of this project. The chapter also consists of a slight description of all the remaining chapters.

Today's car industry is enormously branched and is directed towards autonomous cars in general. Even though these cars might be already capable of driving us wherever we want even better, faster and safer than we could, mankind is not yet ready for such a leap.

Therefore, car industries started working on a smoother transition between man-driven and autonomous vehicles and began developing software and engines that simulate real traffic situations for testing human and autonomous cars.

To maximally simulate and imitate any traffic situation, multiple ways do exist to create such an environment. There are some standards for defining such situations, however, every company has its approach.

In Škoda AUTO company there is an engine that uses a traffic network for such cases and is greatly variable enabling any road situation simulation.

1.1 Motivation

Currently, there is a large progression in the automotive industry. I started working at DigiteqAutomotive, which is Škoda Auto's daughter company. In UXLab department where I work, we design simulations for both VR and physical cockpit simulators to test onboard computers' UI. In addition, I have been developing a GeometryEditor module for VRUT as my bachelor thesis. All this together gave me the motivation and sufficient knowledge to develop RoadEditor module for VRUT.

1.2 Assignment

The aim of this project is to study the methods used for traffic network representation in map and navigation systems. Furthermore, we map used data structures and related attributes requisite for description enabling effective traffic simulation using these data.

We design an interactive editor that enables users to compose a traffic net using pre-prepared blocks (modules) that represent important parts of the traffic net. These blocks will represent road segments of a given number of traffic lanes, different types of intersections, roundabouts, or highway exits. Furthermore, we model fundamental geometry models of these blocks and related traffic net components. We design a way of deformation transformation application for interactive curvature adaptation. We also map the possibility of improving the resulting models' realism using procedural visual details generated for prepared blocks.

In the Virtual Reality Universal Toolkit system (VRUT), we implement a user-friendly editor that enables intuitive traffic net composition using blocks described above, their easy connecting and deformation. Finally, we test editor functionality by creating at least three different traffic nets.

1.3 Work structure

This thesis focuses on other tools that are being developed in the automotive industry for creating driving simulators. A portion of this research is used for the RoadEditor module development for VRUT.

The Chapter 2 maps all other already implemented tools and modules used for road graphs creations and editions including ways of possible traffic representations in VRUT and other engines.

Afterwards, Chapter 3 introduces VRUT 2.0 and highlights some relevant information and modules in the road edition spectrum.

Chapter 4 discusses the idea and possible implementation of the RoadEditor module. It is a brief and decent presumption of how the RoadEditor will work. As for any kind of development, this chapter serves as a concept by which the RoadEditor module itself is inspired.

Chapter 5 describes in detail every part of the implementation and its purpose and usage. There are also already some possible future improvements mentioned as it is related to some of the topics.

In the end, Chapter 6 sums up all the work that has been done and presents a couple of results in the form of traffic networks generated using the RoadEditor module.

Finally, Chapter 7 discusses about the problems during the implementation and possible improvements and Chapter 8 gives a conclusion focusing on the development of the module and discusses possible future for the module and its improvements.

Chapter 2

Road and map navigation system representation

This chapter briefly introduces the problem and how it is approached in other software and modules.

2.1 Other traffic editors

There are many tools used to create and edit traffic networks for simulation purposes. The result of most of these is the same, however the way of building such a network differs. Some of these are based on the idea of the blocks whereas the other are based on a user-defined curve along which the blocks are then generated as is shown in Figure 2.1.

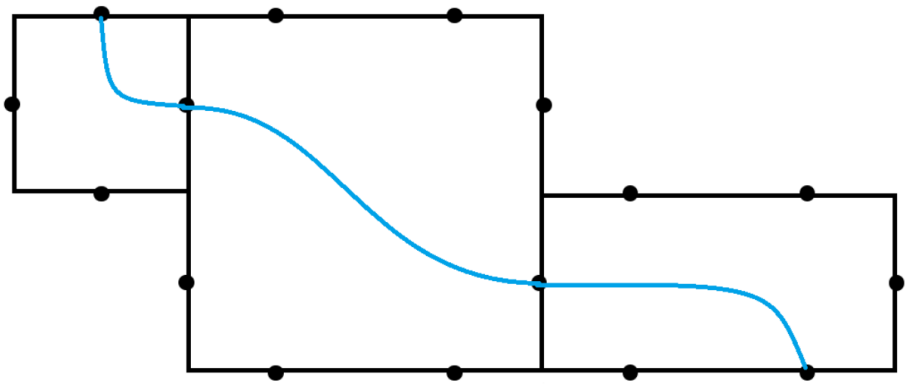


Figure 2.1: Blocks creation along the predefined curve by *thesis of Pudova* [10].

VRUT contains an implemented module that allows users to create and edit traffic networks. This module is mostly used for node segment editions, although its use is not very interactive.

Most of the operations and adjustments that a user is capable of doing using this module are limited to not very user-friendly looking GUI, nevertheless they function properly, so the module serves its purpose with a good way of use.

To summarize, some more interactive ways of editing road graphs would improve user experience during such actions as creating and editing the road graph. The GUI of the road graph editor is shown in Figure 2.2.

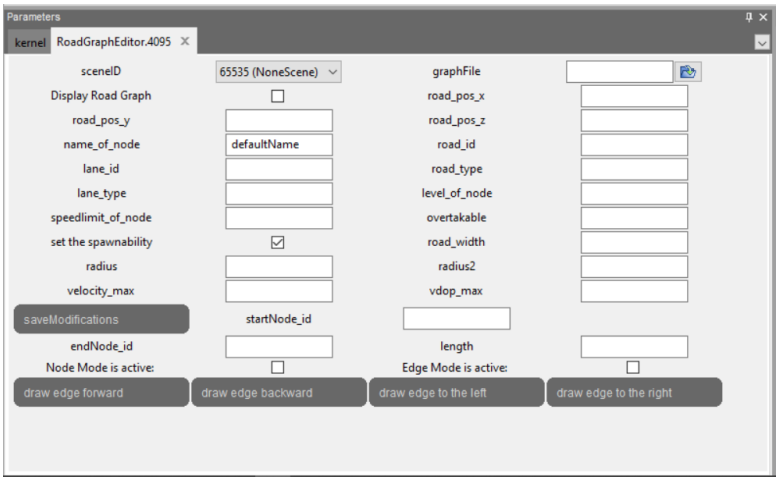


Figure 2.2: Gui of RoadGraphEditor module by *thesis of Kolínský* [9].

2.2 ASAM OpenDRIVE

OpenDRIVE [2] is a standardized format for describing road networks in XML syntax, with the purpose of facilitating data exchange between different driving simulators. It provides a road network description that can be used to develop and validate Advanced Driver Assistance Systems (ADAS) and AD features.

The structure consists of nodes that have the capability to incorporate data defined by the user, allowing for specialization while maintaining interoperability. ASAM OpenDRIVE can be used to construct road networks by connecting individual road sections and can be complemented by other standards for static 3D roadside objects and dynamic content.
Composition of a road network defined by OpenDRIVE is shown in Figure 2.3.

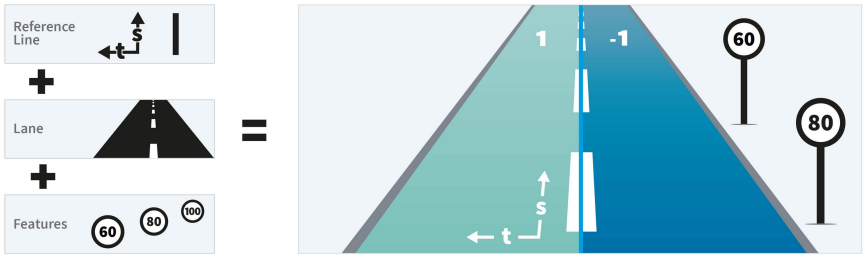


Figure 2.3: Elements of road definition from *project OpenDrive* [2].

Junctions in ASAM OpenDRIVE are represented by <junction> elements and connecting roads by <connection> elements. There are rules for common junctions that apply to clarify ambiguities in linking roads. In Figure 2.4 is shown a complex six-lane junction example.

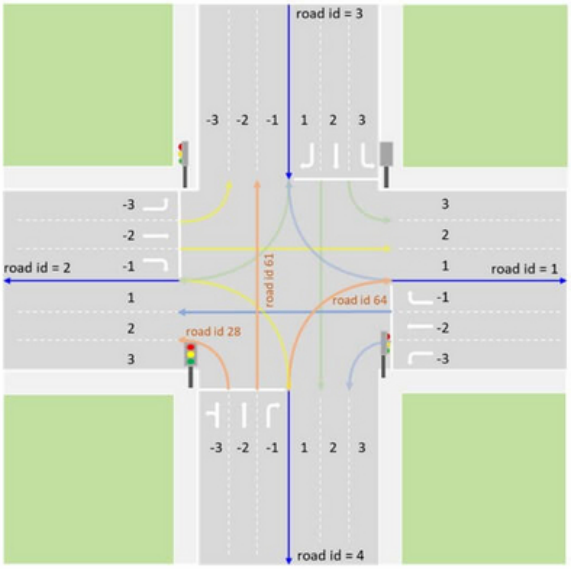


Figure 2.4: Example of a junction in *project OpenDrive* [2].

2.3 EasyRoads (Unity)

EasyRoads is a plugin for Unity that allows users to interactively create any road network including multilane highways and any type of junctions.

The EasyRoads plugin also offers the user to generate surroundings and vegetation. The plugin has multiple tools such as brushes and objects to place on the predefined plane. The package is really advanced and offers an enormous number of road types, side objects, vegetation and more. In Figure 2.5 is an example of a road creation. The user creates a new road by clicking on the plane with **Alt** pressed. Road shape is based on a spline curve and deforms the road shape by pre-clicked control points and interpolation between them.

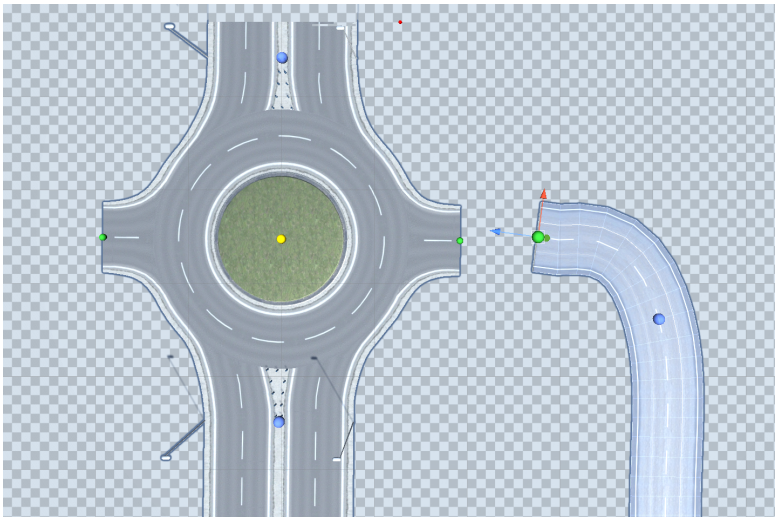


Figure 2.5: Traffic creation example by *EasyRoads* [23].

With more work and scene editing, results with the EasyRoads plugin get to perfection. Naturally, the vegetation and environment give the resulting models authenticity, however bridges, tunnels, crash barriers and other objects like signs have their contribution as well. Example of a scene with highway, vegetation and surroundings created by EasyRoads plugin is shown in Figure 2.6.

Figure 2.6: Highway creation result example by *EasyRoads* [23].

2.4 Mobile Traffic System (Unity)

MTS (Mobile Traffic System) is an Extension Asset for Unity that lets users generate and simulate traffic systems. It is pretty simple to use and easy to learn within hours. The main advantage of this plugin is that it can generate traffic nodes according to EasyRoads predefined roads. In case a certain junction or interconnection is not defined in MTS, the user can add new nodes or adjust the position of any traffic node that is already generated interactively.

The plugin offers multiple scene setups and allows multiple options for traffic density, grid layout, system layers, etc. Although there are already many options, the plugin is not complete. Because of that, further extensions are being developed. There will be further information about this section later in the Discussion section. In Figure 2.7 is shown MTS plugin's menu for the traffic node adjustments and initial scene parameters setting.

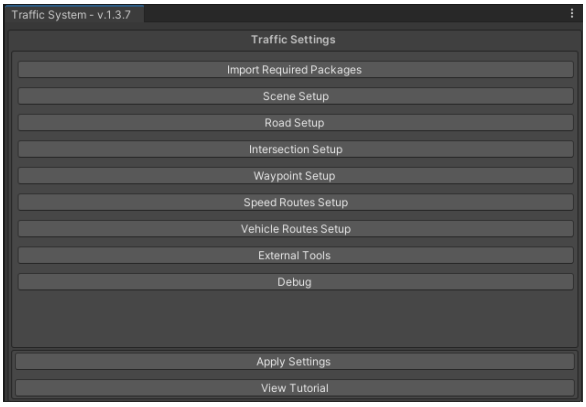


Figure 2.7: Traffic system menu by Gley *Mobile Traffic System* [24].

Unlike the RoadGraph module in VRUT, Mobile Traffic System depicts traffic nodes in real-time and lets the user edit the network interactively. Cars then simply follow by MTS predefined roads like in VRUT (shown in Figure 2.8).

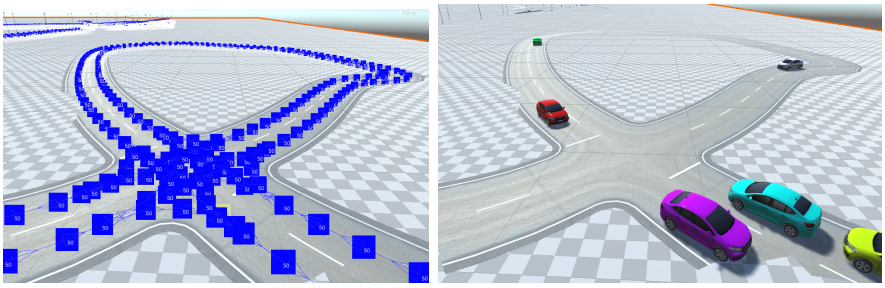


Figure 2.8: Intersection example generation by *Mobile Traffic System* [24].

Nodes can be selected and their parameters can be easily set in the Inspector window (Figure 2.9). There are parameters like max speed, neighbors and previous nodes, allowed cars, give way, spawn prioritization, and other lanes nodes.

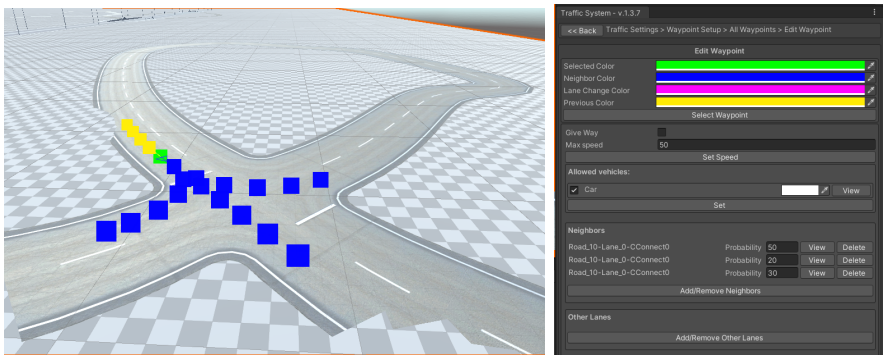


Figure 2.9: Taffic node selection and its menu for parameter settings by *Mobile Traffic System* [24].

In Figure 2.10 are depicted generated nodes along the highway. Although the highway using EasyRoads has to be correctly defined, the road width, speed limit, etc. are converted to the MTS-generated network.

The density of the nodes can also be defined in the modules window as the interconnection between neighbor lanes that are called *other lane*.

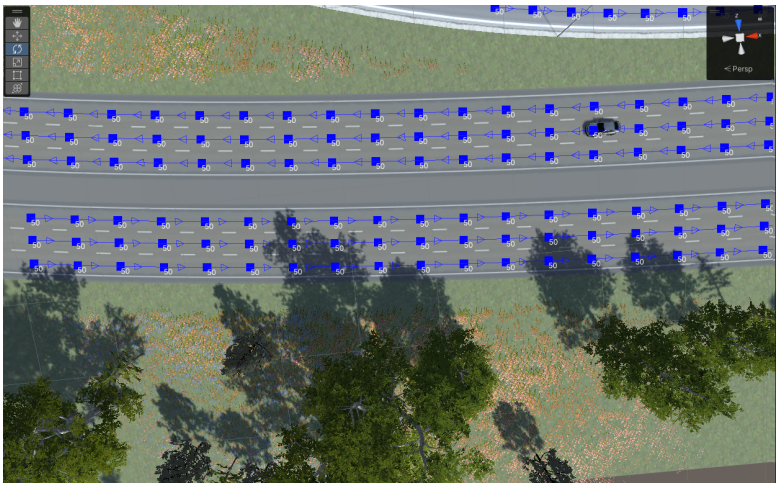


Figure 2.10: Nodes generated by traffic system according to EasyRoads by *Mobile Traffic System* [24].

When a node is selected, the module depicts previous nodes, following nodes, and other lanes nodes. These can be interactively redefined for special cases or whilst interconnecting two different road types. Examples of interactive node selection in MTS are both in Figure 2.9 and Figure 2.11.

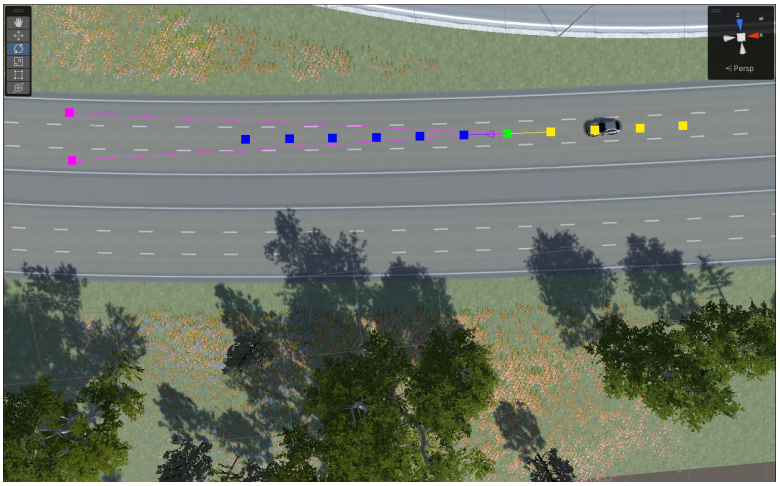


Figure 2.11: Selected nodes predecessors, neighbors, and other lanes nodes by *Mobile Traffic System* [24].

Chapter 3

Introduction to VRUT 2.0

VRUT[1] is application for visualization and editation of 3D data created by Škoda Auto and with participation of other universities and companies.

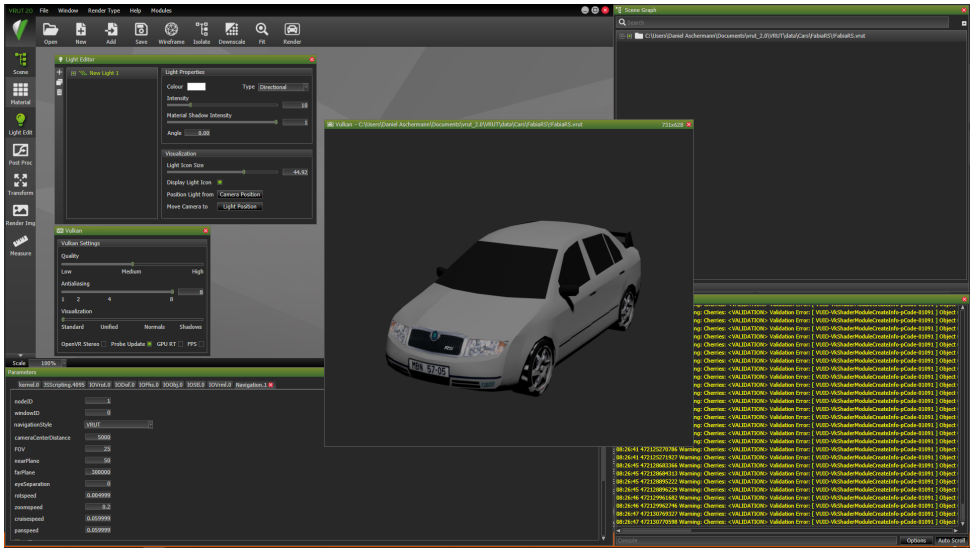


Figure 3.1: VRUT 2.0 - working space example.

The project VRUT 2.0 originated in cooperation with the Department of Computer Graphics and Interaction of CTU FEL with Škoda Auto. The essence of the application stems from the rendering of graphics data and support for modules. The modules extend the functionality of the main application with relative independence on it.

The abbreviation VRUT stands for *Virtual Reality Universal Toolkit*.

[1]Virtual Reality Universal Toolkit

3.1 Building and compilation of VRUT 2.0

VRUT application can be compiled using CMAKE and requires CUDA, GLEW, ODE, and Vulkan libraries for default compilation. VRUT offers various modules to include in the compilation.

```
VRUT_MODULE_IO_IOJTTK
VRUT_MODULE_IO_IOOBJ
VRUT_MODULE_IO_IOOSB
VRUT_MODULE_IO_IOPLMXML
VRUT_MODULE_IO_IOPLY
VRUT_MODULE_IO_IORDF
VRUT_MODULE_IO_IOSPATIAL
VRUT_MODULE_IO_IOSTL
VRUT_MODULE_IO_IOVRML
VRUT_MODULE_IO_IOVRUT
VRUT_MODULE_IO_IOWIRE
VRUT_MODULE_IO_IOXTEX
VRUT_MODULE_JSSCRIPTING
VRUT_MODULE_LEAPMOTION
VRUT_MODULE_LIGHTEDITOR
VRUT_MODULE_MANIP_NAVIGATION
VRUT_MODULE_MATERIALASSIGN
VRUT_MODULE_MATERIALEDITOR
VRUT_MODULE_MEASURE
VRUT_MODULE_NETAPI
VRUT_MODULE_OPTIMIZE
VRUT_MODULE_POSTPROCESS
VRUT_MODULE_RENDERIMAGE
VRUT_MODULE_RENDER_DXRENDER
VRUT_MODULE_RENDER_RAYTRACER
VRUT_MODULE_RENDER_RENDERSTREAM
VRUT_MODULE_RENDER_VRENDER
VRUT_MODULE_RENDER_VRENDER_SHADER_TOOLS
VRUT_MODULE_ROADEDITOR
VRUT_MODULE_SCENEGRAPH
VRUT_MODULE_SEATSTITCHER
VRUT_MODULE_SHADOWCALCULATOR
```

Figure 3.2: CMAKE sample of modules offer.

3.2 How does VRUT 2.0 work?

Main and permanently running part of VRUT is *Kernel*, which represents the essential part of the application. The Kernel provides resources and events to running modules of the application and these can then communicate with the Kernel simultaneously.

Each module then represents different functionality of the application. This project deals with road creation and edition. Other modules are almost indispensable to use not only along with Road Editor, whose implementation is the aim of this project, but in any standalone scene rendered in VRUT 2.0, such as Light Editor, Scene Graph and one of many render modules as Vulkan.

3.3 User Interface

User Interface (UI) in VRUT 2.0 is easily editable and adaptable. We can choose our layout to create the most satisfactory and friendly interface.

To increase ease of work with VRUT 2.0, users can edit their autoexec.cfg scripts as some of the initial engine operations can be automated. Below you can see the script that is used for testing the Road Editor module.

However, RoadEditor module needs to be run also with traffic.cfg script to have several predefined parameters and the user can easily add it as a parameter. This script starts the traffic module and sets all the necessary parameters to display the traffic network.

```
setloglevel all <- sets the log level

runmodule Tesselator
setparam Tesselator.autoTessellation "All scenes"

setparam IOVrml.importLines 0
setparam IOVrut.saveToOneFile 1

runmodule JSScripting
#setparam JSScripting.scriptPath "driving.js"
setparam JSScripting.scriptPath "traffic.js"
<- sets the modules script for testing RoadEditor

#setparam JSScripting.scriptPath "C:/VRUT_data/skoda_menu.js"
setparam JSScripting.runScript 1

runmodule CarHW
runmodule RoadEditor

runmodule Navigation

setparam Navigation.nearPlane 500
setparam Navigation.farPlane 3000000
setparam Navigation.FOV 70

runmodule VRender
usemodule VRender

importscene "..\data\RoadEditor\RoadEditor_test.vrut"
<- imports initial scene
#runmodule RoadEditor
#runmodule LightEditor

renderscene 0

setparam Navigation.nearPlane 500 <- sets the Navigation module parameters
setparam Navigation.farPlane 3000000
setparam Navigation.FOV 60

#fit 0
```

Listing 3.1: VRUT module parameters and scene setup.

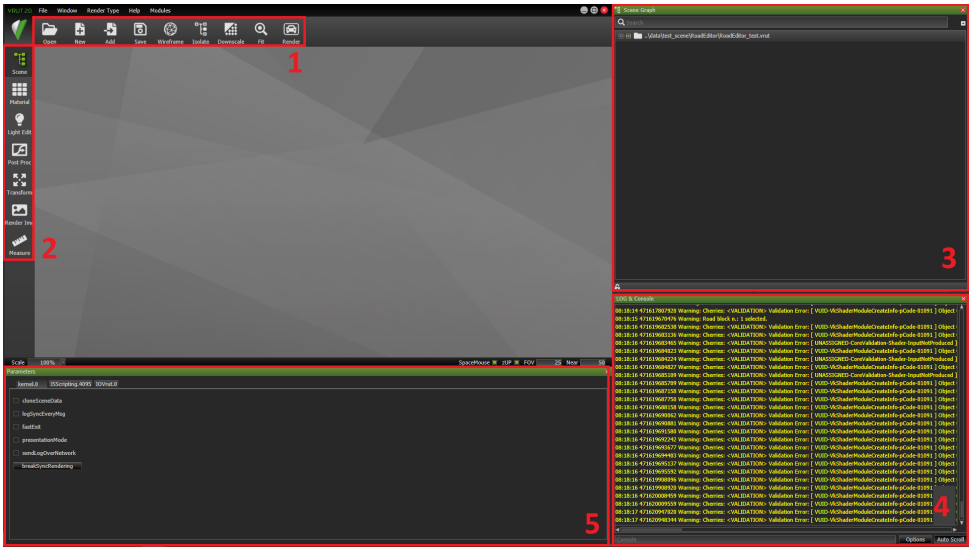


Figure 3.3: VRUT - user interface description; 1 - basic scene operation commands; 2 - available modules; 3 - scene graph module; 4 - LOG & Console; 5 - kernel parameters.

3.4 Other modules in VRUT

As VRUT software is still evolving, the independence of modules is advantageous. The number of modules is enormous.

3.4.1 Module Navigation

Module Navigation allows the user to move in the scene and to set camera parameters. The user needs only a mouse to use the navigation module as the right mouse button moves the camera along the plane given by the camera direction vector and the left mouse button rotates the camera about the given pivot. The module offers numerous settings such as FOV (Field of View), rotation/zoom speed, and far/near plane.

3.4.2 SceneGraph module

SceneGraph module shows the given scene's hierarchy and lets the user do numerous operations on each node and each node type, light, camera, and empty scene node included.

User can move nodes in the hierarchy, delete them and insert new ones, some basic geometry shapes included.

3.4.3 Module Traffic

As there are many operations over the traffic network and its nodes, module Traffic allows the user to load and also export desired graph and its scene.

Such operations also include different parameter settings over nodes spline interpolation, the vehicle control, graph segments displaying and car addition/deletion/parameters settings. The Traffic module also offers to display traffic nodes, which comes in hand while debugging generated networks, which helped during RoadEditor module development. Figure 3.4 shows the Traffic modules GUI.

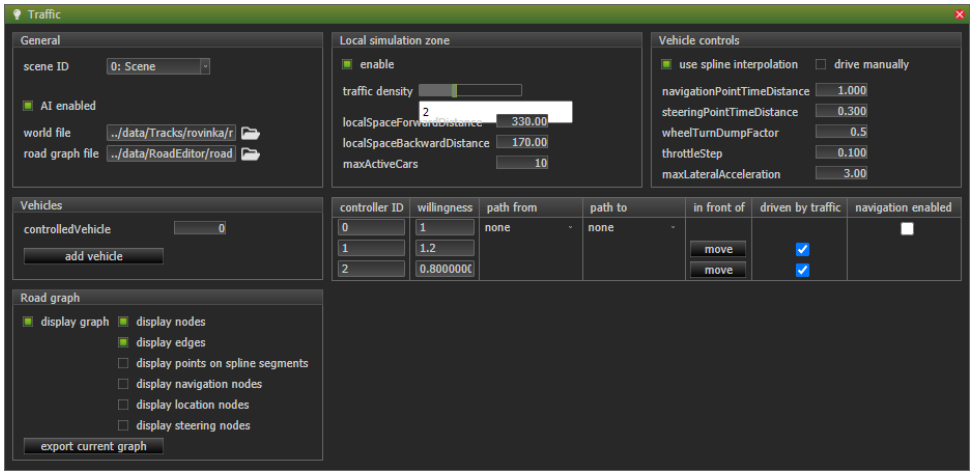


Figure 3.4: Traffic module's GUI, with graph, nodes and edges displayed.

3.4.4 Module VehicleSimulator

VRUT modules are independent and for each functionality, there is a module that provides special functionality. For simulation, there is a VehicleSimulator module that holds the vehicle pool and drives them along the traffic networks. The module also offers multiple parameter settings both for vehicles and the simulation itself.

VehicleSimulator module requires the definition of both traffic network and world file using XML. The traffic network file defining the road graph will be further mentioned later in the section 3.5 as the definition is more complex. On the other hand, the world definition is pretty simple, as the main node has to be defined correctly, the user-driven vehicle's initial position and objects that should be ignored by traffic collision can be defined. World definition also offers to define physics type. Figure 3.5 shows the VehicleSimulator module's GUI while active in the scene.

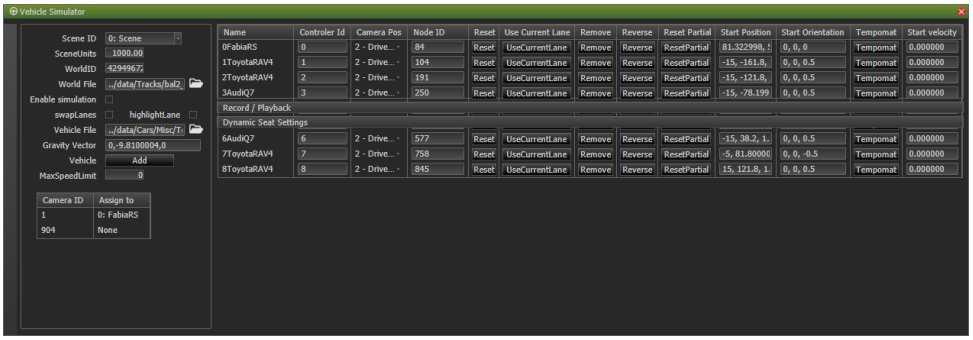


Figure 3.5: VehicleSimulator module's gui depicting in-scene vehicles and their parameters.

3.5 Traffic representation

In VRUT, the traffic network that is used for autonomous vehicles is based on RoadGraph. Such representation has numerous rules and the definition of any complex kind of traffic network might be difficult.

3.5.1 Nodes

Nodes that autonomous vehicles use for defining their path have multiple parameters. Lanes consist of numerous nodes and also have parameters to set up. Road consists of lanes and has only id and type parameters. An abbreviated lane sample is shown bellow in the code snippet.

```
<lane id="1" nodes="3053" type="0" level="3" o="2">
    <node id="3054" x="-3.87186" y="-21.014" z="1" sl="120"
        w="3.6" r="6736.26" r2="10000" vmax="120"/>
    <node id="3055" x="-3.86275" y="-46.0168" z="1" sl="-1"
        w="3.6" r="3.12145e+007" r2="10000" vmax="120"/>
    <node id="3056" x="-3.85366" y="-71.0196" z="0.999999"
        sl="-1" w="3.6" r="3.34467e+006" r2="10000" vmax="120"/>

    <node id="3057" x="-3.84473" y="-96.0995" z="0.999999"
        sl="90" w="3.6" r="6269.86" r2="10000" vmax="120"/>
    ...
    <node id="6104" x="-4.29727" y="48.045" z="1" sl="-1"
        w="3.6" r="896.586" r2="907.247" vmax="120"/>
    <node id="6105" x="-3.91807" y="22.9697" z="1"
        sl="-1" w="3.6" r="1566.33" r2="2023.84" vmax="120"/>
    <node id="6106" x="-3.94037" y="-2.10726" z="1"
        sl="-1" w="3.7" r="4873.21" r2="10000" vmax="120"/>
</lane>
```

3.5.2 Connections

For the vehicles to understand the traffic network structure, the sequences and connections between each node have to be determined. Sequences, interconnections and connections have to be defined so the vehicle can decide to overtake, slow down, speed up, stay in the lane, turn right/left, etc. A short connections sample is shown below in the code snippet.

```
<connections>
    <sequence from="1" to="4" dir="2" closed="0" />
    <sequence from="5" to="8" dir="3" closed="0" />
    <connectiongroup>
        <connection from="17" to="8" type="1" />
        <connection from="18" to="10" type="1" />
        <connection from="10" to="18" type="0" />
        <connection from="16" to="1" type="1" />
    </connectiongroup>
</connections>
```

3.5.3 Junctions

To implement an intersection or any type of junction, a junction tag has to be used. In order to define a junction, path consisting of nodes has to be first defined, and furthermore source node and its successor have to be assigned. Thereafter user can define priority if required.

```
<junction name="T-cross" id="0">
      <path id="0" nodes="2" type="11">
        <node id="0" x="2.75" y="0.65" z="1.7" sl="30" w="3.6"/>

        <node id="1" x="-0.65" y="-2.75" z="1.7" sl="30" w="3.6
            "/>
      </path>
      <laneLink road="1" lane="1" node="9">
        <successor road="1" lane="1" node="6" path="0" />
      </laneLink>
      <priority road="1" lane="1" node="9">
        <check road="1" lane="1" node="2" />
        <check road="1" lane="1" node="7" />
      </priority>
</junction>
```

An example of a road graph generated in VRUT is in scene *dalnice_new* and is depicted in Figure 3.6. The scene is modeled by real road and the graph there is automatically generated along the road textures.

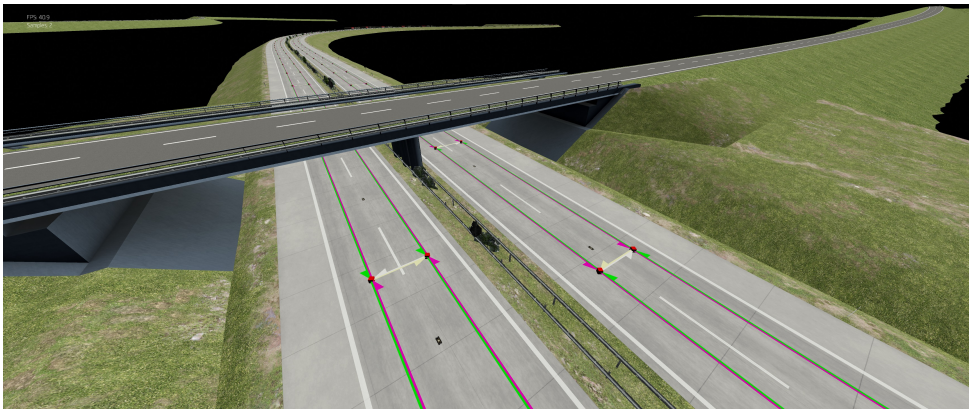


Figure 3.6: Dalnice_new sample scene, traffic nodes generated by the texture parameters.

Chapter 4

Concept design

4.1 General idea

The main purpose of the RoadEditor module implementation is to simplify traffic net creation and edition. There were already numerous modules implemented primarily in the previous VRUT version, for example *Editor silniční sítě v systému VRUT*[1] [9]. This work is focused on the traffic network edition, especially its nodes parameters such as road width, speed, etc.

This module implementation is however focused on a more creative way of designing and generating whole new traffic networks using simple blocks that anyone can use and simply create any kind of network.

Future of such a module could even have a block creation tool implemented and might offer user-friendly interface that allows the creation of custom blocks such as special intersections and road situations. In that case, the hardest part of the implementation would probably rest in traffic nodes connections generation as it is defined using *.xml* files in VRUT.

4.2 Implementation motivation

The ideal way of using the RoadEditor module would be a real-time rendering of traffic network along users blocks positioning, however, such functionality is not yet implemented so for the concept idea I decided to implement a simplified way of the network creation.

However, besides all the parameters that could be set to all the nodes and segments, this module should remain simple and easy to learn to use for any user.

Another important feature would also be a custom block creation, which would be possible with sufficient knowledge of the traffic network representation and also a bit of RoadEditor source code itself.

[1] Road networks editor in system VRUT

4.3 Module design suggestion

Bald and easy proposal of how the module should work was made in Blender (shown in Figure 4.1) as it is a really easy-to-use modeling tool.

Blender mainly served as a modeling tool for blocks used in the RoadEditor module, however, it was also ideal for inventing the core idea of the module's modular approach.

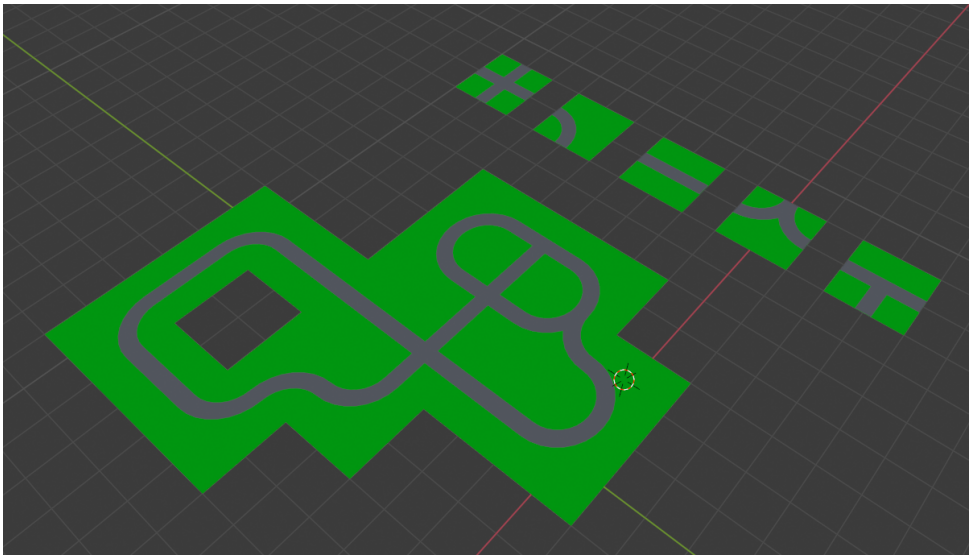


Figure 4.1: Module's blocks design idea (Blender).

4.4 Vegetation generator

Best ways to improve results from such an implementation would be for example including some surroundings, fields, villages, gas stations, etc.

There are many ways of achieving such an improvement to enrich the experience, for example by implementing some cities and village generators or an environment with certain biodiversity. Such a solution is for example implemented in the *thesis by Pudova* [10] (example shown in Figure 4.2).

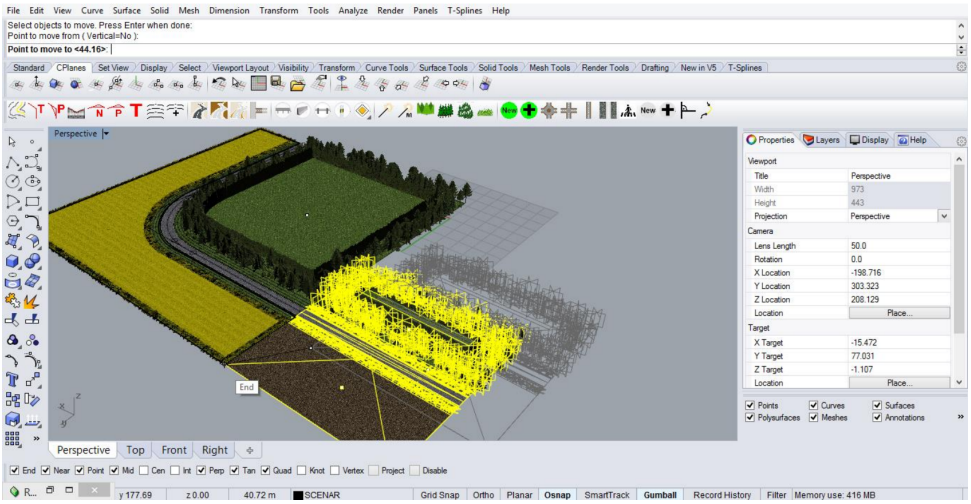


Figure 4.2: Example of a way to generate the road's environment by *thesis of Pudova* [10] (Rhino).

4.5 Adapting blocks

As the implementation went further, numerous other ideas came up. Most of them did stick to the thesis assignment, some were a bit off the grid.

Even though the block solution is appropriate for short testing road sections, it would also come in handy if some kind of block modification would be included in order to adjust the network by a template that would match a real-world network.

4.5.1 Cage based deformations

Cage based deformations [3] are flexible tools for mesh deformation in geometry modeling and computer animation. They allow for easy control of complex models by deforming a cage surrounding the object. Barycentric coordinates, MVC (Mean value coordinates), HC (Harmonic coordinates), and GC (Green coordinates) are methods used in cage based deformations, each with its advantages and disadvantages. MVC is the best for value interpolation, HC is suitable for character articulation, and GC is the best approach for general-purpose in character articulation. Regarding individual approaches, bellow are simple descriptions of each approach.

- Barycentric Coordinates given for point P inside a triangle with vertices A,B and C are given by:

$$\lambda_1 = A_1(A_1 + B_1 + C_1) \tag{4.1}$$
$$\lambda_2 = B_1(A_1 + B_1 + C_1) \tag{4.2}$$
$$\lambda_3 = C_1(A_1 + B_1 + C_1) \tag{4.3}$$
$$P = (\lambda_1, \lambda_2, \lambda_3) \tag{4.4}$$

  where A_1, B_1 and C_1 represent the areas of the triangles formed by the point P and the corresponding vertices.

- Mean value coordinates for point P are computed as follows:

$$\lambda_i = \frac{cot\alpha_i + cot\beta_i}{\sum(cot\alpha_j + cot\beta_j)} \tag{4.5}$$

  where α_i and β_i are the angles between the edge connecting P to the i-th vertex and the two adjacent edges.

- Harmonic coordinates for a point P inside a polygon can be obtained by solving a Laplace equation with Dirichlet boundary conditions. The exact formulas are complex and depend on the specific polygon and its triangulation.

- Green coordinates are a generalization of harmonic coordinates and provide a way to interpolate positions inside a polygonal cage. The exact formulas for Green coordinates are also complex and depend on the specific polygon and its triangulation.

These methods are further compared and described in *Cage based deformations: a survey of Nieto and Susin* [3]. In Figure 4.3 is an example of cage based deformation used on a complex model.

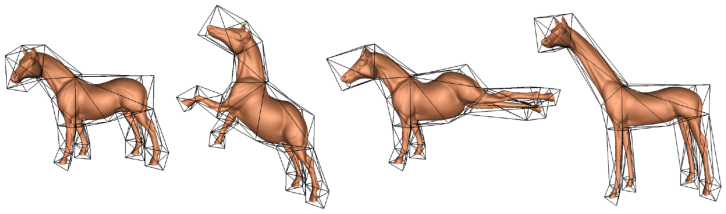


Figure 4.3: Cage based deformation example by *Cage based deformations: a survey of Nieto and Susin* [3].

4.5.2 By curve deformation

Another and perhaps easier way to implement the block deformation would be by the predefined curve. This approach would facilitate slight block deformations, but larger deformations would not be as seamless as with the cage based deformation. New blocks of similar shape would not require their instance creation and rely on already defined block deformation. One of the disadvantages of curve based deformation is that the vegetation and block's surroundings can not be tagged or marked to not have the deformation applied and so the vegetation would have to be generated later with either a different module or with some kind of procedural approach.

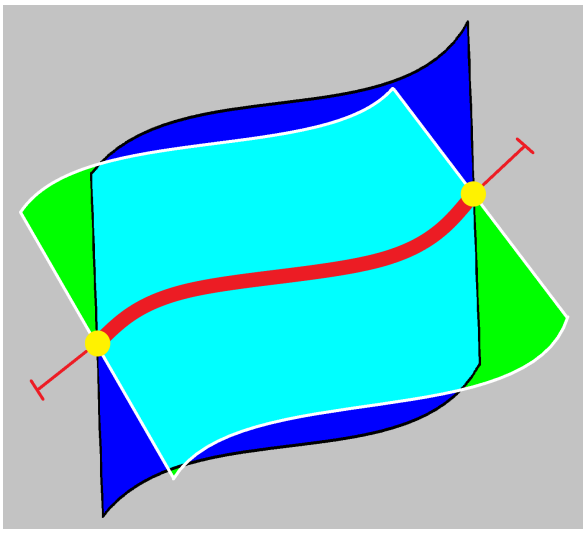


Figure 4.4: Curve based deformation usage proposal.

In Figure 4.4 are two proposed approaches for the block deformation by the curve. Both are based on the Bézier curve where there is one curve defined and vertices and nodes along the curve are assigned to its closest point.

4.5.3 Bézier curve definition

In this section are described three Bézier curve [22] types, there is however an infinite number of definitions as the curve can be defined by n control points (n = 1 for linear, 2 for quadratic, 3 for cubic, etc.).

- Linear is a simple line between two points.

$$B(t) = P_0 + t(P_1 - P_0) = (1 - t)P_0 + tP_1, 0 \leq t \leq 1 \tag{4.6}$$

- Quadratic is a curve traced by an interpolant of points on two linear Bézier curves.

$$B(t) = (1 - t)^2 P_0 + 2(1 - t)tP_1 + t^2 P_2, 0 \leq t \leq 1. \tag{4.7}$$

- Cubic is defined by 4 points, starting at P_0 towards P_1 and arriving at P_3 from the direction of P_2. Can be an affine combination of two quadratic Bézier curves.

$$B(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3, 0 \leq t \leq 1. \tag{4.8}$$

In Figure 4.5 is an example of cubic curve bending based on its control points' positions.

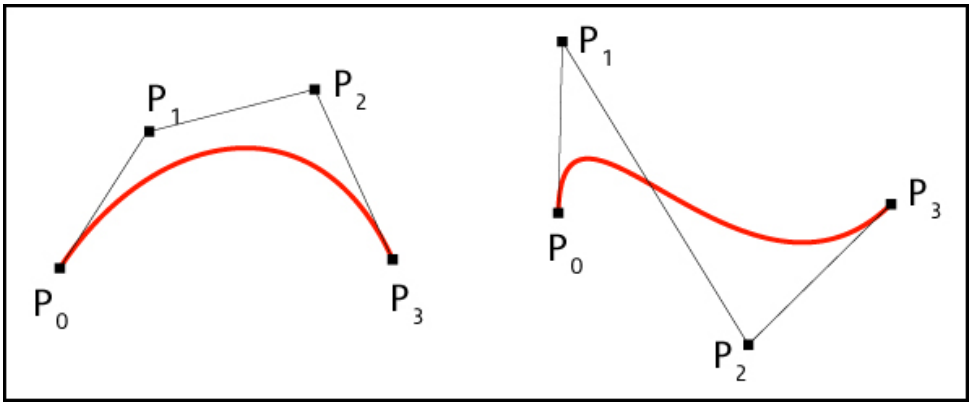


Figure 4.5: Example of Bézier curve from *Quadratic and Cubic Bézier Curves* [22].

4.5.3.1 Bézier segments continuity

In order to connect multiple Bézier segments, certain rules apply for different levels of continuity. An example is given for two segments, each with three control points $((V_0, V_1, V_2), (W_0, W_1, W_2))$. For each continuity C^n, continuity C^{n-1} rules apply and one more is added.

- For the continuity C^0, it is required that the segments meet at the same point, the shared point is then simply resolved:

$$W_0 = V_2 \tag{4.9}$$

- For the continuity C^1, C^0 rules have to be fulfilled and the neighboring control points have to be mirrors of each other:

$$W_1 = 2V_2 - V_1 \tag{4.10}$$

- For the continuity C^2, C^1 rules have to be fulfilled and acceleration continuity is constrained, but we lose the local control of such segment:

$$W_2 = 4(V_3 - V_2) + V_1 \tag{4.11}$$

4.5.4 Euler spiral

Euler spiral [25] is a type of curve used to define road curvature for smooth driving. It's also known as Clothoid or Cornu spiral, with linearly changing curvature along its length.

Euler spirals have applications in diffraction computations, railway and highway engineering, and photonic integrated circuits. They have a geometry that begins with zero curvature and increases linearly until meeting a circular curve, where its curvature becomes equal to that of the circular curve. It's compared with Biarc, which is formed from two circular arcs. For the Euler spiral definition is used an expression of a Fresnel integral.

$$S(z) = \int_0^z sin(\frac{\pi t^2}{2})dt \tag{4.12}$$

$$C(z) = \int_0^z cos(\frac{\pi t^2}{2})dt \tag{4.13}$$

Fresnel integrals are also used to simulate light diffraction, wave interference effects and to model the way light bends and changes direction as it passes through objects such as lenses or water droplets.

A double-ended Euler spiral continues to converge to the points marked in Figure 4.6, where parameter t tends to positive or negative infinity.

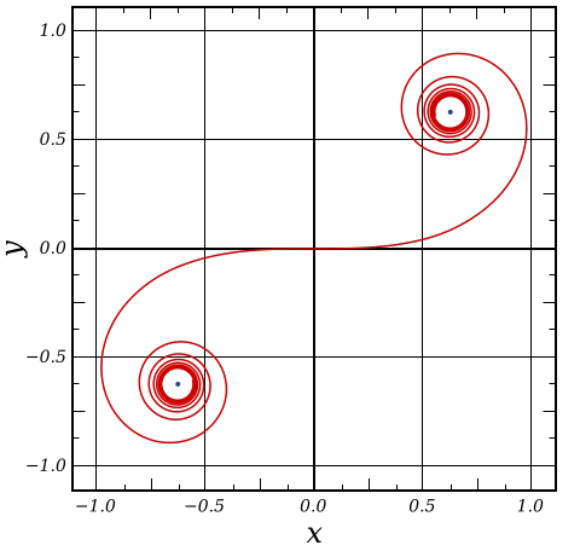


Figure 4.6: A double-end Euler spiral by *MATHEMATICS Euler spiral* [25].

Such an example of a clothoid is depicted in Figure 4.7. The example is depicted in comparison with Biarc which is a smooth curve formed from two circular arcs.

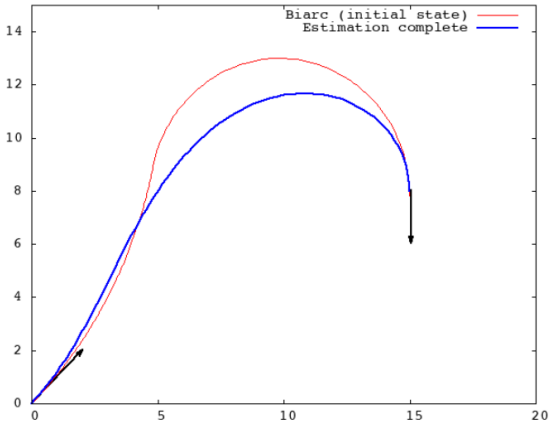


Figure 4.7: Euler spiral demo example by *Euler spiral* [21].

4.6 Procedural visual details generation

There are several methods to increase the road experience. One of them is essentially the topic of this thesis, as the traffic simulation provides drivers with more interaction and realism. Another way of improving the experience is by enriching the environment. Signs, trees, vegetation, barriers, villages, cities and more can rapidly increase the quality of the resulting scene. There may be both manual and automatic approaches. For the manual approaches, there are several tools like *EasyRoads* in Figure 2.5 to facilitate the generation. It provides the creator with several types of brushes and lets him draw into the scene with parameters like randomization, road propagation and terrain adaptation.

For the automatic generation there are also several approaches. Respecting the road curvature and creating only so-called islands of forests, vegetation and villages increase the experience however such an approach does not provide the scene with much detail. To furnish the scene with more detail, terrain and vegetation can be generated along the curve. Such an approach even provides us with sign generation. Such automation could be used for the procedural generation of houses by the modular environment planning as is in the *thesis of Mikushina* [7] (example shown in Figure 4.8).

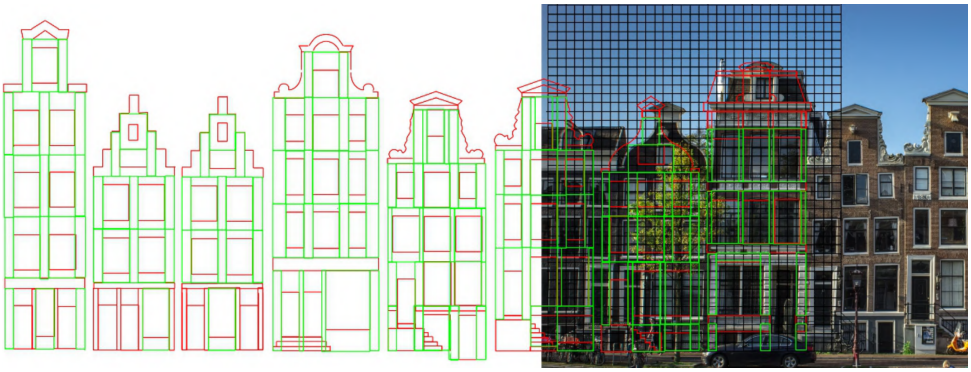


Figure 4.8: Example of the modular environment generation from *the thesis of Mikushina* [7].

This approach can provide us with city blocks as it basically generates Potemkin villages. In addition, as only the front side of these houses could be generated, such a method would not be computationally demanding.

Chapter 5

Implementation description

This chapter focuses specifically on module implementation and all its constituent parts. All specific implementation sections are described in this chapter, starting from its particular coding and way of implementation, leading to their general idea and purpose.

The development of this module was at first focused only on fulfilling project conditions. These conditions were then modified and extended for the master thesis assignment.

The module's implementation consists of *.js*, *.css*, *.html*, but mostly *C++*. There are two classes implemented - RoadEditor and RoadBlock. Class RoadBlock is used as an instance of all the blocks data, it stores all the block's parameters, its traffic network information and geometry. It also provides methods for modifying most of its properties. RoadEditor class handles the mouse and keyboard interactions.

5.1 Module basic properties/functionality

As mentioned in Chapter 4, the basic idea of RoadEditor module was to implement user-friendly interface for any traffic network creation. Such implementation requires dealing with specifics from proper GUI to usable and applicable in-scene interaction with all the module elements such as adjustable traffic parameters, block positioning and deformation. As we dive deeper we start dealing with traffic parsers and road graph creation that the autonomous cars use to drive the predefined roads.

5.2 GUI

Modules GUI (Figure 5.1) is implemented in files *roadeditor.html, roadeditor.css* and *roadeditor.js* and lets the user to easily select any of the predefined blocks to insert them in a scene with preset parameters.

In addition, there is an option for the BlockSize and PlaneSize definitions that define inserted block's size and in-scene plane size.

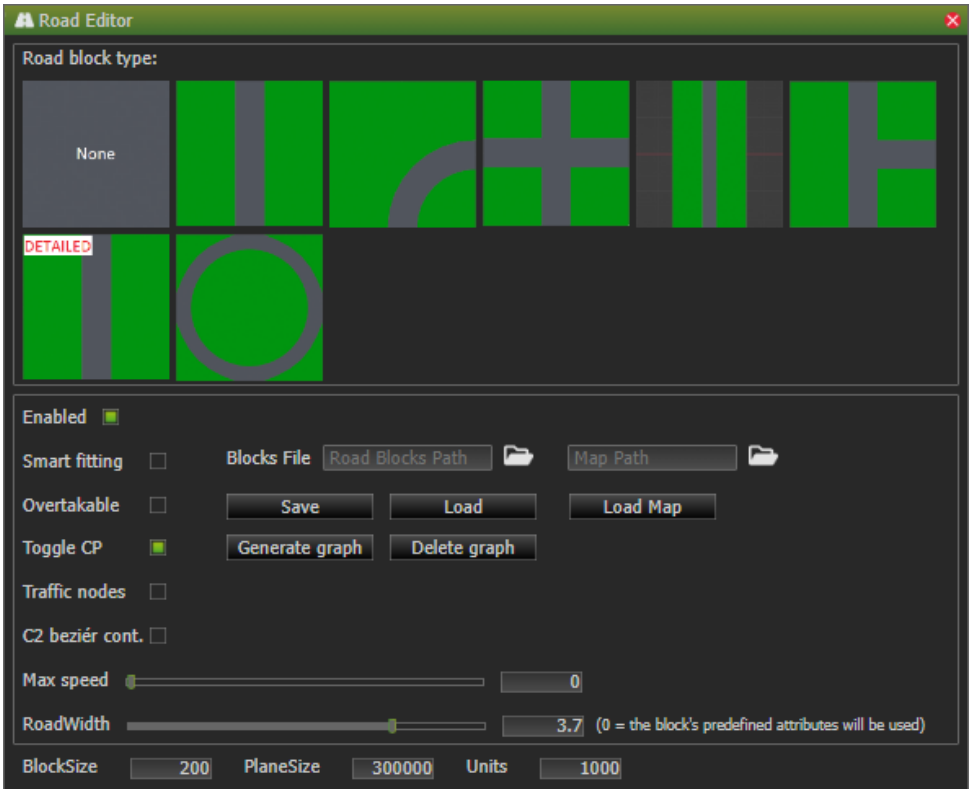


Figure 5.1: GUI of RoadEditor module.

Blocks images are samples of given blocks and are implemented as simple radio buttons. Module then offers numerous other options to be set, there are many parameters to be set for each traffic nodes and segments, most of these may be pre-defined in the block template traffic file, however, they could be modified while inserting a new block.

Some of the basic properties implemented are *Overtaking*, *Max Speed* and *Road Width*. All the GUI parameters' changes then get to be handled in function ProcessParameter.

```
ProcessParameter(const std::string &paramName,
                 const std::string &parameters);
```

Listing 5.1: ProcessParameter function header.

The module GUI is depicted in Figure 5.1 and is further described in subsections below.

5.2.1 Resizable GUI

As the module enables custom block creation, there is *blocks.txt* file that lists blocks currently used by the RoadEditor module. Modules GUI block types menu is based on this file and the GUI resizes based on the blocks number, it depicts **blocksCount % 6** lines and expands the *Road block type* section in the GUI by the value (shown in Figure 5.2).

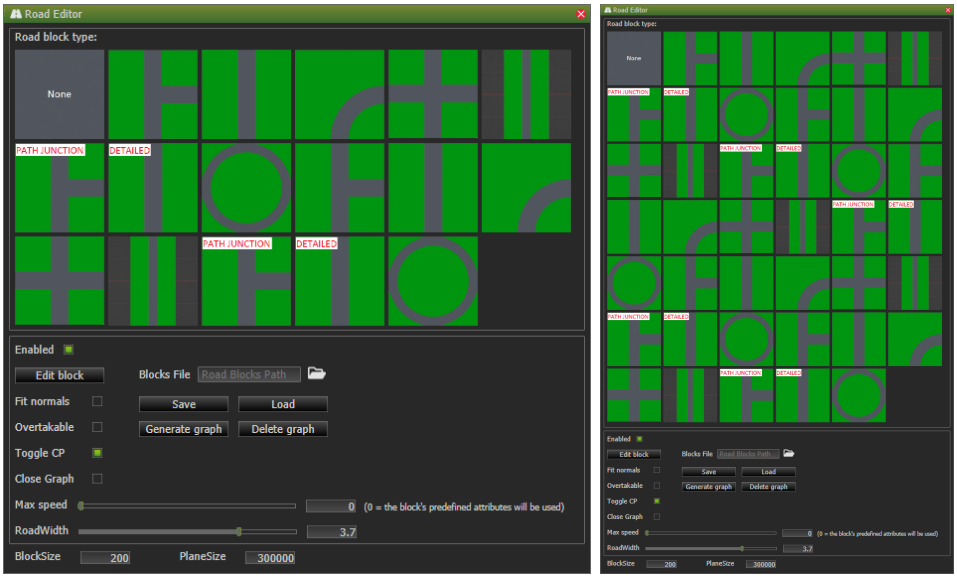


Figure 5.2: Resizable GUI examples.

5.2.2 GUI parameters update

To register GUI messages and more complex interactions than reading values from checkboxes and textboxes, further implementations have to take place. At initialization, it is loading the block's template names. During the program's execution, it is popping up the folder window for the file selection to load the blocks parameters that were previously saved.

```
void RoadEditor::LoadBlocksGUI() {
    CefRefPtr<CefBrowser> cefBrowser = GetCefBrowser();
    if (cefBrowser) {
        CefRefPtr<CefProcessMessage> addMsg =
            CefProcessMessage::Create("initializeBlocks");
        CefRefPtr<CefListValue> argsList = addMsg->
            GetArgumentList();
        argsList->SetSize(templates_names.size());
        for (int i = 0; i < templates_names.size(); i++)
            argsList->SetString(i, templates_names[i]);
        cefBrowser->GetMainFrame()->SendProcessMessage(
            PID_RENDERER, addMsg);
    }
}
```

Listing 5.2: Road Editor blocks GUI update at initialization.

5.2.3 Save/Load blocks

Saving and loading scenes is one of the most desired feature for nearly any software. RoadEditor module handles saving by storing each block's tag/name and its transformation matrix in a *.xml* file.

Button **Load** then just loads blocks as its type new instances, places them in the scene by their assigned transformation matrix and pushes all the in-scene blocks to the vector storing all the in-scene blocks (see Figure 5.3).

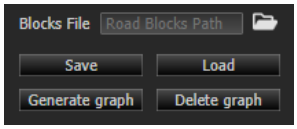


Figure 5.3: Subpart of GUI that handles saving/loading of the blocks and traffic graph generation/deletion.

5.2.4 Map background

There is also an option for setting the *Baseplane*'s texture (see Figure 5.4) to a certain map in order to adjust the traffic network exactly by a given template. In the GUI, user can easily set the texture and load it. Afterwards, using block deformations and other module tools to adjust the network's

shape, the user creates the desired network. Usage of the map background is shown in section 6.2.

Figure 5.4: Subpart of GUI that handles loading a background map on the *Baseplane*.

5.2.5 Graph generation

The generation of graph GUI's part is shown in Figure 5.3. It takes all in-scene blocks traffic definitions and merges them by certain rules into one file that defines the scene's road graph. The detailed graph generation and how it is achieved is described later in Block interconnection definition section subsubsection 5.4.1.3.

In addition the graph nodes of in-scene blocks can be rendered by checking *Traffic nodes* option in the GUI.

5.3 Events handling

Events by each module get to be handled in function *processEvent(Event* evt)*. However, all the event listeners have to be first registered in the module constructor.

```
REGISTER_LISTENER(Event::EVT_INPUT_KEY);
REGISTER_LISTENER(Event::EVT_SCENE_NODES_SELECTED);
REGISTER_LISTENER(Event::EVT_INPUT_MOUSE);
REGISTER_LISTENER(Event::EVT_IO_SCENE_IMPORT_DONE);
REGISTER_LISTENER(Event::EVT_RENDER_SCENE_DONE);
```

Listing 5.3: Events

Yet only a couple of events get to be registered by the RoadEditor module. Events from key input, mouse input and parameters set are in use, however event listener for EVT_PARAM_SET does not have to be registered by the module in the constructor as other events.

5.4 Blocks properties

At the moment there are seven default block types in the module, their geometry is simple and yet for testing purposes, however wider collection and the possibility of custom block designing would eliminate any limitation.

Block types that the RoadEditor module yet offers are so-called: **TCROSS**, **STRAIGHT**, **STRAIGHTLONG**, **STRAIGHTDETAILED**, **TURN**, **INTERSECTION** and **ROUND**.

All these blocks are completely defined and can be used for scene creation right away.

5.4.1 Block definition

Definition of a custom block requires three files: *block_name.obj*, *block_name-traffic.xml* and *block_name.png*. After creating all these files, the new block's name has to be added to *blocks.txt* file in order to load the block at module's initialization and to display it in the module's GUI. Afterward, the traffic file is required during the road graph generation.

5.4.1.1 Block model creation

Block's geometry was created using Blender as it is easy to use tool. The *.obj* format is required as other formats like *.fbx* were not imported correctly as VRUT was having problems with the definition of the materials.

Blocks were evolving over time (see Figure 5.5) as there were problems with several things like z-fighting and cars falling off the road.

In addition there are some data read from the road's surface texture in order to determine the vehicle's relative position to the road lane and so certain textures have to be properly mapped onto the block model.

Furthermore, vehicles were sometimes falling off the road and so the block's geometry had to be extended with a causeway in order to allow vehicles to get back on the road.

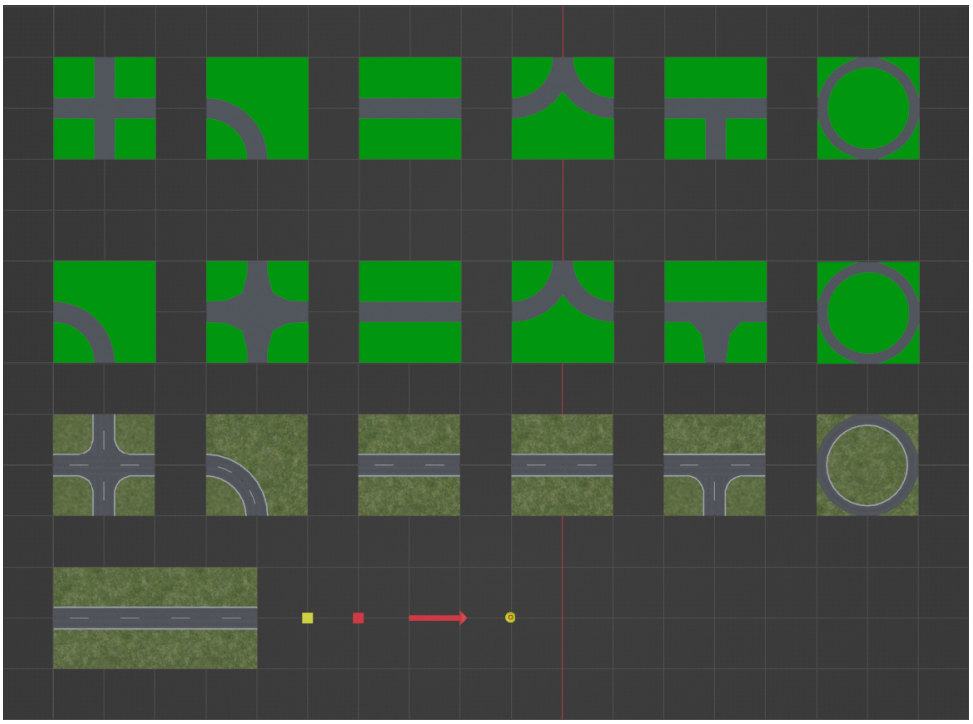


Figure 5.5: Blocks' geometry (Blender).

5.4.1.2 Block nodes layout

Each block's traffic file is defined in *.xml*. The rules and parameters for network definition were mentioned in Traffic representation section 3.5.

I first designed blocks by drawing as is depicted in Figure 5.6. The such layout was helpful for the block design, both for the nodes' positions and their interconnections for both *sequence* and *junction*.

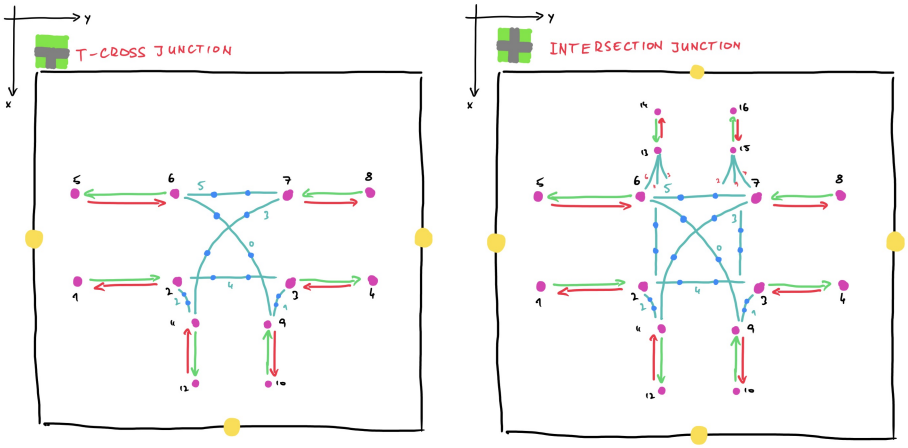


Figure 5.6: Traffic block template layout; nodes, forward/backward edges and junctions; Left - T-cross block, Right - intersection block.

The traffic's definition is only within the spectrum of a small network that will be interconnected with other blocks based on the *connectionpoints* definition.

```
<?xml version="1.0" encoding="UTF-8"?>
<xml>
  <roadgraph version="1.2" />
  <road id="1" type="2">
    <lane id="1" nodes="4" type="0" level="3" o="0"
        nodeidoffset="1">
      <node id="0" x="1.8" y="-15" z="1.7" o="3" navevt="0" sl=
          "10" w="3.74"/>
      <node id="1" x="1.8" y="-5" z="1.7" o="3" navevt="0" sl="
          10" w="3.74"/>
      <node id="2" x="1.8" y="5" z="1.7" o="3" navevt="0" sl="
          10" w="3.74"/>
      <node id="3" x="1.8" y="15" z="1.7" o="3" navevt="0" sl="
          10" w="3.74"/>
    </lane>
    <lane id="1" nodes="4" type="0" level="3" o="0"
        nodeidoffset="1">
      <node id="4" x="-1.8" y="-15" z="1.7" o="3" navevt="0" sl
          ="10" w="3.74"/>
      <node id="5" x="-1.8" y="-5" z="1.7" o="3" navevt="0" sl=
          "10" w="3.74"/>
      <node id="6" x="-1.8" y="5" z="1.7" o="3" navevt="0" sl="
          10" w="3.74"/>
      <node id="7" x="-1.8" y="15" z="1.7" o="3" navevt="0" sl=
          "10" w="3.74"/>
    </lane>
  </road>
  <attributes>
    <speedlimit from="1" to="8" value="30" />
    <vmax from="1" to="8" value="30" />
    <vdop from="1" to="8" value="30" />
  </attributes>
  <connections>
    <sequence from="1" to="4" dir="2" closed="0" />
    <sequence from="5" to="8" dir="3" closed="0" />
  </connections>
  <connectionpoints>
    <node id="3,7" x="0" y="0.5" z="0"/>
    <node id="0,4" x="0" y="-0.5" z="0"/>
  </connectionpoints>
</xml>
```

Listing 5.4: Straight block *.xml* definition.

Such block definition is depicted in Figure 5.7 on the left. More complex blocks do take up to 200 lines of XML code in order to be precisely defined. As for the junctions definition, there has to be path, lanelink and priority defined.

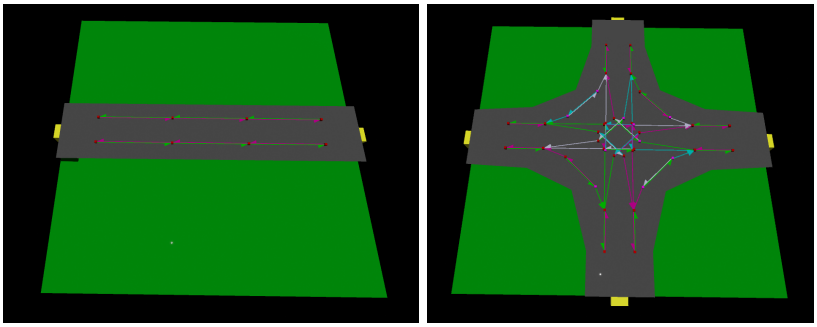


Figure 5.7: Road graph traffic definition, Left - straight block, Right - intersection block.

5.4.1.3 Block interconnections definition

The edge traffic nodes for the block's interconnection are referred to by the called *connectionpoints* tag. The connection points are generated at the block initialization and have node traffic IDs assigned. The connection points are assigned to each other while inserting blocks into the scene (see Figure 5.8).

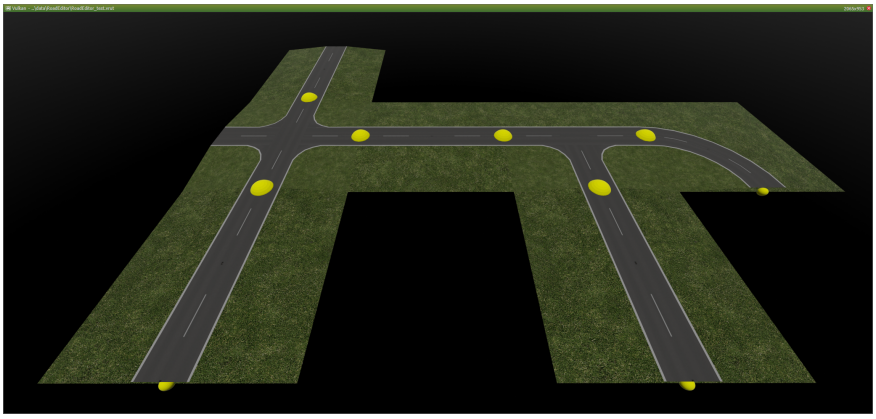


Figure 5.8: Connection points assigned to other blocks - triple size; free CPs[1]still active to be magnetized to.

When two blocks interconnect, the connection points assigned to each other set the local scale to triple size in order to be still visible and possible to click at. In the case of two *straight* blocks interconnection, the rendered CP has a special utilization which is further described in subsection 5.4.7.

[1]Connection points.

5.4.1.4 Connection points

The connection points have more use than just interconnecting block's traffic networks, they are also used for the magnetic feature in order to facilitate the user placing the blocks next to each other. In addition, they are used for block deformation which is mentioned in subsection 5.4.7.

5.4.1.5 Blocks XML parser

In order to the road graph generation, the block's traffic definition has to be parsed first. For such purpose, there are structs predefined in the RoadBlock class that hold this information, there are structs defined for *nodes, lanes, roads, attributes, sequences, connections, connection groups, paths, successors, lanelinks, checks, priorities* and *junctions*. The parser and the node definition have been applied from the Traffic module, which is implemented in the VRUT engine and is used for operating with the traffic network.

```
struct node {
    traffic_node_id id;
    Vector3 pos; // node position in space
    std::string name; // node name used for navigation
    unsigned short roadID;   ///< road number (0...65535)
    unsigned char roadType;  ///< road type (highway, 1st class,
        2nd class, ...) - see RoadNodeRoadType enum
    char laneID;   ///< lane number (-127...127, 0 = right lane,
        1 left lane, -1 right exit, ...)
    unsigned char laneType;  ///< lane type (straight lane,
       exit lane, entrance lane, ...) - see RoadNodeLaneType
    unsigned char level;   ///< autonomous driving level - see
       RoadNodeDrivingLevel enum
    int   navigationEvent = 0; ///< navigation event (like turn
        left, turn right, begin of exit lane, ...)
    float speedLimit; // in km/h
    int   overtakable = -1; // does lane allow overtaking
    bool spawnable; // does node allow spawn?
    float width; // width of the traffic lane
    float radius; // radius of the curve in this lane in meters
    float radius2;
    float vmax; // speed limit [km/h]
    float vdop; // speed limit for turn drive [km/h]
    float vprof[numSpeedProfiles]; // speed profiles
    node(){}
    node(int id, Vector3 pos, int navevt) : pos(pos), id(id),
       navigationEvent(navevt) {}
    node* Clone() const {return new node(*this);}
};
```

Listing 5.5: Road graph node's structure; all parameters defining the node.

5.4.2 Block selection

At the scenes initialization or the RoadEditor module instance creation, all the necessary nodes, objects and materials including roadblocks are imported into the scene. Block selection is by default set to *None*, nevertheless as soon as the user picks any other block from the modules GUI, the selected block appears at the cursor ray intersection with the *Plane* position. The plane is the only object pre-imported in the prepared scene and its size can be changed using GUI.

When block *None* is selected, user can click on any in-scene block and enter edit mode, in which the blocks wireframe appears, traffic nodes are depicted and vectors for block deformation crop up (see Figure 5.9).

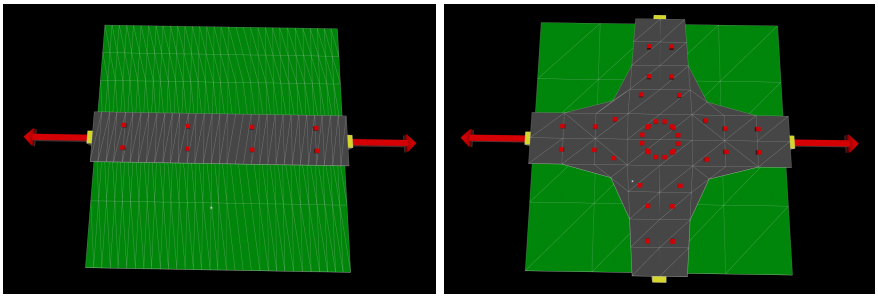


Figure 5.9: Block selection; wireframe, deformation arrows and block's nodes.

5.4.3 Blocks positioning

The left mouse button is used to insert copy of selected block at a given position and the right mouse button is used to rotate the block in CW (clockwise) direction by 90 degrees. Slight rotation changes can be also achieved by keys **O_KEY** (5 degrees to the left) and **P_KEY** (5 degrees to the right).

All the inserted blocks are added to ignore list for ray-cast that is used to detect the intersection point to place the selected block.

The positioning is implemented in function *CalculateBlocksPosition()* and its algebraic approach looks as follows.

First, the connection point's translation has to be calculated. Both for the in-scene block and for the one that is being inserted.

$$M_{insceneCP_{local}} = M_{insceneCP_{global}} \times M^{-1}_{insceneBlock_{global}} \tag{5.1}$$

$$M_{selectedCP_{local}} = M_{selectedCP_{global}} \times M^{-1}_{selectedCP_{global}} \tag{5.2}$$

Furthermore, if the dot product of these two vectors is less than zero, which means they are opposite (facing each other), the in-scene connection point does satisfy the requirements for interconnection.

$$M_{insceneCP_{local}}.vec \cdot M_{selectedCP_{local}}.vec < 0 \tag{5.3}$$

The resulting block's transformation matrix is then computed in the following way:

$$M_{result} = M_{edit\_rot} \times M^{-1}_{selectedCP_{local}} \times M_{insceneCP_{global}} \tag{5.4}$$

In addition in the case of *Smart fitting* checked the M_{edit_rot} is slightly adjusted in order to fit the magnetized CP's normal. First, as the dot product gives us only a positive angle value, we have to compute the sign:

$$a_{vec} = M_{rotZ90deg} \times CP_{normal} \tag{5.5}$$

$$b_{vec} = M_{edit\_rot} \times shift \tag{5.6}$$

$$sign = \begin{cases} -1 & (a_{vec} \cdot b_{vec}) > 0 \\ 1 & (a_{vec} \cdot b_{vec}) \leq 0 \end{cases} \tag{5.7}$$

Then the angle has to be computed using the sign value:

$$angle = sign \times \frac{acos(CP_{normal} \cdot b_{vec})}{CP_{normal}.length \times b_{vec}.length} \tag{5.8}$$

Furthermore, M_{edit_rot} matrix has to be modified by the precomputed angle offset:

$$M_{edit\_rot} = M_{edit\_rot} \times M_{rotZ\_angle} \tag{5.9}$$

The smart fitting feature for magnetic placing (see subsubsection 5.4.3.1) is shown in Figure 5.10.

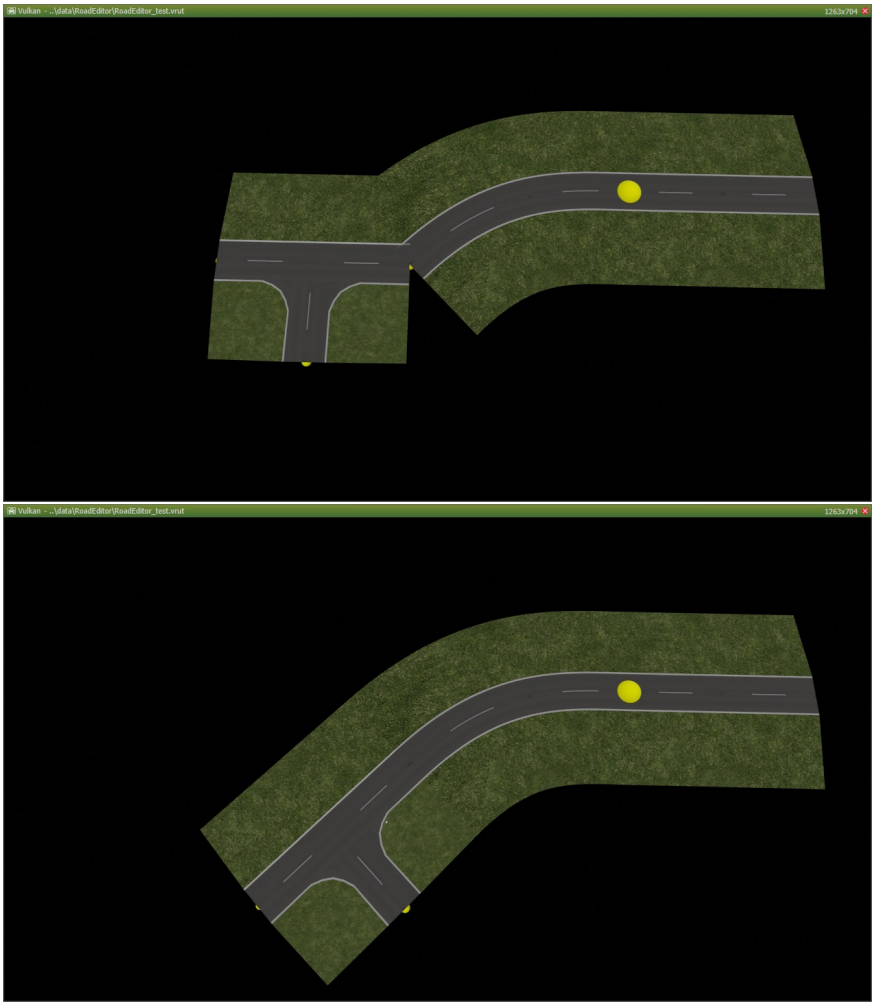


Figure 5.10: Smart fitting feature, Top - feature is off in the GUI, Bottom - feature is on; T-cross block being inserted.

5.4.3.1 Grid and magnetic block placing

Blocks positioning is either based on a grid, that is defined by the first inserted block, where the cursors intersection position is rounded or it can be changed by **TAB_KEY** to magnetic which makes the inserted block stick to in-scene block's connection points, when in certain distance.

Computing the block's transformation matrix for the grid feature is based on the grid value and simply rounds it depending on it.

```
if (!magnetic) {
    transMat = editMAT * Matrix::Translation(Vector3(gridVal *
        round((infoRay.intersection.x - magnetShift.x) /
            gridVal) + magnetShift.x,
        gridVal * round((infoRay.intersection.y - magnetShift.y)
            / gridVal)
        + magnetShift.y, infoRay.intersection.z) + Vector3(0, 0,
            1000));
}
```

Listing 5.6: Grid feature code.

To determine the closest node, the following code is used. In addition, the distance limit can be set, so the magnetic feature works only at a certain distance from each connection point.

```
RoadBlock::connection_point* RoadEditor::getClosestNode(
    Vector3 cursor, long long int limit = MAXLONGLONG) {
     RoadBlock::connection_point* cp_res = nullptr;
     for (auto b : blocks) {
          if (!b || (selectedBlock && selectedBlock->id == b->id
              )) continue;
          for (auto cp : b->cps) {
               if (cp->friendcp != nullptr)
                    continue;
               Vector3 cp_pos = scene->GetWorldTransMatrix(cp->
                   id).ExtractTranslation();
               float curs_cp_dist = sqrtf(pow(cursor.x - cp_pos.
                   x, 2) + pow(cursor.y - cp_pos.y, 2));
               if (curs_cp_dist < limit) {
                    limit = curs_cp_dist;
                    LOGWARNING(std::to_string(limit));
                    cp_res = cp;
               }
          }
     }
     return cp_res;
}
```

Listing 5.7: Magnetic feature code.

In addition when the *smart fitting* option is checked the block's rotation is slightly adjusted in order to fit the magnetized connection point normal.

5.4.4 Blocks parameter set

As it is in the section 5.2 depicted, parameters that are available to set on for each block are *Overtaking*, *Max speed* and *Road width*. These parameters

are in most cases already set in the predefined block's traffic *.xml* file, but can be overwritten by these values.

Overtaking would just set inserted block's nodes **o** value to true. Max speed is in the range from 0 to 200 and the step is set to 10. Road widths range is from 0m to 5m and the step is set to 0,001 m, also the default value is set to 3,74 m as it is the standard road width.

5.4.5 Block insertion/deletion

As the left mouse button inserts a copy of the block at the cursor intersection position, also another already inserted block can be selected and deleted using **DEL_KEY**.

5.4.6 Copy/paste block

Copying blocks was a bit challenging task, because of the block deformation feature. As the block deformation affects the instantiated geometry, new mesh has to be generated and added to the scene. In Figure 5.11 are numerous blocks with the same instance but different deformation.

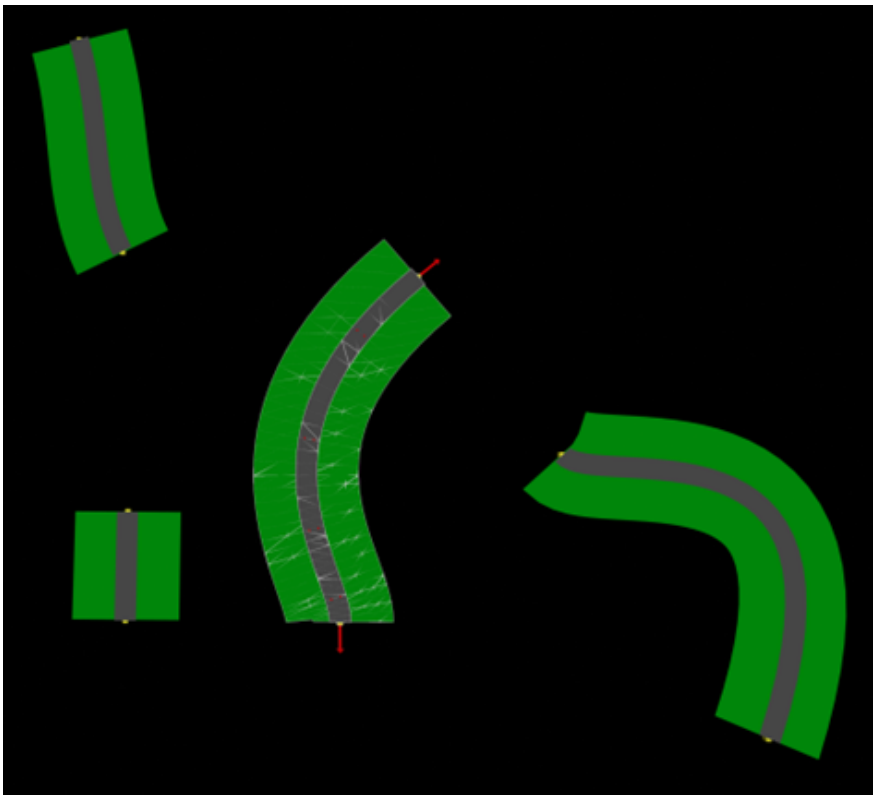


Figure 5.11: Straight block's geometry instances.

5.4.7 Block deformation

When a certain block is selected, the wireframe of its geometry is rendered and two red arrows appear at its connection points. By selecting either any of the connection points or newly rendered red arrows, user can modify the shape of the selected block.

The block's deformation is based on a predefined Bézier curve. The deformation does affect both block mesh's vertices and traffic nodes.

5.4.7.1 Curve deformation application

At initialization is defined each block vertex of the geometry and traffic node's distance from the initial line, which leads from one connection point to its opposing one. While editing the block, the new position of both vertex and traffic node is defined based on the curve's normal and its previous distance from the curve as is in Figure 5.12.

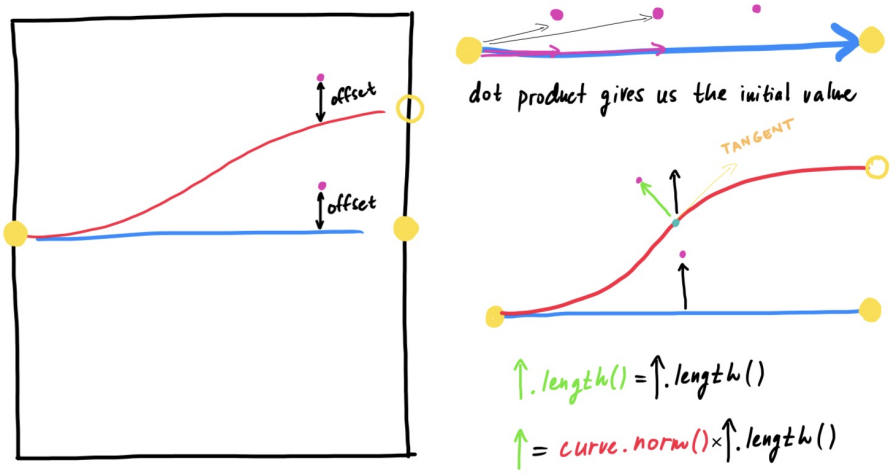


Figure 5.12: Vertex reposition based on the curve formation approach description.

In Figure 5.13 are depicted blocks before and after deformation. In the first row, vertices and traffic nodes are highlighted, however, such an approach was computationally demanding and only the traffic nodes get to be highlighted. In addition, the first deformation was shifting nodes after curve redefinition in the same direction vector as was loaded during the blocks initialization, however, the second deformation does shift traffic nodes and vertices perpendicular to the tangent of the curve.

The second approach retains the width of the road, however, a large curve contortion deforms the block's geometry in a way that some gaps might appear. In such cases, the creation of a new block with geometry that would be closer to the desired shape would be recommended.

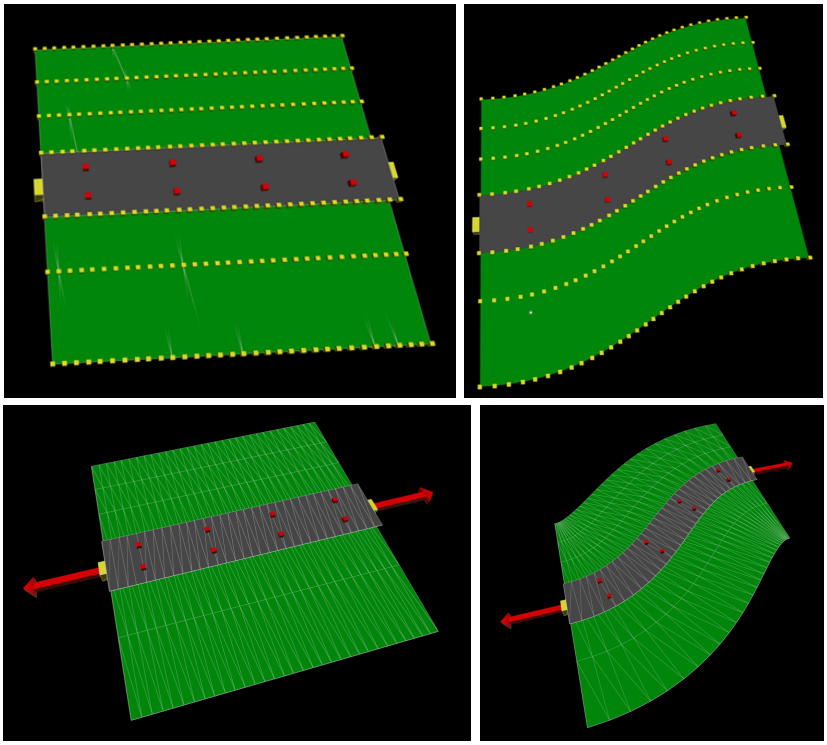


Figure 5.13: Block deformations; the first row with only the offset applied, the second row with the curve normal vector applied.

5.4.7.2 Curvature limitations

As mentioned in subsubsection 5.4.7.1, some limitations to the curve deformation solution appear, an example of an edge case is shown in Figure 5.14.

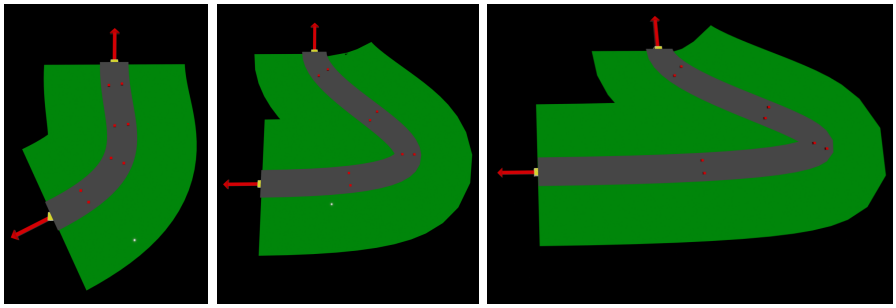


Figure 5.14: Block curvature flexibility limitation.

5.4.7.3 Normals fitting

In order to facilitate any kind of road creation, there is a checkbox in the GUI for fitting the block's normals. If a block gets to be inserted next to any in-scene block's connection point, it is repositioned (in block edit mode), and the curve parameters of these connection points are set to the inverse value of the in-scene connection point's normal value (see Figure 5.15).

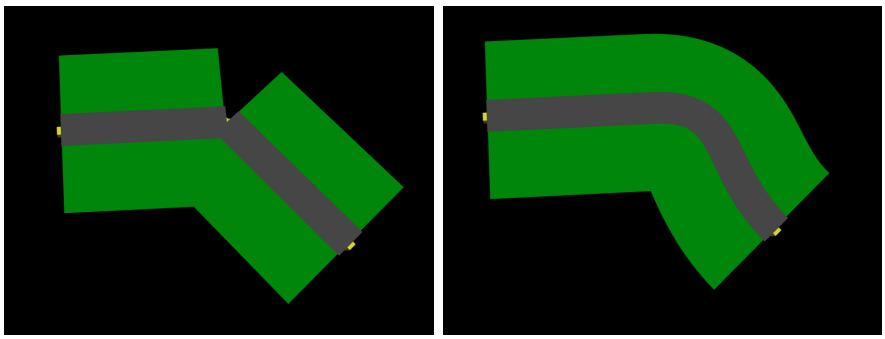


Figure 5.15: Inserted block's normals fitting; Left - before insertion; Right - after.

This feature allows users to create almost any shape of the road. It creates the road scene according to a template or real traffic situation. Interconnection of two separately created traffic networks is even easier with normal fitting feature.

The magnetic feature can be turned on/off by pressing **TAB** during connection point repositioning. The connection point that is being repositioned is either connecting to the closest connection point in a scene or just staying on the cursor's ray intersection position.

Such behavior is depicted in Figure 5.16.

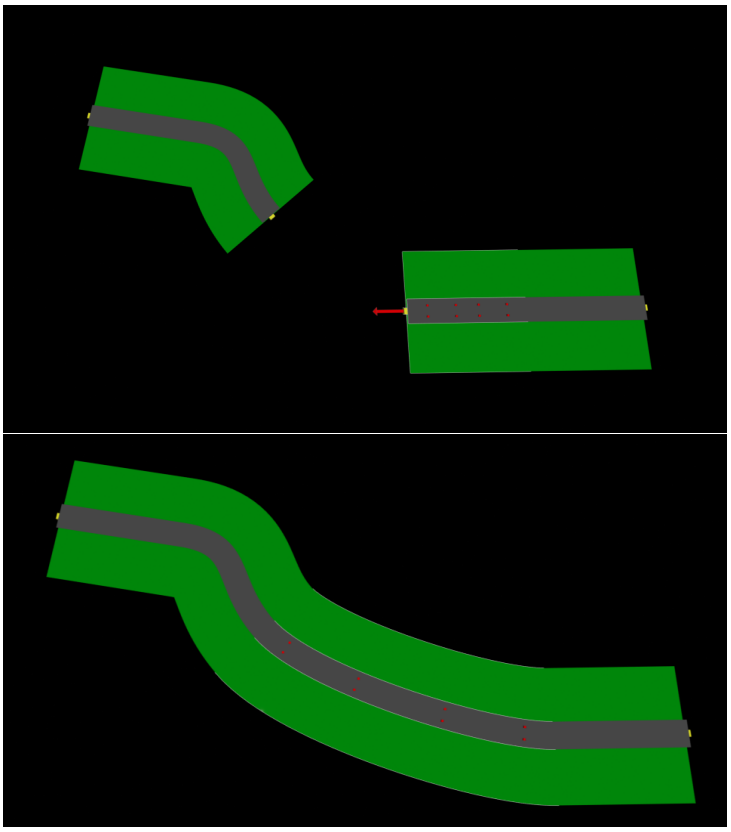


Figure 5.16: Connection point repositioning→fitting neighbour block normals; using magnetic feature.

5.4.7.4 Bézier C^2 continuity

In addition there is an option for the block C^2 Bézier continuity fitting. While checked with *Smart fitting* at the same time the deformation of the block adjusts the way it retains C^2 continuity with the magnetized block as is shown in Figure 5.17.

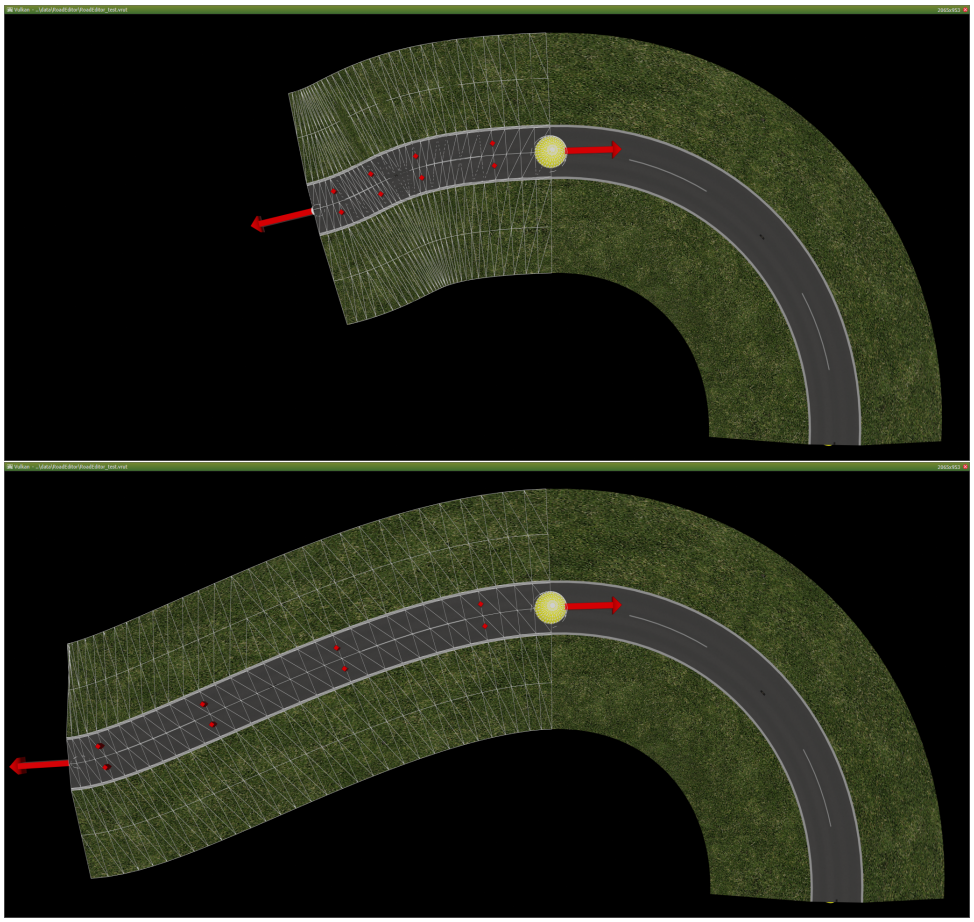


Figure 5.17: Comparison of C^1 (top) and C^2 (bottom) Bézier continuity.

5.4.7.5 Euler spiral

As mentioned in the Chapter 4, the Euler spiral is the best solution for the road curvature definition.

However, while implementing the Euler spiral block deformation into the module, several problems appeared. Firstly, the definition is computationally demanding because of the *Fresnel integral*, so it does not fit the implemented interactive solution. Secondly and more importantly, the implemented library (*EulerSpiral (clothoid)* [16]) that was desired to use, did not allow the point-to-point definition, which is required for this solution.

5.4.7.6 Inner node repositioning

This section draws from the subsubsection 4.5.3.1 and postulates *Bézier* C^2 *continuity* defining formulas. There are two possibilities for the infrastructure's inner node deformation, as there is an option for both C^1 and C^2 continuity.

For the C^1 continuity the solution is easy and the neighbor control points just have to retain mirrored. So the two neighbor control points just retain the initial shift which can be also edited whilst selecting a certain block on a segment and adjusting its curvature using in-scene red arrows.

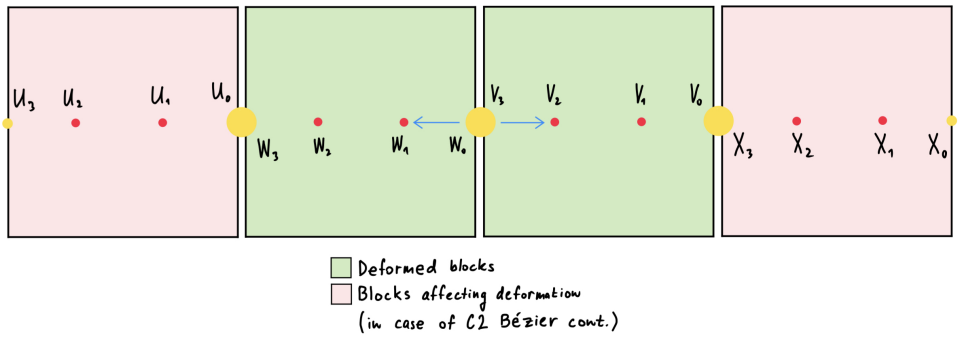


Figure 5.18: Blocks' Bézier curve based deformation (each block defined by a 4 control point Bézier curve).

However, for the C^2 continuity the position definition of side control points is dependent on each other. See the mathematical formulation below and Figure 5.18, the mathematical formulation is given by the C^2 *Bézier segments continuity* described in subsubsection 4.5.3.1:

$$W_2 = 4(V_3 - V_2) + V_1 \tag{5.10}$$
$$V_1 = 4(W_0 - W_1) + W_2 \tag{5.11}$$
$$W_2 = 4(V_3 - V_2) + 4(W_0 - W_1) + W_2 \tag{5.12}$$
$$0 = 4(V_3 - V_2) + 4(W_0 - W_1) \tag{5.13}$$

Which is true however does not define either W_2 or V_1. So a solution dependent on another definition had to be made in order to allow the inner node reposition while C^2 *continuity* is required. The W_2 point is also dependent on its neighbor block U_{0-3}. Although there does not have to be always present block defining such formulation so virtual control point (in this case) U_1 gets to be defined and both W_2 and V_1 are defined symmetrically.

$$W_2 = 4(V_3 - V_2) + V_1 \tag{5.14}$$
$$W_2 = 2U_0 - U_1 \tag{5.15}$$
$$V_1 = 2U_0 - U_1 - 4(V_3 - V_2) \tag{5.16}$$

$$V_1 = 4(W_0 - W_1) + W_2 \tag{5.17}$$
$$V_1 = 2X_3 - X_2 \tag{5.18}$$
$$W_2 = 2X_3 - X_2 - 4(W_0 - W_1) \tag{5.19}$$

5.5 Graph generation

In GUI there is **Generate graph** button that generates a road graph of the in-scene blocks. Two important actions have to take place.

The first one is elementary and just increases each block traffic node's IDs as it generates a graph from the block's subgraphs that would have the same ID. This action has to be applied for all the structures that refer to these nodes, which applies for e.g. sequences, junction's lanelinks and more.

The second action might be more complicated as it interconnects block's traffic nodes in order to make the traffic network continuous.

5.5.1 Traffic node pairing

The first operation is finding the right traffic node's pairs. This is solved in a way that all the possible connections get to be generated, and the one that has the minimal distance over all connections is selected (see Figure 5.19).

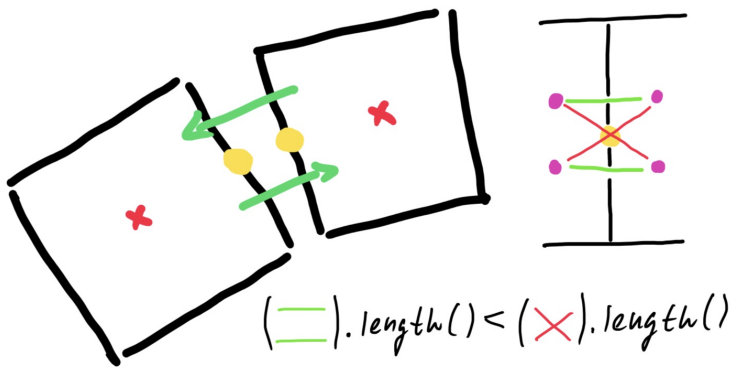


Figure 5.19: Traffic graph interconnection approach description.

5.5.2 Node's direction selection

At this point the direction in which a certain lane is going has to be defined. Such information is already stored in particular lanes, however, there are more options on how to interconnect nodes (sequences, interconnections, connections). However, going through structures and figuring out the correct direction might be more complicated and less reliable. An algebraic approach has been selected for the such a problem as is shown in Figure 5.20. The traffic nodes are already paired and only the direction has to be determined. Each pair's node is rotated around the certain connection point's traffic nodes center by 90 degrees.

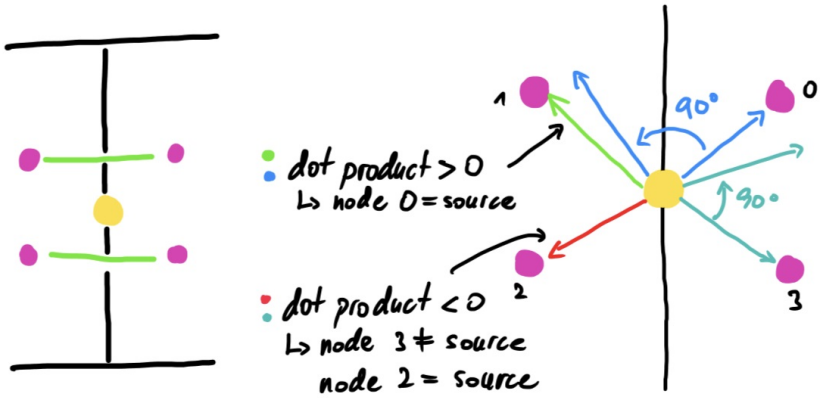


Figure 5.20: Interconnecting lane direction definition description.

In case the resulting vector's dot product with its pair is positive, the previously rotated traffic node is the source; in the other case, it is the sink and the *from* node is switched with the *to* node. Type 0 is the forward edge and type 1 is the backward edge. The generation of such interconnection is shown in Listing 5.8.

```
ss<<"\<connection from\=\""<<from->id+1<<"\"to\=\""<<to->id+1<<
    "\"type\=\""<<0<<"\" \/\>"<<std::endl;
ss<<"\<connection from\=\""<<to->id+1<<"\"to\=\""<<from->id+1<<
    "\"type\=\""<<1<<"\" \/\>"<<std::endl;
```

Listing 5.8: Connections defining direction.

In Figure 5.21 is a small sample of the resulting network interconnections. These interconnections and their correct definitions were tested with the VehicleSimulator module with dozens of cars in-scene.

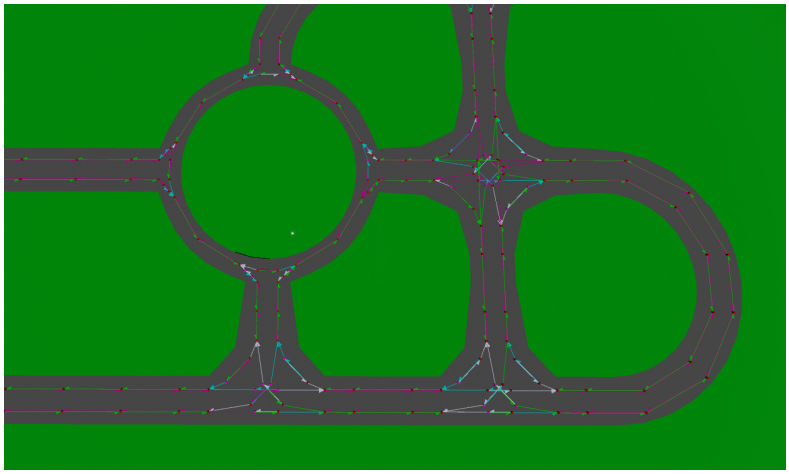


Figure 5.21: Generated road graph with block's traffic interconnections.

5.5.3 Traffic module

To render generated traffic network, the traffic module is required. It has to be run with given parameters like the *.xml* file that is generated by the RoadEditor module and world *.xml* file where physics and car's initial position can be defined including the main scene node that the collisions are computed with. The main node is called *"hlavní"* in all the VRUT scenes.

5.5.4 Traffic display properties settings

The easiest way to test the generated network by the RoadEditor module is to set up a special case for that as is defined below and was inspired by a straight segment case already defined in the *traffic.js* file.

```
case 'test2':
       CAR_COUNT = 9;
       trackFile = "../data/RoadEditor/RoadEditor_test1.vrut";
       carPositions = [
           [4193.14, -8459.2, 1.5, -1.2],
           [-15, -161.8, 1.5, 0.5],
           [-15, -121.8, 1.5, 0.5],
           [-15, -78.2, 1.5, 0.5],
           [-5, -38.2, 1.5, 0.5],
           [-15, -1.8, 1.5, 0.5],
           [-15, 38.2, 1.5, 0.5],
           [-5, 81.8, 1.5, -0.5],
           [15, 121.8, 1.5, 0.5],
           [-3.85366, -71.0196, 1.5, -1.5]
       ]
       carPositions[0] = [81.323, 5.763, 1.275, 0];
       // node for the in-scene vehicles collision detection
       worldNodeName = "hlavni";
       trackCFGFile = "../data/RoadEditor/RoadEditor.xml";
       trafficGraphFile = "../data/RoadEditor/roadGraph-traffic_1.xml"
       updateParamValue("VehicleSimulator", "swapLanes", "1");
       updateParamValue("VehicleSimulator", "sceneUnits", "1000");
       updateParamValue("Navigation", "FOV", "75");
       updateParamValue("Navigation", "nearPlane", "500");
       updateParamValue("Navigation", "farPlane", "5000000");
       break;
```

Some of the parameters do not need to be set, however, it is easier to script some of them to simplify the workflow and so some of the parameters like displaying the nodes and their segments can be instantly marked checked with proper usage of the *traffic.js* script.

5.6 Variable block's size

As the block's size is variable and can be set at the bottom of the GUI, this option is yet not fully supported as it only scales the geometry and transforms the traffic nodes according to its block's size. Magnetic feature and interconnection generation are not affected, however, the roads might

become undrivable as the height of two different-sized blocks could differ as well as is shown in Figure 5.22.

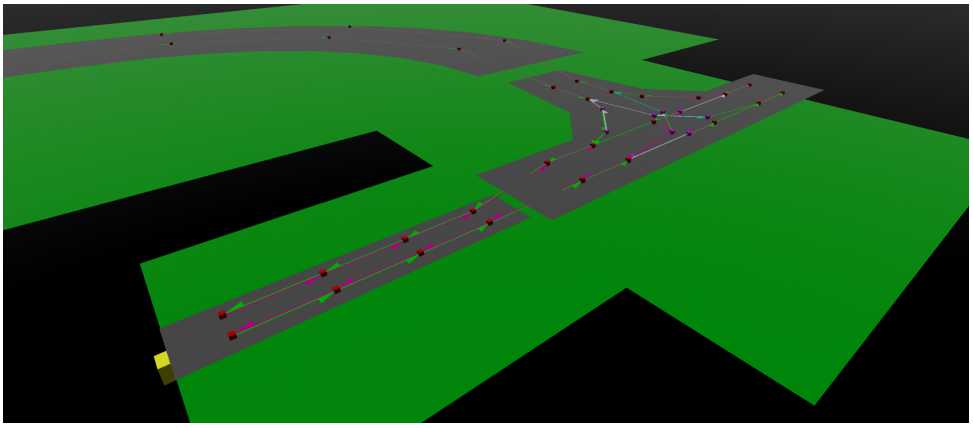


Figure 5.22: Block's interconnection with size disparity.

Chapter 6

Testing and results

This chapter is focused on the way of testing and the results of the RoadEditor module. First, the ways of both blocks testing during the block creation and whole road graph generation testing are explained. Afterwards, multiple scenes generated by the RoadEditor module are tested and evaluated.

6.1 Testing

Module testing was initially focused on the correct traffic generation, so only a traffic module that depicts the road graph network was required. Further, as the road graph generation was handled, testing began also with the VehicleSimulator module, which simulates the traffic.

For such testing, it is necessary for the vehicles to define their initial positions, because of that, also an initial simple scene with a couple of straight blocks has been pre-saved (see Figure 6.1), so there is no need to redefine the vehicle's initial position every time the road graph gets to be generated by RoadEditor module. This approach was useful for testing any new kind of block as the user can see any undesired behavior right away.

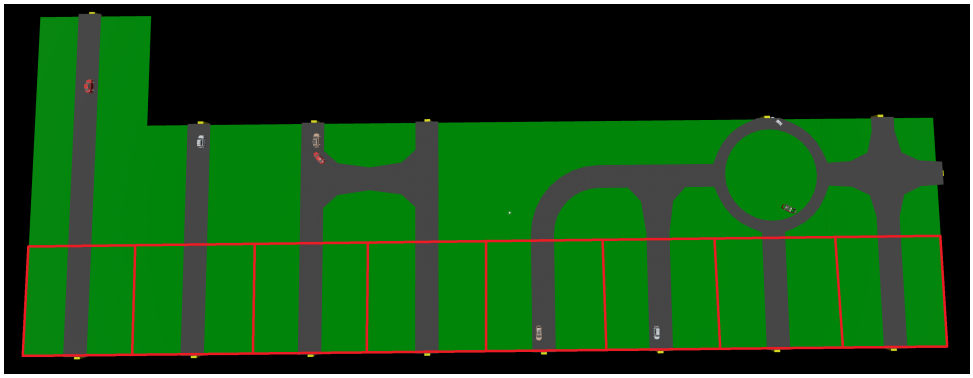


Figure 6.1: Testing blocks with initial pre-saved blocks in red rectangles Vehicles would spawn on these blocks and begin their journey to the neighbor block.

These initial blocks are used in some of the testing scenes in result section 6.2.

6.2 Results

Testing of the module was made on six different tracks. As there was also block deformation implemented, there were more scenes created in order to test as many traffic situations as possible.

First very simple scene is just an oval (Figure 6.2), which proves the by module generated graphs are compatible with both Traffic and VehicleSimulator module.

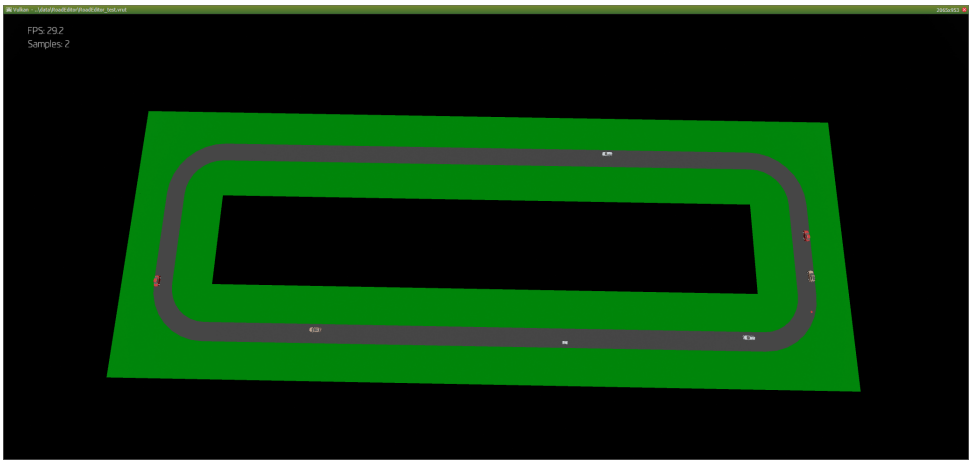


Figure 6.2: Simple oval test result.

Second scene (Figure 6.3) is more complicated as there are already multiple junctions in it. Block types **TCROSS** and **INTERSECTION** are used here for junctions testing.

Figure 6.3: Simple test with junction blocks result.

The next scene is larger and has also some **ROUND** block types (roundabouts) included (see Figure 6.4). This scene could simulate a small city or a housing estate. With some houses, vegetation and more complex block geometry, this scene might be representing real traffic situations for any kind of testing.

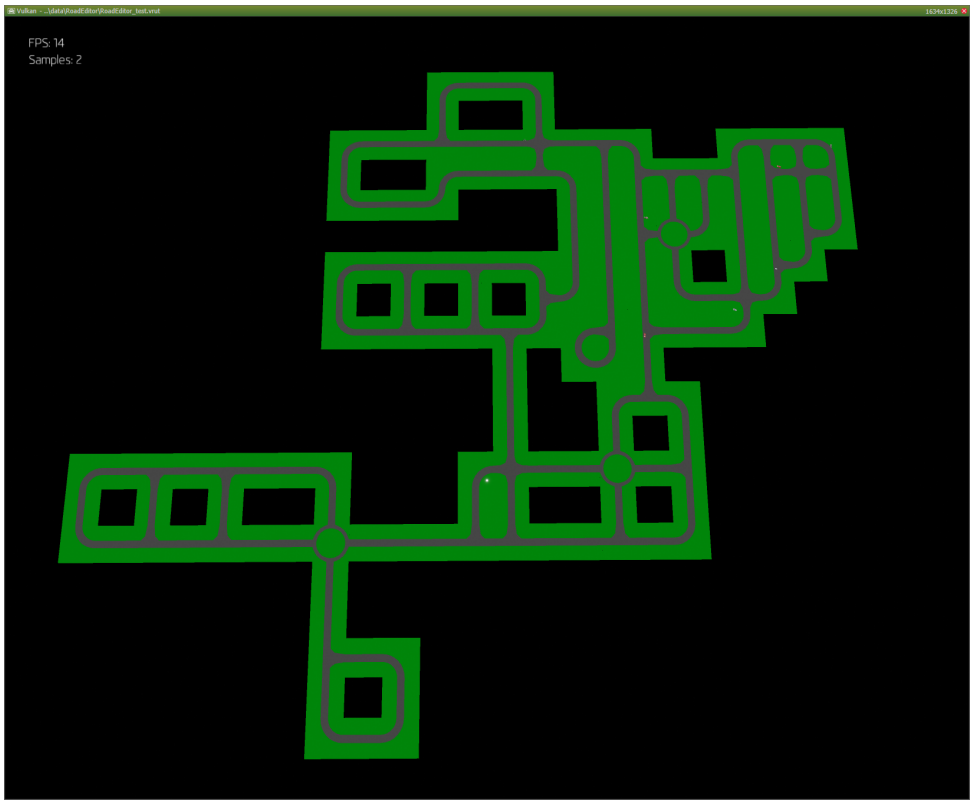


Figure 6.4: Middle sized scene with all block types.

Close look to one of the junctions and its generated road graph and interconnections in the middle sized scene is depicted in Figure 6.5.

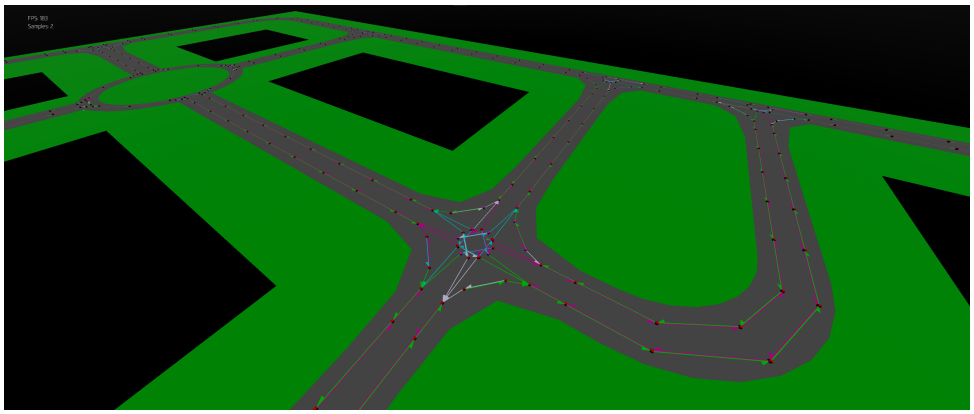


Figure 6.5: Close look to the middle sized scene junction.

The Last scene without block deformation is just larger and has all block types included (see Figure 6.6).

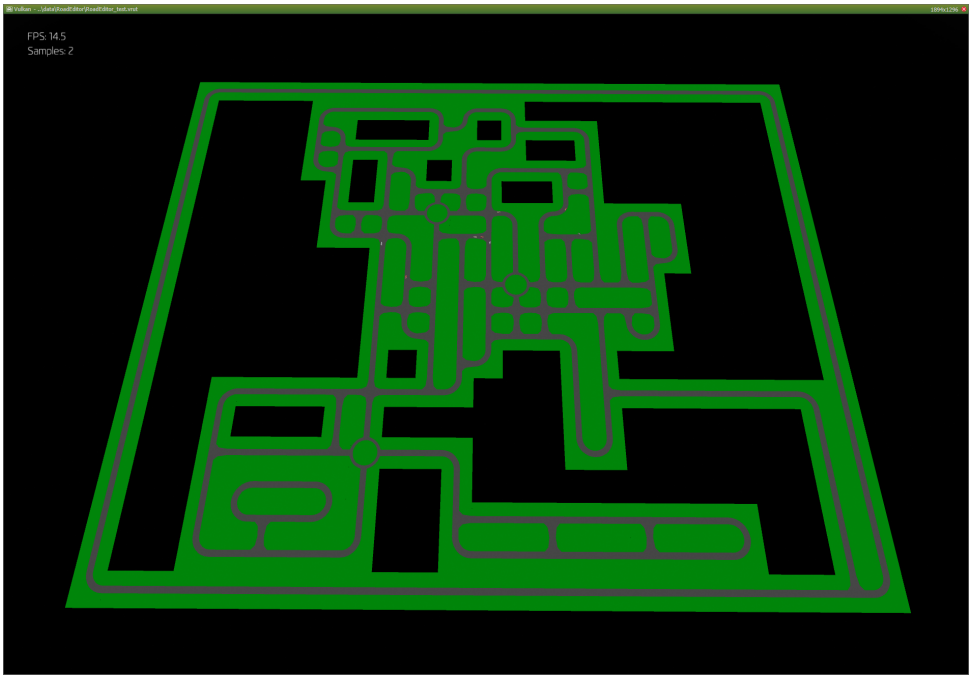


Figure 6.6: Large scene with all block types.

Such scene generates over twelve thousand lines of *xml* code and so the example below is reduced in *Output sample* Listing 6.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<xml>
    <roadgraph version="1.2" />
    <!--Tcross block-->
    <road id="9" type="5">
        <lane id="1" nodes="4" type="0" level="3" o="3"
            nodeidoffset="1">
            <node id="64" x="-38.2" y="105" z="0.5" w="3.74" sl=
                "20" navevt="2"/>
            <node id="65" x="-38.2" y="115" z="0.5" w="3.74" sl=
                "20"/>
            <node id="66" x="-38.2" y="125" z="0.5" w="3.74" sl=
                "20"/>
            <node id="67" x="-38.2" y="135" z="0.5" w="3.74" sl=
                "20" navevt="9"/>
        </lane>
            ...
        <lane id="1" nodes="1" type="6" level="3" o="3"
            nodeidoffset="1">
```

```
        <node id="80" x="-41.4" y="125" z="0.5" w="3.74" sl=
            "20" navevt="5"/>
    </lane>
</road>
...
<!--straightLong block-->
<road id="243" type="2">
    <lane id="1" nodes="8" type="0" level="3" o="0"
        nodeidoffset="1">
        <node id="2838" x="398.201" y="174.999" z="0.5" w="
            3.74" o="3"/>
        ...
        <node id="2845" x="398.201" y="104.999" z="0.5" w="
            3.74" o="3"/>
    </lane>
    <lane id="1" nodes="8" type="0" level="3" o="0"
        nodeidoffset="1">
        <node id="2846" x="401.801" y="174.999" z="0.5" w="
            3.74" o="3"/>
        ...
        <node id="2853" x="401.801" y="104.999" z="0.5" w="
            3.74" o="3"/>
    </lane>
</road>
<attributes>
    <speedlimit from="1" to="8" value="30"/>
    <vmax from="1" to="8" value="30"/>
    ...
    <vmax from="2839" to="2846" value="30"/>
    <vdop from="2839" to="2846" value="30"/>
</attributes>
<connections>
    <connectiongroup>
    <connection from="81" to="72" type="1" />
    <connection from="82" to="74" type="1" />
    ...
    <connection from="2813" to="2814" type="1" />
    <connection from="2814" to="2813" type="0" />
    </connectiongroup>
    <junction name="T-cross 0" id="0">
        <path id="0" nodes="2" type="11">
            <node id="0" x="-37.25" y="120.65" z="0.5" w="
                3.6" sl="30"/>
            <node id="1" x="-40.65" y="117.25" z="0.5" w="
                3.6" sl="30"/>
        </path>
```

```
            ...
            <priority road="9" lane="1" node="81">
                <check road="9" lane="1" node="66"/>
                <check road="9" lane="1" node="79"/>
            </priority>
        </junction>
        ...
        <junction name="T-cross 72" id="72">
            <path id="0" nodes="2" type="11">
                <node id="0" x="397.251" y="79.349" z="0.5" w="
                    3.6" sl="30"/>
                <node id="1" x="400.651" y="82.749" z="0.5" w="
                    3.6" sl="30"/>
            </path>
            ...
            <laneLink road="240" lane="1" node="2811">
                <successor road="240" lane="1" node="2810" path=
                    "5" />
            </laneLink>
        </junction>
        <!-- Block's INTERCONNECTIONS -->
            <connectiongroup>
                <connection from="4" to="672" type="0" />
                <connection from="672" to="4" type="1" />
                ...
                <connection from="2830" to="2820" type="0" />
                <connection from="2820" to="2830" type="1" />
            </connectiongroup>
        <!-- Block's SEQUENCES -->
            <sequence from="1" to="4" dir="2" closed="0" />
            <sequence from="5" to="8" dir="3" closed="0" />
            ...
            <sequence from="2839" to="2846" dir="2" closed="0"
                />
            <sequence from="2847" to="2854" dir="3" closed="0"
                />
    </connections>
</xml>
```

Listing 6.1: Generated road graph by RoadEditor module; originally 12 041 lines.

In Figure 6.7 are initial default blocks with a short section created from deformed straight blocks also using *fitting normals* feature and magnetic feature while repositioning connection point to make the road continuous.

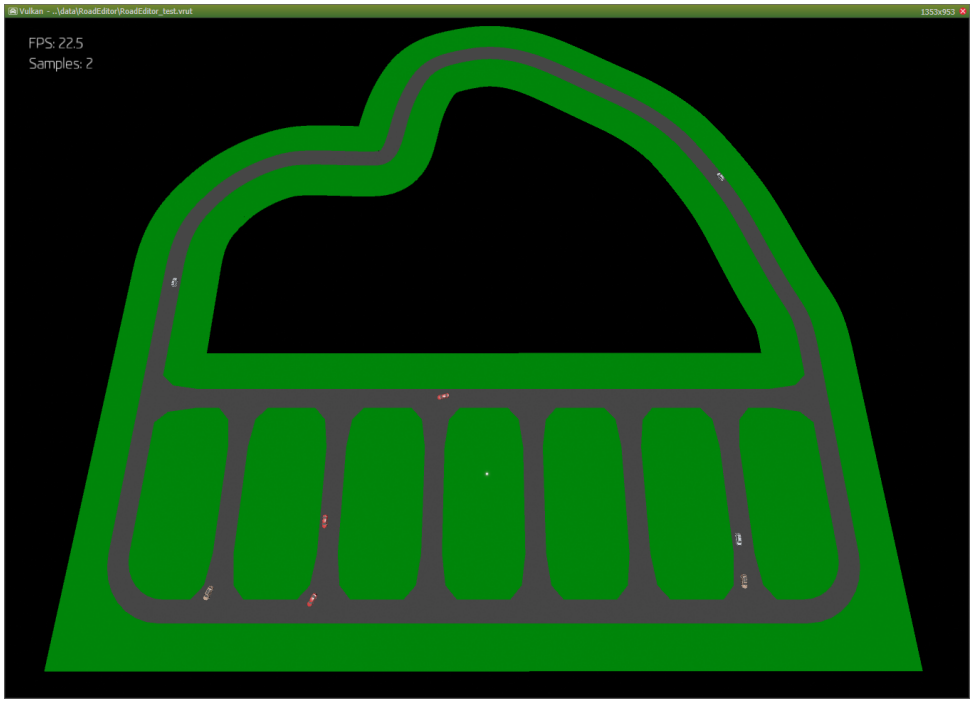


Figure 6.7: Small scene with straight block deformation.

In Figure 6.8 is an example of the possibility to create any kind of network, eventually according to a given template. The fitting normals feature was used only for straight blocks as it should be not desired for the other blocks to respect such a feature.

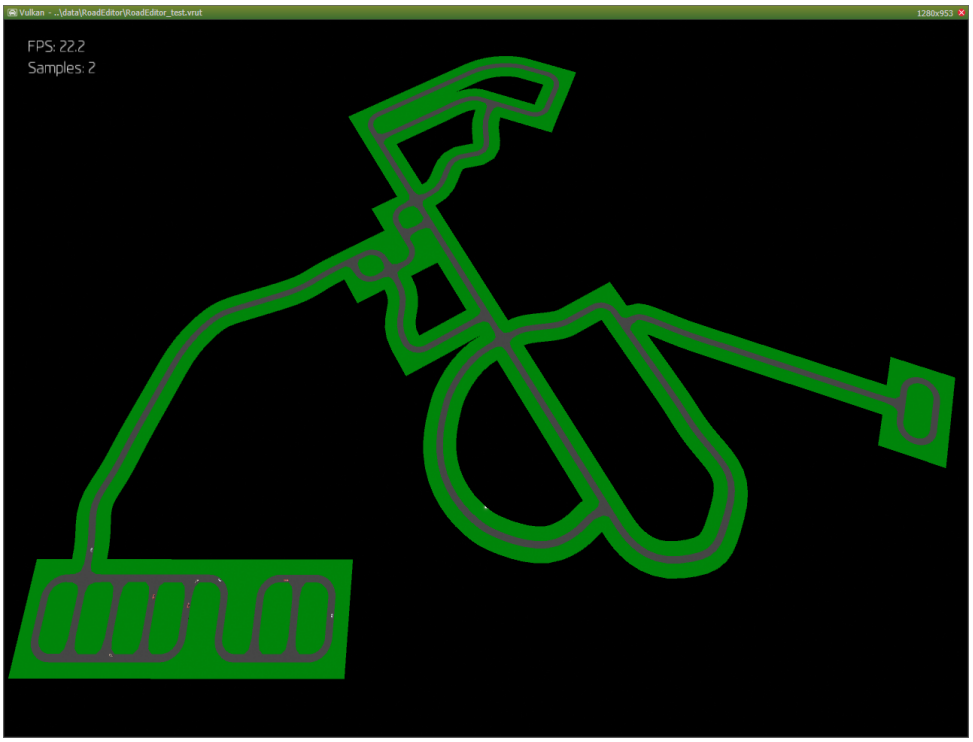


Figure 6.8: Large scene with block deformation.

These scenes are bellow in Table 6.1 depicted in numbers. All scenes have been tested both in debug and release configuration. The scene size correlates with the fps, and the fps drop rapidly with depicting traffic nodes using the Traffic module.

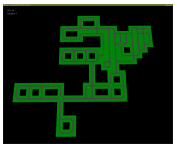
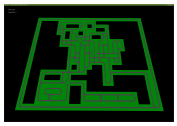
Scene name	Oval	Junctions	Middle	Large	Small_def	Large_def
Scene preview						
Scene nodes	146	356	1470	1964	292	760
Blocks count	18	44	182	243	36	94
Road length(km)	0,90	2,55	10,70	14,30	1,80	5,55
Junctions count	0	18	39	72	14	21
Deformation applied	NO	NO	NO	NO	YES	YES
Polygons count	123	310	1320	1762	280	689
Nodes count	136	668	2720	3688	510	1238
Edge count	136	610	2562	3342	460	1120
Cars count	8	10	10	10	10	10
FPS (debug)	551 (29.2)	510 (24.6)	367 (14.0)	380 (14.5)	492 (22.5)	447 (22.2)
Creation time (min)	0,58	1,58	3,17	5,42	1,75	6,34

Table 6.1: Resulting scenes in numbers. Numbers were generated using the module Optimize which shows all scene parameters.

Lastly, in Figure 6.9 is an example of road network creation using map template from *Geoportal maps* [27]. The yellow spheres in the picture are just a visualization of the block connections that can be easily turned off in the module's GUI.

Figure 6.9: Scene creation using map background; Top - mere map background; Bottom - traffic network composed using RoadEditor module along the map background's roads.

The RoadEditor module was tested with the *VehicleSimulator* module that simulates the traffic behavior over the generated network. Figure 6.10 shows cars driven by *VehicleSimulator* over by-RoadEditor-generated networks.

Figure 6.10: RoadEditor module's resulting scenes driven by vehicles using *VehicleSimulator* module.

The implementation consisted of numerous files and the number of lines of

each file type is depicted in Table 6.2.

File_type	lines(∼)
CPP	3899
HTML	203
JSS	78
CS	66
XML	649
OVERALL	4917

Table 6.2: Number of lines of each file type.

6.3 Manual

In order to control the module the user should be familiar with the module manual that stays in Table 6.3. The module controlling should be intuitive and easy to learn.

Input	Function
MOUSE	
Left mouse button	block insertion, block selection, when *None* block selected → enter *blockEditMode*, selected block's CP or arrow selection
Right mouse button	selected block rotation clockwise by 90 degrees
Mouse movement (inserting blocks)	selected block positioning in scene (grid or magnetic mode)
Mouse movement (editing CPs)	repositioning selected connection point
Mouse movement (editing arrows)	setting Bézier curve parameters, defines direction and length of the arrow defining control point
KEYBOARD	
Del	selected block deletion
Tab	switch between magnetic and grid insertion mode
O	slight rotation of the selected block CCW by 10 degrees
P	slight rotation of the selected block CW by 10 degrees
Ctrl+C	copy selected block
Ctrl+V	paste copied block at cursor's position

Table 6.3: RoadEditor module control manual.

Chapter 7

Discussion

7.1 Problems during the implementation

During the implementation I was facing several problems that are worth mentioning. These are listed here with their solutions.

7.1.1 Block junctions definition

As the problem was subdivided into several small pieces, junctions like intersections and t-cross junctions had to be solved. Traffic representation was more complex with the incomplete documentation of the roadgraph in the system VRUT. Even with an example of an intersection given from already operational road graphs, the definition was complicated. With pieces of advice from Ing. Jaroslav Sloup I implemented these blocks using junction tags for each junction's lane, which finally worked.

7.1.2 Vehicle's falling through the road

One of the largest problem with an easy solution was that the cars spawned in the scene using the VehicleSimulator module were falling through the road and did not collide with any in-scene object at all. The problem here was that there has to be a main scene node defined in the *RoadGraph.xml*, which immediately made the simulation work.

7.1.3 Geometry instancing

Next problem that has been dealt with during the implementation was during block copy implementation. As there has been block deformation feature implemented, the block's original geometry has been deformed and so have all its instances. In order to resolve such a problem, there has to be a new geometry instance created during any new block's deformation.

7.2 Development at DigiteqAutomotive

As I work at DigiteqAuomotive (DQ) company, the development there has been a great help and inspiration for the RoadEditor module implementation. As the problem does not differ mostly, the representation of the network and approach we use there have been advantageous.

The above-mentioned Mobile Traffic System (MTS) is developed and extended in DQ and the RoadEditor module is and can be further inspired by such a solution.

Furthermore, the DQ company, more concretely the VXLab, department is more focused on infotainment and its distraction during driving. We develop simulations and hardware to test the on-board UI (User Interface) of the built-in display

7.3 Implications for further research

This section deals with a couple of ideas for further implementation and highlights its pros, cons, difficulty of the implementation and all the requirements for such a solution.

7.3.1 Realistic blocks

One of the first module extensions could be a realistic block's geometry creation as it would improve the quality of road scenes created using the RoadEditor module. This solution could be used on already created blocks and update only the block's geometry. However, for a better resulting scene experience, the continuity of the block's geometry and textures would have to be solved, including road stripes and road surrounding geometry like barriers.

7.3.2 Any block deformation

As it is possible to deform any kind of block, the feature is yet implemented to support primarily straight blocks. Further implementation by either using Cage box deformation or extending the Bézier curve deformation might be the solution.

7.3.3 Traffic graph depiction

As for a solution yet implemented, in case the user wants to see the whole graph created by the RoadEditor module, the user needs to use the Traffic module for such a depiction. Some elementary ways of rendering such nodes might be predefined in the RoadEditor module in order to make it independent on the Traffic editor during building the traffic network.

Certainly for further use of the created traffic network and simulation, the Traffic module is required.

7.3.4 Saving deformed blocks

As the saving of blocks yet saves only the block's names and transformation matrix, it does not save the instance of newly deformed geometry. This could be either approached the hard way, where we save all the geometry vertices' positions. A better approach would be by creating a new instance of a block and adding it to the block templates by exporting the block's geometry and its traffic nodes to a new *.xml* file.

Chapter 8

Conclusion

This work is focused on studying methods used for traffic network representation in map and navigation systems. The goal was to implement an interactive editor that enables users to compose a traffic network using prepared blocks that represent important parts of the traffic network. First, I analyzed methods in other traffic editors and engines and their representations. Furthermore, VRUT, an application for the visualization and edition of 3D data was analyzed and introduced. Next, the concept design and block deformation approach were analyzed. Gained knowledge was used for the implementation procedure and module development. Based on this concept, the implementation in the system VRUT took place. Results were demonstrated on scenes representing distinct traffic networks.

During the implementation, it was emphasized to design a module that would be intuitive, interactive and easy to use. Blocks parameterization and deformation were described and demonstrated on sample examples. Module control was introduced and possible further module implications were proposed. Furthermore, six different scenes with both deformed and undeformed blocks representing traffic networks were created and tested with the help of other modules.

Bibliography

[1] Paden, B., Čáp, 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.

[2] *Projekt OpenDrive.* http://www.opendrive.org/

[3] Jesús R. Nieto, Antonio Susín. *Cage based deformations: a survey.* University of the Balearic Islands, 2013.

[4] David Tichý. *Simulátor jízdy městem.* Diplomová práce ČVUT FEL, 2006.

[5] Václav Kyba. *Modulární 3D prohlížeč.* Diplomová práce ČVUT FEL, 2008.

[6] Jaroslav Minařík. *Simulace okolních dopravních dějů.* Diplomová práce ČVUT FEL, 2014.

[7] Alena Mikushina. *Tvorba modulárních 3D komponent pro videohry*, Bakalářská práce ČVUT FEL, 2020.

[8] Daniel Aschermann. *Editor geometrie v systému VRUT*, Bakalářská práce ČVUT FEL, 2020.

[9] Vojtěch Kolínský. *Editor silniční sítě v systému Virtual Reality Universal Toolkit*, Bakalářská práce ČVUT FEL, 2020.

[10] Klára Pudová. *Automated creation of traffic in scenarios for driving simulators*, Diplomová práce ČVUT FD, 2019.

[11] Karan Singh, Eugene Fiume. *Wires: A Geometric Deformation Technique*, SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques, 1998.

[12] Jaume Barceló. *Fundamentals of Traffic Simulation*, Dept. Statistics & Operations Research, Universitat Politècnica de Catalunya, Barcelona, Spain, 2010.

[13] Hans-Thomas Fritszsche. *A model for traffic simulation*, Daimler-Benz AG 1994.

[14] G. Kotusevski and K. A. Hawick. *A Review of Traffic Simulation Software*, Computer Science Institute of Information & Mathematical Sciences Massey University at Albany, Auckland, New Zealand, 2009.

[15] J. Montagnat, H. Delingette and N. Ayache. *A review of deformable surfaces: topology, geometry and deformation*, INRIA, 2004 route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex, France, 2001.

[16] *EulerSpiral (clothoid).* https://github.com/CoffeeKumazaki/euler_spiral/

[17] Nicolas Marechal, Eric Galin, Adrien Peytavie, N Maréchal, Eric Guérin. *Procedural Generation of Roads*, Computer Graphics Forum, 2010, 2, 29, pp.429-438.

[18] Tiago Boelter Mizdal, Cesar Tadeu Pozzer. *Procedural Content Generation of Villages and Road System on Arbitrary Terrains*, XVII SBGames, 2018.

[19] *Quadratic and Cubic Bézier Curves.* http://www.e-cartouche.ch/

[20] Matthew Guzdial, Duri Long, Christopher Cassion, Abhishek Das. *Visual Procedural Content Generation with an Artificial Abstract Artist*, College of Computing, Georgia Institute of Technology, Atlanta, GA 30308 USA, 2018.

[21] *Euler spiral.* https://en.wikipedia.org/wiki/Euler_spiral

[22] *Bézier curve.* https://en.wikipedia.org/wiki/Bézier_curve

[23] *EasyRoads3D Pro v3.* https://assetstore.unity.com/packages/tools/terrain/easyroads3d-pro-v3-469

[24] *Mobile Traffic System.* https://assetstore.unity.com/packages/tools/ai/mobile-traffic-system-194888

[25] *MATHEMATICS Euler spiral.* https://math.stackexchange.com/questions/3791546/how-many-points-are-needed-to-uniquely-define-an-euler-spiral

[26] *Bézier C2 continuity.* http://www.linkeova.cz/vyuka/pg_pgs/public/pgr_skripta.pdf

[27] *Geoportal maps.* https://www.geoportalpraha.cz/cs/mapy/mapove-aplikace