Technische Universität München

IDP Final Report

INSTITUTE FOR HUMAN-MACHINE COMMUNICATION
TECHNISCHE UNIVERSITÄT MÜNCHEN
Univ.-Prof. Dr.-Ing. habil. G. Rigoll

Semantic Road Network Description of 3D Urban Environments for the MMK Driving Simulator

Zhechko Zhechev

Advisor:	Patrick Lindemann, M.Sc.
Started on:	01.04.2016
Handed in on:	23.11.2017

Contents

1

Introduction

Augmented reality has not only changed the perception of the world for the ordinary user but it has also allowed researchers to conduct studies which have not been possible before. The chair of Human-Machine Communication is currently developing a mixed reality driving simulator using Unity Engine in a 3D CAVE. It can be utilised to facilitate studies regarding reaction of the driver in different situations on the road and development of better Head-Up Displays.

Currently, the 3D models which simulate the street network are generated using ESRI CityEngine and unfortunately, they are not based on real data. Moreover, a semantic description of the underlaying street network, as well as any properties of its segment, are not present. Street information based on real data exported from open source projects such as OpenStreetMap would improve the quality of the generated models. This would allow the simulation of real scenes and furthermore facilitate the research of the human-machine interaction. Additionally, a semantic description of the street network would allow the simulation of autonomous vehicles and fully simulation of traffic management system.

1.1 Goals of the Project

The main goal of this Interdisciplinary Project is to develop an approach to derive a semantic description of a road network, generated using OpenStreetMap data in CityEngine and SUMO. Moreover, the generated 3D scene of a city model has to be synchronised with the aforementioned description. Additionally, the export file has to contain all details about the lanes of the corresponding road segments and their properties, as well as the coordinates of buildings and parkings which can be found in the generated scene. The export must be accomplished in a widely compatible format which has proven to be suitable for these purposes, as well. Finally, the road network description has to be imported in the MMK Driving Simulator in order to build a navigation system for the traffic participants using connectivity information generated by SUMO.

1.2 Outline

The remainder of this report is organised as follows: Chapter 2 provides background information about the essential tools which are employed to improve the current functionalities of the MMK Driving Simulator. Later, in Chapter 3 we describe thoroughly our approach to derive a semantic description of the road system generated in CityEngine from OpenStreetMap data. Moreover, we discuss how additional road information can be acquired from SUMO simulation files about the same road system. Afterwards, Chapter 4 presents how we can furthermore utilise the SUMO-generated road connectivity information to construct a navigation system. Finally, Chapter 5 makes a summary of the achieved goals and discusses possibilities for future improvements of the project.

2

Background and Related Work

In the following chapter, we introduce all significant entities and tools adopted by the MMK Driving Simulator. Moreover, we discuss other research which is similar to the currently presented project. The simulator itself is implemented in Unity [1], while all of the 3D models and textures are generated by ESRI CityEngine [2]. Additionally, the map information is exported from OpenStreetMap [3] and the traffic simulation is accomplished using SUMO Engine [4]. We discuss essential characteristics of these tools, which are later utilised to achieve the goals described in Chapter 1.

2.1 Related Work

Much research about semantic description of urban street network has already been conducted in the recent years. Haubrich *et al.* have developed a workflow for rapid 3D traffic scenario generation [5]. They use OpenStreetMap (OSM) data and generate the necessary 3D shapes and road description in OpenDRIVE [6] format using *Trian3D Builder*. Both, the shapes and the description are in the end imported into Unity. Their aim was the creation of a road network that can be used for traffic simulation, as part of the *AVeSi* project (*Agentenbasierte Verkehrssimulation*). Shi has also implemented a tool in his master thesis [7] which converts OSM data in OpenDRIVE format. However, the tool is not integrated with a 3D building environment and chooses to interpolate the OSM data using standard techniques. Therefore, the parameters of the generated street attributes may not coincide with separately generated 3D shapes of the whole scene. Lastly, in 2010 Hiblot *et al.* presented ROADS [8], a procedural modelling application for road networks. ROADS offers the possibility to draw roads on a 2D map and export their semantic description.

2.2 OpenStreetMap

OpenStreetMap (OSM) is a open source project that creates and distributes free geographic data for the world. The project was created by the OpenStreetMap Foundation and it is completely supported by volunteers. OSM represents physical features on the

ground (*e.g.*, roads, buildings) using tags attached to its basic data structures: `nodes`, `ways`, and `relations`. Each tag describes a geographic attribute of the feature being shown by that specific OSM object. Among other attributes, each basic data structure of OSM has a `64 bit` integer identification number (OSM ID), which allows every object to be uniquely identifiable.

A `node` represents a specific point on the Earth's surface defined by its coordinates in the WGS-84 coordinate system (latitude and longitude). They can be used to define standalone point features *e.g.* park bench or a water well or they could outline the shape of a way. A `way` is an ordered list of between 2 and 2000 nodes that define a polyline. Ways are used to represent linear features such as rivers and roads, as well as boundaries of areas such as buildings. Finally, a `relation` is a multi-purpose data structure that documents a relationship between two or more data elements, *e.g.* route relation.

There exists a possibility to export a certain area of the map in a `xml` format which can be easily parsed or visualised with different tools, *e.g. JOSM*[1].

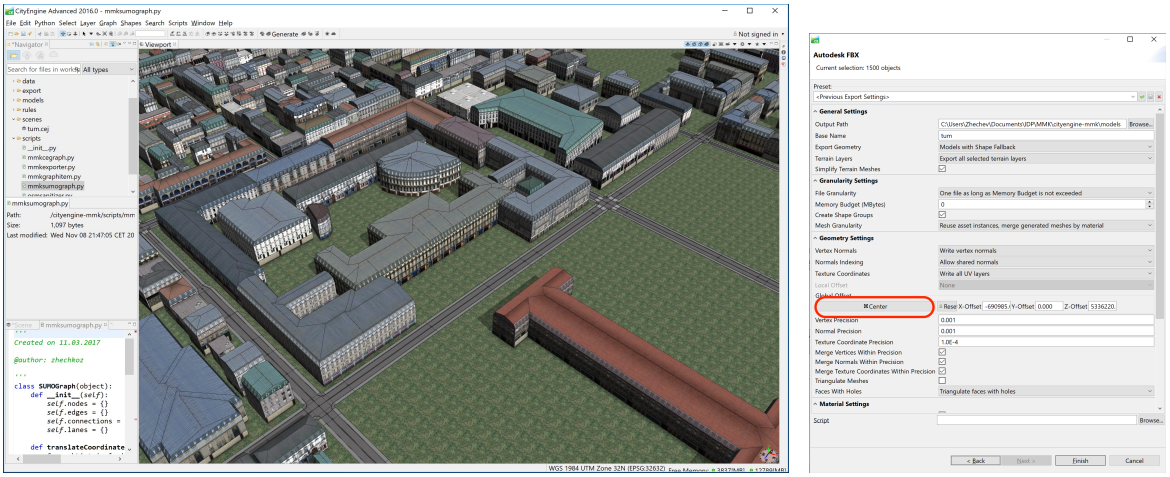
2.3 ESRI City Engine

ESRI CityEngine (CE) is a three-dimensional modelling software application developed by *Esri R&D Center Zurich* and is specialised in the generation of 3D urban environments (Figure 2.1(a)). There are multiple possibilities to create a city model with CE. For instance, one could import a third-party (2D) map data, such as OSM described in the previous section or 3D models in `.shp` or `.fbx` format. Afterwards, the application will try to generate 3D models using the 2D data and additional information (*e.g.* defined tags for every OSM basic data structure) and specified by the user rules. Finally, the resulting shapes can be exported to commonly used formats such as `obj` or `fbx`.

The coordinate system used to represent the location of all objects in the scene is a `y`-up left-handed 3D cartesian coordinate system [9]. In the case of map data imported from OSM, the position of every object in the city model is determined by their own WGS-84 (`(a, b, 0)`) coordinates projected in the Universal Transverse Mercator (UTM) `z`-up coordinate system (`(a, b, 0)`). However, the WGS-84 coordinates are just 2D values, therefore, CE utilises many attributes, defined in the OSM export, in order to interpolate a third dimensional position of each object (`(a, b, c)`). But this is still a `z`-up coordinate system. In order to convert all coordinates to a `y`-up coordinate system, CE applies a rotation by `-90`° and as a result we have `(a, c, -b)`. If one would like to find locations of OSM objects in CE, they have to use a third-party library for WGS-84/UTM conversion (*e.g.* `utm` [10]), as well as follow the aforementioned process. Nevertheless, the `y` cannot be generated correctly because of internal CE routines which are not widely accessible. When exporting 3D models from CE to one of the compatible formats, one could centre the whole scene automatically (as can be seen in Figure 2.1(b)), so the origin of all models is located at the point `(0, 0, 0)`.

The generated street network model is composed of `segments` and `nodes`. Each node is defined by one 3D point in the scene, while each segment composes of exactly

[1] `https://josm.openstreetmap.de`



(a) (b)

Figure 2.1: Finished CityEngine scene on the left (a) and the export window on the right(b), where with red is marked the *Center* button.

two nodes which define its form. Both, nodes and segments, have unique id values (`OID`) and can hold shapes as children objects, which are referenced with the same id as their parent segment or node and an index (`OID:INDEX`). Maximilian Murauer has already implemented a Python script in his research project at the Institute of Human-Machine Interaction, TUM which exports the properties of all nodes, segments and their corresponding shapes in a JSON format. However, this was never imported in the MMK Driving Simulator.

CE includes a `Jython` interface which can be utilised to access and change scene's objects properties. CE 2016 uses `Jython 2.7b21`, which already has the most common Python modules, however there are some necessary software which has to be manually included to the `Jython` environment. In order to use the export scripts which will be described in the next chapter, one have to download and copy `simplejson` installation folder to `C:\Users\<USERNAME>\.CityEngine\2016.0R.win32.win32.x86_64\jythonCache\ThirdParty`.

2.4 Simulation of Urban MObility (SUMO)

SUMO is a open-source (licensed under the GPL) traffic simulation software which was developed by the National Aeronautics and Space Research Center of the Federal Republic of Germany in Berlin. It is highly portable, microscopic (*i.e.* each vehicle is controlled separately) and continuous (*i.e.* runs in predefined steps) road traffic simulation package designed to handle large road networks. SUMO's traffic simulation runs in separate steps where each vehicle, by following its *lane*, is moved to the next destination. However, some of these actions, such as lane switching, are accomplished not in a smooth fashion. Before the simulation begins, one has to define the geometry

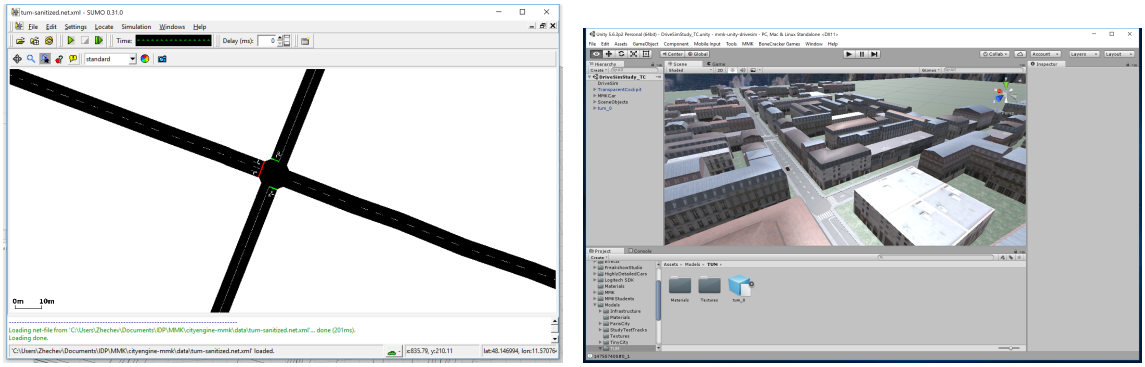
and the properties of the street network, as well as each vehicles's characteristics. This is realised by generating a couple of `xml`-based configuration files. In the following section, we will describe some of the most relevant to this project fields from `net.xml` configuration file.

Firstly, five main objects are used in `net.xml` to describes the whole street network: `location`, `junction`, `edge`, `lane`, `connection`. This configuration file can be generated using `netconvert` tool which is available in the SUMO package from many input formats, *e.g.* OSM Export Format, OpenDRIVE. Additionally, the final `net.xml` can be illustrated using SUMO's GUI tool (Figure 2.2(a)). There are many options which can be passed to `netconvert` either by command line or as a configuration file. Using these options, one could customise the street network generation process. To generate the necessary `net.xml` file in this project we used two optional parameters which were passed to `netconvert`: `output.street-names true` and `output.original-names true`, which preserve the original OSM information. SUMO adopts again a new custom coordinate system which uses translation and projection to convert the original objects' coordinates. This is done in such a way that after the transformation the left, top corner of the scene has the coordinates `(0, 0)`. The applied transformation and projection parameters can be found in the `location` tag. CE and SUMO usually use the same parameters for the projection but they differ in their choice of translation values for each object in the scene.

The most substantial object in the `net.xml` file for this project is `edge`. It holds an information about the track between two nodes, which is equivalent to a `segment` in CE. There are generally two types of `edges`: *internal* and *ordinary*. The latter specify the connection between two junctions. One of the important attributes of this kind of `edge` is `id`, which is a unique identification of a street. If the source of the map data is OSM, then this `id` has the form `OSM-ID#INDEX`, where `INDEX` is necessary since it is possible that one `way` in OSM to correspond to more than one `edge` in SUMO. Next, `edge` has `from` and `to` attributes which specify its start and end node. Lastly, the *internal* type of `edge` define a node. In this case the `id` has the form `:OSM-ID#INDEX` and *function* is marked as *internal*. Both `edge`-types posses *lanes* as child nodes.

As already mentioned `edge` contains a list of lanes, which specify the actual vehicle's driving lanes. One lane always corresponds to exactly one `edge`. Lane's attributes which are beneficial for this project are `id`, `index`, which represents the position of the lane on the street, `length` and `shape`. If the map data are imported from OSM then the *id* of a lane has the following format `OSM-ID-of-parent-edge_INDEX`. All `shape` values are defines as space-separated list of 2D points (`x` and `y` values separated again by a comma) and mark the middle of the lane.

Finally, `connection` objects in `net.xml` facilitate the description of links between different `edges`. There are multiple metadata options which ease the driving of vehicle through the connections. Moreover, the connections are built according to the driving law. By employing the connectivity information between the lanes one could construct a path from one point on the street network to another. The most prominent connection's attributes are `from/to`, `fromLane/toLane` and `via` (possible intermediate lanes which has to be driven through in order to reach the destination lane).



(a) (b)

Figure 2.2: SUMO (a) and Unity (b) interface.

2.5 Unity

As previously mentioned the MMK Driving Simulator has been currently developed in Unity `v5.6.3` which uses `dotNet v4`. In the following, we highlight the most prominent conjunctions of Unity and the rest of the software introduced in this chapter. Moreover, we discuss the necessary extensions which have to be added to Unity in order to utilise them later in this project.

The generated urban network, together with the 3D models can be exported from CE and imported in Unity using the following steps: (1) Create a folder `Assets\Models\<name of scene>`, (2) Create another folder `Assets\Models\<name of scene>\Textures` and copy all pictures from models folder in CityEngine to this folder, (3) Copy the exported file (*e.g.* `fbx`) file to `Assets\Models\<name of scene>` (the materials folder will be created automatically). These steps have to be executed in the presented order, so the textures are added automatically to all models. Finally, if the format of the exported model from CE is `fbx`, all objects in the Unity scene has to be scaled in all directions by `100`. An example result can be seen in Figure 2.2(b).

In this project we utilise Dijkstra's Algorithm to find the shortest path between two lanes in the street network. As explained in Chapter 4, we need a priority queue which will always track the nearest lane according to the currently visited lane. Unfortunately, the current version of Unity does not implement a priority queue in their standard library. In fact, one could use `Dictionary` and the `Sort` method to implement a priority queue. However, there exist an open-source *High Speed Priority Queue for C#* [11] implemented by BlueRaja and licensed under *MIT*. The project is referenced in multiple Unity forums and has also a high star rating on *GitHub*. By default, the `SimplePriorityQueue.cs` class uses float numbers to manage priority of elements but in the Driving Simulator we use double precision numbers to measure the distances. Therefore, before using it, one have to change all occurrences of `float` with `double`.

Another third-party software which was necessary for the project is *SimpleJSON* C# script which can be found on UnityWiki. The format we have chosen for the semantic description of the street network is JSON but Unity API does not natively support it.

Therefore, we had to find an implementation of a JSON parser and *SimpleJSON* was pointed out again as a considerable option.

3

Semantic Description of Road Systems

The following chapter provides a thorough description of the models and semantic description generation process for an urban aria from ESRI CityEngine (CE) using data provided by OpenStreetMap (OSM) combined in the end with SUMO simulation information. Moreover, we provide a detailed explanation of the resulting JSON format which holds data about essential scene objects. Lastly, we explain how the resulting file is parsed and used by Unity. Additionally, we defend design decisions, as well as describe failed attempts in the process.

A general overview of the whole process can be seen in Figure 3.1. Firstly, OSM data of a given area is sanitised and imported in CE. The same sanitised data is provided to SUMO, which generates additional road details which are not present in CE. Afterwards, 3D models are generated together with the combined from CE and SUMO semantic road description data. In the end, they are imported and utilised in Unity. The navigation module which can be seen in *Unity Game Engine* will be discussed in Chapter 4.

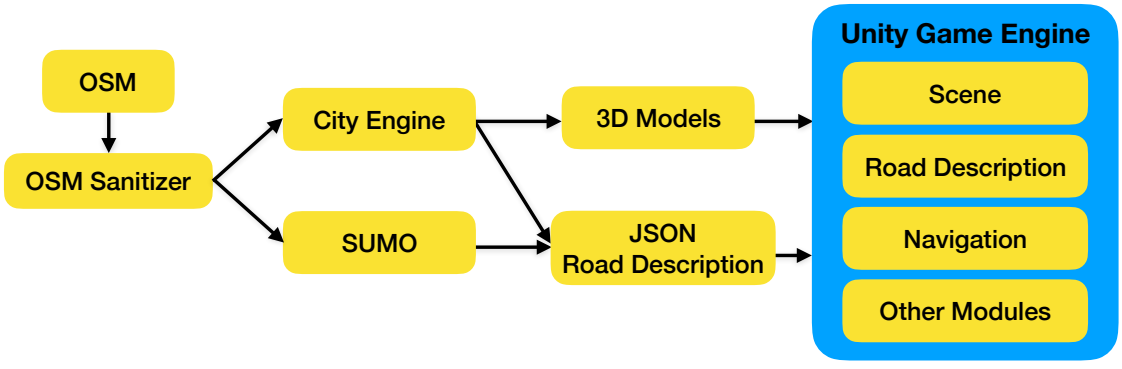
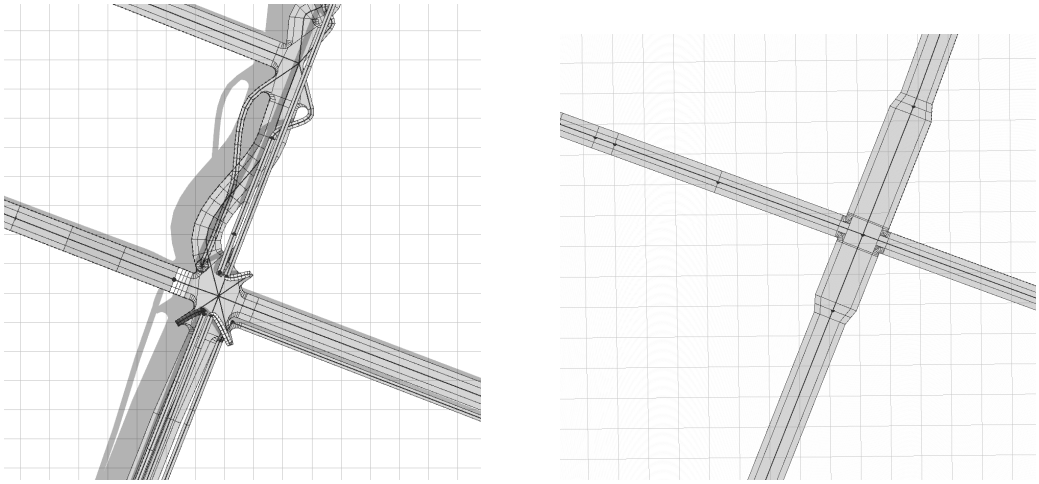


Figure 3.1: Overview of the scene creation workflow.

3.1 OSM Export

As already introduced, the whole map data used in the MMK Driving Simulator is exported from OSM. In Figure 3.1 we can see that exported *OSM* file is given to the *OSM Sanitizer* script, which removes some of the data in it. There are multiple reasons why this is necessary. Firstly, the objects contained in the export include some unnecessary information which is not utilised in the simulation, *e.g.* underground network, footpaths. Moreover, this additional objects in the map represent an obstacle for both SUMO and CE in the scene generation process. For example, in Figure 3.2(a) and 3.2(b) one could see the difference in the shape creation between respectively an unsanitised and a sanitised version of the same *OSM* export data.



(a) Unsanitised (b) Sanitised

Figure 3.2: Comparison of the same CityEngine generated shapes from unsanitised and sanitised *OSM* export file.

Since the information in the OSM export is only 2D, CE attempts to determine the perfect location for every shape, although most of them overlap. Therefore, it chooses `y` values which would generate reasonable shapes. Clearly, the scene generated in this way could not be used in Unity, so we have to delete some of the data. This was achieved with a Python 3 script which can be found in `cityengine-mmk/scripts/osmsanitizer.py`. The script accepts only one input parameter, which is the OSM export file and produces a sanitised version of the same export with the appendix `-sanitized` in its name. It can be executed in the console as following, assuming we are in the same folder as the script: `python3 osmsanitizer.py ../data/map.osm`. The resulting file `map-sanitized.osm` will be again situated in `cityengine-mmk/data` folder.

The sanitisation process follows a straightforward whitelisting fashion. Firstly, we have to delete all roads which are not drivable by a vehicle. Additionally, we want to preserve information about buildings and parkings which should be also present in the export. As we know from Chapter 2, there are two main objects in OSM which hold information about the infrastructure of the road network: `way` and `relation`. Ac-

cording to the OSM documentation[1], all streets (`way` objects) must have a `highway` tag which marks its type. The valid highway tags are `motorway`, `motorway_link`, `trunk`, `trunk_link`, `primary`, `primary_link`, `secondary`, `secondary_link`, `tertiary`, `tertiary_link`, `unclassified`, `residential`, `living_street`, `unsurfaced`. If a `way` object does not contain a `highway` tag or the type of the road is not one of the aforementioned fourteen values, then it is deleted. Similarly, we check whether a `way` object contains a `building` tag which is set to `yes` and an `amenity` tag set to `parking`. This approach leaves in the OSM export buildings and parkings which coordinates and properties can be later imported in CE.

The sanitiser tries to solve another problem with OSM data, too. Sometimes information about streets, *e.g.* speed limits or number of lanes, could be missing. When this is the case CE and SUMO try to extrapolate this information and suggest some valid values in order to create the resulting models. Unfortunately, it could happen that both softwares provide different values for the same part of the road system. Because we want to integrate the information streams coming from both, CE and SUMO to create a semantic description of the road network, this data extrapolation has to be accomplished one step ahead in the sanitisation process. Therefore, we provide default valid values for possibly missing or damaged information and add it to the OSM objects in the final sanitised file.

Finally, we have an OSM data which is valid and contains only the necessary information to build 3D shapes from it. Note that we do not delete any `node` objects because unused nodes, *i.e.* not connected by a way or a relationship, are by default discarded in both, CE and SUMO.

3.2 Building the Semantic Road Description

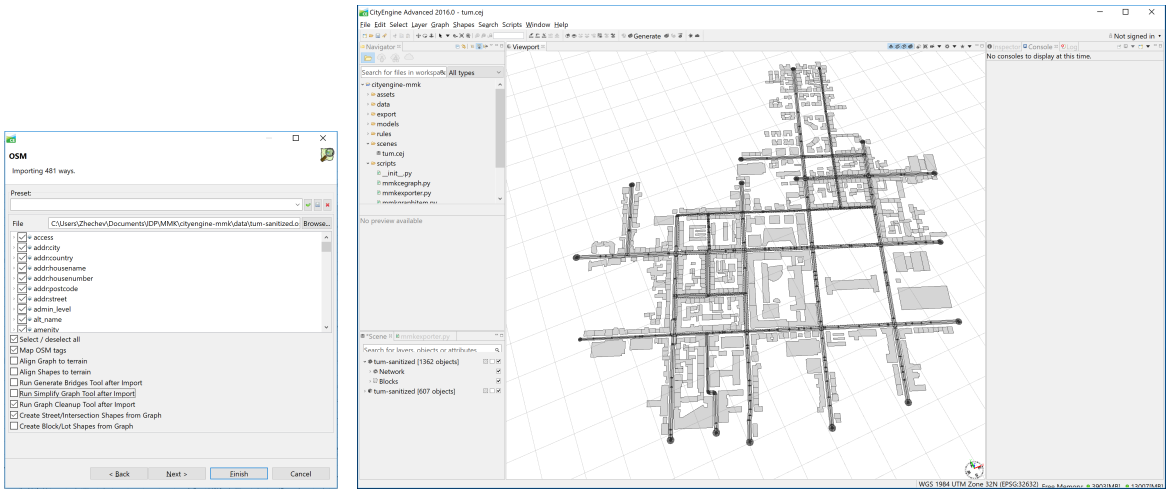
After we have accomplished the sanitisation of the OSM data, we can provide the resulting file to both CityEngine and SUMO as can be seen in Figure 3.1. Overall, we use the CE's Python interface to traverse all necessary objects and extract their properties. Afterwards, using the OSM ID, we extract lane information about the road network generated by SUMO, which is not available in CE. In the end, the combined information from CE and SUMO is exported as JSON format, which structure is described in Section 3.3.

3.2.1 OSM Data Import Process

In order to build a 3D urban scene accompanied by a semantic description of the road network, the user has to execute the steps which will be described in this section. Some of the steps were not able to be automated because of limitations related to CE's Python interface. After starting a CE instance and opening the `cityengine-mmk` project which contains some essential rules to generate the 3D models, the user has to create a new empty scene. The sanitised OSM file has to be situated in `cityengine-mmk\data`

[1] `http://wiki.openstreetmap.org/wiki/Key:highway`

folder. Now the user can drag the file to the 3D viewport and a similar window as in Figure 3.3(a) will show up. Since we have removed all of the unnecessary objects from the OSM file, we can just select all properties to be imported. Next, we can leave the *Map OSM tags* selected, because we will need them later in the export process. Although our sanitiser has prepared the data, we still need *Run Graph Cleanup Tool after Import* option because sometimes OSM data can be still imprecise. This option will delete unnecessary nodes and segments which overlap each other or are incomplete. Finally, we want CE to generate shapes for the road network, therefore we mark the *Create Street/Intersection Shapes from Graph* option, too. There is a way to import an OSM map data directly with the CE's Python interface using `CE.importFile` and `OSMImportSettings`. Unfortunately, it is impossible to select which of the each object's attributes to be imported. Moreover, only some of the attributes are selected by default which is not sufficient for our purposes. In the end, one have to select *Finish* and the software will import the data and generate 3D shapes. An example import of the *TU Viertel* in München[2] can be seen in Figure 3.3(b).



(a) (b)

Figure 3.3: Example import dialog of OSM data in CE (a) and the resulting scene (b).

3.2.2 CE Settings and Way-Segment Inconsistencies

If we happen to leave the *Run Simplify Graph Tool after Import* option selected, it may alter the imported OSM information, thus making it useless for cross referencing data between OSM and CE. The simplifying strategy of CE tries to merge some neighbouring segments in order to make the underlying road system's shapes more straight. Unfortunately, the CE's algorithm tries to merge properties of segments which are not meant to be combined. If it happens that the two segments, which are going to be combined,

[2] `https://www.openstreetmap.org/#map=16/48.1493/11.5678`

posses different OSM ID values, then the new segment receives the arithmetic mean of the two values. Of course, this new ID is not valid and we cannot restore the original segments' IDs which renders it useless. To address this problem, we selected separately segments which have the same OSM IDs and called the simplifying tool only for them. Fortunately, this solved the problem with the destroyed IDs, nonetheless it broke junction's connections. In that case, two ways which were crossing in the middle forming a junction, were separated by CE in 4 segments, one for each exit of the junction. Calling the simplifying tool separately for the two pairs deleted the junction and placed one of the roads above the other by combining the two segments again. Clearly, this was not a solution to this problem, so we decided to unmark this tool since all of the generated shapes until this moment were acceptable.

Another setting which could negatively influence the shape generation process and therefore decrease the reality of the produced scene is *Run Generate Bridges Tool after Import*. This tool determines whether a segment's `y` value can be changed so it is lifted above another, underlying structure. This approach introduces some inaccuracies in the generated road models. Since this tool is absent from SUMO (it is only a 2D simulation) and we have to keep the shapes of the two generated urban networks as close as possible, we chose to switch off this option. However, this also means that the user has to furthermore adjust the shapes of some objects to represent the reality while staying coherent with SUMO.

Our first approach to create a semantic description of the road network in CE generated using OSM data involves completely ignoring the imported parameters in CE. We still have to export the vertices of the CE's shapes using the CE's Python interface but all of the information about a given street segment such as number of lanes or speed limits will be directly taken from the OSM export file. The biggest advantage of this approach is that we can accomplish the `xml`-parsing in Python 3 which might be more performant than using the Jython interface in CE to read all of the attributes. Although this is generally true, we would break the export functionality in two big modules which are highly coherent and they normally belong in one place. Moreover, there was another obstacle which seemed crucial for the success of this approach. CE does place almost (actually this causes yet another problem which will be discussed later in this section) every OSM node on its correct place and uses its original OSM ID, but this is not the case with segments. According to the shape of a given way (which is determined by its nodes), CE decides in how many actual segments to partition it. In other words, it may happen that one `way` object has more than one corresponding segments in the CE scene. Unfortunately, this introduces unnecessary difficulties in reconstructing a way from its segments which would be later crucial for the lane generation. Therefore, this first approach was abandoned in favour of a script, purely implemented in the CE's Jython environment which depends on the imported attributes in CE from OSM to reconstruct each object's behaviour.

3.2.3 CE Export

After importing the OSM data in CE as previously explained, the user has to assign rules to every object (*e.g.* buildings, streets) and generate the 3D models. Some minor adjustments, such as moving buildings or placing crossing marks manually, may be necessary in order to fully prepare the scene for an export. Next, we can start the exporting process which is conducted by the `MMKExporter` class which can be seen in Figure 3.4. Firstly, we only consider the `MMKCEGraph`.

The `MMKExporter` traverses all `nodes`, `segments` and `shapes` in CE and parses them together with their attributes in the corresponding `CEGraphItem` objects (Figure 3.5). Utilising the CE's Python interface (`ce.getAttribute(item, 'NameOfAttribute')`), it is easy to access all of the essential attributes of each object which were imported from OSM. As you can see in the UML class diagram in Figure 3.5, we have adopted a simple inheritance hierarchy which allows extendability, *i.e.* adding more CE objects to the export. As already explained in Chapter 2, the whole scene has to be exported in the end to some compatible format. Moreover, one have to specify the `x`, `y`, and `z` offsets which will translate all object's coordinates to the origin. We always leave the `y` to `0` since this data is interpolated anyways by CE. However, `x` and `z` has to be provided as input parameters to `MMKExporter` as `ox` and `oz`. This is necessary, since the parsed object's coordinates has to be also translated in order to stay coherent with the corresponding 3D models. Additionally, the `x` and `y` coordinates of each CE object has to be negated (`180°` rotation) in order to correspond to the exported 3D models. Finally, the `exportJson` method can be invoked after the parsing procedure has finished and the semantic description is saved in the `export` folder of the CE project in JSON format.

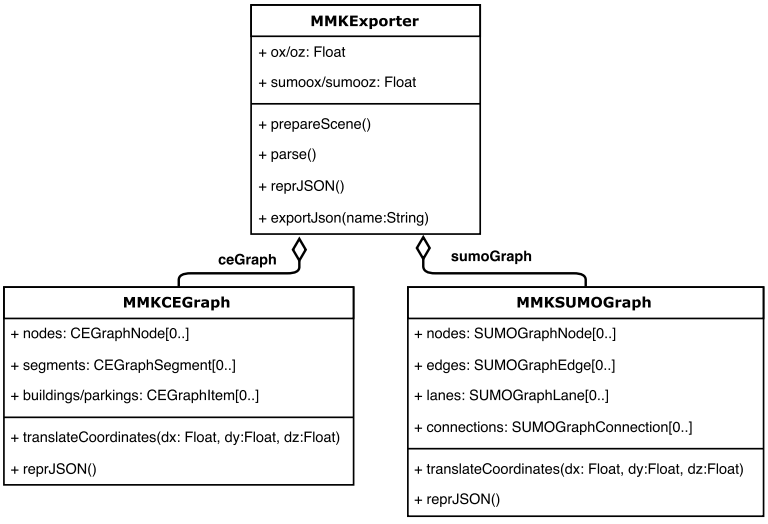


Figure 3.4: UML class diagram of `MMKExporter`, `MMKCEGraph`, and `MMKSUMOGraph`

3.2.4 Additional Information from SUMO

Now, that we have the basic properties of the road network we can try to extend it with additional data by benefiting from OSM ID values. For instance, we can add lane information and their connectivity to every street segment. Unfortunately, OSM data holds only the number of lanes for each street segment, therefore, CE does not create any lane shapes. We could try to generate lane shapes by interpolating the given data: number of lanes per segment and size of the segment. However, we know that MMK Driving Simulator uses SUMO to simulate other traffic in the scene, which defines its own lane information. Therefore, we can try to synchronise CE and SUMO shapes coordinates and adopt SUMO lanes in our CE export process. This approach was associated with numerous problems which will be discussed in the following.

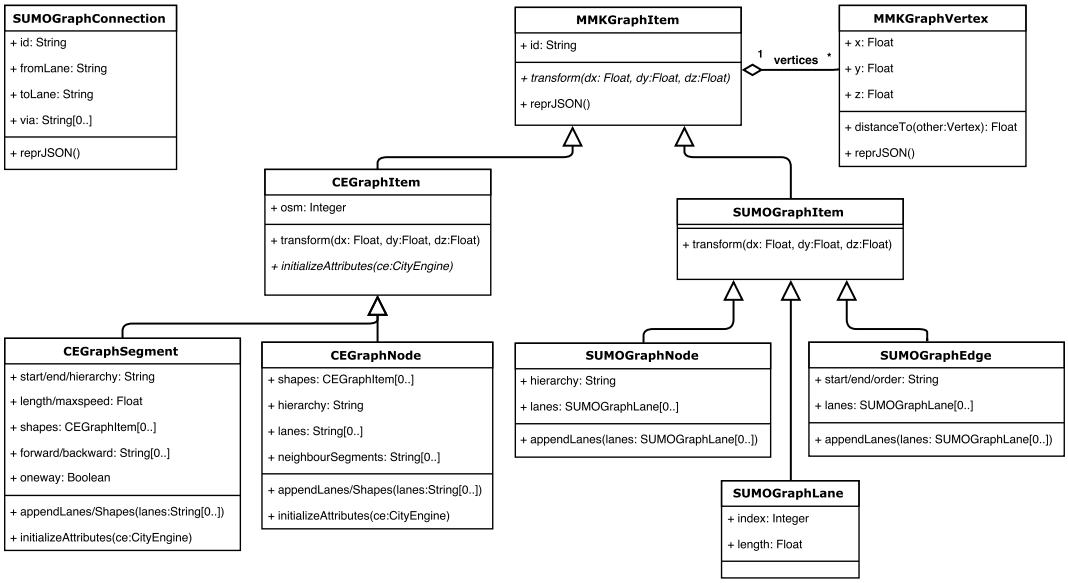
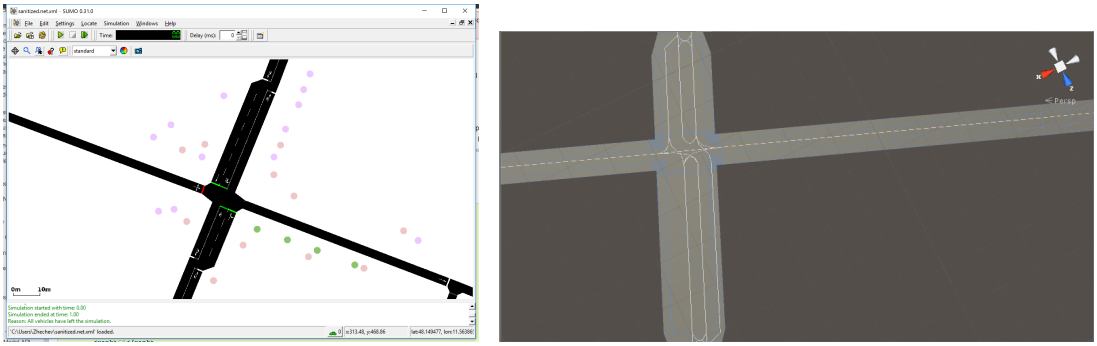


Figure 3.5: UML class diagram showing the `MMKGraphItem` hierarchy and other classes used by the `MMKExporter`.

Firstly, we parsed SUMO objects in corresponding `MMKGraphItem` objects from the `net.xml` file (Figure 3.5). We also parsed the connectivity information of the lanes in `SUMOGraphConnection` objects. As introduced in Chapter 2, the imported OSM data in SUMO is projected and translated using some parameters similarly to CE centring export feature. CE and SUMO use the same default projection settings but this is not the case with the translation offsets. Therefore, we had to find these offsets and adjust SUMO object's translation to match the CE configuration. Firstly, it was unknown to us that the `net.xml` contained a `location` object which specifies exactly the translation offsets used to adjust all coordinates. Therefore, in order to find the lane's coordinates generated by SUMO in CE's coordinate system, we had to find some correlation between the objects. We know that OSM `way` objects are sometimes separated in more than one segments in CE. The same is valid for SUMO, so we could not use segments to calculate

the necessary translation since more than one segment in CE have the same OSM ID. Then, we decided to take under consideration the nodes which are uniquely identifiable by their OSM ID values in both SUMO and CE. Overall, we searched for the same node in CE and SUMO objects and calculated the difference in their coordinates. These values were then used to translate all SUMO object's coordinates to match CE's models. Moreover, we discovered that SUMO coordinates are rotated by `180°` in comparison to CE coordinate system which had to be reversed before translation. This process was accelerated when we started using the offset values defined in `net.xml`'s node `location`.

Nonetheless, this was not the only incompatibility between the coordinates of the CE and SUMO objects. Apparently, the algorithms which CE and SUMO employ to generate junctions' shapes have very different features. In Figure 3.2(b) and Figure 3.6(a) we can see the same junction created by CE and SUMO. Moreover, Figure 3.6(b) shows lanes's figures generated by SUMO next to their corresponding CE models. Clearly, SUMO tries to center the streets joining in a junction according to their number of lanes. Since, we cannot interfere in the junction creation algorithms in SUMO and CE, we tried to shift the street models in CE's scene to be alined to SUMO's shapes. This is done in `prepareScene` method in `MMKExporter` by adjusting the CE's segments' `/ce/street/streetOffset` attribute. Firstly, we search for segments which are joining in a junction, then for each segment we subtract its number of forward and backward lanes. The resulting value is then used to shift the current segment either left or right according to its orientation in the whole road network. As already mentioned, OSM `way` objects can be split into more than one segments in CE and SUMO. Therefore, it could happen that the number of corresponding edges in SUMO and segments in CE is not the same. Thus, one SUMO edge may be represented in CE by two (or more) adjacent segments (from which exactly one is at the junction). Therefore, we have to propagate our shifting algorithm in all directions away from the junction for similar segments, *i.e.* same number and kind of lanes, as well as hierarchy. As a result, we have achieved a cohesion between CE's and SUMO's shape generation process (Figure 3.7). However, there might be some cases which could not be handled because of the versatile nature of both softwares and the user has to still resolve inconsistencies manually by herself.



(a) (b)

Figure 3.6: Inconsistency between SUMO and CE junction creation.

Finally, we can start ordering the parsed lanes to their corresponding segments which is only possible because they share the same OSM ID values. We note that in the final format, one lane can be associated to more than one CE segment. However, this ordering process can stumble upon yet another difference between SUMO's and CE's import processes. Even after disabling all shape optimisation routines in CE, it might happen that some nodes are not imported but rather their corresponding segments are attached to another, neighbouring node. We can handle that by looking for nodes which are present in SUMO but missing in CE and finding the nearest existing node in CE. Then, we attach to it the lane's information. However, this operation could be ambiguous because there could be more than one neighbouring nodes.

In conclusion, we have managed to integrate SUMO generated road information into CE urban network. As a result, we can export this combined semantic road description data as a whole. In the next section we throughly describe the JSON export format which holds this data.

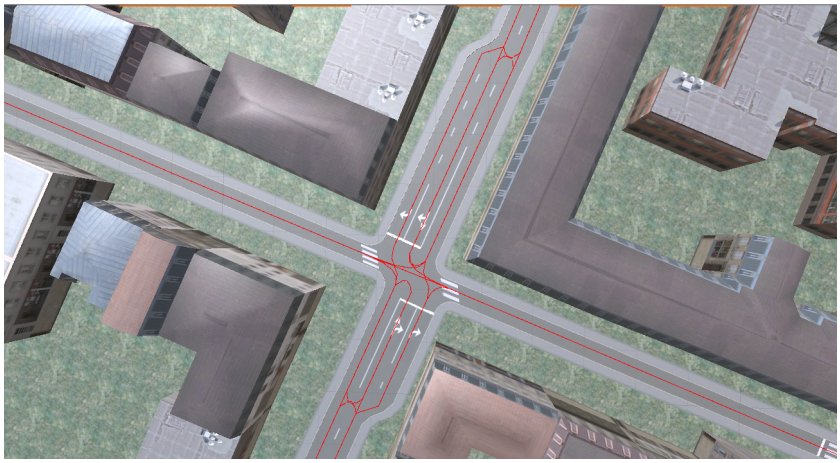


Figure 3.7: Appropriate shifting of the created junction shapes in CE aligns them to the corresponding SUMO junction objects.

3.3 Format Description

As already introduced, we adopt a JSON format to hold the contents of the semantic description of a road network. In the following, we discuss its advantages and disadvantages, as well as present all essential key value pairs used in the format.

In the beginning of this project, there was an idea to export the street network in OpenDRIVE format which was mentioned in Chapter 2 in connection to other related work. Although OpenDRIVE allows a precise description of a road network and moreover, it is open-source and can be freely adopted, we withdraw from using it because of synchronisation reasons. More precisely, the data saved in OSM format has to be interpolated [7], which would introduce once again another desynchronisation between

the road networks built by CE and the SUMO simulation. If this happens, the interpolated lanes in a hypothetical OpenDRIVE format would not coincide with the lane information generated by SUMO. Therefore, SUMO may not be able to be integrated as a simulation environment. Additionally, a JSON format, which we have chosen, has proven to be more human readable and easier to parse. Moreover, there exists a Unity module (Chapter 2) which allows easier parsing. A JSON-based format gives us the opportunity to simultaneously provide a thorough description of a certain road network but also adopt and combine other data without breaking their coexistence.

In Listing 3.1 one can see a general overview of the used JSON format. Firstly, we define the `author`, `date` of the export and the name of the `project`. These can be also changed in the `MMKExporter` script to arbitrary values. The more crucial parts of the export are `nodes` and `segments`. Obviously, each of these arrays hold all of the nodes, respectively segments in the CE scene. Previously, in the research project by Maximilian Murauer, next to these values, there were also the number of all elements which the both arrays hold. We have chosen to eliminate these values since every parsing process can count the elements in each array. Next, the JSON export contains the `offsets` of CE and SUMO which were applied to each vertex coordinates in order to compose one coherent scene. Then, we have the `connections` array which holds all possible links between lanes which is later used to build a navigation system in the next chapter. Lastly, `sceneObjects` holds vertex information about other objects in the scene which have no direct relation to the road network, *e.g.* buildings, parkings.

```
{
    "author": "TUM - MMK",
    "connections": [...],
    "date": "2017-11-13",
    "lanes": [...],
    "nodes": [...],
    "offsets": {...},
    "project": "cityengine-mmk",
    "sceneObjects": {...},
    "segments": [...]
}
```

Listing 3.1: General overview of the used JSON format for description of road networks.

A sample `node` value can be seen in Listing 3.6. Firstly, we define its `hierarchy` which is specified by SUMO and it can be one of the following values: `unknown`, `priority` or `traffic_light` depending on the kind of node. Secondly, each node holds information about every lane which is part of it. Next, we find two id values in the export: `id` is the OID value generated by CE and `osm` is obviously its OSM ID. Each node's neighbouring segments are also referenced in the export by their id values. A node can have zero or more shapes which properties can be found in the `shapes` array. Additionally, each shape has again two kinds of id values (identical to the node's ids). Finally, the coordinates of each node are kept in the `vertices` key.

```
...
"nodes": [
    {
        "hierarchy": "priority",
        "id": "40170885-31eb-11b2-8d80-00e8564141ba",
        "lanes": [
            ":21457467_0_0",
            ...
        ],
        "neighbourSegments": [
            "46535ea4-31eb-11b2-8d80-00e8564141ba",
            ...
        ],
        "osm": 21457467,
        "shapes": [
            {
                    "id": "40170885-31eb-11b2-8d80-00e8564141ba:0",
                    "osm": 21457467,
                    "vertices": [
                        {
                            "x": 227.979736328125,
                            "y": -0.0,
                            "z": 109.73065185546875
                        },...
                    ]
            }
        ],
        "vertices": [
                {
                    "x": 232.09375,
                    "y": -0.0,
                    "z": 106.30369567871094
                }
            ]
        },
        ...
],
...
```

Listing 3.2: General overview of the node key in the JSON exportformat.

Similarly to nodes, each segment has two id values (OID and OSM ID), as well as analogous `shapes` key. The `hierarchy` key provides the type of road defined by OSM and each possible value has been described in Chapter 2. Next, segments define in their `lanes` key the direction of each lane (`forward` or `backward`). There are some self-explanatory attributes which are specific to segments such as `maxspeed`, `oneway`, `length`. Additionally, each segment holds a reference to its `start` and `end` node. Lastly, there is a `vertices` array holding a list of 3D polygon points which describe the overall shape of the segment.

```
...
"segments": [
    {
        "end": "4545af7a-31eb-11b2-8d80-00e8564141ba",
        "hierarchy": "tertiary",
        "id": "4545fc94-31eb-11b2-8d80-00e8564141ba",
        "lanes": {
            "backward": [
                "-392069506#0_0"
            ],
```

```
            "forward": [
                "392069506#0_0"
            ]
        },
        "length": 27.097801453973258,
        "maxspeed": "50",
        "oneway": false,
        "osm": 392069506,
        "shapes": [...],
        "start": "43422a00-31eb-11b2-8d80-00e8564141ba",
        "vertices": [...]
    },
],
...
```

Listing 3.3: General overview of the segments key in the JSON export format.

Lanes objects' structure is straightforward to describe. They posses an `id` field where their SUMO generated id is saved. Additionally, they have an index, length and a list of vertices describing their shape.

```
...
"lanes": [
        {
            "id": ":315667013_8_0",
            "index": "0",
            "length": "7.88",
            "vertices": [
                {
                    "x": 34.27750000000003,
                    "y": 0.0,
                    "z": -110.02389160156247
                },...
            ]
        },
    ],
...
```

Listing 3.4: General overview of the lanes key in the JSON export format.

Next, we take under consideration the connection objects. The only information they hold is that there is a connection between two lanes (`fromLane` to `toLane`). Since, two lanes might be connected through intermediate links, these links are noted in the `via` array.

```
...
"connections": [
        {
            "fromLane": "-155040996_0",
            "id": 0,
            "toLane": "392103894#1_0",
            "via": [
                ":4972221472_0_0"
            ]
        }, ...
    ]
...
```

Listing 3.5: General overview of the connection key in the JSON export format.

The scene objects are defined as shapes and have all of their properties as described above. Finally, we consider the `offsets` key which holds the translation parameters applied to every object from CE and SUMO in order to match both coordinate system.

```
...
"offsets": {
        "ce": {
            "x": -690985,
            "z": 5336220
        },
        "sumo": {
            "x": 605.7575,
            "z": 595.5761083984376
        }
    },
...
```

Listing 3.6: General overview of the offsets key in the JSON export format.

To summarise, the resulting export file with all of its features described in this section holds the necessary information about a given urban network. In the next section we discuss how this file is imported in Unity.

3.4 Unity Import

Finally, we discuss how the generated 3D models and the corresponding semantic road description are imported and used in a Unity scene as can be seen in Figure 3.1. We explained in Chapter 2 how CE exported models are loaded in Unity. The next step involves dragging and dropping the model to the Unity scene. Then, the user has to change the scaling of the street model to `(100, 100, 100)` and add a *BoxCollider* to the ground of the scene. In the end, `Network` script has to be attached to the model and `CarRoadDescriptor` script to *MMKCar*. In `Network` we have to specify the path to the road description file and in `CarRoadDescriptor` we have to check the name of the 3D urban model.

All scripts responsible for the parsing of the road network description file and their communication behaviours are depicted in the UML class diagram in Figure 3.8. Similarly to the export process, there are classes which encapsulate road structures, such as edges (`NetworkEdge`) and lanes (`NetworkLane`). `Network` holds the actual graph and it is responsible for its creation from a JSON format file described in the previous section. Upon creation, to every edge and node a *BoxCollider* is assigned which encapsulates its vertices. This is used in the `CarRoadDescriptor` for optimisation reasons when calculating the currently occupied lane by the vehicle. In order to determine that, we have to perform costly geometrical calculations involving the current coordinates of the car and **all** lanes in the scene. Instead of traversing all lanes, we could consider only lanes from the current edge or node where the vehicle is situated. This is achieved by tracking which colliders the vehicle is currently triggering. The `RoadPosition` method is responsible for determining the current lane and segment of the vehicle.

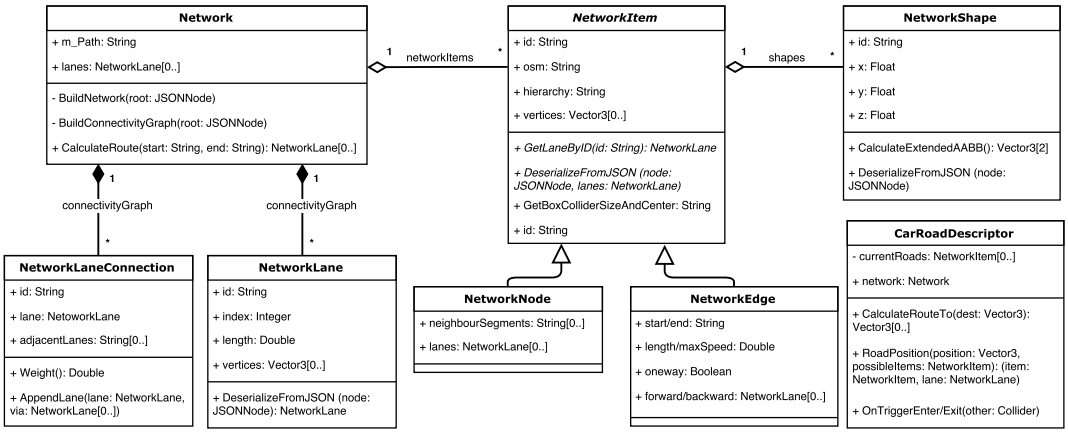


Figure 3.8: UML class diagram of modules related to semantic road description in Unity's project.

The whole scene generation process, as well as the semantic road description export have been applied to an OSM export data from different parts of the world. A 3D scene of *TU Viertel* in München and the corresponding description of the urban network can be found in the CE project.

4

Routing

There are many applications of the semantic road description which can be used to improve the usability of the MMK Driving Simulator. One such application is navigation which we will discuss in this chapter. In Figure 4.1 one can see a vehicle (above, right) which wants to reach the blue point (down, right). Utilising the connectivity of the lanes derived by SUMO, as explained in the previous chapter, we can calculate the shortest possible path. This path is shown in the figure using red dots. Moreover, one can see that we have included to the path the `via` lanes, too. In the following, we describe the employed algorithms to calculate an ordered list of points which a vehicle has to follow in order to reach a predefined destination. Additionally, we discuss our design decisions, as well as some limitations which were recognised during development.

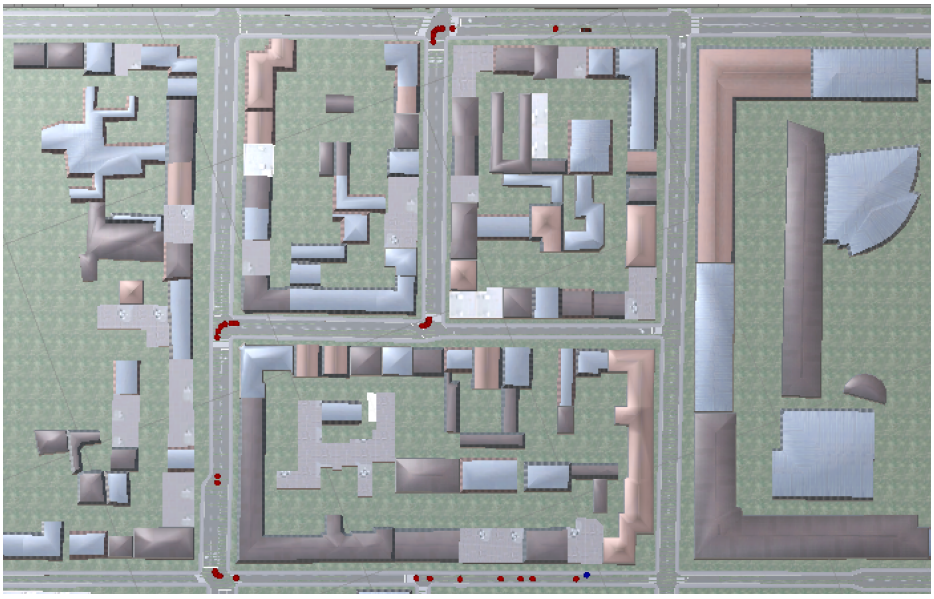


Figure 4.1: MMK Car, above right and the calculated shortest path (red) to a predefined user destination (down right, blue).

4.1 Routing Algorithm

Currently, there exist many algorithms able to calculate the shortest path between two points, *e.g.* Dijkstra, Bellman-Ford, A*. Some of these algorithms work under special conditions, *e.g.* Dijkstra can be applied to graphs where no edge has negative weight. On the other hand Bellman-Ford is more versatile, as it is capable of handling graphs in which some of the edge weights are negative. Since in our graph there are only edges with positive weights, we decided to choose Dijkstra for our implementation.

Dijkstra's algorithm is a shortest-path algorithm that iteratively searches from a start node s to a target note t [12]. From a currently visited node v, all neighbouring, unvisited nodes get a weight set based on the edge weights. Each node encountered gets a weight w assigned which denotes the distance from s to this node via the current node v. If a node already has a weight, it is only changed if the new weight would be lower, ensuring an optimal solution.

The algorithm forms a circle around the start node, with a radius increasing in each step. The search can be stopped once t has been visited. The resulting path is the shortest path from s to t.

4.2 Implementation Details

First of all, we have to build a graph from the provided lane connectivity information in order to apply Dijkstra to it and calculate the shortest path between two points. However, this cannot be done in a straightforward way because in this case the nodes of the graph are the actual lanes and the connections are links between them which have no length. Of course, some connections have "via" lanes which can be used to calculate the distance between two lanes but this cannot be applied everywhere. We note that, in order a vehicle to drive from the start of a given lane A to the start of another, adjacent to A, lane B, it has to drive through the whole lane A. This intuition helped us to design the actual graph. Since links between lanes do not provide weight information (apart from the sum of the length of via lanes if present), we defined **the weight of an edge** connecting given lane A and lane B (in the direction from lane A to lane B) as the **length of lane A**. Since, there is no connectivity information for neighbouring lanes, we do not consider these connections.

After designing the graph, we have to decide its representation in memory. There are many possibilities to achieve this, *e.g.* adjacency matrix/array/list, edge list. The most crucial part of the Dijkstra algorithm as already explained is the finding of neighbouring nodes, which has to be done in an efficient way. Therefore, we have chosen to utilise an adjacency list to hold the graph information. Additionally, we implemented a special object `NetworkLaneConnection` (Figure 3.8). In the `Network` class, using the `BuildConnectivityGraph(_: JSONNode)` method we parse all connections defined in the semantic road description export file. Firstly, we initialise a data structure to store all `NetworkLaneConnection` objects. Each object in this structure has, as can be seen in Figure 3.8, a lane object and a list with all adjacent to it other lanes. Next, for

every connection between two lanes, we either create a new `NetworkLaneConnection` using the `fromLane` and `toLane` as respectively the `lane` attribute and the first lane in the `adjacentLanes` list. Otherwise, the corresponding `NetworkLaneConnection` object with the current lane already exists and we just have to store the `toLane` to the list of adjacent lanes. Additionally, we store all `via` lanes to the corresponding destination lanes.

Next, we implemented Dijkstra in the `Network` class which calculates the shortest path between two lanes given their id values. `SimplePriorityQueue` class presented in Chapter 2 was utilised to further optimise the algorithm. The id values of the start and destination lane were calculated using the `RoadPosition` method detailed in Chapter 3. There are two requirements, which have to be fulfilled in order to execute the algorithm. Firstly, the vehicle has to be on a valid lane and secondly, the destination point has to lay again on a valid lane. Using the id values of the start and destination lane, we can invoke the `CalculateRoute` method in `Network` class. The result is a list of `NetworkLane` objects which has to be traversed in order a vehicle to reach the desired destination, starting from its current location. If there is no route found or some of the aforementioned two requirements are not met, then the method returns `null`. Nonetheless, this is not the result we expected in the beginning because the vehicle awaits from us an ordered list of navigation points which it can follow.

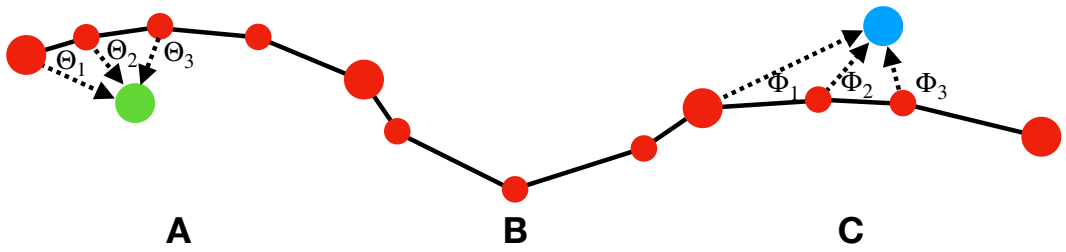


Figure 4.2: Given a path of lanes (direction from A via B to C) and the coordinates of the vehicle at the start (green point) and the destination (blue point), we use the *dot* product to calculate which points have to be excluded from the final path.

Lastly, we have to collect all points which construct each lane on the calculated shortest path. Generally, there are two cases where lanes have to be handled separately: (1) the first and the last lane, (2) all lanes between them. Since the vehicle does not necessarily start driving through the given route from the start of the first lane, but rather from where it is currently situated, we have to exclude all lane points which are "before" the vehicle. Similarly, all points "after" the destination coordinates have to be discarded from the route. An example is shown in Figure 4.2, where the green point represents the vehicle, while the blue point is the destination. There are three lanes (A, B, and C), which starts and ends are represented by bigger red dots, and each point on every lane is a smaller red dot. As we can see, the vehicle is already situated after the first two points on the start lane, so they don't have to be included in the navigation route. In order to determine which points have to be excluded from the path, we start

in the beginning of the first lane and for each point on it we calculate the *dot* product between the vector built by the current point and the vehicle coordinates and the vector built by the current point and the next point on the lane. If the resulting value is negative, *i.e.* the angle between the two vectors is larger than `90°` (or smaller than `-90°`), then this point is excluded from the list. Analogously, the same operation is performed for the destination coordinates.

To summarise, in this chapter we designed a graph from the lane connectivity information provided by SUMO and implemented a routing algorithm in order to determine the shortest path between a given start and destination.

5

Conclusion and Outlook

5.1 Conclusion

In this Interdisciplinary Project we have extended the functionality of the MMK Driving Simulator developed at the Institute for Human-Machine Communication, TUM. Some of these functionalities include the ability to generate a semantic description of the road network and ways to utilise it in the simulator.

Firstly, we designed a JSON format which holds semantic description of a road network. We used this format to describe a 3D urban environment generated in CityEngine adopting OpenStreetMap (OSM) data from an arbitrary chosen areal. In order to achieve this, we employed the Python interface provided by CityEngine and implemented a Python script which exports the properties of the scene. However, this approach was not enough for a complete description of the considered road network. Therefore, we also imported the same OSM data in SUMO and used it to extend the semantic description with data about valid lanes which facilitate the movement of vehicle in the scene. Furthermore, we implemented an interface which imports this description format in Unity and can be utilised for different purposes, *e.g.* navigation or traffic signalisation enhancements.

Finally, we used the exported semantic description of a road network to build a navigation functionality for vehicles in Unity. This can be used to calculate the shortest path from one point to another point on the map.

5.2 Outlook

While in this project we have developed a way to semantically describe a road network and showed some of its applications, there are many opportunities for extending the scope of this project in the concept of the Driving Simulator. This section presents some of these directions and gives some suggestions how they can be achieved.

First of all, one could use the buildings' vertex information contained in the exported file to place colliders in Unity's scene so the vehicle could not drive through them. This could be done for each building or even better: one collider could contain more than

one building which are in close proximity. This would have a positive effect on the performance of the simulator. Furthermore, the same idea could be used to combine some of the nearly situated colliders which are generated for every road segment. Finally, the road information can be used to implement autonomous vehicles.

Bibliography

[1] Unity Game Engine, "Unity game engine-official site," *[Online][Cited: October 9, 2008.] http://unity3d. com*, pp. 1534–4320.

[2] Esri R&D Center Zurich, "CityEngine, 2016," http://www.esri.com/software/city-engine.

[3] OpenStreetMap contributors, "Planet dump retrieved from https://planet.osm.org," https://www.openstreetmap.org.

[4] D. Krajzewicz, J. Erdmann, M. Behrisch, and L. Bieker, "Recent development and applications of SUMO - Simulation of Urban MObility," *International Journal On Advances in Systems and Measurements*, vol. 5, no. 3&4, pp. 128–138, December 2012.

[5] T. Haubrich, S. Seele, R. Herpers, M. E. Müller, and P. Becker, "Semantic road network models for rapid 3d traffic scenario generation," in *Tagungsband ASIM/GI-Fachgruppentreffen STS/GMMS, Workshop Simulation technischer Systeme-Grundlagen und Methoden in Modellbildung und Simulation*, 2013, pp. 51–55.

[6] M. Dupuis *et al.*, "Opendrive format specification," *VIRES Simulationstechnologie GmbH*, 2010.

[7] H. Shi, "Automatic generation of opendrive roads from road measurements," 2011.

[8] N. Hiblot, D. Gruyer, J.-S. Barreiro, and B. Monnier, "Pro-sivic and roads. a software suite for sensors simulation and virtual prototyping of adas," in *Proceedings of DSC*, 2010, pp. 277–288.

[9] Esri R&D Center Zurich, *CityEngine Manual*, 2016.

[10] B. van Andel, T. Bieniek, and T. I. Bø, "utm - Bidirectional UTM-WGS84 converter for python ," https://github.com/Turbo87/utm, 2012.

[11] B. Raja, "A c# priority queue optimized for pathfinding applications," https://github.com/BlueRaja/High-Speed-Priority-Queue-for-C-Sharp, 2015.

[12] S. S. Skiena, *The algorithm design manual*. Springer Science & Business Media, 1998, vol. 1.