DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Semesterarbeit in Informatics

Thesis title

Autonomous Driving Simulator and Benchmark on Neurorobotics Platform

Liu, Hongshen

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Semesterarbeit in Informatics

Thesis title

Autonomous Driving Simulator and Benchmark on Neurorobotics Platform

Author:	Liu, Hongshen
Supervisor:	Knoll Alois Christian; Prof. Dr.-Ing. habil.
Advisor:	Zhou, Liguo
Submission Date:	22.12.2022

I confirm that this semesterarbeit in informatics is my own work and I have documented all sources and material used.

Munich, 22.12.2022 Liu, Hongshen

Abstract

In our days, vehicle automation is in a continuous evolutionary phase consisting of experiments, testing and validation. Accelerating the development and deployment of autonomous vehicles and infrastructure is a real demand as these technologies have a great potential to improve traffic safety and resolve road transport problems. The Vehicle-In-the-Loop testing is therefore indispensable throughout the development process.This article describes two potential solutions. The first one is a simulator built with ROS as the communication framework and Gazebo as the physical simulation platform. The motion control of the test vehicle and the implantation of the target detection algorithm are implemented. The second simulator is based on SUMO and Unity 3D. large scale traffic flow is generated in SUMO and mapped to Unity 3D via TraCI server and TCP/IP communication. unity 3D generates high fidelity mapped vehicles and performs real time simulation.ROS2 is used as the communication framework of the algorithm module to facilitate the implantation and testing of various perception, decision-making, and planning algorithms.

Kurzfassung

Heutzutage befindet sich die Fahrzeugautomatisierung in einer kontinuierlichen Entwicklungsphase, die aus Experimenten, Tests und Validierung besteht. Die Beschleunigung der Entwicklung und des Einsatzes von autonomen Fahrzeugen und Infrastrukturen ist ein echter Bedarf, da diese Technologien ein großes Potenzial zur Verbesserung der Verkehrssicherheit und zur Lösung von Straßenverkehrsproblemen haben. Vehicle-In-the-Loop-Tests sind daher während des gesamten Entwicklungsprozesses unverzichtbar, und in diesem Artikel werden zwei mögliche Lösungen beschrieben. Bei der ersten handelt es sich um einen Simulator, der mit ROS als Kommunikationsrahmen und Gazebo als physikalische Simulationsplattform aufgebaut ist. Die Bewegungssteuerung des Testfahrzeugs und die Implantation des Zielerkennungsalgorithmus sind implementiert. Der zweite Simulator basiert auf SUMO und Unity 3D. Ein groß angelegter Verkehrsfluss wird in SUMO generiert und über einen TraCI-Server und TCP/IP-Kommunikation auf Unity 3D abgebildet. Unity 3D generiert realitätsgetreu abgebildete Fahrzeuge und führt Echtzeitsimulationen durch.ROS2 wird als Kommunikationsrahmen des Algorithmusmoduls verwendet, um die Implantation und das Testen verschiedener Wahrnehmungs-, Entscheidungs- und Planungsalgorithmen zu erleichtern.

Contents

1 ROS-based autonomous driving platform

1.1 Concept

Robot system simulation is a technology that simulates a physical robot system by computer. In ROS, simulation implementation involves three main elements: modeling the robot (URDF), creating a simulation environment (Gazebo), and sensing the environment (Rviz), and other systematic implementations.

1.1.1 Advantages and disadvantages

Simulation plays a pivotal role in the development of robotic systems and has the following significant advantages over physical robot implementation in R&D and testing:

1. **low cost:** the current high cost of robotics, often hundreds of thousands of dollars, simulation can greatly reduce the cost and reduce the risk.
2. **High efficiency:** The built environment is more diverse and flexible, which can improve test efficiency and test coverage
3. **High safety:** In the simulation environment, there is no need to consider the problem of wear and tear.

The performance of the robot in the simulation environment and the actual environment is different, in other words, the simulation does not completely simulate the real physical world, there are some "distortion" situation, the reason:

- The physics engine used by the simulator is not yet able to simulate the real-world physics completely and accurately.
- The simulator is built for the absolute ideal situation of joint drive (motor & gearbox), sensor and signal communication, and currently does not support the simulation of actual hardware defects or some critical states and other situations.

1.1.2 Related components

1. **URDF**
   URDF is an acronym for Unified Robot Description Format, which directly translates to Uniform (Standardized) Robot Description Format, and can be used to describe parts of the robot's structure in an XML way, such as the chassis, camera, LIDAR, robot arm, and the degrees of freedom of different joints ..... This file can be converted into a

visual robot model by the C++ built-in interpreter, and is an important component for implementing robot simulation in ROS.

2. **Rviz**
   RViz is an acronym for ROS Visualization Tool, which translates directly to ROS 3D visualization tool. Its main purpose is to display ROS messages in three dimensions, allowing visual representation of the data. For example: robot models can be displayed, sensor-to-obstacle distances from laser range finder (LRF) sensors can be expressed without programming, point cloud data from 3D distance sensors such as RealSense, Kinect or Xtion, image values from cameras, etc.

3. **RoadRunner**
   RoadRunner is an interactive editor that lets you design 3D scenes for simulating and testing automated driving systems. You can customize roadway scenes by creating region-specific road signs and markings. You can insert signs, signals, guardrails, and road damage, as well as foliage, buildings, and other 3D models. RoadRunner provides tools for setting and configuring traffic signal timing, phases, and vehicle paths at intersections.

   RoadRunner supports the visualization of lidar point cloud, aerial imagery, and GIS data. You can import and export road networks using OpenDRIVE. 3D scenes built with RoadRunner can be exported in FBX, glTF™, OpenFlight, OpenSceneGraph, OBJ, and USD formats. The exported scenes can be used in automated driving simulators and game engines, including CARLA, Vires VTD, NVIDIA DRIVE Sim, Baidu Apollo, Cognata, Unity, and Unreal Engine.

4. **Gazebo 11.0**

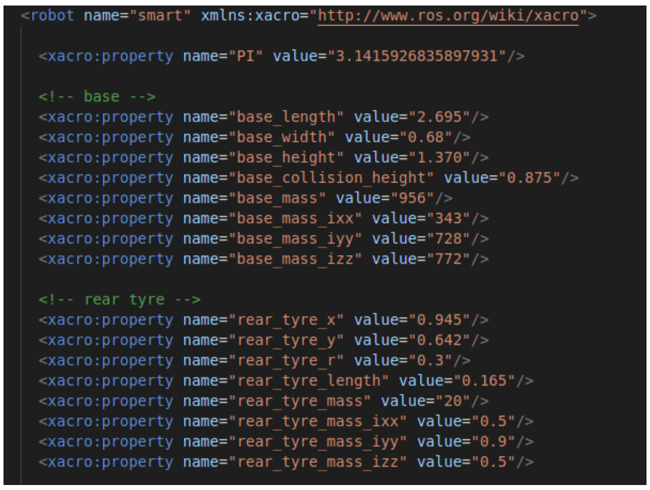
   Gazebo is a powerful 3D physics simulation platform with a powerful physics engine, high quality graphics rendering, easy programming and graphics interface, and most importantly, its open source and free nature. gazebo has the same robot model as the one used by rviz, but requires the addition of physical properties of the robot and its surroundings to the model, such as mass, friction coefficient, coefficient of elasticity, etc. The sensor information of the robot can also be added to the simulation environment in the form of a plug-in for visualization.

1.2 Construction of simulation platform

1.2.1 Create vehicle model

The experimental vehicle is a highly detailed model of a car with independent controllable steering for Ackermann steering control of the two front wheels, free front and rear wheels and a high definition camera. The vehicle model is drawn and shaped by URDF language rules. The specific idea of the implementation is:

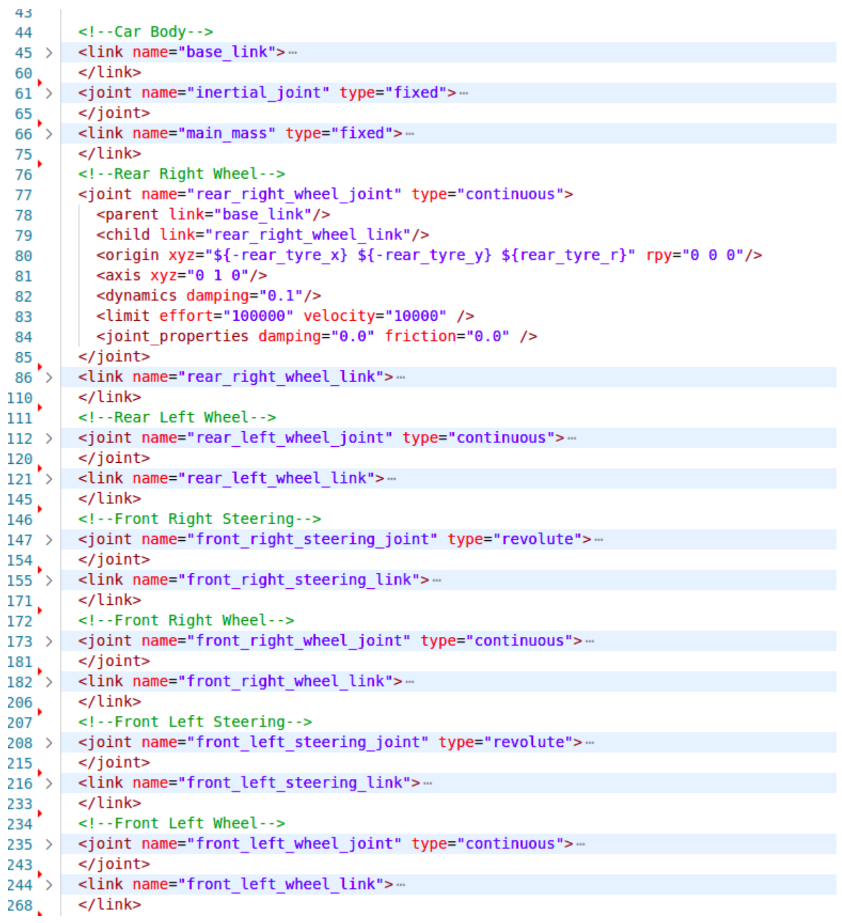
1. **Use xacro to optimize URDF.**
   Use xacro to optimize URDF. Wrap some chassis parameters, variables as xacro:property. e.g.: PI value, cart chassis radius, ground clearance, wheel radius, width, etc.



```xml
<robot name="smart" xmlns:xacro="http://www.ros.org/wiki/xacro">

  <xacro:property name="PI" value="3.1415926835897931"/>

  <!-- base -->
  <xacro:property name="base_length" value="2.695"/>
  <xacro:property name="base_width" value="0.68"/>
  <xacro:property name="base_height" value="1.370"/>
  <xacro:property name="base_collision_height" value="0.875"/>
  <xacro:property name="base_mass" value="956"/>
  <xacro:property name="base_mass_ixx" value="343"/>
  <xacro:property name="base_mass_iyy" value="728"/>
  <xacro:property name="base_mass_izz" value="772"/>

  <!-- rear tyre -->
  <xacro:property name="rear_tyre_x" value="0.945"/>
  <xacro:property name="rear_tyre_y" value="0.642"/>
  <xacro:property name="rear_tyre_r" value="0.3"/>
  <xacro:property name="rear_tyre_length" value="0.165"/>
  <xacro:property name="rear_tyre_mass" value="20"/>
  <xacro:property name="rear_tyre_mass_ixx" value="0.5"/>
  <xacro:property name="rear_tyre_mass_iyy" value="0.9"/>
  <xacro:property name="rear_tyre_mass_izz" value="0.5"/>
```

Figure 1.1: Wrap chassis parameters, variables as xacro:property

2. **Use links and joints to draw chassis entities and chassis-to-wheel connections.**
   The link element describes a rigid body with an inertia, visual features, and collision properties.The joint element describes the kinematics and dynamics of the joint and also specifies the safety limits of the joint. Attension, the name of model or link must be specific and has no same name with other objects.The below figure 1.2 shows the base architecture frame to describe the physical relationship of the whole vehicle in URDF file:



```
<!--Car Body-->
<link name="base_link">⋯
</link>
<joint name="inertial_joint" type="fixed">⋯
</joint>
<link name="main_mass" type="fixed">⋯
</link>
<!--Rear Right Wheel-->
<joint name="rear_right_wheel_joint" type="continuous">
  <parent link="base_link"/>
  <child link="rear_right_wheel_link"/>
  <origin xyz="${-rear_tyre_x} ${-rear_tyre_y} ${rear_tyre_r}" rpy="0 0 0"/>
  <axis xyz="0 1 0"/>
  <dynamics damping="0.1"/>
  <limit effort="100000" velocity="10000" />
  <joint_properties damping="0.0" friction="0.0" />
</joint>
<link name="rear_right_wheel_link">⋯
</link>
<!--Rear Left Wheel-->
<joint name="rear_left_wheel_joint" type="continuous">⋯
</joint>
<link name="rear_left_wheel_link">⋯
</link>
<!--Front Right Steering-->
<joint name="front_right_steering_joint" type="revolute">⋯
</joint>
<link name="front_right_steering_link">⋯
</link>
<!--Front Right Wheel-->
<joint name="front_right_wheel_joint" type="continuous">⋯
</joint>
<link name="front_right_wheel_link">⋯
</link>
<!--Front Left Steering-->
<joint name="front_left_steering_joint" type="revolute">⋯
</joint>
<link name="front_left_steering_link">⋯
</link>
<!--Front Left Wheel-->
<joint name="front_left_wheel_joint" type="continuous">⋯
</joint>
<link name="front_left_wheel_link">⋯
</link>
```

Figure 1.2: Relationship of the whole vehicle

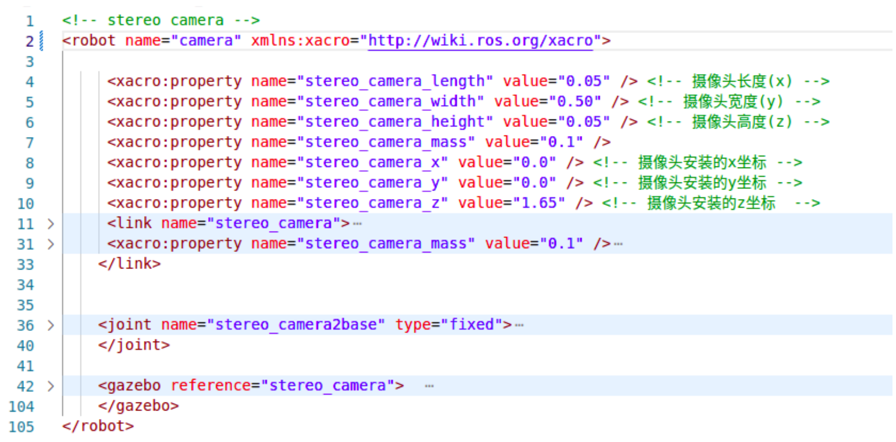
a) **Description of URDF Labels**

- **<link>** — The corresponding model as component from the entirety model.
- **<joint>** — Description of relationship between link-components <joint type> — Type of the joint:
  - **revolute** — a hinge joint that rotates along the axis and has a limited range specified by the upper and lower limits.
  - **continuous** — a continuous hinge joint that rotates around the axis and has no upper and lower limits.
  - **prismatic** — a sliding joint that slides along the axis, and has a limited range specified by the upper and lower limits.
  - **fixed** — this is not really a joint because it cannot move. All degrees of freedom are locked. This type of joint does not require the <axis>,<calibration>, <dynamics>, <limits> or <safety_controller>.
  - **floating** — this joint allows motion for all 6 degrees of freedom.

    - **planar** — this joint allows motion in a plane perpendicular to the axis.

- **<parent>/<child>**
    - - the secondary label as element of <joint> label
    - — declaration for the belonging relationship of referring "links"

b) **Sensor Label**
A new URDF file is created to describe the camera configuration. Here the stereo camera is selected for the implantation of the algorithm related to binocular ranging And the detailed construction for the camera sensor seeing blow scripts:



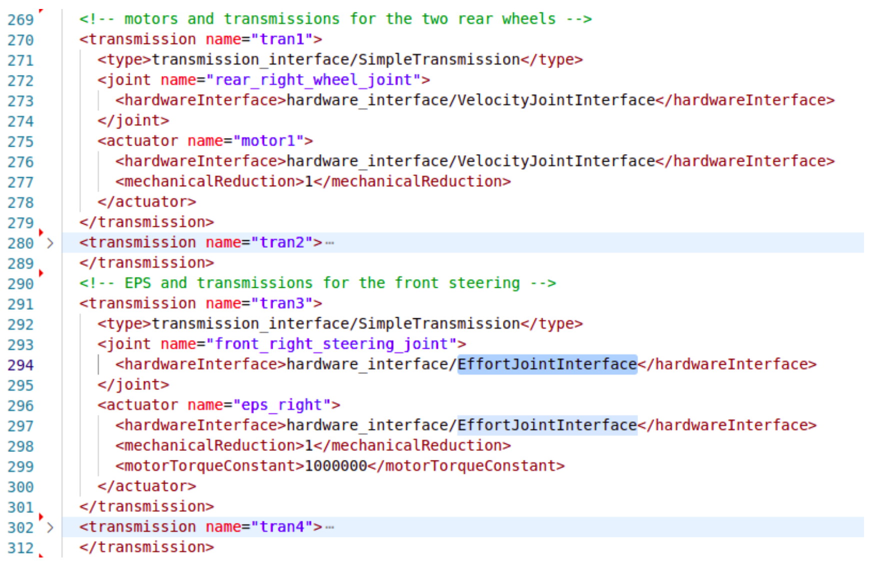
```
<!-- stereo camera -->
<robot name="camera" xmlns:xacro="http://wiki.ros.org/xacro">

    <xacro:property name="stereo_camera_length" value="0.05" /> <!-- 摄像头长度(x) -->
    <xacro:property name="stereo_camera_width" value="0.50" /> <!-- 摄像头宽度(y) -->
    <xacro:property name="stereo_camera_height" value="0.05" /> <!-- 摄像头高度(z) -->
    <xacro:property name="stereo_camera_mass" value="0.1" />
    <xacro:property name="stereo_camera_x" value="0.0" /> <!-- 摄像头安装的x坐标 -->
    <xacro:property name="stereo_camera_y" value="0.0" /> <!-- 摄像头安装的y坐标 -->
    <xacro:property name="stereo_camera_z" value="1.65" /> <!-- 摄像头安装的z坐标  -->
    <link name="stereo_camera">⋯
    <xacro:property name="stereo_camera_mass" value="0.1" />⋯
   </link>


   <joint name="stereo_camera2base" type="fixed">⋯
   </joint>

   <gazebo reference="stereo_camera">   ⋯
   </gazebo>
</robot>
```

Figure 1.3: Implementation of stereo camera

According to the requirement of YOLO detect algorithm the width and height of camera should be set as integral multiples by 32.

3. **Construction of vehicle controller**
Simulating a robot's controllers in Gazebo can be accomplished using ros_control and a simple Gazebo plugin adapter. ros_control is a framework for implementing and managing robot controllers, and is dedicated to providing a robot-agnostic approach to controller design with real-time performance. ros_control is derived from the PR2 robot controller pr2_mechanism, but ros_control is completely robot-agnostic. It is now a standard controller interface in ROS. The specific implementation method is as follows:

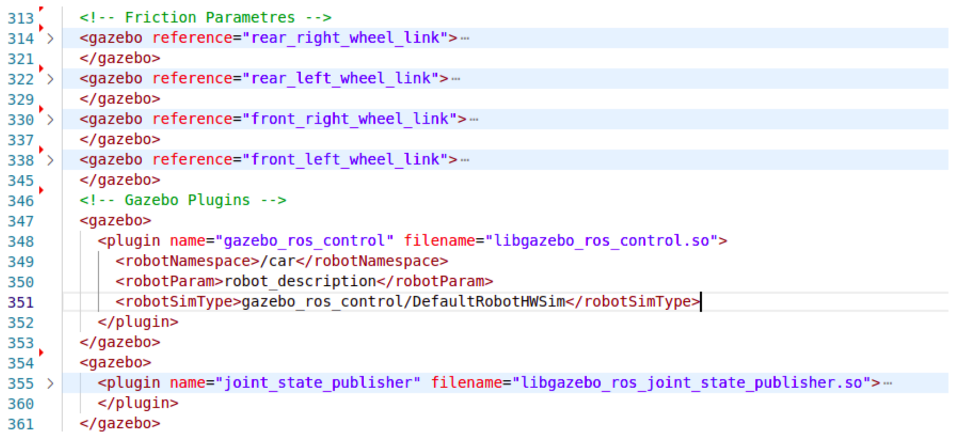
    a) **Add the gazebo_ros_control plugin to URDF file**
    The transmission element is an extension to the URDF robot description model that is used to describe the relationship between an actuator and a joint. Four transmission tags are used here to set the relationship between each of the four wheels and the controllers. Four transmission tags are used here to set the relationship between each of the four wheels and the controller. The rear wheels are the drive wheels and the front wheels are the steering wheels. As shown in the figure belowthe <hardwareInterface> attribute of the rear wheel controller is set to VelocityJointInterface, and the front wheel driver is set to EffortJointInterface.



```xml
<!-- motors and transmissions for the two rear wheels -->
<transmission name="tran1">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="rear_right_wheel_joint">
    <hardwareInterface>hardware_interface/VelocityJointInterface</hardwareInterface>
  </joint>
  <actuator name="motor1">
    <hardwareInterface>hardware_interface/VelocityJointInterface</hardwareInterface>
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>
</transmission>
<transmission name="tran2">…
</transmission>
<!-- EPS and transmissions for the front steering -->
<transmission name="tran3">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="front_right_steering_joint">
    <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface>
  </joint>
  <actuator name="eps_right">
    <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface>
    <mechanicalReduction>1</mechanicalReduction>
    <motorTorqueConstant>1000000</motorTorqueConstant>
  </actuator>
</transmission>
<transmission name="tran4">…
</transmission>
```

Figure 1.4: Transmission for wheels and controllers

b) **Add the gazebo_ros_control plugin to URDF file**
In addition to the transmission tags, a Gazebo plugin needs to be added to your URDF that actually parses the transmission tags and loads the appropriate hardware interfaces and controller manager. By default the gazebo_ros_control plugin is very simple, though it is also extensible via an additional plugin architecture to allow power users to create their own custom robot hardware interfaces between ros_control and Gazebo. Pluginlib-based interface provided by the gazebo_ros_control Gazebo plugin is usd to implement custom interfaces between Gazebo and ros_control for simulating more complex mechanisms (nonlinear springs, linkages, etc).



```xml
<!-- Friction Parametres -->
<gazebo reference="rear_right_wheel_link">…
</gazebo>
<gazebo reference="rear_left_wheel_link">…
</gazebo>
<gazebo reference="front_right_wheel_link">…
</gazebo>
<gazebo reference="front_left_wheel_link">…
</gazebo>
<!-- Gazebo Plugins -->
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    <robotNamespace>/car</robotNamespace>
    <robotParam>robot_description</robotParam>
    <robotSimType>gazebo_ros_control/DefaultRobotHWSim</robotSimType>
  </plugin>
</gazebo>
<gazebo>
  <plugin name="joint_state_publisher" filename="libgazebo_ros_joint_state_publisher.so">…
  </plugin>
</gazebo>
```

Figure 1.5: Implementation for Gazebo_ros_controller plugin

c) **Create a .yaml config file**
The PID gains and controller settings must be saved in a yaml file that gets loaded to the param server via the roslaunch file. The .yaml file is created in the config

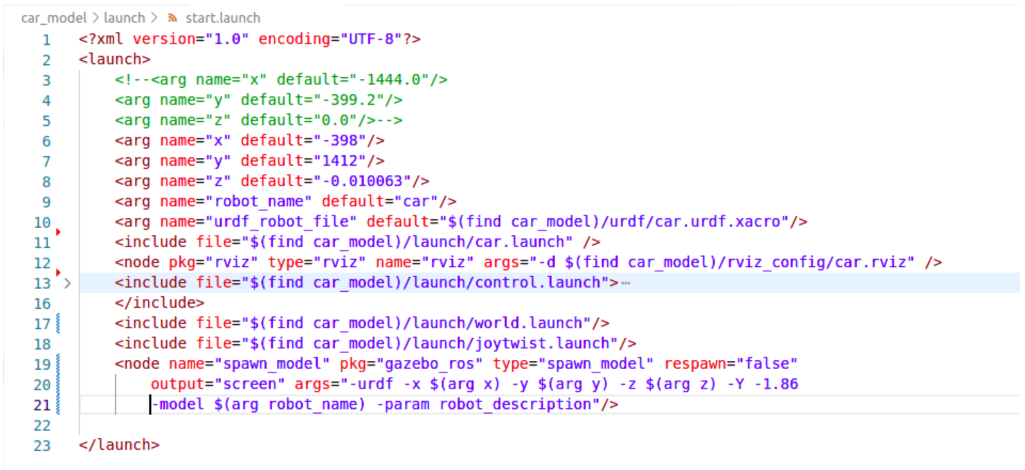
folder of the package and the PID parameters for each wheel controller are written in it. See the official documentation for details on how to do this.

d) **Graphical visualization in Rviz and Gazebo**
The URDF model file of the vehicle is integrated into Rviz and Gazebo via the launch file. Rviz and Gazebo will render the URDF text into a graphical model of the car.

i. **Create roslaunch file for starting the ros_control controllers.**
First the config file is uploaded to the parameter server via the rosparam tag.The controller_spawner node starts the four joint position controllers for the vehicle by running a python script that makes a service call to the ros_control controller manager. The service calls tell the controller manager which controllers you want. It also loads a third controller that publishes the joint states of all the joints with hardware_interfaces and advertises the topic on /joint_states. The spawner is just a helper script for use with roslaunch. The final line starts a robot_state_publisher node that simply listens to /joint_states messages from the joint_state_controller then publishes the transforms to /tf. This allows you to see your simulated robot in Rviz as well as do other tasks.

ii. **Create roslaunch file for starting Gazebo**
the map file path is wirten to gazebo's launch file. This map is drawn using the Roadrunner.This map model is based on the high accuracy satellite generated and very similar to origin location.

iii. **Integration of launch file.**
A new launch fild named start.launch is created for starting the whole simulation. As shown in the figure belowfirst the car initial position parameters and the car model URDF file are uploaded to the parameter server. Then the controller launch file and the gazebo launch file and rviz node are written to this launch file.



car_model > launch > start.launch

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
    <!--<arg name="x" default="-1444.0"/>
    <arg name="y" default="-399.2"/>
    <arg name="z" default="0.0"/>-->
    <arg name="x" default="-398"/>
    <arg name="y" default="1412"/>
    <arg name="z" default="-0.010063"/>
    <arg name="robot_name" default="car"/>
    <arg name="urdf_robot_file" default="$(find car_model)/urdf/car.urdf.xacro"/>
    <include file="$(find car_model)/launch/car.launch" />
    <node pkg="rviz" type="rviz" name="rviz" args="-d $(find car_model)/rviz_config/car.rviz" />
    <include file="$(find car_model)/launch/control.launch">···
    </include>
    <include file="$(find car_model)/launch/world.launch"/>
    <include file="$(find car_model)/launch/joytwist.launch"/>
    <node name="spawn_model" pkg="gazebo_ros" type="spawn_model" respawn="false"
        output="screen" args="-urdf -x $(arg x) -y $(arg y) -z $(arg z) -Y -1.86
        -model $(arg robot_name) -param robot_description"/>

</launch>
```

Figure 1.6: Launch file for Simulation

e) **Start the Simulation using roslaunch**
Run start.launch with the roslaunch command, the result is shown in the figure below.

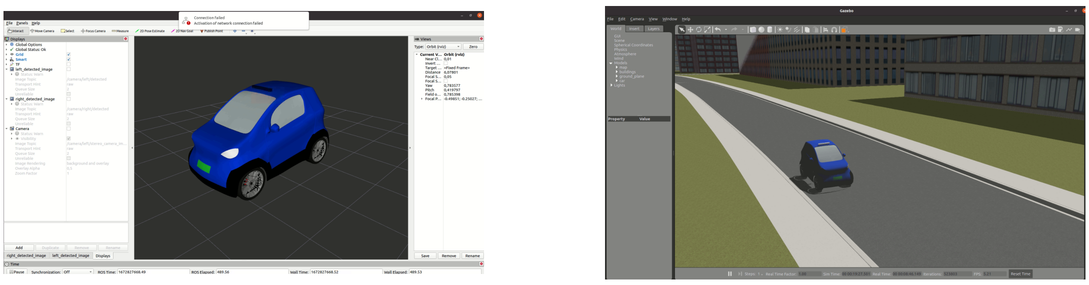


Figure 1.7: Visualization in Rviz and Gazebo

1.2.2 Implementation of Ackermann steering in ROS

Ackermann steering mechanism was proposed by the German vehicle engineer Lankensperger in 1817 and later patented by his British agent Rudolph Ackermann in 1818. The Ackermann steering mechanism (Ackermann steering) is designed to solve the problem of different steering angles of the left and right steering wheels caused by the different steering radii of the left and right steering wheels when the vehicle is steering. The vehicle is designed with the steering mechanism according to the Ackermann steering geometry, and when the vehicle turns along the curve, the equal crank of the four-link can make the steering angle of the inner wheel bigger than the outer wheel by about 24̃ degrees, so that the center of the circle of the four wheel paths roughly meet at the instantaneous steering center on the extension line of the rear axle, thus allowing the vehicle to turn smoothly. Therefore, in order to ensure the realism of the simulation and the stability of the experimental vehicle it is necessary to implement Ackermann steering of the experimental vehicle in ROS.

1. **Implementation of Ackermann steering node in Python**
   The speed and angular velocity of the car in the forward direction can be obtained by subscribing to the cmd_vel topic. As shown in the figure below, the steering angle of the dashed wheel is the subscribed angular velocity. This steering angle is the Ackermann steering angle.

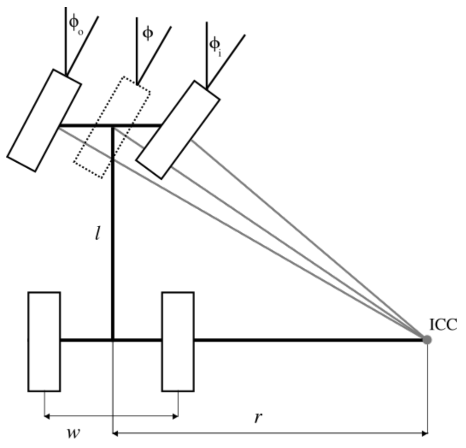


Figure 1.8: Ackermann steering mechanism

According to the structural analysis Ackermann kinematic model, the the following relation is derived.

$$r = \frac{l}{\tan \Phi} \tag{1.1}$$

$$r\_R = r - \frac{w}{2} \tag{1.2}$$

$$r\_L = r + \frac{w}{2} \tag{1.3}$$

$$\phi\_i = \arctan\left(\frac{l}{r\_R}\right) \tag{1.4}$$

$$\phi\_o = \arctan\left(\frac{l}{r\_L}\right) \tag{1.5}$$

$$v\_R = v * \frac{r\_R}{r} \tag{1.6}$$

$$v\_L = v * \frac{r\_L}{r} \tag{1.7}$$

The angle of the two steering wheels and the speed of the two driving wheels can be obtained by the above relation.It is then posted separately to the topic of the corresponding controller speed command. Note here that the name of the topic to be published should be the same as the name of the topic to which the gazebo_ros_control plugin is subscribed.

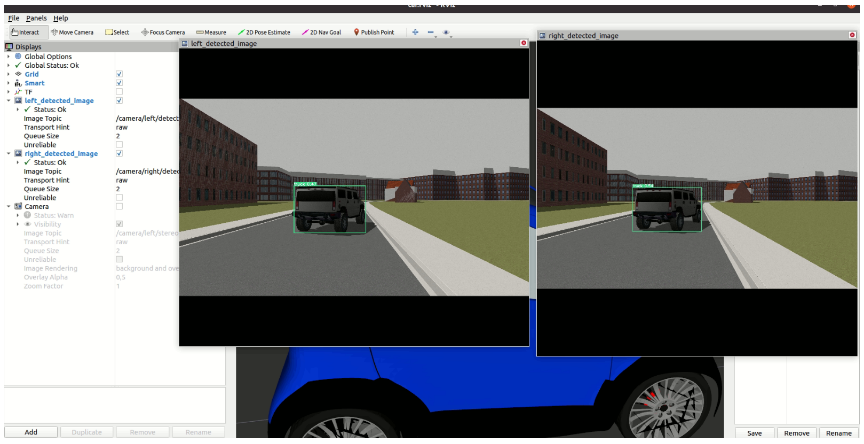
2. **Add node to launch file**
The Ackermann steering node is integrated into the start.launch written in the previous subsection. Once the simulation starts, the Ackermann steering node receives the speed message from cmd_vel, and after calculation, sends the Twist of each wheel to the controller state topic. the Gazebo controller plugin receives the speed information to control the vehicle movement and implement Ackermann steering. **[(TODO: rqt bild)]**

1.2.3 Implementation of Object detection

Object detection is an important research branch in the field of computer vision, which is the basic link of object recognition and tracking, and its main research content is to find out the object of interest in the image, including object localization and classification. Among them, traffic scene object detection and recognition is a hot problem in the field of computer vision research, and its purpose is to detect and identify vehicles, pedestrians and other traffic scene object information in the traffic scene using image processing, pattern recognition, machine learning, deep learning and other technologies to achieve the goal of intelligent transportation and automatic driving.
This subsection describes how to integrate YOLO v5 into the ROS environment.

1. **Impementation of YOLO v5 node in ROS.**
   Based on original execute-python-file "detect.py" has another python file "Yolov5Detector.py" with self-defined Yolov5Detector class interface been wrote in "yolov5" package. To use YOLO v5 should in main progress validate the yolo-v5 class, second use warm-up function "detectorWarmUp()" to initiate the neural network. And "detectImage()" is the function that sends image-frame to main preidict detection funtion and will final return the detected image with bonding-boxes in numpy format.

2. **Publish the detected images.**
   In the previous subsection stereo camera parameters are set by the built-in plugin of gazebo. gazebo publishes topics for each camera image, subscribes to the initial image topic in the detection node, and then passes the image to Yolo_v5_detector for processing to get an image with a booding box. Finally, the processed images are published, and the detected images are displayed by selecting the topic published by the detection node in rviz, as shown in the following figure. See the official documentation on how to write post and subscribe topics.



```
Global Options
Global Status: Ok
Grid                      ✓
Smart                     ✓
TF
left_detected_image       ✓
  Status: Ok
  Image Topic             /camera/left/detect
  Transport Hint          raw
  Queue Size              2
  Unreliable
right_detected_image      ✓
  Status: Ok
  Image Topic             /camera/right/detec
  Transport Hint          raw
  Queue Size              2
  Unreliable
Camera
  Status: Warn
  Visibility              ✓
  Image Topic             /camera/left/stereo
  Transport Hint          raw
  Queue Size              2
  Unreliable
  Image Rendering         background and ove
  Overlay Alpha           0,5
  Zoom Factor             1
```

Figure 1.9: Detected image with bonding bix

2 Autonomous driving simulator based on sumo and Unity3D

2.1 Introduction

In our time, vehicle automation is in a continuous evolutionary phase consisting of experimentation, testing and validation. Accelerating the development and deployment of self-driving vehicles and infrastructure is a real need, as these technologies have great potential to improve traffic safety and solve road transport problems. Therefore, vehicle-in-the-loop testing is essential throughout the development process. However, most simulation software today is unable to create a realistic simulation environment with high definition. This leads to less accurate testing of perception algorithms. As a potential solution to meet this need, we develop an autonomous driving simulation platform based on SUMO and Unity 3D. The platform is able to simulate real traffic around the self-driving test vehicles. A realistic simulation environment is generated and rendered in Unity 3D to meet the high fidelity required for computer vision. Provides interfaces to perception, planning, and decision algorithms to facilitate algorithm validation.

2.1.1 Software requirements

Based on the research of the software and the requirements of the simulation platform, SUMO and Unity 3D were finally selected as the basic development platform.

1. **SUMO**

   SUMO is an open source microscopic continuous traffic flow simulation software developed by the German National Aerospace Center. It comes with a traffic simulation road network editor that allows interactive editing to add roads, edit lane connections, handle intersection areas, edit signal timings, etc. It is also possible to convert road networks from Vissim, OpenStreetMap, OpenDrive through a separate conversion program. Routing can be specified for each vehicle by editing the routing file, or randomly generated using parameters. At runtime, it is possible to handle continuous traffic simulation requirements for several square kilometers and up to tens of thousands of vehicles simultaneously, and also provides an OpenGL-based visualization to display the traffic simulation results in real time.In addition, SUMO also supports secondary development based on C++.

2. **Unity 3D**

Unity 3D, also known as Unity, is a comprehensive multi-platform game development tool developed by Unity Technologies that allows players to easily create interactive content such as 3D video games, architectural visualizations, real-time 3D animations, and other types of interactive content.We chose Unity for the following main reasons.

- Unity has a great ecosystem. Unity has a great community where we can get feedback on various issues.
- Unity has a very good Asset Store resource store, we can reuse a lot of third party tools, we don't have to start everything from scratch and build the wheel from scratch.
- Unity has a very efficient graphics rendering system, and we can render realistic images in real time.
- Unity's performance is also getting better and better now, and it can help us a lot in terms of cloud acceleration and GPU acceleration.
- Last but not least, Unity is a traditional game engine, but now Unity is also looking more and more at the automotive industry, including the autonomous driving field, and Unity has recently launched a toolkit for autonomous driving, which is one of the main reasons why we are working with Unity.

3. **RoadRunner**

   RoadRunner is an interactive editor that lets you design 3D scenes for simulating and testing automated driving systems. You can customize roadway scenes by creating region-specific road signs and markings. You can insert signs, signals, guardrails, and road damage, as well as foliage, buildings, and other 3D models. RoadRunner provides tools for setting and configuring traffic signal timing, phases, and vehicle paths at intersections.

   RoadRunner supports the visualization of lidar point cloud, aerial imagery, and GIS data. You can import and export road networks using OpenDRIVE. 3D scenes built with RoadRunner can be exported in FBX, glTF™, OpenFlight, OpenSceneGraph, OBJ, and USD formats. The exported scenes can be used in automated driving simulators and game engines, including CARLA, Vires VTD, NVIDIA DRIVE Sim, Baidu Apollo, Cognata, Unity, and Unreal Engine.

2.1.2 Softwares Version

- **SUMO 1.2**
- **Unity 3D 2021.3.10.f1**
- **RoadRunner R2022a**
- **Python 3.8**

2.2 building of Unity 3D simulation scene

2.2.1 Mapping and integration of map

The map is drawn using the Roadrunner[1]. Export the drawn scene to the fbx model format for unity, and then place the model file in the unity assets folder. Finally, put the fbx file into the unity scene[2]. This map includes Garching, Garching hochbrueck and Garching TUM campuses. It is based on the high accuracy satellite generated and very similar to origin location. 2.1 shows a full view of the map.

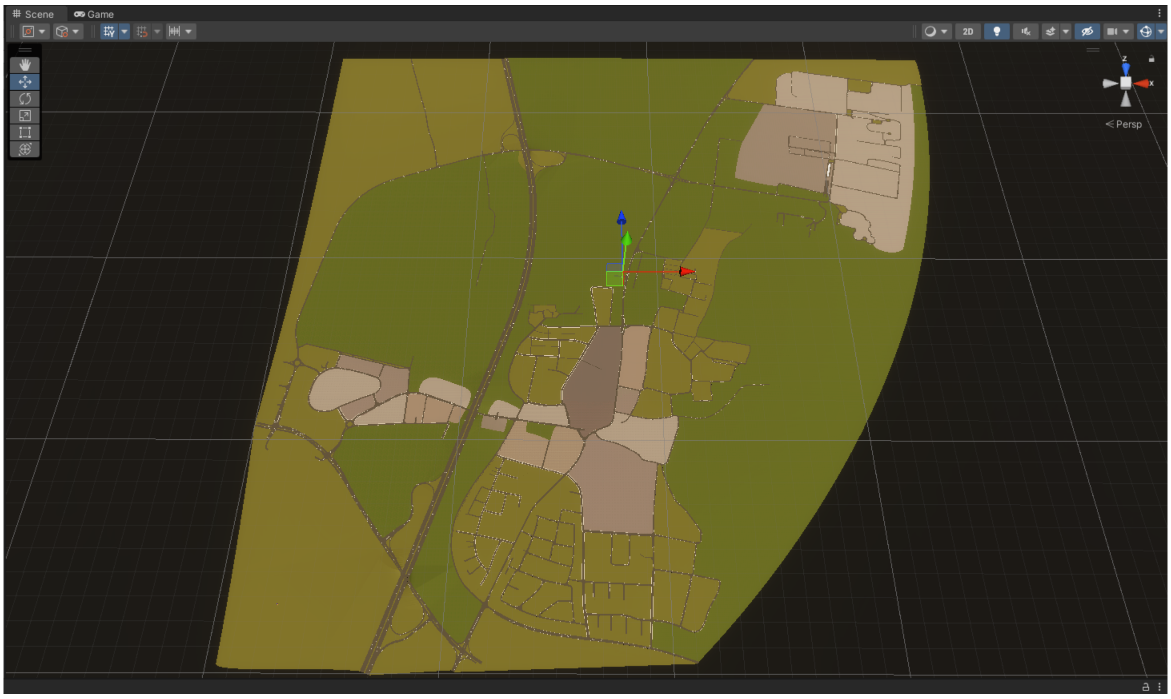


Figure 2.1: Map for simulation

Then, buildings and cityscapes are then placed into the scene. As shown in the 2.2, the simulation scenario is built.

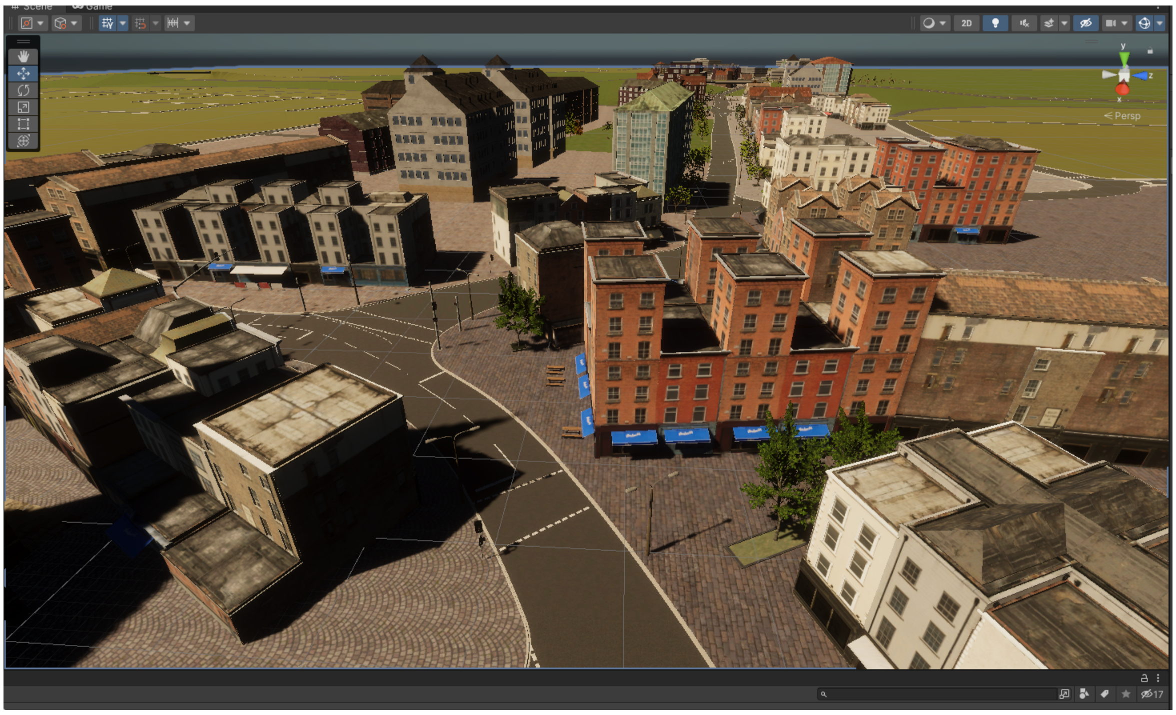


Figure 2.2: City scene for simulation

2.2.2 Add test vehicle model

The vehicle model in the "NWH Vehicle Physics2" package[3] is used as the test vehicle. NWH Vehicle Physics 2 is a complete vehicle simulation package for Unity. It simulates the complete vehicle characteristics and each module to realistically simulate the kinematic and dynamical characteristics of a real vehicle. The highly optimized code runs on both desktop and mobile devices. All vehicles combined, the desktop demo takes less than 0.5 milliseconds of total CPU time per frame. The "Playground" scene of NWH Vehicle Physics 2 pakeage is opened as shown in the 2.3.

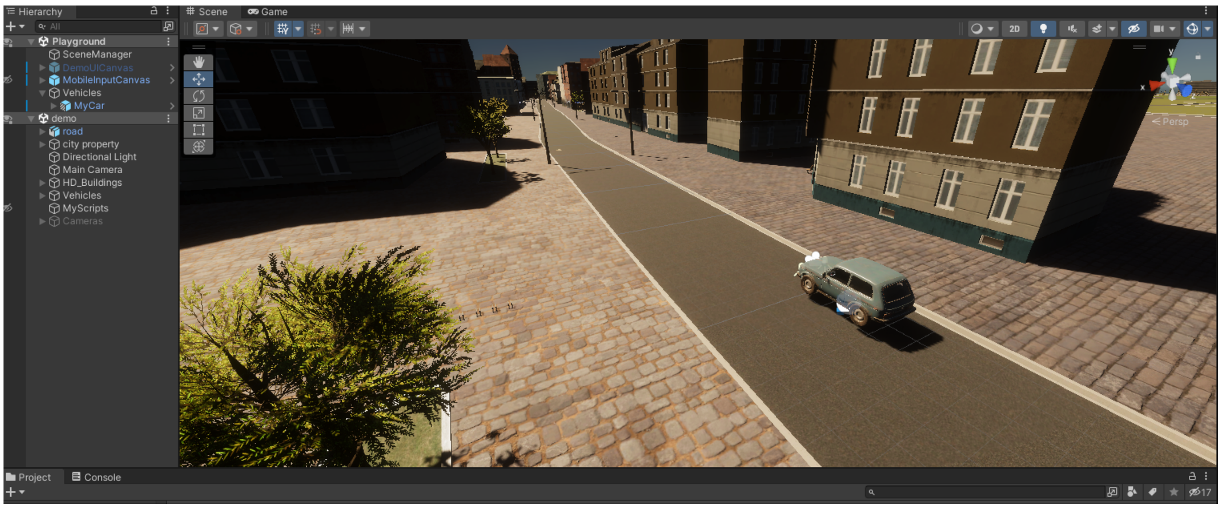


Figure 2.3: controlled vehicle

2.2.3 Preinstantiation of random vehicle models

Frequent generation and destruction of random vehicles are required in the simulation of traffic flow. If we use dynamic generation method, it will take up a lot of cpu resources. Instead of creating new vehicles and destroying old ones, all the required vehicle models are pre-instantiated before the simulation starts and the model state is set to inactivate, so that only the required vehicle models are activated or deactivated when the simulation runs. This can greatly improve performance. You can also create an Object pooling[4] with c# script to pre-instantiate the models.

2.3 Co-simulation based on Sumo and Unity 3D

2.3.1 Simulation in Sumo

Exporting the fbx map model in Roadrunner also exports a file in opendrive format, which is a popular standard file format for describing road network information. SUMO has a built-in netconvert command to generate road networks in SUMO format (.net.xml). Figure 2.4 shows the road network generated using the netconvert command based on the scenario in Figure 2.1.

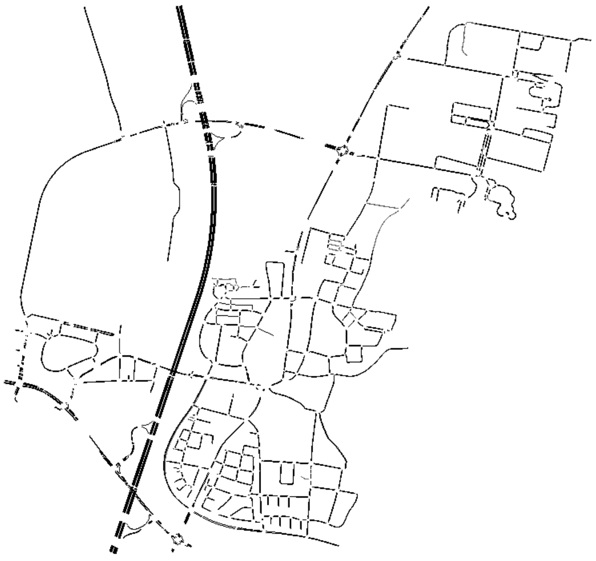


Figure 2.4: Road network in SUMO

The /randomtrips and /duarouter commands are then used to generate random trips and routes for all vehicles in the scenario. The /randomtrips and /duarouter commands are then used to generate random trips and routes for all vehicles within the scenario. The .sumocfg format file allows the integration of the road network file and the routing file and the simulation in SUMO-GUI. As shown in the figure 2.5, open the network.sumocfg file in SUMO-GUI and click the play button to simulate it.

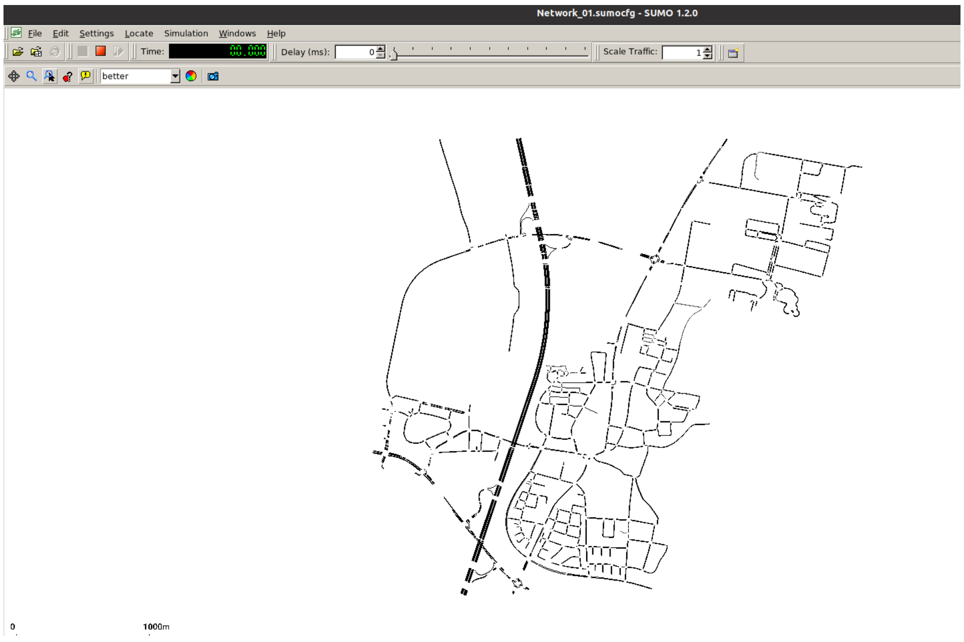
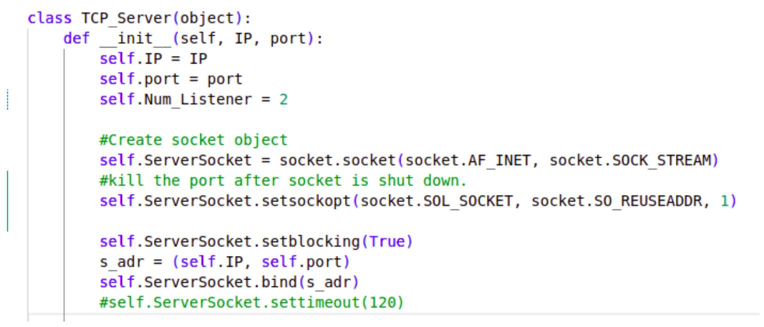

Network_01.sumocfg - SUMO 1.2.0
File Edit Settings Locate Simulation Windows Help
Time:
Delay (ms): 0
Scale Traffic: 1
better
0
1000m

Figure 2.5: Simulation in SUMO-GUI

2.3.2 connection between Sumo and Unity 3D

A secondary development based on the work of Tettamanti[5] and Horváth[6] was made. Communication is based on TCP/IP protocol. The detailed implementation is as follows.

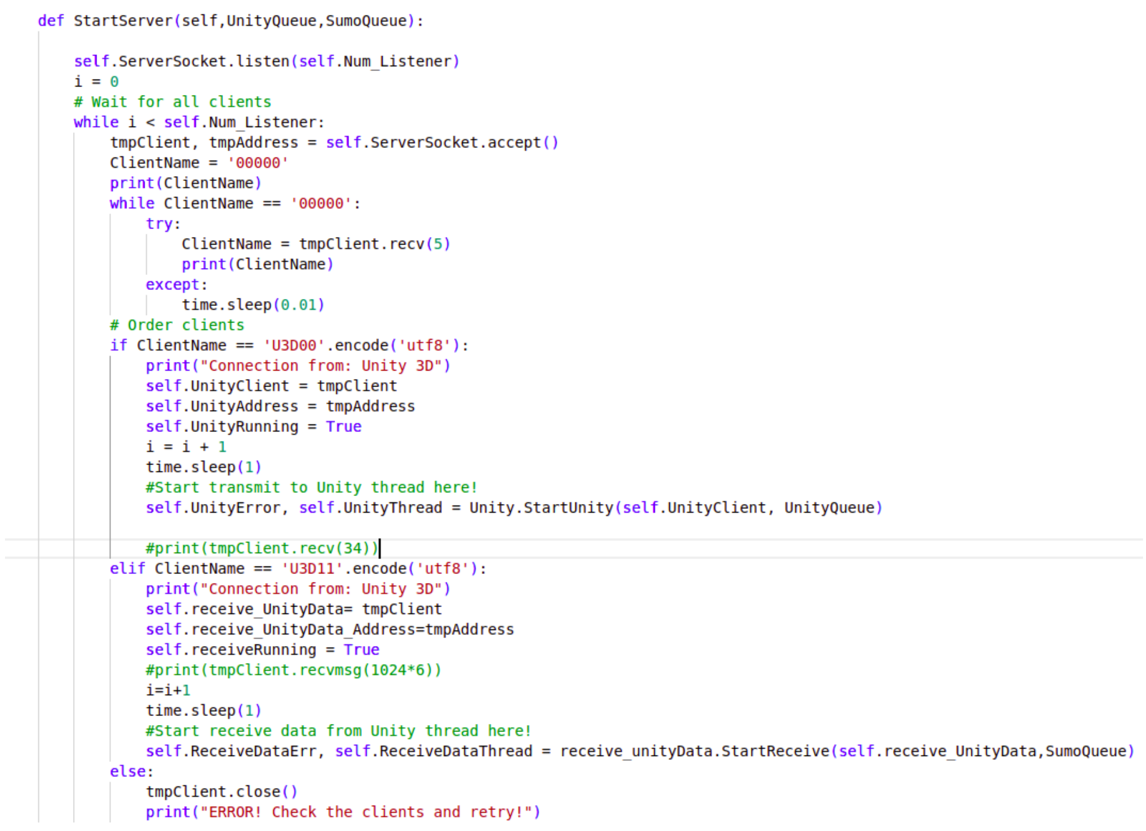
1. **TCP_server in Python**
   A new TCP_Server class is Created. The IP address, port, and the number of listening clients in the constructor are Initialized. And a TCP/IP socket is created. Then **bind()** is used to associate the socket with the server address.



```
class TCP_Server(object):
    def __init__(self, IP, port):
        self.IP = IP
        self.port = port
        self.Num_Listener = 2

        #Create socket object
        self.ServerSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        #kill the port after socket is shut down.
        self.ServerSocket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

        self.ServerSocket.setblocking(True)
        s_adr = (self.IP, self.port)
        self.ServerSocket.bind(s_adr)
        #self.ServerSocket.settimeout(120)
```

Figure 2.6: Initialization of TCP_server Class

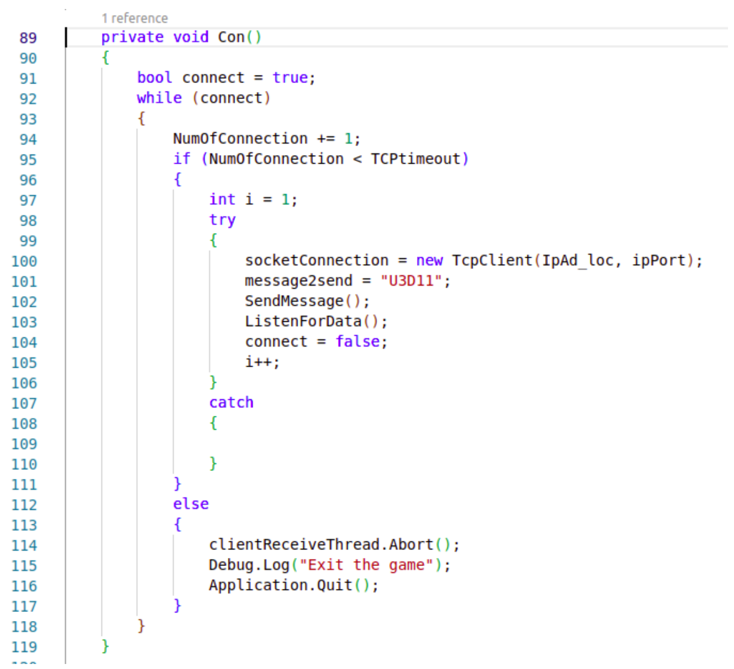
   Calling **listen()** puts the socket into server mode, and **accept()** waits for an incoming connection. **accept()** returns an open connection between the server and client, along with the address of the client. The connection is actually a different socket on another port (assigned by the kernel). Here we need to set up two clients in Unity 3D with the names 'U3D00' and 'U3D01'. One client is used to receive the status information of the vehicle in SUMO, and the other one delivers the information of the test vehicle.



```python
def StartServer(self,UnityQueue,SumoQueue):

    self.ServerSocket.listen(self.Num_Listener)
    i = 0
    # Wait for all clients
    while i < self.Num_Listener:
        tmpClient, tmpAddress = self.ServerSocket.accept()
        ClientName = '00000'
        print(ClientName)
        while ClientName == '00000':
            try:
                ClientName = tmpClient.recv(5)
                print(ClientName)
            except:
                time.sleep(0.01)
        # Order clients
        if ClientName == 'U3D00'.encode('utf8'):
            print("Connection from: Unity 3D")
            self.UnityClient = tmpClient
            self.UnityAddress = tmpAddress
            self.UnityRunning = True
            i = i + 1
            time.sleep(1)
            #Start transmit to Unity thread here!
            self.UnityError, self.UnityThread = Unity.StartUnity(self.UnityClient, UnityQueue)

            #print(tmpClient.recv(34))
        elif ClientName == 'U3D11'.encode('utf8'):
            print("Connection from: Unity 3D")
            self.receive_UnityData= tmpClient
            self.receive_UnityData_Address=tmpAddress
            self.receiveRunning = True
            #print(tmpClient.recvmsg(1024*6))
            i=i+1
            time.sleep(1)
            #Start receive data from Unity thread here!
            self.ReceiveDataErr, self.ReceiveDataThread = receive_unityData.StartReceive(self.receive_UnityData,SumoQueue)
        else:
            tmpClient.close()
            print("ERROR! Check the clients and retry!")
```

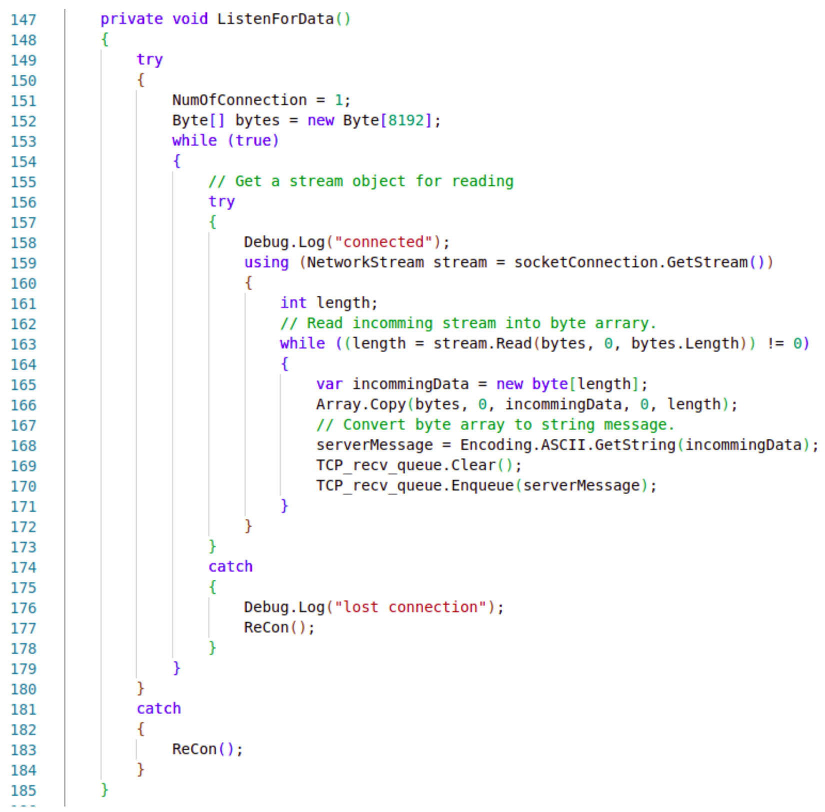
Figure 2.7: Start server

2. **TCP_Clients in Unity 3D**
   Call the **ConnectToTcpServer()** function in the **start()** initialization function. In **ConnectToTcpServer()**, create and enable a new process and call the **con()** function. By executing the **con()** function, you can create a client socket and send the client name "U3D00" to the TCP server in python to verify if the connection is successful. Note that the IP address and port of the server and client must be the same. **ListenForData()** function is used to receive messages from the server and store them in the enqueue. con() and **ListenForData()** functions are shown in Figure 2.8 and Figure 2.9.



```
1 reference
private void Con()
{
    bool connect = true;
    while (connect)
    {
        NumOfConnection += 1;
        if (NumOfConnection < TCPtimeout)
        {
            int i = 1;
            try
            {
                socketConnection = new TcpClient(IpAd_loc, ipPort);
                message2send = "U3D11";
                SendMessage();
                ListenForData();
                connect = false;
                i++;
            }
            catch
            {

            }
        }
        else
        {
            clientReceiveThread.Abort();
            Debug.Log("Exit the game");
            Application.Quit();
        }
    }
}
```

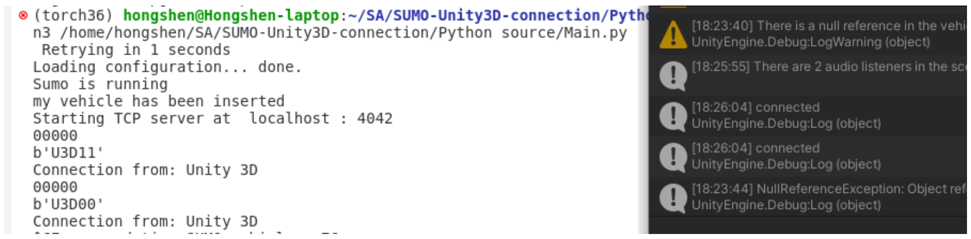
Figure 2.8: Implementation of connection to Server



```
private void ListenForData()
{
    try
    {
        NumOfConnection = 1;
        Byte[] bytes = new Byte[8192];
        while (true)
        {
            // Get a stream object for reading
            try
            {
                Debug.Log("connected");
                using (NetworkStream stream = socketConnection.GetStream())
                {
                    int length;
                    // Read incomming stream into byte array.
                    while ((length = stream.Read(bytes, 0, bytes.Length)) != 0)
                    {
                        var incommingData = new byte[length];
                        Array.Copy(bytes, 0, incommingData, 0, length);
                        // Convert byte array to string message.
                        serverMessage = Encoding.ASCII.GetString(incommingData);
                        TCP_recv_queue.Clear();
                        TCP_recv_queue.Enqueue(serverMessage);
                    }
                }
            }
            catch
            {
                Debug.Log("lost connection");
                ReCon();
            }
        }
    }
    catch
    {
        ReCon();
    }
}
```

Figure 2.9: Implementation of listening data

The figure 2.10 shows the result of successful authentication. For sending messages client "U3D01" can be connected to the server in the same way.



```
(torch36) hongshen@Hongshen-laptop:~/SA/SUMO-Unity3D-connection/Pyth
n3 /home/hongshen/SA/SUMO-Unity3D-connection/Python source/Main.py
 Retrying in 1 seconds
Loading configuration... done.
Sumo is running
my vehicle has been inserted
Starting TCP server at  localhost : 4042
00000
b'U3D11'
Connection from: Unity 3D
00000
b'U3D00'
Connection from: Unity 3D
```

```
[18:23:40] There is a null reference in the vehi
UnityEngine.Debug:LogWarning (object)
[18:25:55] There are 2 audio listeners in the sc
[18:26:04] connected
UnityEngine.Debug:Log (object)
[18:26:04] connected
UnityEngine.Debug:Log (object)
[18:23:44] NullReferenceException: Object ref
UnityEngine.Debug:Log (object)
```

Figure 2.10: the result of connection authentication

2.3.3 Data transfer between SUMO and Unity 3D

This subsection introduces the acquisition of real-time vehicle data in Sumo scenes, the transfer of data and the real-time mapping of Sumo to unity's vehicle motion state.

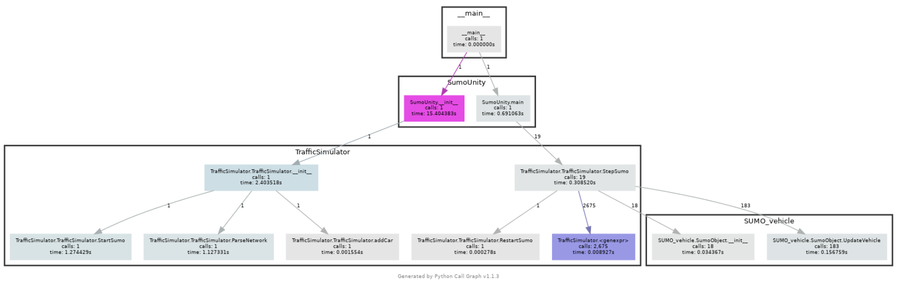
1. **Data transfer of SUMO simulation**

   a) **Retrieval of Vehicle motion states by using TraCi**

      TraCI is the short term for "Traffic Control Interface". Giving access to a running road traffic simulation, it allows to retrieve values of simulated objects and to manipulate their behavior "on-line".[7]

      i. **SUMO Startup**
         TraCI uses a TCP based client/server architecture to provide access to sumo [8]. Thereby, sumo acts as server that is started with additional command-line options: –remote-port <INT> where <INT> is the port sumo will listen on for incoming connections. Note that the ip address and port of the sumo startup function need to be the same as the socket created in the previous subsection. SUMO's official documentation describes the API usage of TraCi.

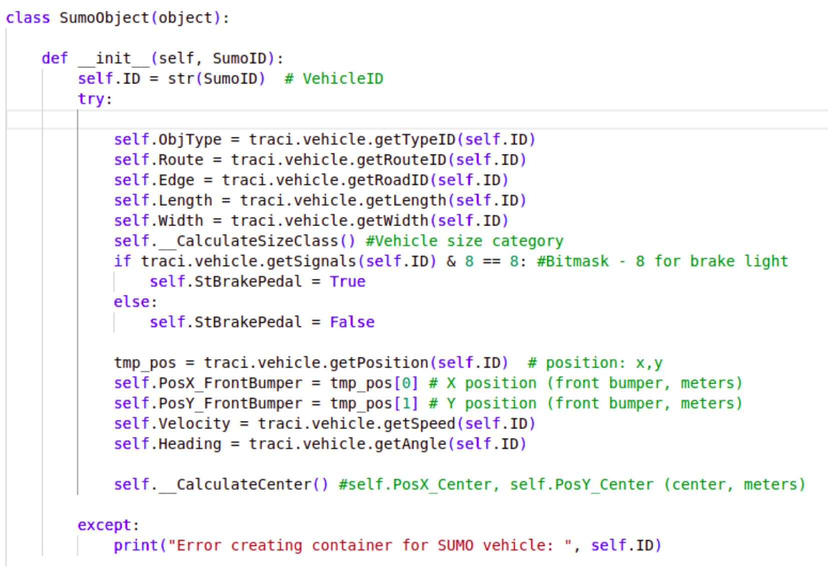
      ii. **Retrieval of Vehicle motion states**
         In Figure 2.13represent the Function callgraph which shows the process of acquisition of real-time vehicle data in Sumo scenes and initialzation of this Vehicles in Python.



__main__
__main__ calls: 1 time: 0.000000s
1
1
SumoUnity
SumoUnity.__init__ calls: 1 time: 15.404383s
SumoUnity.main calls: 1 time: 0.691063s
1
19
TrafficSimulator
TrafficSimulator.TrafficSimulator.__init__ calls: 1 time: 2.403518s
TrafficSimulator.TrafficSimulator.StepSumo calls: 19 time: 0.308520s
1
1
1
1
2675
18
183
TrafficSimulator.TrafficSimulator.StartSumo calls: 1 time: 1.274429s
TrafficSimulator.TrafficSimulator.ParseNetwork calls: 1 time: 1.127331s
TrafficSimulator.TrafficSimulator.addCar calls: 1 time: 0.001554s
TrafficSimulator.TrafficSimulator.RestartSumo calls: 1 time: 0.000278s
TrafficSimulator.<genexpr> calls: 2,675 time: 0.008927s
SUMO_vehicle
SUMO_vehicle.SumoObject.__init__ calls: 18 time: 0.034367s
SUMO_vehicle.SumoObject.UpdateVehicle calls: 183 time: 0.156759s
Generated by Python Call Graph v1.1.3

Figure 2.11: Call graph of Retrieval of Vehicle motion states

The SumoObject class is created in the **SUMO_vehicle** module. The parameters of the vehicle in the constructor are initialized by using TraCi's API. For example, **traci.vehicle.getPostion()** is used to get the vehicle position, **traci.vehicle.getAngle()** is used to get the vehicle head angle i.e. attitude, etc. The following figure 2.12 shows the parameters in the constructor.



```
class SumoObject(object):

    def __init__(self, SumoID):
        self.ID = str(SumoID)  # VehicleID
        try:

            self.ObjType = traci.vehicle.getTypeID(self.ID)
            self.Route = traci.vehicle.getRouteID(self.ID)
            self.Edge = traci.vehicle.getRoadID(self.ID)
            self.Length = traci.vehicle.getLength(self.ID)
            self.Width = traci.vehicle.getWidth(self.ID)
            self.__CalculateSizeClass() #Vehicle size category
            if traci.vehicle.getSignals(self.ID) & 8 == 8: #Bitmask - 8 for brake light
                self.StBrakePedal = True
            else:
                self.StBrakePedal = False

            tmp_pos = traci.vehicle.getPosition(self.ID)  # position: x,y
            self.PosX_FrontBumper = tmp_pos[0] # X position (front bumper, meters)
            self.PosY_FrontBumper = tmp_pos[1] # Y position (front bumper, meters)
            self.Velocity = traci.vehicle.getSpeed(self.ID)
            self.Heading = traci.vehicle.getAngle(self.ID)

            self.__CalculateCenter() #self.PosX_Center, self.PosY_Center (center, meters)

        except:
            print("Error creating container for SUMO vehicle: ", self.ID)
```

Figure 2.12: Construction of class SumoObject

The **stepsumo()** function in the **TrafficSimulator** module implements the update of vehicles in sumo. The Sumobject class of SUMO_vehicle is called and instantiated as shown in the figure 2.13. When a new vehicle is created in SUMO or a vehicle reaches its destination and is destroyed, an instance of SumoObject is created or destroyed with it. The existing instances are stored in the **SumoObjects** list. The vehicle data is updated once at each simulation step by calling **UpdateVehicle()**. A new vehicle **"MyVehicle"** in SUMO as a mapping of the test vehicle in unity is insert by calling the **AddCar** API of TraCi.

b) **Transfer Data between SUMO and Unity 3D**
This subsection describes how to implement data transfer based on the TCP communication framework established in the previous subsection **1.3.2**.

i. **send Data to Unity**
In Figure 2.13represent the Function callgraph which shows the process of acquisition of sending the real-time vehicle datas to Unity 3D.

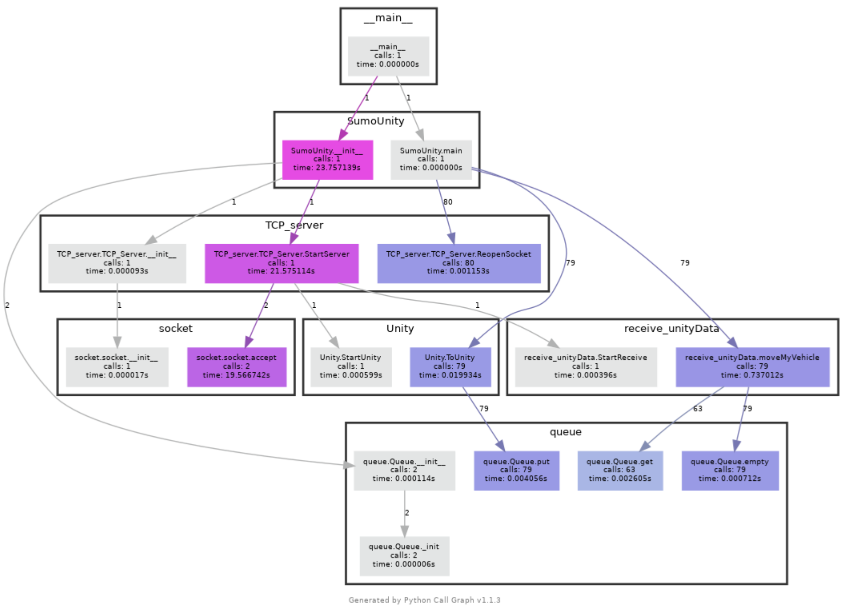
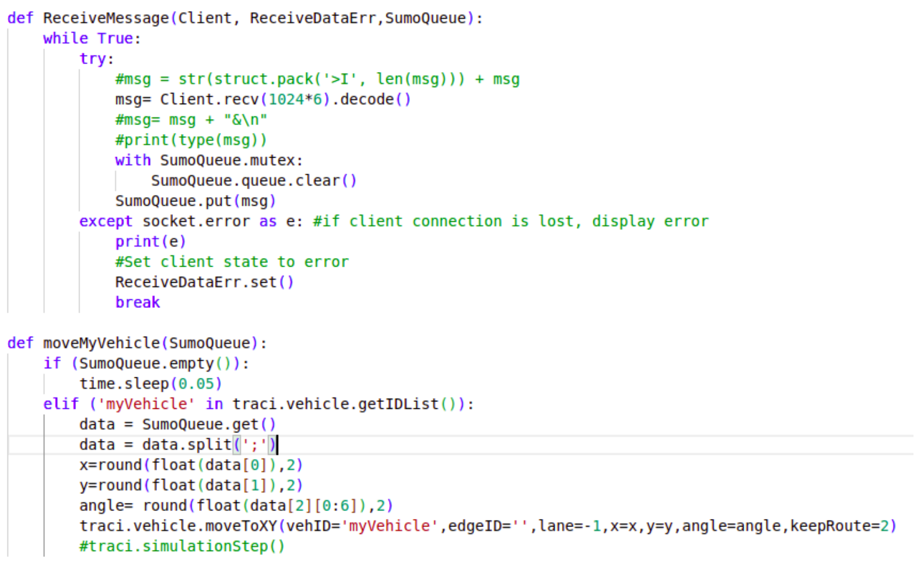

__main__
__main__ calls: 1 time: 0.000000s
1
1
SumoUnity
SumoUnity.__init__ calls: 1 time: 23.757139s
SumoUnity.main calls: 1 time: 0.000000s
1
1
80
79
79
2
TCP_server
TCP_server.TCP_Server.__init__ calls: 1 time: 0.000093s
TCP_server.TCP_Server.StartServer calls: 1 time: 21.575114s
TCP_server.TCP_Server.ReopenSocket calls: 80 time: 0.001153s
1
2
1
1
socket
socket.socket.__init__ calls: 1 time: 0.000017s
socket.socket.accept calls: 2 time: 19.566742s
Unity
Unity.StartUnity calls: 1 time: 0.000599s
Unity.ToUnity calls: 79 time: 0.019934s
receive_unityData
receive_unityData.StartReceive calls: 1 time: 0.000396s
receive_unityData.moveMyVehicle calls: 79 time: 0.737012s
79
63
79
queue
queue.Queue.__init__ calls: 2 time: 0.000114s
queue.Queue.put calls: 79 time: 0.004056s
queue.Queue.get calls: 63 time: 0.002605s
queue.Queue.empty calls: 79 time: 0.000712s
2
queue.Queue._init calls: 2 time: 0.000006s
Generated by Python Call Graph v1.1.3

Figure 2.13: Call graph of Sending and receiving message to Unity3D

The attributes of all vehicle instances into a string message are packed to put into the UnityQueue by calling the **ToUnity()** function. Open a new thread in **startUnity()** to continuously call **sendmessage()** to send messages from the UnityQueue to the unity client. The messages are sent once per simulationstep.

ii. **Receive data from Unity 3D**
Open a new thread in**startReceive()** to continuously call **ReceiveMessage()** to receive messages from the unity client and put the decoded data to the SUMO-Quene. Call **moveMyVehicle()** to get the data in SUMOQueue and update the pose of the test vehicle in SUMO scene by API **traci.vehicle.moveToXY()**.The specific call relationship is shown in the figure 2.13.The figure 2.14 shows the concrete implementation of the functions **ReceiveMessage()** and **moveMyVehicle()**.

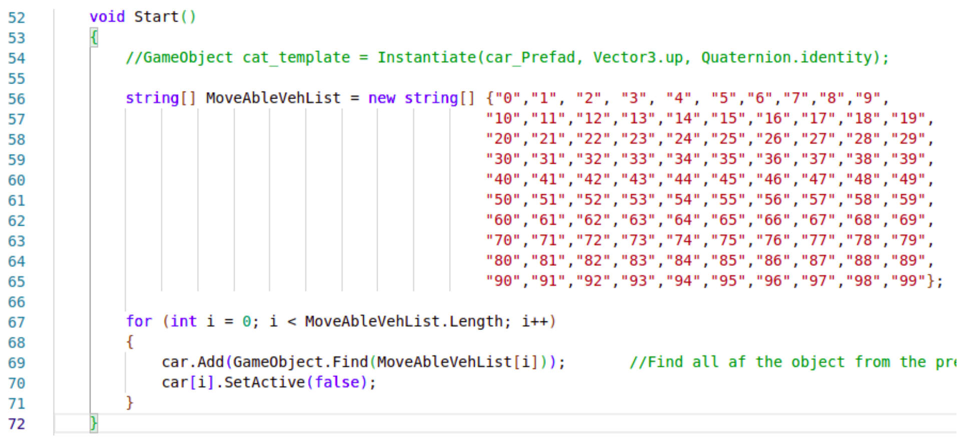


```python
def ReceiveMessage(Client, ReceiveDataErr,SumoQueue):
    while True:
        try:
            #msg = str(struct.pack('>I', len(msg))) + msg
            msg= Client.recv(1024*6).decode()
            #msg= msg + "&\n"
            #print(type(msg))
            with SumoQueue.mutex:
                SumoQueue.queue.clear()
            SumoQueue.put(msg)
        except socket.error as e: #if client connection is lost, display error
            print(e)
            #Set client state to error
            ReceiveDataErr.set()
            break

def moveMyVehicle(SumoQueue):
    if (SumoQueue.empty()):
        time.sleep(0.05)
    elif ('myVehicle' in traci.vehicle.getIDList()):
        data = SumoQueue.get()
        data = data.split(';')
        x=round(float(data[0]),2)
        y=round(float(data[1]),2)
        angle= round(float(data[2][0:6]),2)
        traci.vehicle.moveToXY(vehID='myVehicle',edgeID='',lane=-1,x=x,y=y,angle=angle,keepRoute=2)
        #traci.simulationStep()
```

Figure 2.14: Code of Functions **ReceiveMessage()** and **moveMyVehicle()**

2. **Data transfer of Unity 3D**
   The code framework of Unity 3D part follows the work of Tettamanti and Horváth, and rewrites some of the functions.

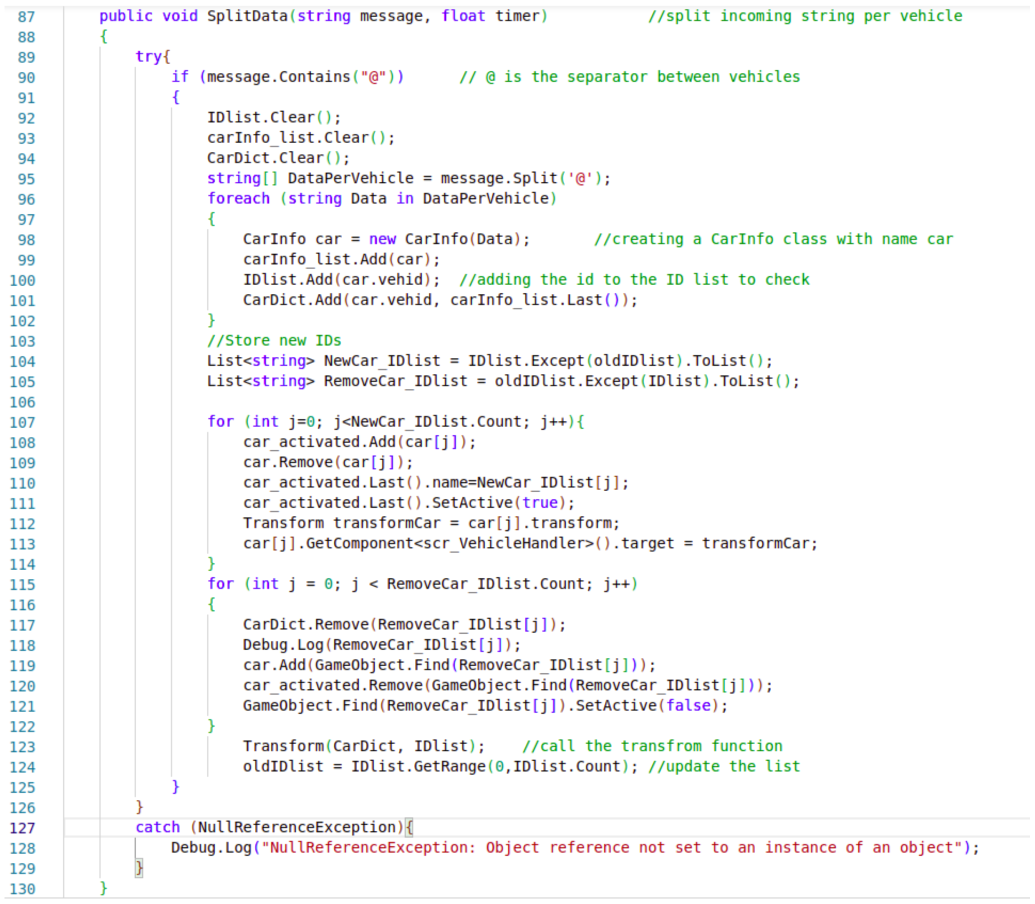
   a) **Receive Data from Sumo Motion Mapping of vehicles**
      The Figure 2.15 **start()** in the Main.cs store the external vehicle objects preloaded in the Unity scene to the **car** List and deactivate all vehicle Objects.



```csharp
void Start()
{
    //GameObject cat_template = Instantiate(car_Prefad, Vector3.up, Quaternion.identity);

    string[] MoveAbleVehList = new string[] {"0","1", "2", "3", "4", "5","6","7","8","9",
                                             "10","11","12","13","14","15","16","17","18","19",
                                             "20","21","22","23","24","25","26","27","28","29",
                                             "30","31","32","33","34","35","36","37","38","39",
                                             "40","41","42","43","44","45","46","47","48","49",
                                             "50","51","52","53","54","55","56","57","58","59",
                                             "60","61","62","63","64","65","66","67","68","69",
                                             "70","71","72","73","74","75","76","77","78","79",
                                             "80","81","82","83","84","85","86","87","88","89",
                                             "90","91","92","93","94","95","96","97","98","99"};

    for (int i = 0; i < MoveAbleVehList.Length; i++)
    {
        car.Add(GameObject.Find(MoveAbleVehList[i]));        //Find all af the object from the pr
        car[i].SetActive(false);
    }
}
```

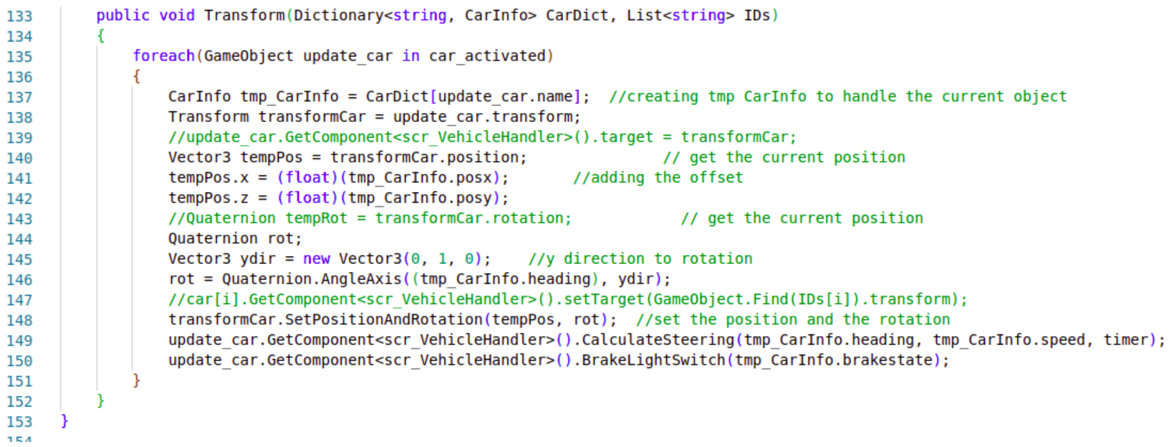
Figure 2.15: Code of **start** function

The **SplitData()** and **Transform()** functions are overridden here. **SplitData()** implements updating the vehicle status information based on the data received from the TCP_server, activating and deactivating the preloaded vehicle, and calling the **Transform()** function to update the vehicles status. The figure 2.16 and 2.17 shows the code details.



```
    public void SplitData(string message, float timer)          //split incoming string per vehicle
    {
        try{
            if (message.Contains("@"))      // @ is the separator between vehicles
            {
                IDlist.Clear();
                carInfo_list.Clear();
                CarDict.Clear();
                string[] DataPerVehicle = message.Split('@');
                foreach (string Data in DataPerVehicle)
                {
                    CarInfo car = new CarInfo(Data);       //creating a CarInfo class with name car
                    carInfo_list.Add(car);
                    IDlist.Add(car.vehid);  //adding the id to the ID list to check
                    CarDict.Add(car.vehid, carInfo_list.Last());
                }
                //Store new IDs
                List<string> NewCar_IDlist = IDlist.Except(oldIDlist).ToList();
                List<string> RemoveCar_IDlist = oldIDlist.Except(IDlist).ToList();

                for (int j=0; j<NewCar_IDlist.Count; j++){
                    car_activated.Add(car[j]);
                    car.Remove(car[j]);
                    car_activated.Last().name=NewCar_IDlist[j];
                    car_activated.Last().SetActive(true);
                    Transform transformCar = car[j].transform;
                    car[j].GetComponent<scr_VehicleHandler>().target = transformCar;
                }
                for (int j = 0; j < RemoveCar_IDlist.Count; j++)
                {
                    CarDict.Remove(RemoveCar_IDlist[j]);
                    Debug.Log(RemoveCar_IDlist[j]);
                    car.Add(GameObject.Find(RemoveCar_IDlist[j]));
                    car_activated.Remove(GameObject.Find(RemoveCar_IDlist[j]));
                    GameObject.Find(RemoveCar_IDlist[j]).SetActive(false);
                }
                    Transform(CarDict, IDlist);    //call the transfrom function
                    oldIDlist = IDlist.GetRange(0,IDlist.Count); //update the list
            }
        }
        catch (NullReferenceException){
            Debug.Log("NullReferenceException: Object reference not set to an instance of an object");
        }
    }
```

Figure 2.16: Code of **SplitData()** function

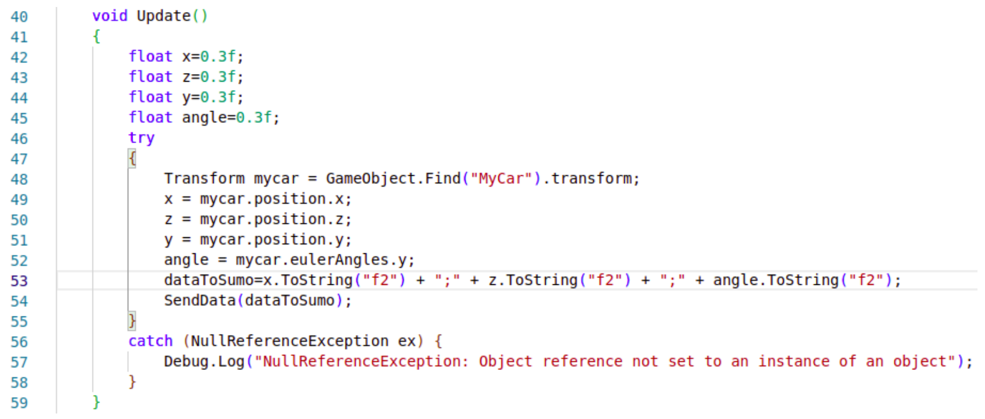


```
    public void Transform(Dictionary<string, CarInfo> CarDict, List<string> IDs)
    {
        foreach(GameObject update_car in car_activated)
        {
            CarInfo tmp_CarInfo = CarDict[update_car.name];  //creating tmp CarInfo to handle the current object
            Transform transformCar = update_car.transform;
            //update_car.GetComponent<scr_VehicleHandler>().target = transformCar;
            Vector3 tempPos = transformCar.position;                 // get the current position
            tempPos.x = (float)(tmp_CarInfo.posx);        //adding the offset
            tempPos.z = (float)(tmp_CarInfo.posy);
            //Quaternion tempRot = transformCar.rotation;              // get the current position
            Quaternion rot;
            Vector3 ydir = new Vector3(0, 1, 0);    //y direction to rotation
            rot = Quaternion.AngleAxis((tmp_CarInfo.heading), ydir);
            //car[i].GetComponent<scr_VehicleHandler>().setTarget(GameObject.Find(IDs[i]).transform);
            transformCar.SetPositionAndRotation(tempPos, rot);  //set the position and the rotation
            update_car.GetComponent<scr_VehicleHandler>().CalculateSteering(tmp_CarInfo.heading, tmp_CarInfo.speed, timer);
            update_car.GetComponent<scr_VehicleHandler>().BrakeLightSwitch(tmp_CarInfo.brakestate);
        }
    }
}
```

Figure 2.17: Code of **Transform()** function

The **transform()** calls the gamecomponent scr_VehicleHandler script of the vehicle object to calculate the wheel speed and implement the animation effect.

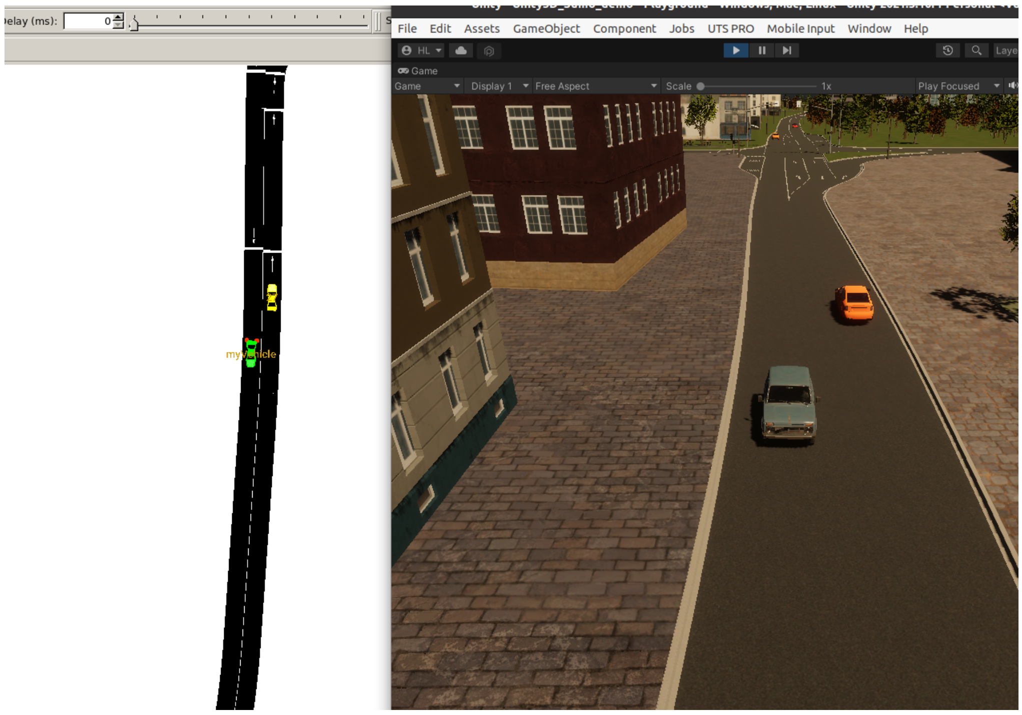
b) **Send Data of the test vehicle**
Create a new unity c# script **scr_DatatoSumo.cs**. Call the unity API **GameObject.Find().Transform** in the **update** function to get the pose information of the test vehicle. Pack the pose information into string format and send it to TCP_server with **SendData()**. synchronize the movement of the test vehicle **"MyVehicle"** in SUMO with the in **1(b)ii** executed **MoveMyVehicle()** function. See the official documentation for how to use the unity API[9].



```
void Update()
{
    float x=0.3f;
    float z=0.3f;
    float y=0.3f;
    float angle=0.3f;
    try
    {
        Transform mycar = GameObject.Find("MyCar").transform;
        x = mycar.position.x;
        z = mycar.position.z;
        y = mycar.position.y;
        angle = mycar.eulerAngles.y;
        dataToSumo=x.ToString("f2") + ";" + z.ToString("f2") + ";" + angle.ToString("f2");
        SendData(dataToSumo);
    }
    catch (NullReferenceException ex) {
        Debug.Log("NullReferenceException: Object reference not set to an instance of an object");
    }
}
```

Figure 2.18: Code of **update()** function

The following figure 2.19 shows the real-time synchronization of the test vehicle in SUMO and Unity 3D.



elay (ms):
myVehicle
File Edit Assets GameObject Component Jobs UTS PRO Mobile Input Window Help
Game
Display 1
Free Aspect
Scale
1x
Play Focused

Figure 2.19: synchronization of the test vehicle in SUMO and Unity 3D

2.4 To do in the future

So far, we have completed the construction of unity 3D HD scene demo, the connection between SUMO and Unity based on TCP/IP, and the real-time synchronization of vehicle movement. We still need to add further features and improve the code in the future. The works that needs to be done are as follows.

1. **Solve the Unity simulation lag**
   The problem of vehicle motion lagging occurs during the simulation. After analysis, it is due to some code running inefficiently and taking up too much computing resources. The code needs to be further improved and modified.

2. **Automatic building of scenes.**
   Automatically generate high precision maps based on real scenes using the Here high precision map API in Roadrunner. Create generator for unity scenes. Generate city scenes such as buildings, landscapes, weather, etc. by importing 3D high precision maps.

3. **Simulation of traffic light system and pedestrians system**
   Design traffic light timings based on real traffic flow conditions and simulate them in SUMO and Unity in real time. Design the pedestrian system in SUMO and synchronize it in Unity.

4. **Integration of automated driving system**
   Implement a modular interface for algorithms based on the ROS2 framework for perception, decision making, and planning.

5. **GUI design.**
   Encapsulates the use of SUMO and Unity. Users can simply start the simulation and adjust parameters to validate their algorithms.

List of Figures

Bibliography

[1] *https://de.mathworks.com/help/roadrunner/fundamentals.html*.

[2] *https://de.mathworks.com/help/roadrunner/export-scenes.html*.

[3] *https://assetstore.unity.com/packages/tools/physics/nwh-vehicle-physics-2-166252reviews*.

[4] *https://learn.unity.com/tutorial/introduction-to-object-pooling?uv=2019.4*.

[5] T. Tettamanti. *Vehicle-In-the-Loop Test Environment for Autonomous Driving with Microscopic Traffic Simulation*. IEEE International Conference on Vehicular Electronics and Safety (ICVES), 2018.

[6] M. T. Horváth. *Vehicle-In-The-Loop (VIL) and Scenario-In-The-Loop (SCIL) Automotive Simulation Concepts from the Perspectives of Traffic Simulation and Traffic Control*. Transport and Telecommunication Journal, 2019.

[7] *TraCi Reference: https://sumo.dlr.de/docs/TraCI.html*.

[8] *https://sumo.dlr.de/docs/TraCI/Interfacing$_T$raCI$_f$rom$_P$ython.html*.

[9] *Unity Scripting Reference: https://docs.unity3d.com/ScriptReference/*.