TUM School of Computation, Information and Technology
Technical University of Munich

Bachelor's Thesis in Games Engineering

Improving the Realism of a Real-Time Digital Twin of Road Traffic Using the CARLA Simulator

Verbesserung der Realitätsnähe eines digitalen Echtzeit-Zwillings des Straßenverkehrs auf Basis des CARLA-Simulators

Supervisor Prof. Dr.-Ing. habil. Alois C. Knoll

Advisor Walter Zimmer, M.Sc.

Author Robin Brase

Date February 15, 2023 in Munich

Disclaimer

I confirm that this Bachelor's Thesis is my own work and I have documented all sources and material used.

Munich, February 15, 2023

(Robin Brase)

Abstract

With an ever-increasing need for high-quality training data for machine learning, photorealistic simulations of digital twins are becoming more than just immersive visualization tools. To validate the safety of autonomous vehicles, virtual replicas of real test tracks are often created nowadays. In these virtual replicas, the systems can be tested under a variety of scenarios. While most existing research focuses on validation and generation of training data, there is little work to synchronize such a digital twin with actual traffic in real time. Using a part of the A9 test track [Krä+19] as an example, a process is presented that shows how the real world can be reproduced virtually with close attention to detail. It is also shown how such a simulation can be connected to an existing detection pipeline to spawn vehicles in a realistic manner. The resulting extension of the digital twin can now be used as a basis for real-time visualization and data generation.

Zusammenfassung

Angesichts des ständig steigenden Bedarfs an hochwertigen Trainingsdaten für das maschinelle Lernen werden fotorealistische Simulationen digitaler Zwillinge zu mehr als nur immersiven Visualisierungstools. Um die Sicherheit von autonomen Fahrzeugen zu überprüfen, werden heutzutage oft virtuelle Nachbildungen von realen Teststrecken erstellt. In diesen virtuellen Nachbauten können die Systeme unter einer Vielzahl von Szenarien getestet werden. Während sich die meisten Forschungsarbeiten auf die Validierung und Generierung von Trainingsdaten konzentrieren, gibt es bislang nur wenige Ansätze, um einen solchen digitalen Zwilling in Echtzeit mit dem realen Verkehr zu synchronisieren. Am Beispiel eines Teils der A9-Teststrecke [Krä+19] wird ein Verfahren vorgestellt, welches eine detailgenaue virtuelle Nachbildung der realen Welt ermöglicht. Außerdem wird gezeigt, wie eine solche Simulation mit einer bestehenden Detektionspipeline verbunden werden kann, um Fahrzeuge auf realistische Weise zu platzieren. Die so entstandene Erweiterung des digitalen Zwillings kann nun als Basis für die Echtzeit-Visualisierung und Datengenerierung genutzt werden.

Contents

Chapter 1

Introduction

1.1 Providentia

> "Providentia [Krä+19] is a research project that has been funded by the Federal Ministry of Transport and Digital Infrastructure (BMVI) since early 2017. Since early 2020, it has been continued under the name Providentia++ with the Chair of Robotics, Artificial Intelligence and Real-time Systems at the Technical University of Munich's Department of Informatics serving as consortium leader. The project's goal is to research the flow of information between vehicles and infrastructure along the A9 highway extending into the urban area, to create a digital twin of the current traffic situation [...]" [pro].

For this purpose, several masts and sign bridges are outfitted with a multitude of sensors that record the current road conditions. This sensor data is then used to detect and track objects such as cars, pedestrians and cyclists so that their position can be transmitted to the Digital Twin (DT).

Figure 1.1: Sensors mounted on the S110 gantry.

1.2 Motivation

A 3D digital twin brings several advantages. Unlike a live stream from a real camera, that of a digital twin does not contain any personal data and can therefore be streamed to everyone with fewer restrictions. Furthermore, it is easier to collect training data in the virtual environment because the simulation can also produce ground truth in addition to sensor data. This eliminates the need for time-consuming labeling of the data. It is also possible to simulate situations that would pose a danger to everyone on the road in the real world, such as disregarding traffic rules or even accidents. To keep the so-called Reality Gap [JHH95] to a minimum, the simulation must be as realistic as possible. As demonstrated in [Ric+16] (images) and [Des+21] (LiDAR), synthetic training data created using computer game technology can significantly improve the training of machine learning models. The digital twin can help other researchers through better visualization and provide them with novel perspectives on their data. Other stakeholders and the public can also experience the research in a more interactive and engaging way, which can lead to a better understanding of the research accomplishments.

1.3 Contribution

At the moment two different visualization tools are used within the project. On the project's website, the data of the digital twin is presented with the help of an application based on the *three.js* library. The rendering of the scene takes place on the client side, which reduces the amount of data to be transferred. The other one is based on the driving simulator CARLA, which in turn is based on the popular game engine Unreal Engine 4. It offers many existing high quality assets and therefore a high degree of realism. Furthermore, it can simulate different sensors like cameras and LiDARs and generate ground truth e.g. for semantic segmentation. Since the goal of this work is to create a digital twin with the highest possible degree of realism, the CARLA-based simulator was chosen.

Static environment The first part will show how the static environment of the A9 test field can be represented as accurately as possible within the game engine. This includes the creation of the terrain model based on an existing HD map as well as the modeling of buildings in the surrounding area.

Dynamic environment Some aspects of the environment like the light conditions and ground textures are constantly changing. This part will explore how these changes can be reflected in CARLA.

Detected road objects The second part is about the better representation of dynamically detected objects such as cars, trucks or pedestrians. This includes displaying them in the correct color and size.

Panorama stream One disadvantage of the CARLA simulation is the high demand on the required hardware and the necessity to install the simulation locally. Therefore this project will demonstrate how a 360 degree panorama video of the simulation can be produced and be streamed live.

Color detection To be able to display the colors of the vehicles, a color detection based on [Özl18] was integrated into the existing detection pipeline. The classifier was trained on the A9 dataset [Cre+22] and has been optimized for real-time detection. The classifier can detect colors with an accuracy of 42.7%.

Chapter 2

Related Work

2.1 Digital Twins

The term Digital Twin (DT) refers to a virtual representation of a physical object or system that can be used for a variety of purposes. While they are already becoming increasingly prevalent in research [Liu+21] and industry [Wor22], there is still no agreed definition of what exactly constitutes a digital twin. Due to the focus of this thesis, this chapter will mainly concentrate on the digital twins, which also have a virtual 3D representation of a real-world location commonly based on an existing game engine. A far more in depth review and classification of digital twins an be found in [Jon+20] and [VM21]. [Coe+21] provides an overview of DTs focused on urban areas. It proposes a general architecture for DTs and illustrates some areas of application such as air quality monitoring or flood protection. The city of Zurich provides 3D spatial data models of city features such as buildings to foster the development of novel applications which are supposed to address problems such as population growth [SH20].

With the help of NVIDIA, Deutsche Bahn is working on its own digital twin of its rail network based on existing HD maps and CAD models [Gey22]. This DT is supposed to be used to improve the efficiency of the existing network. The individual trains are connected to the digital twin via 5G and can warn staff and each other if an unexpected obstacle has been detected by one of the sensors in the trains. To improve this object detection, the DT is also supposed be used to generate synthetic training data for the machine learning applications. [Heg+22] shows the creation of a DT for a highway tunnel in the Netherlands. Based on Unity, the application contains a complete 3D model of the tunnel and it's access control system. The goal of the application is the validation of the control systems, which should prevent too large vehicles from entering the tunnel. It is also proposed that the application may be used for training as well.

In addition to video recordings from drones, the CitySim-Dataset [Zhe+22] also provides high resolution 3D maps for the locations where the video was taken. This allow for further testing and data generation using the CARLA simulator. A pipeline for creating a 3D digital model of a university campus based on geodata is described in [Azf+22]. The goal is to import terrain data and buildings into CARLA without much effort. Another pipeline describing how different sources of open geospatial data such as aerial imagery and digital terrain models can be integrated into a game engine was presented in [BZH22]. The paper from [WBE22] introduces a pipeline for creating 3D virtual environments from different data sources (OpenStreetMap, OpenDRIVE maps, CityGML, and digital terrain models) in an automated way. The low-fidelity models of building are enhanced by procedurally generating additional details like windows and facade textures. The user study concluded that the best way for participants to make connections between the real and the virtual environment was

through an accurate road geometry. X presents a whole simulation stack for autonomous vehicle using ROS, the Autoware Software Stack and CARLA. For this a small area of Ingolstadt was created in RoadRunner based on data from OpenStreeMap and subsequently imported into CARLA. The project SAVeNow[1] also created a digital twin of Ingolstadt, which can be seen in this video[2] . The same area around Garching Hochbrück, which is the focus of this thesis, was also modeled by [Cao+22] using only the outlines of the buildings provided by OpenStreetMap.

2.2 Simulators

[Epi] Unreal Engine 4 The most widespread engine is the Unreal Engine 4 , which stands out due to its realistic graphics and free availability. The newer version of this engine, Unreal Engine 5, has been available for some time and offers much more realistic graphics and better performance, but there is still no simulator based on this platform.

[Uni] Unity Another popular engine is Unity, which is a bit more beginner friendly compared to other engines. However, it is more difficult to achieve photorealistic graphics, which is why this engine has seen little adoption in the field of digital twins.

[KH04] Gazebo Gazebo is the primary simulation software used in combination with the popular robotic framework ROS [Sta18]. Therefore it is very easy to use gazebo to test applications that already use ROS. Due to the large community there are also a lot of plugins that add more sensors or other functionality. As a primary rendering engine gazebo uses OGRE3D [OGR], which also does not offer photorealistic graphics.

[UNI] UNIGINE 2 UNIGINE is a proprietary game engine developed primarily for applications such as simulations and digital twins. The engine offers photorealistic visuals, simulation of various sensors and other features that are particularly useful for digital twins. There is a free community version that can be used for academic and non-commercial projects, but it does not offer the full functionality [3].

[NVI] NVIDIA Omniverse NVIDIA Omniverse is a 3D simulation software created by NVIDIA, also focusing on digital twins and simulation. Due to the support of real-time ray tracing, the graphics are very realistic. NVIDIA offers further solutions for special application areas like NVIDIA DRIVE Sim which focuses on simulation of autonomous vehicles. The *Neural Reconstruction Engine*, based on [Mül+22a], offers the possibility to generate 3D meshes of the environment based on simple videos.

[Fou] O3DE Open 3D Engine is a fully open source 3D game engine targeting video games and simulations. The engine is still in an early stage of development and does not offer all functionalities compared to established engines like Unity or Unreal. Nevertheless, there are already plugins that provide a connection to ROS2, so that O3DE can already be used as simulation software.

When choosing a simulator, two aspects should be considered. The first is the availability of virtual sensors and the ability to generate ground truth data. A short overview of of such in open source simulation tools can be found in Table 2.1. The second aspect is the functionality available through the API, such as the spawning of models at runtime or a dynamic change

[1] https://savenow.de/
[2] https://www.youtube.com/watch?v=kQ_ns5JAX34&t=199s
[3] see comparison table at https://unigine.com/get-unigine/

of the weather. Another summary of simulators based on additional criteria can be found in [Loc20].

Name	Engine	RGB	LiDAR	3D BB	Segm.	Depth	DVS	Flow	Source
CARLA	UE4	✓	✓	✓	✓	✓	✓	✓	[Dos+17]
CiThruS2	UE4	✓	?	?	?	?	?	?	[GNV21]
AirSim	UE4	✓	✓	✓	✓	✓	✓	✓	[Sha+17]
UnrealROX+	UE4	✓	x	✓	✓	✓	x	x	[Mar+21]
UnrealCV	UE4	✓	x	x	✓	✓	x	x	[QY16]
NDDS	UE4	✓	?	✓	✓	✓	?	?	[To+18]
Gazebo	ogre2	✓	✓	✓	✓	✓	✓	x	[KH04]
O3DE ROS2	O3DE	✓	✓	?	x	x	x	x	[Rob]

Table 2.1: Comparison of offered virtual sensor in Open Source simulation platforms.

2.3 Related Games

While computer games are primarily designed to provide a fun gaming experience, some of them also provide a realistic representation of the real world. Especially games from the simulator genre are often developed to recreate real-life situations as detailed as possible. One example of such a game that specializes in simulating traffic scenarios is the Euro Truck Simulator [SCS12]. In this game, players can drive their truck along individual European highways that are roughly based on existing counterparts. Other games, such as Tram Simulator [Vie22] or CityDriver [Vie22], offer a truly detailed representation of the real world, although on a much smaller scope. The two games mentioned above, for example, depict a few selected places within the city of Munich. In stark contrast to this is the playable world of Microsoft Flight Simulator, which spans across the entire globe. Instead of having the entire game world built by human artists, most of the environment is automatically generated. This is done by combining satellite imagery from Bing Maps and vector data from OpenStreetMap to create an accurate three-dimensional representation of the world [Fue22]. Following the definition of a DT used in this chapter, the flight simulator can also be called a digital twin, since it displays live data such as the weather or air traffic data in real-time inside the game (Figure A.7) [Dev21].

Furthermore, games can also be helpful in generating large datasets for machine learning applications. For general object recognition models, any game with realistic graphics can be used. [Ric+16] presents a way to generate ground truth data from the game GTA V and shows that such synthetic data can significantly increase the accuracy of machine learning models. Since such AAA games are often developed with very high budgets, the models and textures they contain are of high quality. Other games like the Euro Truck Simulator have a very active modding community that offers additional high quality vehicle models as a free download [Ste].

2.4 Environment modeling

Some semi-automatic pipelines for importing buildings based on aerial photographs have already been described above [Azf+22]. However, in the context of the A9 test stretch and this thesis, these have several drawbacks, including outdated aerial imagery that does not contain all the buildings and a poor level of detail found in aerial imagery. Thus some buildings have to be created by hand in a 3D application like Blender. [Bra14] offers a comprehensive overview on alternative applications that can be used as well to create a detailed LOD3 building model from an existing lower resolution LOD2 model. A specification of the different levels of detail can be found in [BLS16]

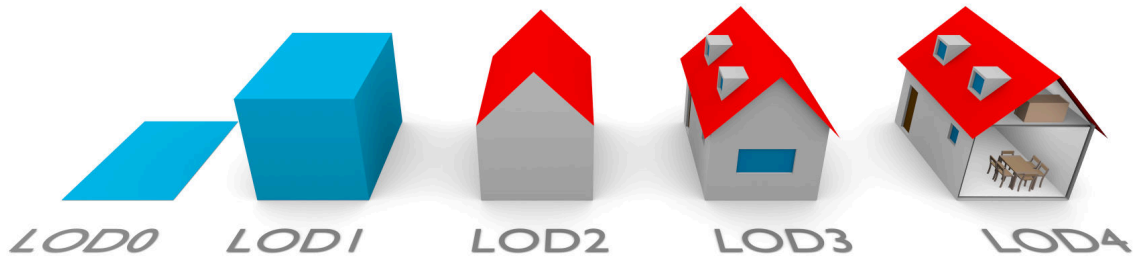


Figure 2.1: The five LODs of CityGML 2.0 from [BLS16].

2.5 Model scaling

In order to display the vehicles in the correct size, they have to be scaled at run time. If the deviation between the size of the original model and the desired size is too large, significant deformations could occur. This would be recognizable, for instance, by the tires, which would no longer be circular. By analyzing the vulnerability of surfaces [Kra+08] builds a protective grid around the model which allows for scaling of a complex model without destroying important features. The expensive calculation of the vulnerability map has to be done only once for each model, it can then be used for any scaling factors. More recent work like [She+22] use a neural network to learn how much influence a control point has over a vertex in the model which should be scaled.

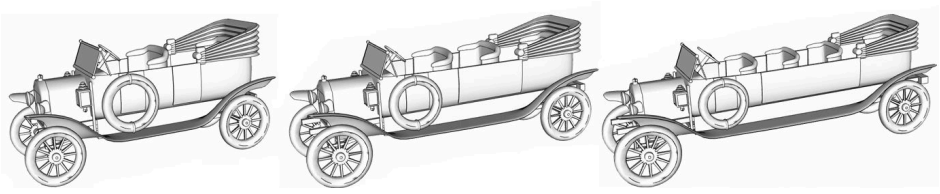


Figure 2.2: Car scaled by the method described in [Kra+08]. Seats were added manually.

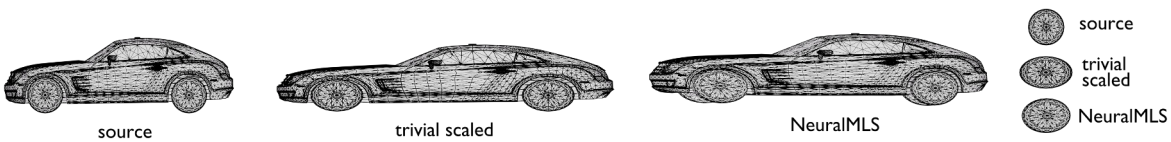


Figure 2.3: Comparison between homogeneous scaling and NeuralMLS.

Chapter 3

Background

3.1 CARLA

CARLA is a free and open source simulation platform that allows users to create and test autonomous vehicle systems in a virtual environment. CARLA builds on top of the Unreal Engine 4 [Epi] and provides additional functionality for simulating autonomous vehicles, such as the ability to customize vehicle models and sensor configurations and offers a selection of high quality assets (cars, pedestrians, props etc.). It uses a client-server architecture, which can be distributed across multiple computers.

Server The server runs the Unreal game and the simulation within. This includes the physics calculations and rendering. Since this can be quite resource hungry, especially for a high image quality, this component should be run on a powerful computer with sufficient GPU performance.

Client One or more clients can control the simulation, such as spawning new objects in the world or changing the weather parameters. The clients can receive the data from the virtual sensors and process them further. There are official APIs for both Python and C++.

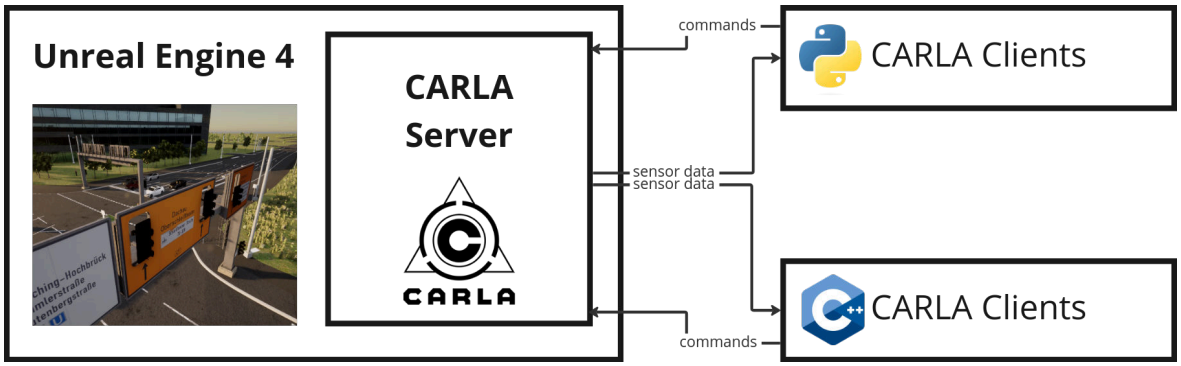


Figure 3.1: High-level overview of a system using CARLA.

3.2 Map Projections

In order to display geo-referenced data in a game engine, the coordinates, given in latitude and longitude, must first be converted into a Cartesian coordinate system. The problem is

that the coordinates describe a point on a three dimensional spherical surface that must be projected onto a two dimensional map. From a geometrical point of view there are three basic types of projection surfaces (fig. 3.2): planes (accurate for polar regions), cones (distortions are constant along parallels) and cylinders (accurate at the equator). Based on this, there are many different map projections available, each with its own set of characteristics and trade-offs.

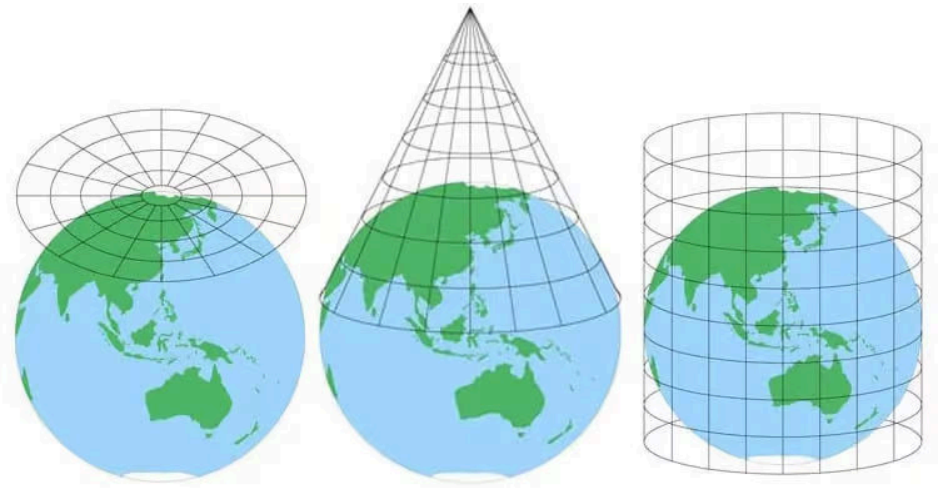


Figure 3.2: Map projection surfaces [Map20]

Equidistant Cylindrical (Plate Carrée) A cylindrical projection in which angles are preserved, but surfaces and shapes are distorted. The Plate Carrée is often used in scientific research and data visualization because of its simplicity and ease of use. It is also utilized for 360-degree videos.

Transverse Mercator (Gauss-Krüger) A cylindrical projection in which shape and direction are preserved, but areas are distorted. It is widely used in surveying, navigation, and mapping.

UTM Divides the Earth into 60 zones, each 6 degrees of longitude wide. Each zone is based on the Transverse Mercator projection and uses a different central meridian, ensuring minimal distortion within that specific zone.

(a) Equidistant Cylindrical **(b)** Transverse Mercator **(c)** UTM Zones

Figure 3.3: Comparison of map projection types [Graphics from https://proj.org/].

It is not possible to accurately represent the curved surface of the earth on a flat surface without distorting some aspect of the map, such as area, shape, distance, or direction. Since there is no perfect solution for the choice of the projection, it is only important that all programs within the project use the same projection.

3.3 Vehicle Color Detection

The color classification algorithm introduced by [Özl18] uses histograms and the K-Nearest Neighbors (KNN) algorithm to classify the primary color in an image. The algorithm starts by clustering the image using K-Means to create a segmented image. Based on this image, a histogram for each color channel is calculated, which represents the distribution of colors within the image. The peak pixel values of these histograms are then used as features in the KNN algorithm, which compares the histograms of the image to a set of histograms acquired during training and assigns the color of its closest match.

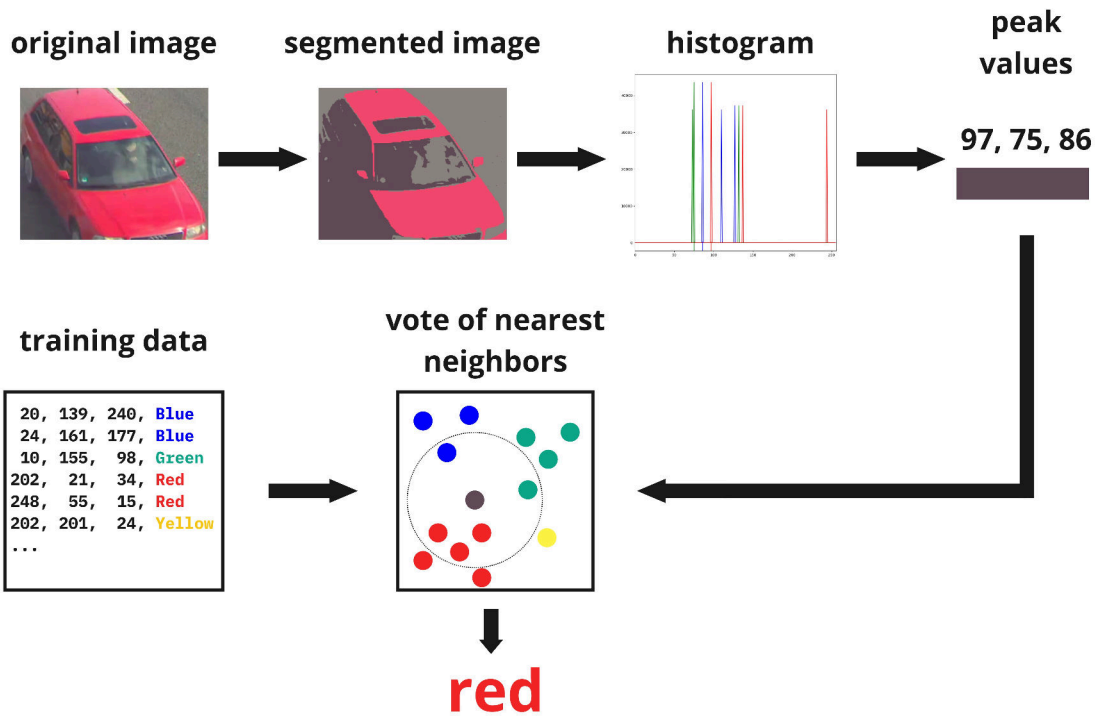


Figure 3.4: Color classification pipeline.

Chapter 4

Implementation Details

4.1 Static Environment Modeling

4.1.1 Full Pipeline

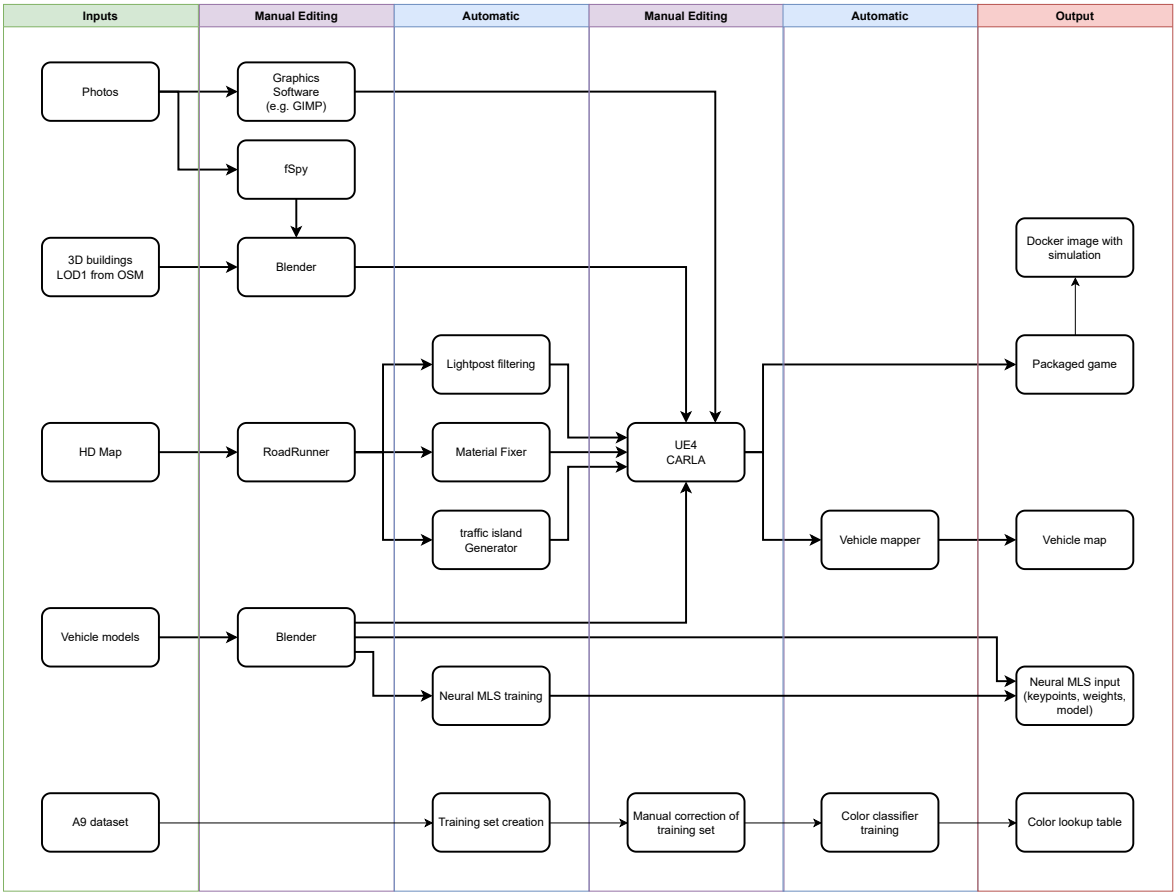


Figure 4.1: Overview of the full creation pipeline.

The following chapter describes the creation process of the digital twin in detail. The goal of the creation process for the DT is to recreate the real world on site as closely as possible in a virtual environment. For this, a variety of tools come into play that can either process information about the environment automatically or help a user to recreate the environment manually. The top part of the diagram, concerning the creation of textures and building models based on the reference image, will be explained in Sections 4.1.2 and 4.1.3 respectively.

The creation of the virtual environment based on an existing HD map, shown in the centre of the diagram, will be dealt with in Section 4.1.4 for the map in general and Section 4.1.6 for the traffic island generation. The procedure for creating and rigging the vehicle models is entirely based on the official instructions[1] and is not covered in this thesis. Neither will the training of weights for use with NeuralMLS, which also simply follows the existing example code in the official repository[2]. At the end of this chapter, the training of the color recognition and the runtime optimizations are discussed.

4.1.2 Texture Creation

Without textures, the game environments would look flat and uninteresting, which would make the game world look much less convincing. In addition, important details, such as the writing on traffic signs or the rough surface of the road would be completely missing. That is why the use of high quality textures in games is essential.

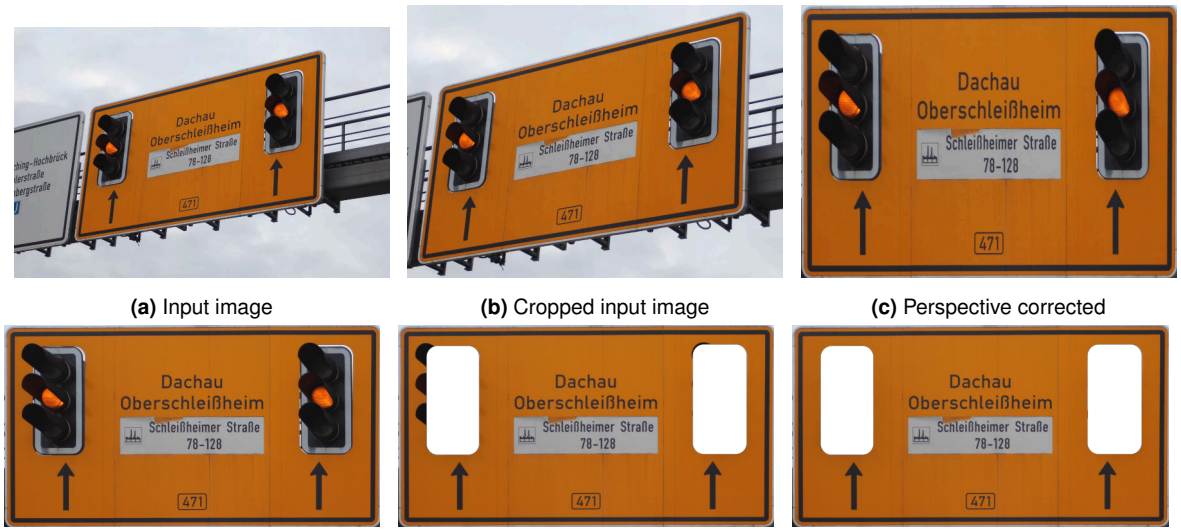
Since all games are confronted with this problem, not every single texture had to be created from scratch for this thesis. CARLA itself offers a good selection of important textures and materials like road surface, grass or reflective window glass. There is also the Unreal marketplaces and a big number of websites that offer further textures for download, some of them completely free of charge. For this work some textures and assets (see Figure A.8) from the site ambientcg[3]. were used, which are all available under the Creative Commons CC0 1.0 Universal License. However, some textures, such as the lettering of traffic signs, are very location-specific and had to be specially created for this work.

The first step in this process is to take high resolution photos of the signs to be reproduced later in the textures. Cloudy days are particularly suitable for this, as on these days the lighting is consistent and no harsh shadows are present. This simplifies the post-processing of the images, since such lighting effects have to be removed from the images. Furthermore, care should be taken to look as straight as possible at the signs in order to minimize perspective distortions. Special attention should be paid to the protruding traffic lights, which should not obscure any writing or symbols. For the post-processing of the images, the free open source graphics software GIMP was chosen, but the required functionality should be available in any other commonly used program.

If there are several signs on one image, they should be cut out and processed individually. The next step is to correct the perspective deformation of the images, which can be accomplished with the perspective tool (Shift+P) in GIMP. To do that, the grid lines must be dragged to the edges of the sign (Figure A.9). The direction must also be set to Corrective (Backward) in the tool options (Figure A.10a).

After applying the transformation, it is possible that the proportions in the image are not quite correct. This can be checked with the help of the pictograms, which should be square. They can be selected using the rectangle selection tool and their aspect ratio can then be seen in the status bar. The image should then be scaled (Image -> Scale Image) to correct this aspect ratio. The holes for the traffic lights can be cut out using the Rectangle select tool, the corners of which should be rounded accordingly (Figure A.10b). A layer mask can be created from the holes, so that they will not be filled again in the subsequent steps. Now the overhanging sunblinds of the traffic lights and their shadows have to be removed. For this other areas of the image can be cut out and placed on top of it. As an alternative the clone tool can be used to copy the texture from nearby places. As a last step, the white balance of the texture can be corrected, if this should be necessary. An example can be seen in Figure 4.2

[1] https://carla.readthedocs.io/en/0.9.14/tuto_content_authoring_vehicles/
[2] https://github.com/MeitarShechter/NeuralMLS
[3] https://ambientcg.com/



(a) Input image **(b)** Cropped input image **(c)** Perspective corrected

(d) Fixed aspect ratio **(e)** Removed traffic lights **(f)** Final texture

Figure 4.2: Steps of the texture creation process

4.1.3 Building Modelling

Similar to the textures, there are also some unique buildings in the area that should be represented by the digital twin. For the modeling, there is a choice between the classic, manual creation of the 3D models or the automatic creation with the help of aerial and drone photography. Although a very detailed model can be created through such automatic generation, the whole process is also much more complex. In order to render the models in real-time in Unreal Engine 4, the created models would also have to be simplified to reduce the number of polygons. Since the intended buildings all have a rather modern architectural style, they consist mainly of simple geometric shapes with many straight lines. As such buildings are still quite simple to create, it was decided for this work to create them manually. If it is desired to create buildings with more details, like ornaments on facades within a city center, processes like photogrammetry might be the better choice instead. More information about this process will be provided in the outlook chapter.

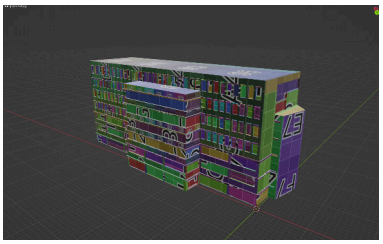
It is neither possible nor intended to provide a detailed and all-encompassing guide to the creation of such 3D models in this thesis. However, a rough process is described below, which can be used as a possible starting point.

The process starts with the creation of images that show the entire building in one picture. The image should also contain enough lines that are parallel in reality but converge in the image itself. These lines are needed to reconstruct the camera's perspective later in *fSpy*[4]. The perspective can then be controlled in *fSpy* by moving the axes or one of the 3D guides. The saved *fSpy* project can then be imported into the 3D software using the *fSpy* Blender plugin. This sets the calculated camera perspective and inserts the image as a reference image in Blender, which can be viewed by switching to the camera perspective. Since the correct size cannot be calculated from a single image, some additional information is needed. Fortunately, *OpenStreetMap* has the correct outlines of most buildings that can be used as a reference. The simplest way is to use the *blender-osm* to load all buildings in the area into a Blender project and then copy the desired building into the project for the corresponding building. If this is not possible because the building is not yet included in *OpenStreetMap* or a different object than a building is to be modeled, a known length within the image can be

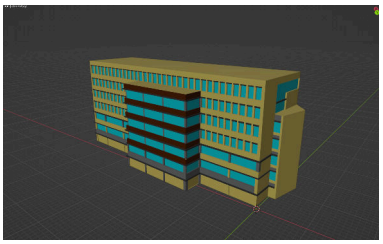
[4]https://fspy.io/

used as a reference. The reference should then be moved in Blender until the actual edge lengths match those in the reference image. For buildings with simple windows without much decoration, it is sufficient to cut the recesses of the windows into the outline using loop cuts. For more complex windows, the *archi-mesh* plugin included in Blender can be used to create detailed windows. Depending on the proximity of the building to the cameras in the digital twin, further details of the building should then be modeled. A correct UV-unwrapping is important to be able to put textures on the building correctly. Texturing does not necessarily have to be done in Blender, but the different materials (e.g. wood, wall, glass) have to be created as dummies in Blender to be able to assign the correct material to the corresponding faces.

(a) Building created just using loop cuts



(b) UV mapping of the building shown using a color grid texture



(c) Dummy materials assigned to the faces

4.1.4 HD Map Conversion in RoadRunner

In this project, the conversion of the HD map into a 3D model was done with the help of *RoadRunner*, a commercial software from *MathWorks* [5]. It allows the automatic generation of most of the map like the terrain, roads and traffic signs. The interactive editor still offers the possibility to correct errors that occur during the generation and to add additional details that are not included in the HD map.

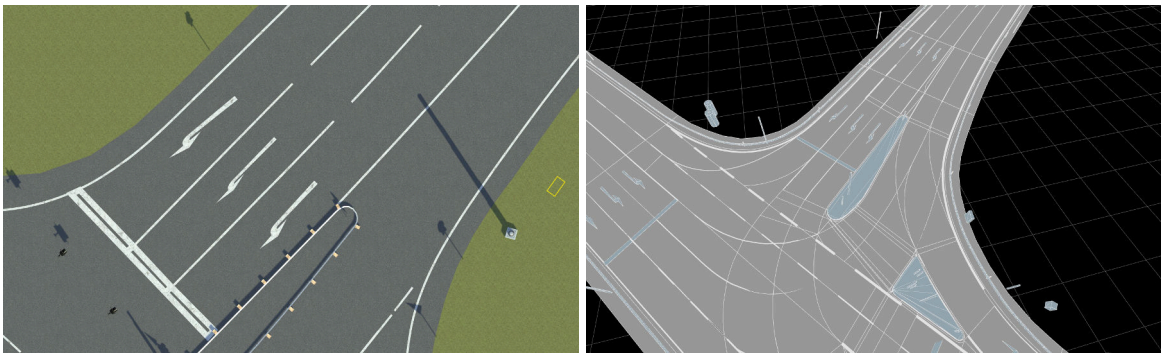
One such error that could be observed was the representation of directional arrows on the ground, which all pointed to the left in *RoadRunner* (Figure 4.4a). These arrows were pointing in the wrong direction, despite the information in the underlying high-definition map being correct. Other software, like the browser based ODR viewer[6], show the correct arrows (Figure 4.4b). This error was solved by simply editing the affected arrows within the vicinity of a relevant intersection (S110).

Furthermore, the mapping of object IDs in the HD map to textures was configured incorrectly, which resulted in a number of places where traffic signs were either completely missing or American signs were being displayed. The mapping, which is used for the generation, can be found under the menu item `assets/asset mapping`. The missing IDs, which were obtained from the error messages during the generation, can be entered in the signals tab. Most textures for German signs were already included in the used RoadRunner version and could therefore be linked directly. Some more special variations of traffic signs (e.g. variation of no parking signs) had to be created first, using the public domain graphics from Wikipedia.

The HD map lacked detailed information about the traffic islands and green stripes between the driving lanes around the S110 intersection. These were then created manually with the help of the traffic islands tool in RoadRunner. Since these are not yet exported as a mesh, an extra workaround had to be found, which will be explained in the next chapter.

Further parts that were added or edited manually:

[5] https://de.mathworks.com/products/roadrunner.html
[6] https://odrviewer.io/



(a) Incorrect arrows displayed by RoadRunner **(b)** Correct arrows displayed using the ODR viewer

Figure 4.4: Ground arrows displayed in different programs.

- Ground markings at the pedestrian crossings
- Noise barriers and other height differences of the terrain, especially near the highway and the bridge
- Bollards at the corners of buildings to be able to place the manually created models in the right positions later.

(a) Ground markings at the northern end of the S110 intersection **(b)** Noise barrier next to an exit of the A9 highway **(c)** Bollards placed to mark the corners of buildings

Figure 4.5: Map elements which were edited manually.

4.1.5 Map Import

In order to be able to transfer the model to CARLA, RoadRunner offers an extra option for its export. In addition to the model in `.fbx` format, a new HD map with the changes made is also exported. These files can then be imported into the Unreal Engine project using the import script described in the CARLA documentation[7]. However, when running this script, there were some small problems that had to be solved in order to successfully import the map into CARLA.

The first of these were segmentation faults within the XODR parser of CARLA, which is needed in several places during the import. These were due to apparently faulty nodes in the HD map. The parser (`LibCarla/source/carla/road/MapBuilder.cpp`) was then modified in such a way that it skips the corresponding lanes, which do not have the expected structure, during some steps. At a later point during the execution of the script, crashes due to array out of bounds exceptions during the lookup of materials occurred. These could be traced back to faulty materials in the FBX file of the map, which also occurred when opening

[7] https://carla.readthedocs.io/en/0.9.14/tuto_content_authoring_maps/#importing-your-road-network-into-carla

the file in other 3D programs such as Blender. Blender, however, could handle these broken materials and simply removed them. For this reason a script (`fix_fbx.py`) based on the Python API of Blender was developed, which loads the FBX files into Blender and exports them again without any further modifications. This exported version then only contains intact materials and can be imported into the Unreal Engine without any problems.

The last problem was the absence of some objects like lamp posts and street signs in the imported variant of the map. Due to the previous step, a version of the map imported into Blender already existed that still contained these objects. Another script (`filter_wanted.py`) based on the Blender API filtered out these desired objects, which were then saved in an extra FBX and Blender file. The FBX file could then be imported via the normal import functionality of the Unreal Engine and the objects can then be added to the world. The markers contained in the Blender file for objects such as the gantries were also used for the correct placement of their 3D models, which were then also added to the project via the normal import.

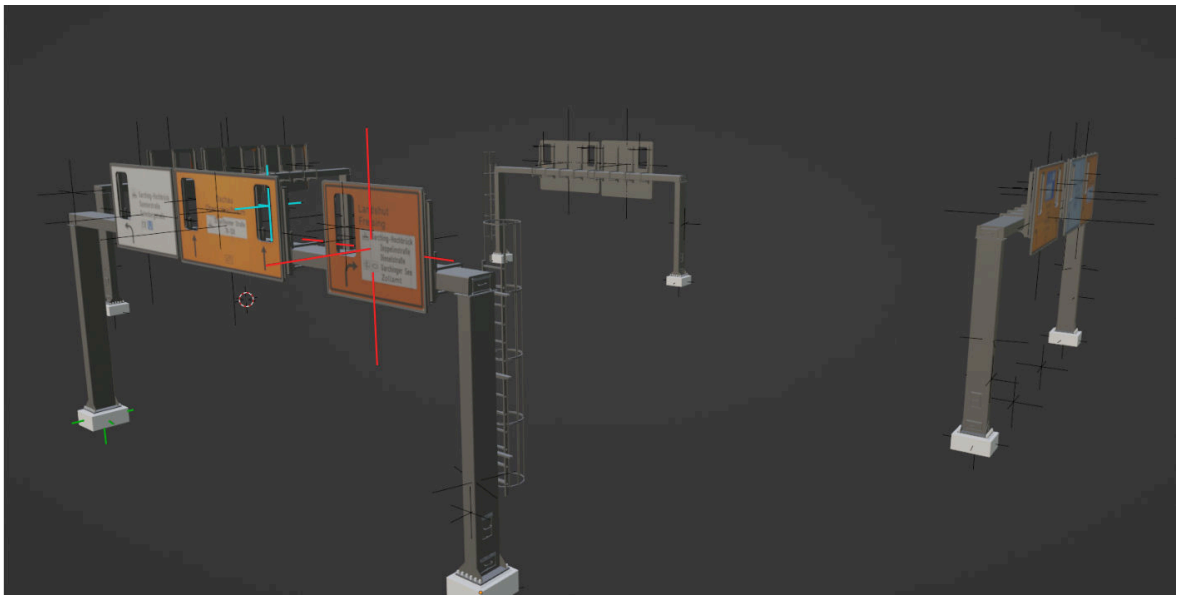


Figure 4.6: Models of the gantries with the markers showing the correct positions of e.g. signs (red), traffic lights (cyan) or gantries (green).

When importing, it is important to ensure that a certain folder structure is adhered to in order to obtain the correct semantic tags for the relevant sensors. For example, buildings must be placed in the Static/Building folder (relative to the imported package) in order to receive the building class during segmentation. The same applies to vehicles, whose skeletal mesh must be stored e.g. under Static/Bus or Static/Car.

4.1.6 Traffic Island Generation

Since the creation of traffic islands in RoadRunner is still a very new feature, it is currently not yet possible to export them directly from there as a mesh. However, they are included in the exported HD map and can be created based on that. The traffic islands are objects within a road element with the type `trafficIsland` and store the outline in road coordinates (Figure 4.8a). These are transformed into global coordinates using the libOpenDRIVE library. For each of these points on the ground, three more vertices are added to the model, forming the upper and inner edges of the traffic island respectively (fig. 4.8b). These are shifted a little towards the centre of the islands, which is controlled by the $i1$ and $i2$ parameters in

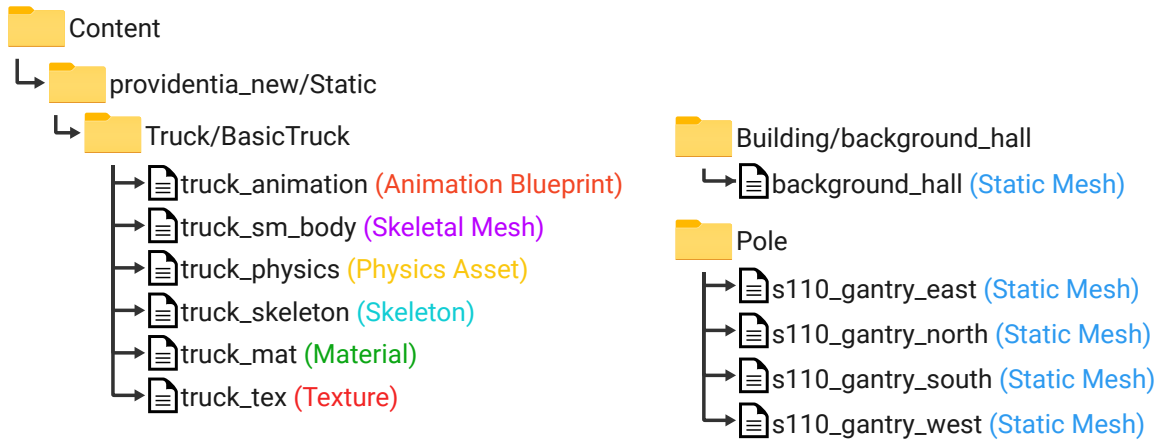


Figure 4.7: Excerpt of the folder structure within the UnrealEngine project

Equations 4.1 and 4.2.

$$B_n = (1 - \frac{i_1}{|A_n|}) \cdot A_n + \begin{pmatrix} 0 \\ 0 \\ h \end{pmatrix} \tag{4.1}$$

$$C_n = (1 - \frac{i_2}{|A_n|}) \cdot A_n + \begin{pmatrix} 0 \\ 0 \\ h \end{pmatrix} \tag{4.2}$$

$$D_n = C_n - \begin{pmatrix} 0 \\ 0 \\ 0.05 \end{pmatrix} \tag{4.3}$$

They are added to the index array of the model in this order, so the points based on the first outline-point have a vertex index of 1-4 (vertex indices in the obj file format start with 1).

$$POINTS = [A_1, B_1, C_1, D1, A_2, B_2, C_2, D_2, ..., A_n, B_n, C_n, D_n] \tag{4.4}$$

The faces then simply connect the adjacent points to form the outer and upper surfaces. The bottom of the island simply connects all the outline points, just as the inner patch (the patch that is covered with vegetation in real life) connects the innermost points. The program itself does not produce triangle faces, because the models have to be converted to a FBX file anyway (again with the help of Blender). Furthermore, the triangulation can be done automatically during this conversion.

$$f_{Base} = [4i | i < n] \tag{4.5}$$

$$f_{Mid} = [4i + 3 | i < n] \tag{4.6}$$

$$F_{Outer} = \{[4i + 1, 4i, 4(i + 1), 4(i + 1) + 1] | i < n\} \tag{4.7}$$

$$F_{Top} = \{[4i + 1, 4(i + 1) + 1, 4(i + 1) + 2, 4i + 2] | i < n\} \tag{4.8}$$

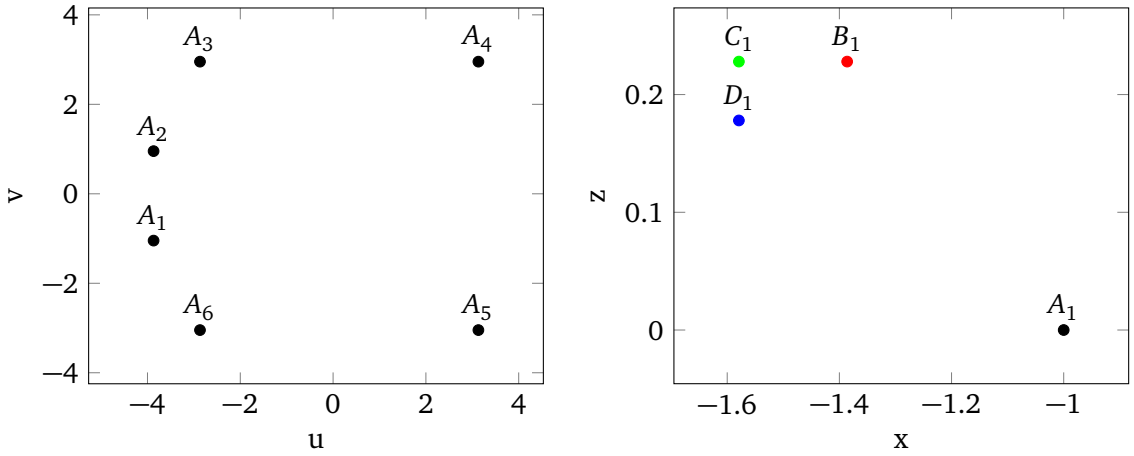
$$F_{Inner} = \{[4i+2, 4(i+1)+2, 4(i+1)+3, 4i+3] | i < n\} \tag{4.9}$$

$$F_{All} = \{f_{Base}, f_{Mid}\} \cup F_{Outer} \cup F_{Top} \cup F_{Inner} \tag{4.10}$$

In order to be able to put the correct stone and grass textures on the model later on, two dummy materials are created and the faces are assigned accordingly. The resulting model can then, like most other models, be imported with the help of the normal Unreal import functionality. Once again, care must be taken to ensure that the folder structure is correct.

$$F_{tex\_outer} = \{f_{Base},\} \cup F_{Outer} \cup F_{Top} \cup F_{Inner} \tag{4.11}$$

$$F_{tex\_grass} = \{f_{Mid}\} \tag{4.12}$$



(a) Example outline points for a simple traffic island

(b) Cross-section showing the vertices based on one point of the outline

(c) 3D view of a traffic island

Figure 4.8: Visualization of an exemplary traffic island.

4.1.7 Sensor Coordinates Systems

As mentioned in the introduction to map projections, it is important that all programs involved in working with geo-referenced data use the same projection. By default, RoadRunner uses a *Transverse Mercator* projection with the projection center placed on the middle of the HD map. This minimizes the distortions that would grow with increasing distance from this center. This could be the case, for example, if the conversion is done using the UTM system, which would result in a significant deviation of the positions (Figure 4.9).

The exact parameters of the projection are included in the exported HD map and can be used with the help of the open source library *proj*[8]. The positions of the sensors can then be stored in geo-referenced coordinates (longitude and latitude) and can be converted into in-game coordinates with the help of the appropriate projection. This allows for an accurate calculation of the x and y coordinates within the CARLA map. For the z position, an offset of 2.78m is applied, obtained by comparing the intended and actual z position of the S110 base station (more precisely, the elevation of the road there). Since the geo-referenced coordinates and the measured rotations of the sensors are not always completely correct, there is also the option of specifying a manual offset for the position and the rotations. This makes it possible to calibrate the sensors in such a way that the resulting measurements are similar to those of the real sensors.

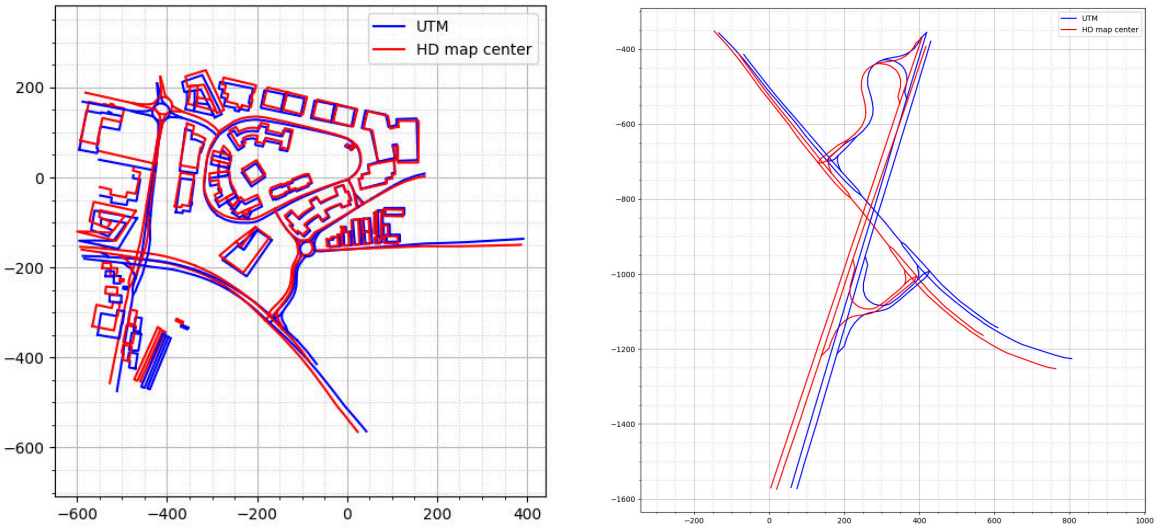


Figure 4.9: Deviation caused by different map projections.

4.1.8 Color Detection Training

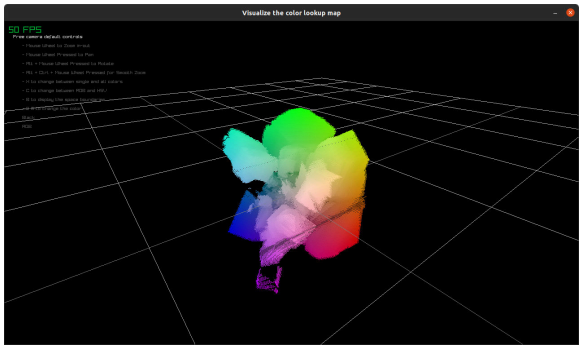
Since every camera displays the color slightly different, the color recognition system should also be trained for the cameras used in this project. Parts of the A9 dataset [Cre+22] already contain labels for the correct colors of the labeled vehicles, so that these could then also be used as a basis for the training. In the dataset, the colors black, blue, grey, green, red and white were labeled, which was manually extended by the color yellow for the training set (these vehicles were classified as other in the original dataset). With these colors, about 95% of the vehicles that the system will get to encounter should be covered [21].

For the training, the pictures of the vehicles in the dataset are cropped and sorted into

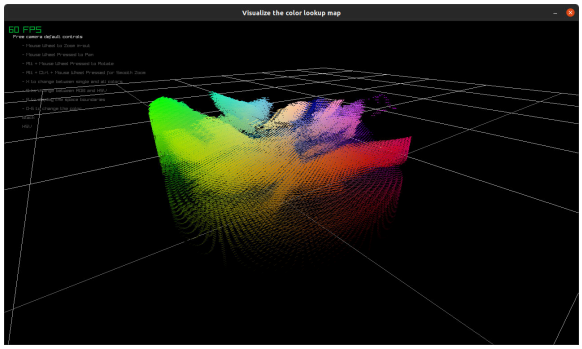
[8] https://proj.org/

different folders according to their color. Some pictures were removed manually, because, among other things, their quality was too low or the assignment to a color was ambiguous.

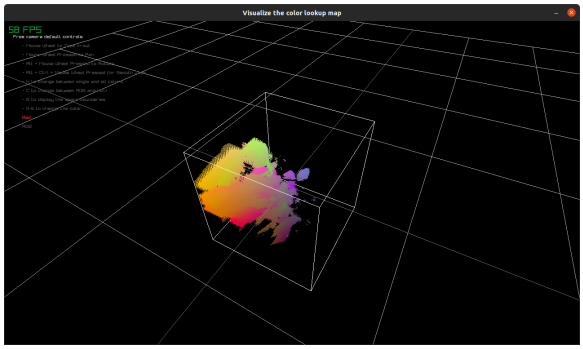
The procedure described in Section 3.3 was then applied to each image in the training set and the resulting RGB values are saved together with the label in a `.csv` file. These points are then used as the basis for the nearest neighbor algorithm at runtime. However, due to the high number of data points, one for each of the up to 30,000 training images, the performance of this procedure drops. To counteract this, the calculation for the runtime is instead replaced by a suitable lookup table. This table contains the corresponding color classification for each of the approximately 16.8 million RGB values contained in the 24-bit RGB color space. This also enables an intuitive visualization of the training result by displaying all points within the color space that are assigned to a certain class. For example, the graphic in Figure 4.10 shows that a surprisingly large number of colors are classified as black, because the windows and tires occupy a significant part of the image in many pictures.



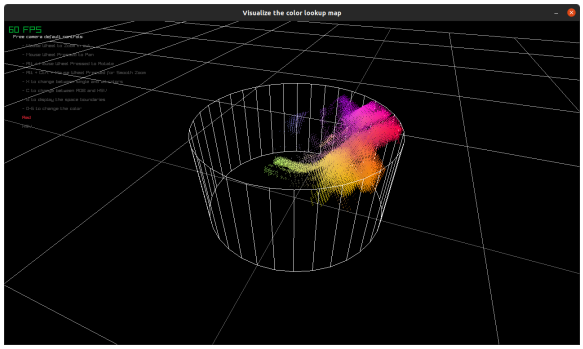
(a) Colors classified as black in the RGB space



(b) Colors classified as black in the HSV space



(c) Colors classified as red in the RGB space



(d) Colors classified as red in the HSV space

Figure 4.10: Visualization of the color training

4.2 Dynamic Behavior

4.2.1 Panorama Client

To create a panorama picture, several images have to be combined into a single image so that it can be projected onto a sphere at the end. This is basically the same problem that is solved by map projections, which is why these equations are also used for panorama pictures. The selected *JavaScript* library *Pannellum* [Pet19] for displaying panorama pictures best supports the equirectangular projection, so that it is used in this thesis as well. The method for creating a single equirectangular image by merging multiple pictures is achieved by determining the origin of each pixel in the final image from the individual captured images. Since the input images are purely synthetic and therefore the rotation is perfect and there are no other distortions in the images, no blending is required. Instead, the projection of the pixel is made for each camera and the first return value, which is actually in the range of the captured image, is selected. These calculations do not have to be made from scratch for each image, since they only depend on the rotation, the field of view and the resolution of the images. Thus, at the start of the program (or even at compile time), the pixel whose color value is to be copied can be calculated for each pixel in the output image. The images of the virtual camera are all written to a one dimensional buffer, each offset by $w_{img} \cdot h_{img} \cdot 4$. When creating the projection map, only the index into this input array has to be stored, indicating which pixel values should be read out regularly. This enables a frame rate of about 17 FPS.

In order to calculate these projections, the parameters (resolution and fields of view) of all images first must be known.

$$i = (w, h, \text{fov})^\intercal \in \mathbb{Z}^2 \times \mathbb{R} \tag{4.13}$$

Since the goal is to create a 360 degree panorama, 360x180 degrees is also chosen as the field of view for the output image. Of course, the field of view for the input images should be as large as possible in order to reduce the number of cameras and the associated performance costs. With values of more than 90 degrees field of view, however, visual defects occurred frequently in CARLA, especially with shadows, which is why this limit value was chosen in both dimensions. Thus, a total of six cameras, arranged according to the sides of a cube, are needed to be able to cover the entire field of view. for each of these cameras a projection function can now be defined which projects the position of a pixel in the equirectangular image into its own image.

$$f((x_{in}, y_{in}), i_p, i_c) : \mathbb{Z}^2 \times (\mathbb{Z}^2 \times \mathbb{R}) \times (\mathbb{Z}^2 \times \mathbb{R}) \mapsto \mathbb{N}^2 \tag{4.14}$$

As a first step, the image must first be centered by moving the origin of the coordinates to the middle of the image. Therefore, half of the width (w_p) or height (h_p) of the panorama image is subtracted from each of the coordinates.

$$x_1 = x_{in} - \frac{w_p}{2} \qquad y_1 = y_{in} - \frac{h_p}{2} \tag{4.15}$$

Then the yaw of the camera is reversed by shifting the image along the x-axis:

$$\Delta x = \frac{yaw \mod 2\pi}{2\pi} \cdot w_p \tag{4.16}$$

$$x_2 = x_1 - \Delta x \pmod{w_p} \qquad y_2 = y_1 \tag{4.17}$$

With the help of the reverse projection equation for the equirectangular projection, the latitude and longitude of the pixel can now be calculated. The radius of the sphere on which the image is projected scales with the resolution of the output image. Since the image has already been centred by the previous steps, most parameters like the central meridian (λ_0) can be set to 0.

$$R = \frac{w_p}{\text{fov}_p} = \frac{w_p}{2\pi} \tag{4.18}$$

$$\lambda = \frac{x_2}{R\cos\varphi_1} + \lambda_0 = \frac{x_2}{R\cos 0} + 0 = \frac{x_2}{R} \qquad \varphi = \frac{-y_2}{R} + \varphi_0 = \frac{-y_2}{R} + \frac{\pi}{2} \tag{4.19}$$

The resulting spherical coordinates must now be converted into the Cartesian coordinate system for further calculation.

$$x_{cart} = \sin\varphi \cdot \cos\lambda \qquad y_{cart} = \cos\varphi \qquad z_{cart} = \sin\varphi \cdot \cos\lambda \tag{4.20}$$

These can now be rotated to incorporate the pitch of the camera. In this step it would also be possible to take the roll of the camera into account, but this is not necessary with the chosen camera configuration.

$$\begin{pmatrix} x_3 \\ y_3 \\ z_3 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(-pitch) & \sin(-pitch) \\ 0 & \cos(-pitch) & -\sin -pitch \end{bmatrix} \begin{pmatrix} x_{cart} \\ y_{cart} \\ z_{cart} \end{pmatrix} \tag{4.21}$$

Now the three dimensional point can be projected onto the two dimensional image plane. The equations needed for this are adapted from the *libpano13*[9] library.

$$r_2 = \sqrt{x_3^2 + y_3^2} \qquad \begin{pmatrix} x_4 \\ y_4 \end{pmatrix} = \frac{atan2(r_2, z_3)}{r_2} \begin{pmatrix} x_3 \\ y_3 \end{pmatrix} \tag{4.22}$$

$$\theta = \sqrt{x_4^2 + y_4^2} \qquad \begin{pmatrix} x_5 \\ y_5 \end{pmatrix} = R\frac{\tan\theta}{\theta} \begin{pmatrix} x_4 \\ y_4 \end{pmatrix} \tag{4.23}$$

Finally, the coordinate must be correctly scaled and shifted so that the origin of the image lies in the upper left corner, as usual.

$$s = \frac{w_c}{2\frac{w_p}{\pi}\tan\frac{\text{fov}_c}{2}} \qquad x_{out} = x_5 \cdot s + \frac{w_c}{2} \qquad y_{out} = y_5 \cdot s + \frac{h_c}{2} \tag{4.24}$$

A visualization of these steps and a result can bee seen in Figure 4.11.

4.2.2 Weather

Having realistic lighting and weather conditions is important to enhance the overall accuracy and realism of the digital twin. This is because the way light behaves in the real world can have a significant impact on how objects appear and the overall atmosphere of a scene.

[9] https://panotools.sourceforge.net/

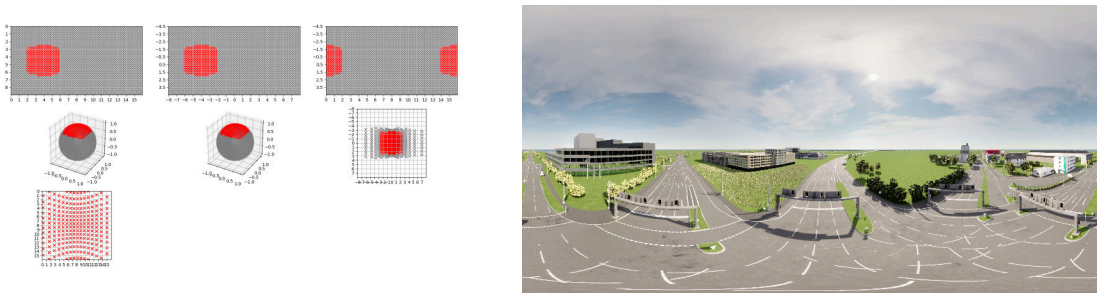


Figure 4.11: Steps to create a panorama image (left, pitch=0, yaw=270) and example of a panorama image over the S110 intersection (right).

Objects like street signs and the gantries throw distinct shadows on the ground which can change the output of detection algorithms significantly (i.e. by adding additional hard edges which would be detected by edge detection algorithms). So in order to be able to verify and test such programs with virtual data, such factors must also be part of the virtual twin. There are also many other characteristics of the camera such as white balance, lens distortion, chromatic aberration etc. that can affect the final image. However, these are not covered in this thesis.

To replicate the direction and length of the shadows as realistically as possible, the correct position of the sun at the desired time has to be calculated. More precisely, the azimuth (between 0 and 360 degrees) and the altitude (between -90 and 90 degrees) angle of the sun must be calculated, as the weather parameters of the CARLA API offer these values as a parameter. For this purpose the Python library *suncalc* [Bar] is used, which in turn is based on the *JavaScript* library of the same name [Aga23]. It offers the function `get_position()` with which the two required parameters can be calculated depending on the position (latitude and longitude) and the time. Because of differences in the coordinate system, the azimuth angle for CARLA has to be rotated by -90 degrees. A detailed explanation of the equations used for the calculation can be found in [Str21]. However, only calculating the sun's position correctly is not quite enough to be able to display the weather conditions accurately. CARLA offers a few more options for setting the weather, such as the degree of cloudiness. There are also preconfigured presets such as `ClearNoon` or `WetNight` that already set these parameters to suitable values. In order to set this, a parameter called preset can be passed to the relevant method (`DynamicEnvironmentController.apply`), which is then used as a base preset. This value could be adjusted based on real time weather data as well.

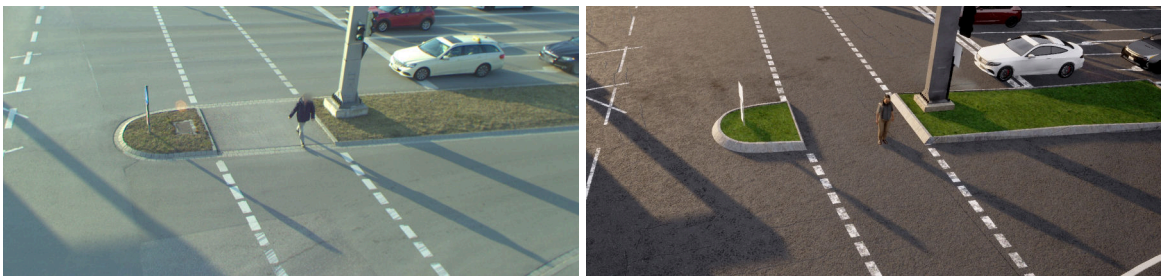


Figure 4.12: Comparison of the shadows between the real image (left) and an image generated with the CARLA simulation (right)

4.2.3 Dynamic Textures

Similar to light and weather conditions, the surfaces of the environment can also change over time. These can be, for example, weather-related changes such as water and snow, or dynamic traffic signs. Puddles are already visualized by CARLA and can be set via the weather parameters, which is why they do not need to be explained here. Since there are no dynamic traffic signs at the S110 intersection, the principle is explained using a clock that is located on one of the buildings in the background.

The clock to be displayed is a large seven-segment display which can alternate between showing the current time and the date. The clock has four digits with seven segments each, a colon and a decimal point. Therefore there are 31 segments (the two parts of the colon can be controlled separately) which can either be switched on or off. With the help of a seven-segment graphic from wikipedia, the general layout of the clock can be created. Then each of the segments can be exported as individual image in the active (red) and inactive (grey) state. These images all have the size of full clock texture but just contain a single segment with an otherwise transparent background. To create a texture at runtime, it is only necessary to determine which segment is to be active and which is to be inactive for each digit. The corresponding images are then simply summed up to obtain the final texture. Using the `apply_float_color_texture_to_objects()` function of CARLA, this texture can then be transferred to the server. There, the created texture is then passed as a texture parameter to the associated material of the clock model.

Figure 4.13: Clock on top of a building

This approach of calculating the texture completely in the client and then transferring it offers a great deal of flexibility, as any textures can be displayed without having to rebuild and export the game every time. However, the transfer of a complete image is quite expensive, since a `carla.Color` object has to be created for each pixel of the image. With higher resolution textures, the creation and transfer of a texture may take several seconds. Another disadvantage is that the client only has a fraction of the information that is available to a material in the Unreal Engine. For example, the client has no access to the UV coordinates, the normal vector or the absolute world coordinates of a pixel. Since these parameters are very useful in the creation of a material, creating a texture on the client side is not always the best idea.

Nevertheless, the `apply_color_texture_to_objects()` function is not useless in these cases, but can still be very useful in a slightly different way. Essentially, the function only provides an interface to pass arbitrary data from the client to a material in the Unreal Engine. The material that is applied to the ground and is supposed to depict snow also makes use of this. However, instead of mixing the snow on the client side with the ground texture, only a single floating point number between zero and one is transferred via the interface, which represents the strength of the snow. For this purpose, a texture with only one pixel is created on the client side, which contains the desired value in its red part. In the material this value can then be used by sampling the value of a 2D parameter with the name `BaseColor` at the UV coordinate $(0.5, 0.5)$. To calculate the snow texture, this value is multiplied by a scaling factor that controls the maximum amount of snow that can be on the ground. The result is then multiplied with a perlin noise texture. This then serves as an alpha value for a lerp between the actual ground texture and a completely white texture. This is of course only a very simple way of applying snow, as the geometry of the object is not taken into account here. The snow is applied to all sides of the object instead of just the top. However, as the terrain is mostly flat, this is still a useful demonstration of this feature.

Figure 4.14: Traffic islands with different amounts of snow.

4.2.4 Vehicle Spawning

Since the calibration of the sensors and the recognition of the vehicles are not always one hundred percent accurate, it can happen that vehicles collide with other objects when spawning in the world. This quickly becomes obvious especially with the road surface, in case a vehicle is not spawned at the correct height. By itself, the Unreal Engine has a very powerful physics system that could be used to place objects in the correct position. However, vehicles spawned on a light slope would start rolling when the physics simulation is enabled for them. Since vehicles should only be moved by the client based on the detection, using the physics system is not a viable option, when accurate positions are required (e.g. when semantic masks for real images should be generated.). To spawn vehicles at the correct height anyway, a line-trace can be used. This sends a ray in a desired direction and returns the first position where this ray collides with an object. By sending a ray straight down at the desired spawn position, the height of the ground at that point can be determined. But even with this method, collisions can still occur, e.g. if the position of the vehicle is very close to the curb so that the ray hits the street while the wheels would collide with the curb. Should the spawn fail due to a collision, the position must be corrected slightly upwards and a new attempt to spawn must be made.

The CARLA API provides all the required functionality to implement this on the client side. The method `ground_projection()` can perform the line-trace to determine the ground height at the spawn location. In addition, this method also returns the semantic tag of the hit object, which can be used to prevent vehicles from being spawned in certain locations, e.g. on top of another vehicle. For spawning, the `try_spawn_actor()` function can be used, which just returns `None` if the object cannot be spawned. This can then be called

until the spawn either succeeds or a maximum upper bound is reached and the spawn is aborted. If the spawn is successful, the physics for this actor is disabled immediately (by calling the `set_simulate_physics method()`). While this approach does work, it has the disadvantage that it is very time-consuming. Since there is a remote procedure call (RPC) behind each of these method calls, i.e. the client process has to communicate with the server process, each of them also has a non-negligible runtime. It takes about 2.2 seconds to spawn 15 individual vehicles with this method, even if both processes are running on the same computer. With 146 milliseconds per spawn, even a single new vehicle would be too for a live system running at just 10 Hz, as each frame had to be processed in under 100 milliseconds.

CARLA offers the possibility to send some commands in batches, so that the overhead for the call can be reduced. This is possible for spawning and deactivating the physics, but not for the ground projections. There is also the additional problem that when the spawn method is called asynchronously, it is not clear which vehicles could be spawned and which failed. Due to these limitations, it was not possible to ensure that the live system would be stable enough, so most of the logic had to be moved to the server side. For this purpose, the spawn logic of vehicles in the `VehicleFactory` blueprint of CARLA was adapted accordingly. With the help of the `LineTraceByChannel` block, a line trace can be executed at the desired spawn position. As with the client-side implementation, this is then used as an initial value and increased until the spawn actually works. The maximum number of attempts can be passed by the client by setting the newly added attribute `spawn_retries`. If this attribute is not set, the new spawn logic is completely skipped and the spawn works exactly as before. With this modification, the client has to send only one batch command to the server, in which the spawning of all desired vehicles is included. Since it takes time after the call until the next detection frame is received anyway, this can be done asynchronously in order not to block the client for an unnecessarily long time. When the processing of the new frame begins, all active actors in CARLA are first requested and then assigned to the internal objects. This works by storing the internal tracking id, which comes from the detection pipeline, in the `role_name` attribute of the blueprint that is to be spawned. If there is an internal object that is not active in CARLA, its spawn has failed even after multiple attempts. After a certain amount of such failures, the object is ignored for a short period of time to avoid wasting time on each failed spawn attempt. This reduced the total time for each frame to about 80 milliseconds, which is fast enough for a live system running rate at 10 hertz. Another change was to disable the physic of an actor directly after spawning, so that no further method calls are needed.

Another problem that arises from disabling physics is properly adapting the z-position of vehicles after they have moved. since there is a small height difference across the S110 intersection, vehicles would either float or drive into the ground after crossing the intersection. in the client-only implementation, the vehicle had to be regularly removed and spawned again to solve this problem, as the line trace would have hit the vehicle otherwise. Since the server-side implementation invokes the Unreal Engine functions directly, there is a better way to solve this problem. It is possible to restrict the line-tracing to certain objects, so that the rays can pass through the vehicles or the gantries and only return collisions with the ground. For this purpose a new `TraceChannel` with the name `DrivableSurface` was created under the project settings. The default value for the collision response was set to ignore, so that rays can fly through all objects by default. Furthermore, a new preset was created which sets the collision response within the newly created trace channel to block. This was then applied to all static actors in the scene on which objects could be spawned (terrain, roads, road markings, etc.) so that only these are hit by the rays. This means that the vehicle does not have to be removed regularly as it is ignored by the line-traces and the rays still hit the ground only. The implementation of this additional functionality was done in the `SetActorGlobalTransform` which is called when the client wants to move an actor

using the API method `set_transform()`. Instead of simply setting the position, additional line traces are executed here to keep the vehicle at the correct height. Unlike spawning, teleporting an object will not fail if it collides with another object at the new position. Therefore, a single line-trace is no longer sufficient, as in a scenario like the one described above, the vehicle would simply be pushed into the pavement. Instead, four line-traces are performed at the respective corners of the bounding box and the highest z-value is taken. At this point, it would also be possible to rotate the vehicle correctly instead of keeping it parallel to the xy-plane as is currently the case. To avoid unnecessary calculations, this is not done with every call but only if the object has moved a certain distance since the last line-tracing.

The entire functionality is also implemented for pedestrians, the only difference is that the radius of the bounding has to be added to their z-position, since their origin is placed differently.

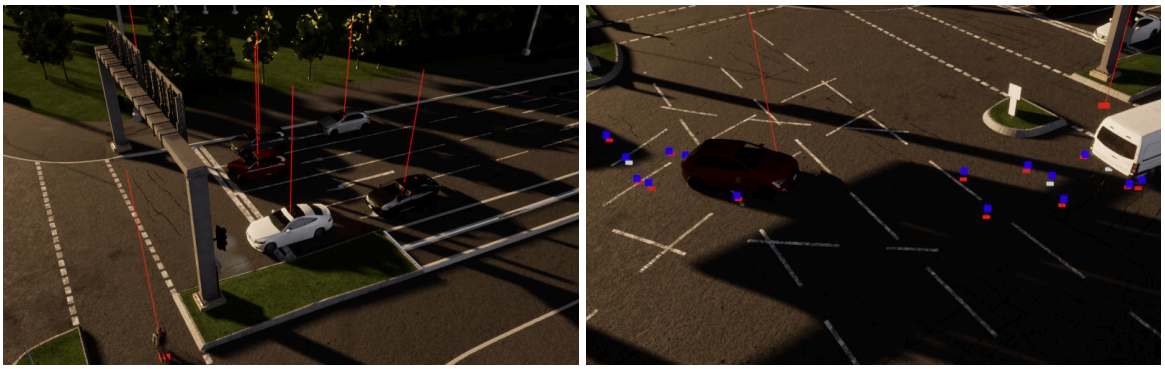


Figure 4.15: Line-traces done at spawn (left) and after movement (right). The blue points in the right image show the corners of the bounding box, the red ones the intersection with the ground and the white one the chosen location.

4.2.5 Vehicle Sizes

Trivial Scaling

In order to change the scaling of a vehicle, the new parameters `size_x`, `size_y` and `size_z` were added, in which the desired size can be passed. These can then be used in the spawn logic of the object to calculate the required scaling based on the bounding box of the object. With props this is quite easy, as they only consist of a single static mesh and thus the size of the bounding box can be obtained before spawning. The desired size along each axis can then be divided by the actual size of the object to calculate the scaling factor. This can be entered directly into the transform during spawning, so that these objects spawn with the correct size right away. This is not so easy with vehicles, because they consist of several individual parts such as the body and the individual wheels. Because of this, the final size can only be determined after the vehicle has been spawned, which is why the vehicle will first spawn unscaled for a short moment. After the vehicle has been spawned, however, the scaling can then be calculated in the same way and applied with the help of the set `SetActorScale3D()` function. As a result, the unscaled size is still used for the initial collision detection and the scaling can slightly push the object into other objects. Only after the object has moved enough and the logic explained in the last section takes effect, the scaled size is also taken into account during the collision calculation.

NeuralMLS

Since the method described above can lead to strong distortions when using larger scaling, the scaling with the help of *NeuralMLS* [She+22] was also implemented. The implementation is designed in such a way that models that are created by any method can be spawned. For this, they only have to be saved as a valid `obj` file. The path to this file can be specified in the parameter `mesh_path` when spawning any `prop`, which will replace its static mesh with the mesh from the `obj` file. However, not all `obj` features are currently supported, e.g. normal vectors and UV-coordinates are completely ignored at the moment. Extra materials defined in an extra `.mtl` are also not supported, but the materials body, glass, tire and metal can be set if the faces in the `obj` use a material with the same name. In CARLA, these are replaced with a corresponding material from the Unreal Engine. These limitations are due to the fact that the parsing of `obj` files is currently done by a simple self-developed parser, which serves more for demonstration purposes. To spawn the resulting mesh, the parser divides the vertices and faces into four parts according to the materials, which are then used to create a single mesh section.

In order to be able to use this functionality with *NeuralMLS*, the scaled model has to be created first. For this purpose, the trained weights, keypoints and the original `obj` file for the desired car model are needed. In this `obj` file, the faces must already be assigned to the four materials mentioned above. The keypoints can then be moved according to the desired size and the procedure described in the *NeuralMLS* paper can be used to calculate the vertices of the rescaled model. The points along with the faces from the original model are then written to an `obj` file, which is sent to the CARLA server as described above. Due to the limited materials, less functionality (e.g. missing lights that can be switched on and off), the complicated preparation (for each model keypoints have to be selected and weights have to be trained) and an increased runtime this method is currently not used for the developed live system. It is still unclear whether the better scaling has an impact on trained data and whether more work in this direction is justified.

4.2.6 Choosing Vehicles

Although there are many different sub-types of vehicles, the current detection pipeline does not contain a detection of the exact vehicle model. In order to be able to spawn vehicles as accurately as possible, the model that comes closest in size is selected for each recognized vehicle. This also minimizes the distortions that can arise from trivial scaling of the models. In order to avoid having to request the existing models and their sizes from the server each time, this is done once and the result is saved in a `json` file. Besides the bounding box, this also contains other attributes about the vehicle that are necessary for a correct selection, such as the exact vehicle type, whether the vehicle is an emergency vehicle or whether it has lights that can be switched on. Based on this information, the vehicle model that best matches the detection received can then be selected at run time.

4.2.7 Interpolation

As already mentioned in the previous chapters, the spawning of new actors in CARLA is always associated with certain costs, which is why unnecessary spawns should be avoided. For this reason, after an update, the vehicles that already existed in the last frame are only moved instead of deleting and re-spawning everything. For this, the program has to keep track of the vehicles and their velocities so that it can also predict new positions in the event of short disruptions.

As soon as a new detection frame has been received, the current state of the CARLA simulation must first be determined. To do this, all actors that are currently available are queried with a single call, which again keeps the number of *RPC* calls to a minimum. The objects that are spawned by this program are all given a role name that starts with a triple at-sign (`@@@`) so that they can be easily filtered. This filtered list can then be compared with the internal desired state, based on the last frame, to establish a link between the internal tracking IDs of the detected objects and the CARLA IDs of their corresponding actors. If there are internal objects without an associated actor in CARLA, their spawn must have failed due to collisions with other objects. The IDs of those objects are stored in a map to get an overview of how many times the spawn of an object has already failed. After too many consecutive failures, an object is ignored for a short time to avoid unnecessary increases in runtime. Now that the correct current state has been determined, it is compared with the desired state, which is based on the frame that has just been received. The first step is to decide which objects need to be either moved, removed or newly spawned. In special cases, a vehicle must be respawned (i.e. both removed and newly spawned) in a frame, for example, if its color or size changes significantly, which could result in a different car model being selected. Objects that should be deleted are first checked to see for how long they have not been detected by checking their time-to-life (*TTL*) value. Since objects are sometimes only absent for a few frames before being detected again, they are kept alive for a short period of time. The positions of such objects in the current frame are then estimated based on their last known position and velocity. The positions of the objects, for which both the old and the new position are now known, can now be interpolated linearly between these two points. This allows smooth movements to be displayed even with a low frequency of detection. After these objects have reached the actual state of the current frame, their velocities are recalculated based on the movement. Now the objects that are to be removed are actually removed and the new objects are spawned. For this, first the correct vehicle models are selected as described in Section 4.2.6 and the spawn commands are assembled. These commands are then transmitted asynchronously to the CARLA server in a single batch.

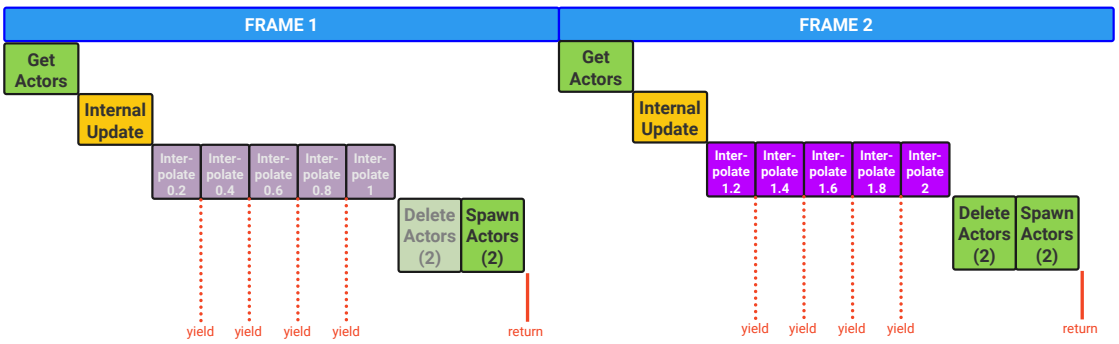
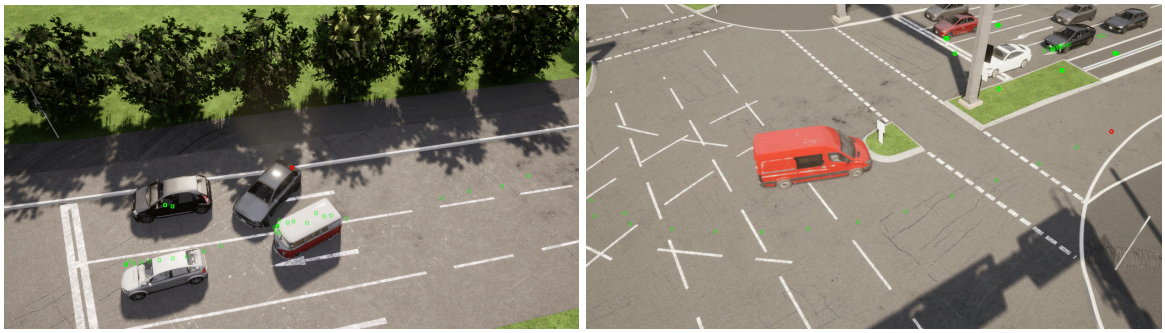


Figure 4.16: Timeline of the actions. The interpolation and deletions in the first frame are *NOPs* because there are no previous actors.

4.2.8 Physics Based Control

Although the procedure described in the last section offers the possibility to position vehicles as accurately as possible at the detected positions, this is only possible at a fairly low frequency. This means that the movements of the vehicles are not very smooth, which reduces the perceived realism of the virtual livestream. Therefore another approach was attempted to solve this problem by using a PID controller for each vehicle, which controls their speed and steering angle. This controller then continuously tries to drive the vehicle to the last received detection. The implementation is based on the `VehiclePIDController`[10] of CARLA but had to be modified to work in this context.The first modification was that the controller applies the brakes to the vehicle when the distance to the target is small enough. This causes vehicles to properly stop at traffic lights instead of circling around the detection. The second change is a dynamic adjustment of the target speed based on the distance to the next waypoint instead of the fixed target speed of the CARLA controller.

The detections can now be received at lower frequencies, as the server will move the vehicles based on the physics simulation. This will make the vehicles move smoothly and physically correct. But in some cases this physical accuracy can lead to problems. As mentioned in the last section, it sometimes happens that vehicles spawn partially on the sidewalk and then roll onto the road due to their physics. This causes the distance to the target position to be too big and the controller tries to drive back to the desired position by executing a turning maneuver. By doing so, it drives across several lanes, which can lead to collisions with other vehicles. Sometimes the vehicles also cut the corners, collide with signs and block the road for oncoming vehicles. To use this variant in the live system, some further modifications are required, which will be explained in chapter 6.3.



(a) Gray vehicle in the center of the image trying to reach its target point (red) and crashing into another vehicle in the process.

(b) Van cutting the corner and crashing into a traffic island because its heading directly to the last received goal point (red).

Figure 4.17: Issues with the current PID control approach.

[10] https://github.com/carla-simulator/carla/blob/master/PythonAPI/carla/agents/navigation/controller.py

4.2.9 Memory Usage

Since the live system, which is designed to portray the digital twin in real-time, is expected to run uninterrupted for a long time, special care must be taken to manage the use of resources. It is particularly important that the memory usage does not increase too quickly, otherwise it could fill up at some point. However, analyses of the first versions of the program showed that this was the case with this system. Two different issues could be identified where memory was not released. The first problem was the dictionaries in Python that store the relationship between internal tracking IDs and CARLA actor IDs. Although all keys were always removed correctly, the removal of keys did not clean up the internal hash table, which grew larger and larger. To counteract the problem, this and other dictionaries have to be recreated from time to time. The second problem was an internal cache of the CARLA client library, which cached actors locally to avoid unnecessary requests to the server. Since CARLA is not necessarily designed to run for a long time, objects are never removed from this cache. As a simple solution for this project, this cache was adapted in such a way that all objects are just deleted when the cache becomes too large. A better solution for this would certainly be a more complex data structure such as a *LRU* cache, which first removes the rarely used objects.

Chapter 5

Evaluation

5.1 Visual Comparison

The images in this chapter attempt to show the results of this work. Figure 5.1 shows a picture of the south1 camera at the S110 sensor station, where the labeled vehicles and pedestrians have been spawned. It can be seen that the static environment has been reproduced quite accurately. Smaller details, such as the manhole cover on the traffic island, the path in the background or the textures on the traffic signs are still missing and could be added in further work. The dynamic objects are also close in appearance and position to their real life counterparts. The taxi in the foreground is marked as white in the A9 dataset, which is why it has been colored as such in CARLA. The position and rotation of the camera also match the real image almost perfectly, even though no manual offset was necessary for this camera. The small deviation of the positions can be seen even better in Figure 5.2. Most objects are a bit too far down within the image, for example the traffic island or the lines of the pedestrian crossing in the background. Other things like the shadow of the gantry bridge on the left side, however, fit much better.

What is still not quite right is the overall look of the image, which is probably mainly due to the too perfect sharpness of the CARLA image and the different white balance (the real image is a bit greenish in this case). This could be improved with better camera settings in CARLA or additional post-processing, which will be discussed in the next chapter (Section 6.8 and 6.9).

Figure 5.1: Comparison of the CARLA image (left) and the real camera image (right) [s110_camera_basler_south1_8mm]

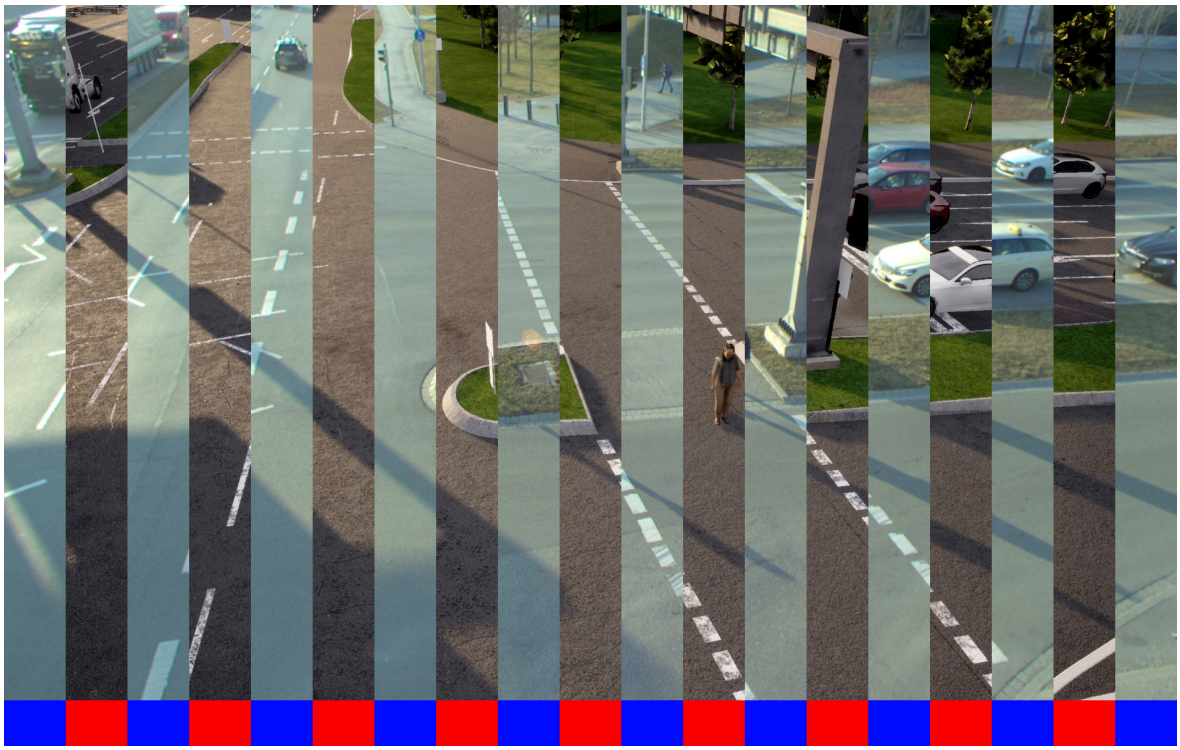


Figure 5.2: Comparison of the real image (blue border) with the CARLA image (red border)

The other camera perspective (s110_camera_basler_south2_8mm) can be compared with the real image in Figure 5.3. In this image, the replicated buildings in the background can be seen well. A small detail which is also modeled incorrectly is the green strip between the road and the sidewalk in the lower left corner. The positions of the vehicles show a slight offset, which is probably caused by an inaccurate conversion from the sensor to the S110 base coordinate system. Unfortunately, the rotation of this camera is not as good as the other one, at the moment the yaw angle is set to an offset of 11 degrees, which needs to be refined further.

Figure 5.3: Comparison of the CARLA image (left) and the real camera image (right) [s110_camera_basler_south2_8mm]

While the lighting is reasonably accurate in the previous daytime pictures, it is different in the nighttime scenes (Figure 5.4 and Figure 5.5). In contrast to the real pictures, the CARLA picture is still far too bright and the reflections from the wet ground are still too weak. In addition, not all vehicles have lights that can be switched on and that could be reflected at all. Likewise, static lights such as street lamps and lights on buildings are also missing. Figure 5.4

demonstrates that special vehicles such as police cars can also be displayed in CARLA if they are recognized as such.

Figure 5.4: Comparison of the CARLA image (left) and the real camera image (right) [s110_camera_basler_south1_8mm]

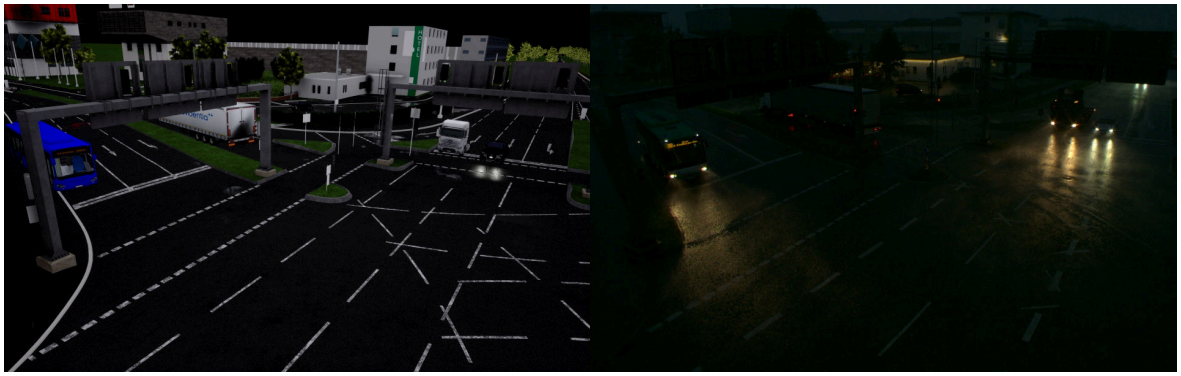


Figure 5.5: Comparison of the CARLA image (left) and the real camera image (right) [s110_camera_basler_south2_8mm]

The possibility of further synthetic data such as semantic or instance masks is shown in Figure 5.6 and Figure 5.7. Such data is especially interesting for the training of e.g. instance segmentation, because the instance masks would not have to be labeled manually. Whether and how well this data can improve such a training has to be evaluated in further work.

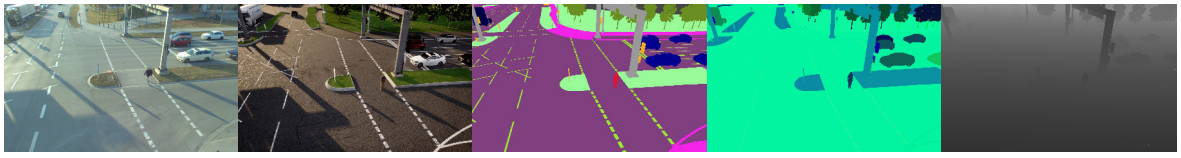


Figure 5.6: Comparison of the real image (left) and the synthetic images (from left to right: rgb, semantic segmentation, instance segmentation, depth image) [s110_camera_basler_south1_8mm]

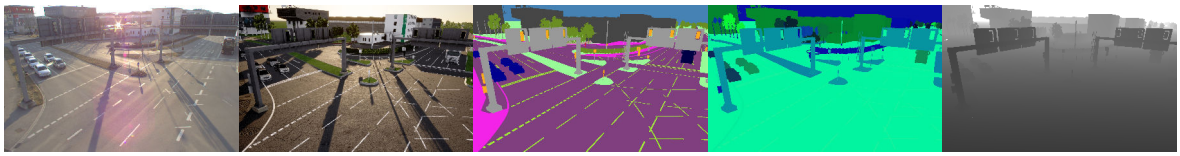
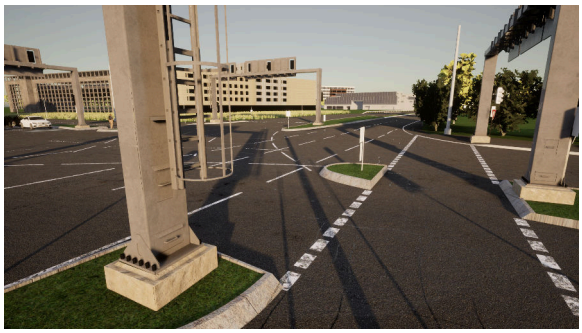


Figure 5.7: Comparison of the real image (left) and the synthetic images (from left to right: rgb, semantic segmentation, instance segmentation, depth image) [s110_camera_basler_south2_8mm]

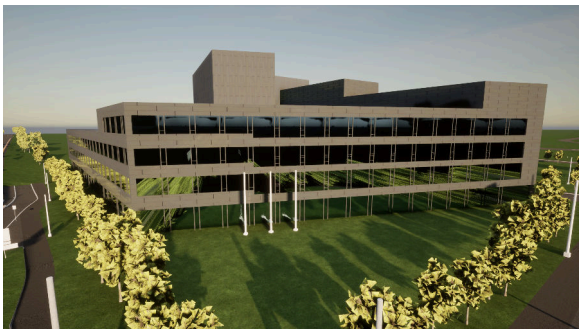
The textures that were created for the signs can be seen in Figure 5.8a and Figure 5.8b. Figure 5.8b also shows vegetation that has been added in some places to give the world a more vivid appearance. A more detailed look of one of the gantry bridges can be seen in Figure 5.8d. However, not all properties of the real world can be completely replicated, for example Figure 5.8d shows broken reflections, as these are sometimes difficult to realise in games engines.

(a) Gantries with accurate traffic sign at the S110 intersection

(b) Foliage at the ground near the S110 intersection



(c) Detailed look at the foot of a gantry bridge

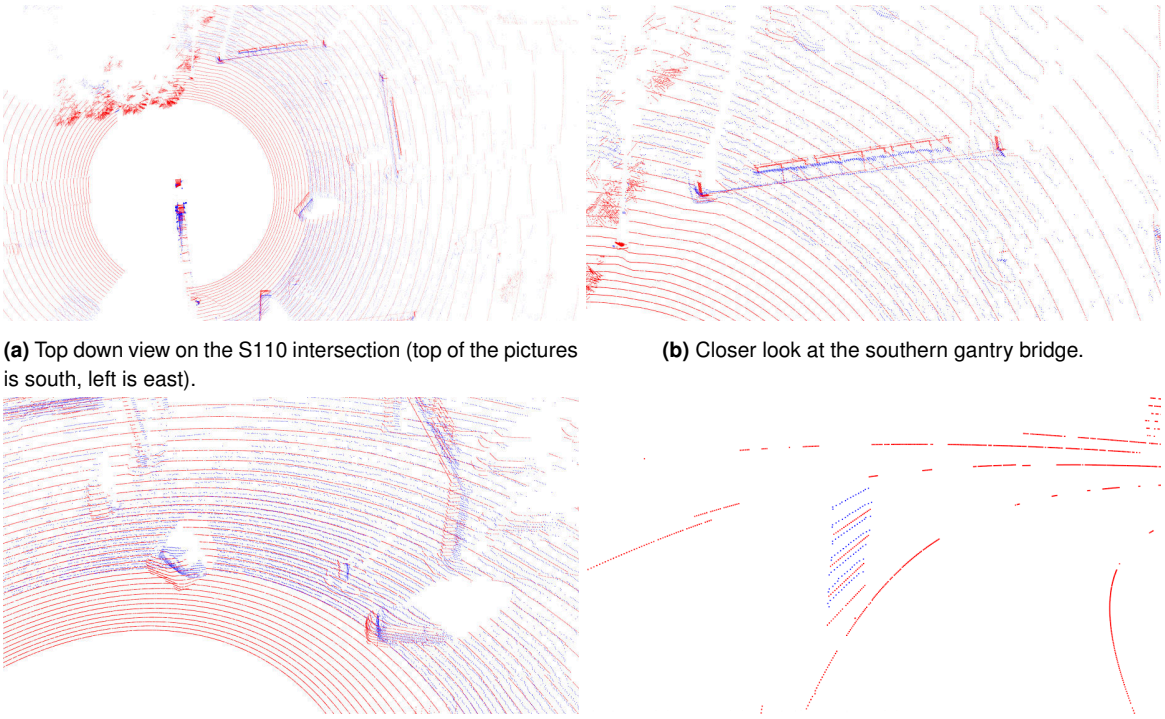


(d) Broken reflections

Figure 5.8: Details images of certain objects in CARLA

5.2 Point Cloud Similarity

Figure 5.9 shows that point clouds generated with CARLA can be similar to data acquired from a real sensor. Some manual calibration was done ($\Delta_z = -0.6, \Delta_{roll} = -1, \Delta_{yaw} = -7$) to get the synthetic point clouds closer to the original but the yaw could be improved further. The problem with offsets in the vehicle positions, which was already noticeable in the pictures, is also visible here.



(a) Top down view on the S110 intersection (top of the pictures is south, left is east).

(b) Closer look at the southern gantry bridge.

(c) Vehicles in the center of in the intersection.

(d) Point cloud of a traffic sign roughly 100m east of the S110 intersection

Figure 5.9: Comparison of the real point clouds (blue) with a synthetic one generated with CARLA (red)

5.3 Color Detection Accuracy

As already mentioned in the chapter about the color recognition training, many vehicles are currently erroneously classified as black (Figure 5.10, Table A.1). In order to reduce the influence of the ground and tires, an additional option has been added to cut off parts of the image border. this improves the accuracy from 37.4 to 42.7 percent, if 40 percent of the border is cut off. For some colors, such as green and yellow, there are not enough images to achieve a good result. Some of the vehicles that were falsely identified as red are images from the rear, where the brake lights take up part of the image. Considering the rather poor results of the current recognition, there is definitely still a lot of potential for improvement for future work.

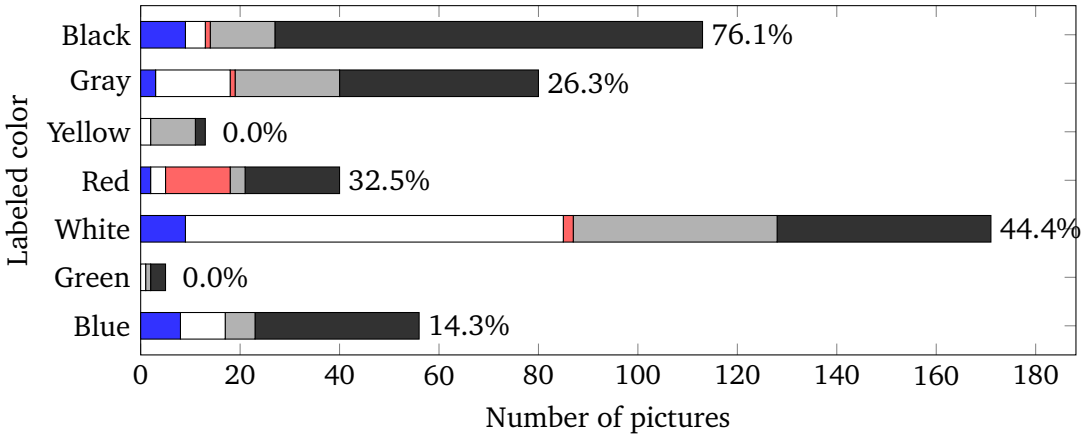


Figure 5.10: Color classifications on the A9 dataset

Chapter 6

Outlook

6.1 Static pipeline

While the method of converting the HD map into a 3D model using RoadRunner shown in this thesis works reasonably well, there are some disadvantages to this approach. One big problem is the long development times that arise when an object in the map needs to be edited. Depending on the problem, editing and exporting in roadrunner should be quite fast. Importing the map into CARLA, however, can take several hours (about 4-5 hours for this map), making even minor changes very time-consuming. After the new import of the map, all other objects that were manually added to the level in the meantime must also be copied into the newly created level. This might be solved by exporting the map in chunks, which could be imported back into the Unreal Engine individually (using the `Reimport Mesh` feature) after editing in RoadRunner. The initial import of such a separated world always failed with the current map, which is why this approach was not pursued further so far. Another possibility would be to generate the road models separately, e.g. with the help of the ODR-viewer or a similar solution as described in x. Another possible improvement would be to use the native terrain feature of the Unreal Engine instead of importing it as a static mesh. This landscape feature is based on chunked height maps, in which the height of the terrain at each position is stored. These can also be edited directly in the Unreal Engine editor, which eliminates the need for time-consuming re-imports. They also bring other advantages, such as the reduction of memory usage (static meshes can need up to 7 times the memory), smooth transitions between levels-of-detail and more efficient collision checks.

The situation is similar with the foliage, especially the trees. Again, the Unreal Engine offers a more efficient way to represent plants in the engine with its built-in foliage system. When importing trees from RoadRunner, for example, a separate static mesh is created for each tree, even if most of the trees are only copies of each other. This leads to a higher consumption of memory and hard disk space because each copy has to be loaded. In the exported game, the trees alone account for 2.4 of the 9.5 GB of files.

6.2 Photogrammetry and NeRFs

Photogrammetry is a technique that automatically creates 3D models from a large number of photographs, usually taken with the help of drones. It is an alternative method to traditional 3D modeling, where a 3D model is created based on drawings or individual reference images. The great advantage of photogrammetry is the accuracy of the results. By using high-resolution photographs and optional laser scans, a high level of detail can be achieved in the 3D models. This is particularly advantageous for buildings with a lot of fine details or

curves, as it takes a lot of time and experience to create them manually. For larger computer games, this is usually done by teams of up to several hundreds of artists, which is not feasible for most research projects.

However, there are also some disadvantages to using photogrammetry. One of the most important disadvantages is that photographs used as a basis for a 3D model cannot be used under certain conditions, such as bad weather or poor lighting conditions. Sometimes a lot of effort has to be put into fixing error and cleaning up the automatically created objects. It is also not always possible to use only textures based on photographs as the surface material, as for example windows need physically based materials to show reflections correctly. The high accuracy of the generated 3d models can also be a disadvantage, because older engines (like the Unreal Engine 4) cannot display unlimited high resolutions in real-time. Programs that can be used to create such models include the proper solution RealityCapture[1] and the open source software Meshroom[2].

A newer approach to create such models are so-called Neural Radiance Fields (NeRF) [Mül+22b], where a machine learning application is trained to reconstruct a 3D scene from a single image (or a small set of images). The advantage of this is that not all viewpoints have to be captured, as the AI can fill in the gaps. The scenes are usually represented as volumetric data, from which meshes can then be generated using e.g. Poisson Surface Reconstruction or the Marching Cubes algorithm [Chi+20].

6.3 PID Controller

To use the PID controller approach in a meaningful way, it needs to be extended by a few additional features. First of all, special handling should be introduced for vehicles that collide with the environment or other vehicles. These vehicles could, for example, be teleported back to their lane or destroyed immediately so that they do not interfere with the surrounding traffic. The problem with cutting the corners could be solved by a FIFO buffer for the waypoints. Instead of always heading straight for the last received point, several points could be stored and successively traversed. In the end, this approach should also be checked with regard to its runtime and memory utilization.

6.4 Network Architecture

As already pointed out in section 4.2.4, the CARLA architecture is already reaching its limits at around 10 Hz when vehicles should be spawned in real-time. If a higher frequency is desired, either more tasks have to be handed over to the server or a solution to spawn vehicles that is not handled by CARLA has to be found. If the vehicles should only move more fluently but the detections are still only updated with 10 Hz, then the interpolation of positions described in section 4.2.7 could also be implemented on the server side. If this is not possible or desired, it would also be an option to implement the spawning of vehicles with the help of the multiplayer feature[3] of the Unreal Engine. This network stack is already optimized for low-latency applications and is used in many popular multiplayer games.

[1] https://www.capturingreality.com/
[2] https://alicevision.org/#meshroom
[3] https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/Networking/

6.5 Improved Detections

The detection pipeline, which runs before the CARLA visualization, could be extended with a few more features that would improve the visualization. For example, the exact vehicle type (SUV, sports car, coupe, etc.), the brand or even the complete vehicle model could be recognized to improve the selection. There is also a lot of room for improvement in the color recognition, such as the identification of the correct region of interest to ignore the wheels and windows. Further pre-processing could be done to compensate for the white balance of the cameras, for example, or to be able to function in poor lighting conditions. In addition, a similar approach to the color recognition of vehicles could also be used to recognize the current weather instead of only determining it based on the current date (and possibly data from a weather API). This would allow, to automatically recognize and set the correct amount of snow on the ground.

6.6 Unreal Engine 5

As already mentioned in some other sections, the Unreal Engine 4 is reaching its limits in some areas, some of which are being solved by its successor, the Unreal Engine 5. An important new feature is the Nanite renderer which is able to handle massive amounts of geometry data while still providing real-time performance. This allows photogrammetry scans to be loaded directly into the engine without the need for extensive cleaning. The new Lumen illumination system can also display much more realistic lighting conditions and reflections. The use of Unreal Engine 5 would be a big step towards a photorealistic simulation.

6.7 Virtual Reality in CARLA

To experience the digital twin in a more immersive way, it might be a good idea to add virtual reality support. This would give the user the feeling of actually standing in or driving through the test track. To make this really immersive, it would also be necessary to simulate the ambient sound of the vehicles. With such a virtual environment, one could then research the behavior of people in e.g. accident scenarios without having to expose them to any danger. The simulator DReyeVR offers exactly these features based on CARLA. Further work could therefore integrate the map created in this thesis into this simulator.

6.8 Intrinsic Camera Parameters

The current virtual camera images look quite flawless compared to real images. If data is used for training machine learning applications, this training may not be transferable to real data. CARLA offers the possibility to adjust some internal parameters like the intensity of chromatic aberration or the white balance of the camera. Future work could investigate how far these can close the gap between virtual and real data.

6.9 Style Transfer

Instead of adjusting the camera parameters and render settings to make the virtual image look as real as possible, one could also use generative adversarial networks (GAN) for this purpose. These can be trained to convert the synthetically generated image to have the same style as the images from the training sets. A very promising implementation of this concept can be found in [RAK21]. This work used synthetic data from the computer game GTA V and generated images in the style of e.g. the cityscapes dataset (see Figure 6.1). The runtime specified in the paper is about 0.5 seconds for an image, so this would not be realtime capable yet.

Figure 6.1: Synthetic input image (left) and the corresponding result from [RAK21] (right)

Chapter 7

Conclusion

The pipeline outlined in this thesis provides a solid foundation for the creation and development of realistic 3D digital twins. The steps show how essential objects such as buildings and road signs can be digitally recreated with great detail using simple reference images. In addition, the process to create an environment model based on a detailed HD map is illustrated. The described steps were applied to one part of the A9 test track, namely the area surrounding the S110 intersection. Further work should investigate how the development cycles can be shortened to iteratively add additional parts of the test track. In this context it could be evaluated to what extent automated processes like photogrammetry can be helpful, what kind of data is needed for such processes and how this can be gathered. It is important to keep in mind that the resulting models still have to be rendered in real time by a game engine and therefore the resource utilization should be considered as well.

For the dynamic behavior of traffic, it was shown how the data of the existing recognition pipeline can be used to display the vehicles at the correct positions in a realistic fashion. In addition to the position and rotation, further properties (size and color) of the traffic objects can now be displayed as a result of this work. The exact subtype of the vehicle is estimated based on the size, by selecting the best fitting vehicle model. Even properties of the environment itself, such as the current position of the sun or snow on the ground can be dynamically adjusted. With the help of a clock model, it was also shown how textures can be created at runtime and displayed in the digital twin. For the dynamic behavior, special attention was given to the runtime of the programs in order to be able to continue executing the system in real time. however, there is still a need for further development to be able to display smooth traffic. Further potential for improvements can be found in the detection pipeline. On the one hand, the color recognition introduced by this work can be further optimized to improve the current accuracy of 42 percent and the runtime at higher resolutions. On the other hand, the pipeline can be extended to determine more detailed information such as the manufacturer or model of the vehicles.

To be able to experience the enhanced realism of the digital twin in a more immersive way, it was also made possible to stream a 360 degree video of the simulation. This could be followed by the introduction of virtual reality support, so that users can experience and control the simulation by themselves.

To improve the realism of the simulation further, some metrics should be established, which can serve as an orientation for further steps. These steps could include better configuration of the current virtual sensors, the use of a newer rendering engine, or post- processing using generative adversarial networks

Appendix A

Appendix

A.1 Known issues

Z-Offset As stated in section 4.1.7, the x and y coordinates are calculated correctly with the help of the projection details. However, the Z coordinate is increased by an extra offset of 2.78 metres.

Hard-coded projection details The projection details of the map are still hard-coded in the file `coordinate.py`. Here it would be better to read them out of the current map at runtime. For this the current HD map could be requested via the CARLA API to then parse the projection details from it.

Hard-coded map position The coordinates of the map, which are necessary for the correct calculation of the sun's position, are currently also hardcoded in `dyn_env.py`. These could also be read from the HD map as described above, or the geo references of the sensors could be used.

Size of two-wheeled vehicles Two-wheeled vehicles (bicycles, motorcycles) return a actor bounding box of (0, 0, 0) which prevents the scaling from being calculated correctly. As a consequence, changing the size of these vehicles is deactivated at the moment

Size of pedestrians The scaling logic implemented for props and vehicles needs to be transferred to the pedestrian blueprint.

Correct position of traffic islands The code to generate traffic islands contains an offset of (x=-89.96, y=+25.19), which was acquired through manual alignment of the traffic islands in the Unreal Engine. It is not known where this offset comes from and why the position of the traffic islands is otherwise calculated incorrectly

Hard-coded sensor frames The sensor frame which used to calculate the global position within CARLA is currently hard-coded to *S110_base*. It would be better to read this information from the OpenLABEL file or the ROS message.

Special vehicles Special vehicles like police cars or firetrucks are currently handled correctly in the selection logic. The information about the vehicle is present in the `vehicle_map.json` but not used at the moment.

Calls to `get_map()` A client currently crashes if `get_map()` method of a `carla.World` object is called and the `providentia_new map` is loaded. The OpenDRIVE parser used to import the HD Map into CARLA needed some tweaks to prevent nullptr accesses, maybe a similar issue causes these crashes. Some API calls can be fixed by changing the CARLA library to just skip problematic lanes.

Pedestrian navigation Trying to build the pedestrian navigation fails when the official tutorial[1] is followed. To fix this, only the needed actors should be selected in the export steps within the Unreal Engine. The pedestrians crossing across the S110 intersection are also missing and would have to be created as described in the tutorial.

Bad UV mapping The UV map of the the kreuzmayr building was generated automatically using the *Smart UV project* function in Blender. This leads to sub-optimal UV-map which has a visible seam on the side of the building (see Figure A.1).

Missing details Some rather important objects like the cameras on the S110 sensor station or the sensor masts of other station are currently missing. These where available in previous iterations of the map but have yet to be recreated for the current one (see Figure A.2 and Figure A.3). A suitable 3D model for the camera can be found here https://poly.pizza/m/a6J7IDufQP.

Z-Fighting Some objects, like the blinds on the kreuzmayr building or lane markings, suffer from z-fighting which causes flickering when viewed from certain angles.

State of the highway Because this works mainly focused on the recreation of the environment around the S110 intersection, little attention was given to the A9 parts of the testfield (Figure A.4). This involves, among other things, missing sign bridges, green middle sections and more appealing trees next to the highway.

Missing signs Only the signs mounted to the gantries where created as described in 4.1.2. The other signs on test-stretch still need to be created to make the map complete.

Textures on traffic signs Although most of the signs in RoadRunner have correct textures, these are missing in CARLA (Figure A.5). It is still unclear at which step of the export these textures got lost. Besides, the meshes exported from roadrunner do not seem to be quite right yet, as round signs are still exported as squares.

Autopilot accidents On this map, the CARLA autopilot often makes mistakes and runs into obstacles. It is not clear whether these are problems with the map or with the autopilot. However, the former seems to be the more plausible, as such accidents are not observed on the official maps.

Dynamic meshes As only file paths are sent to the server for the spawning of dynamic meshes, which was implemented for the tests with NeuralMLS, the sever and client have to run on the same computer or both have to have access to the corresponding path.

Truck/Trailer collisions To avoid collisions between trucks and their trailers, the collision boxes have been greatly reduced in size (see Figure A.6. However, when the physics are activated, as is necessary for the PID controller, this leads to problems as these vehicles can easily overturn. An improved handling of trucks as shown in https://github.com/frankeng/CarlaSemiTruckTrailer would be beneficial here.

Dynamic weather The selection of a suitable weather preset is only implemented for some weather types in OpenLABEL files. Currently no ROS-node for weather handling is implemented.

[1] https://carla.readthedocs.io/en/0.9.14/tuto_M_generate_pedestrian_navigation/

A.2 Further Graphics

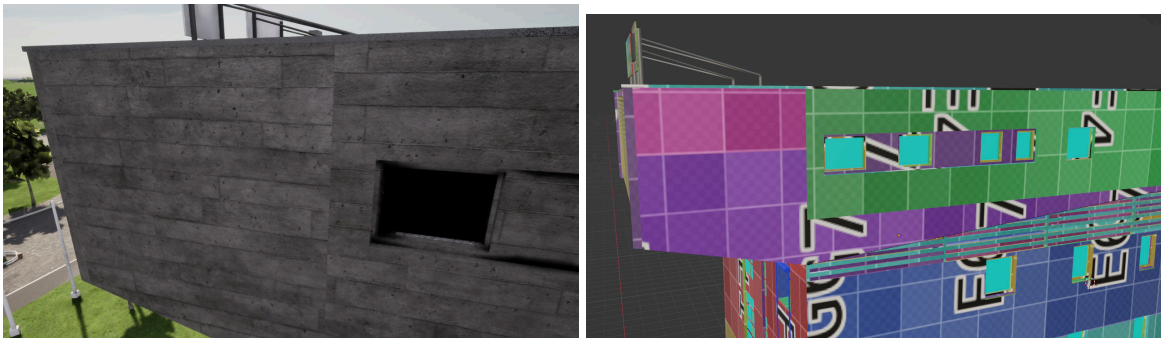


Figure A.1: UV-map issues causing a visible seam in the concrete texture

Figure A.2: Old version of the S110 gantry (left) with cameras which are not present in the current one (right).

Figure A.3: The 5G mast (left) and other sensor stations (right) in older versions of the CARLA map

Figure A.4: Old version of the highway (left) compared to the new version (right).

Figure A.5: Traffic sign with correct texture in RoadRunner (left) and without in CARLA (right).

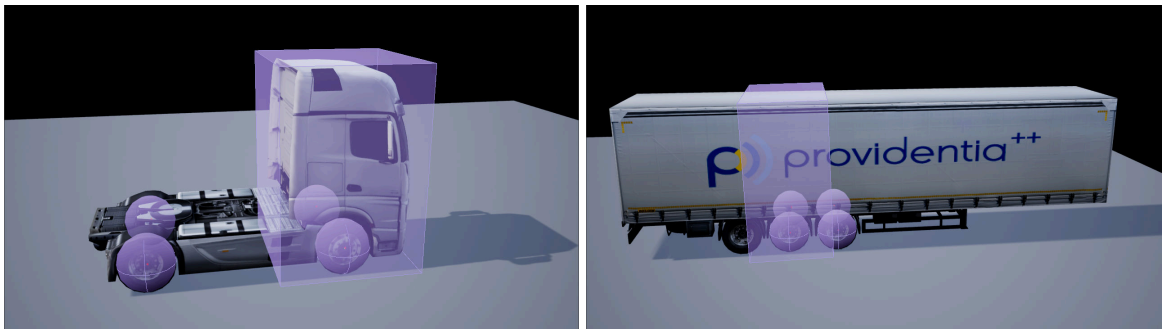


Figure A.6: Reduced collision boxes for the truck (left) and the trailer (right)

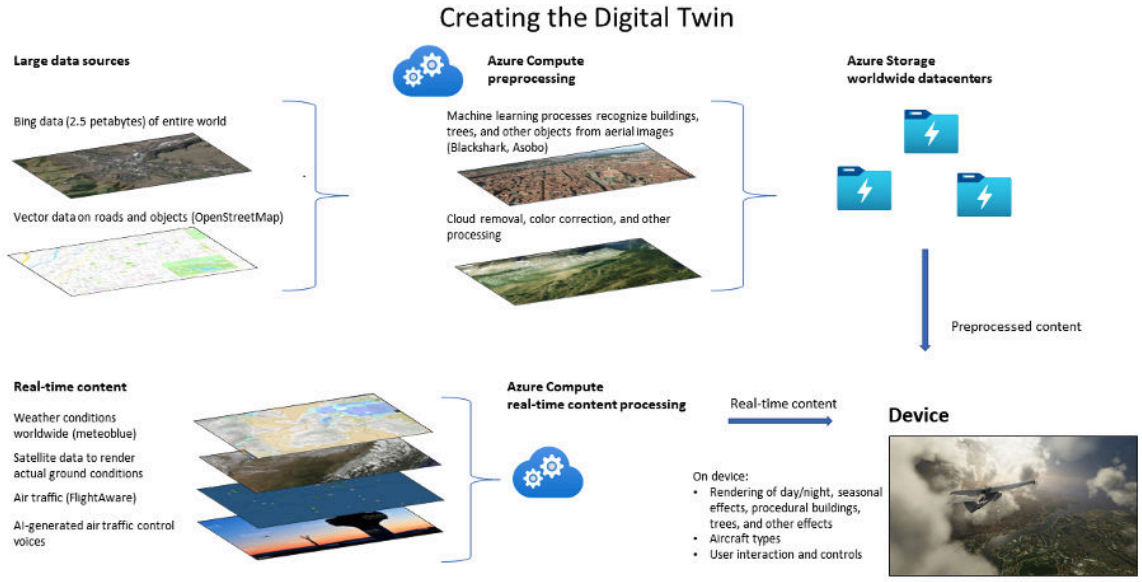
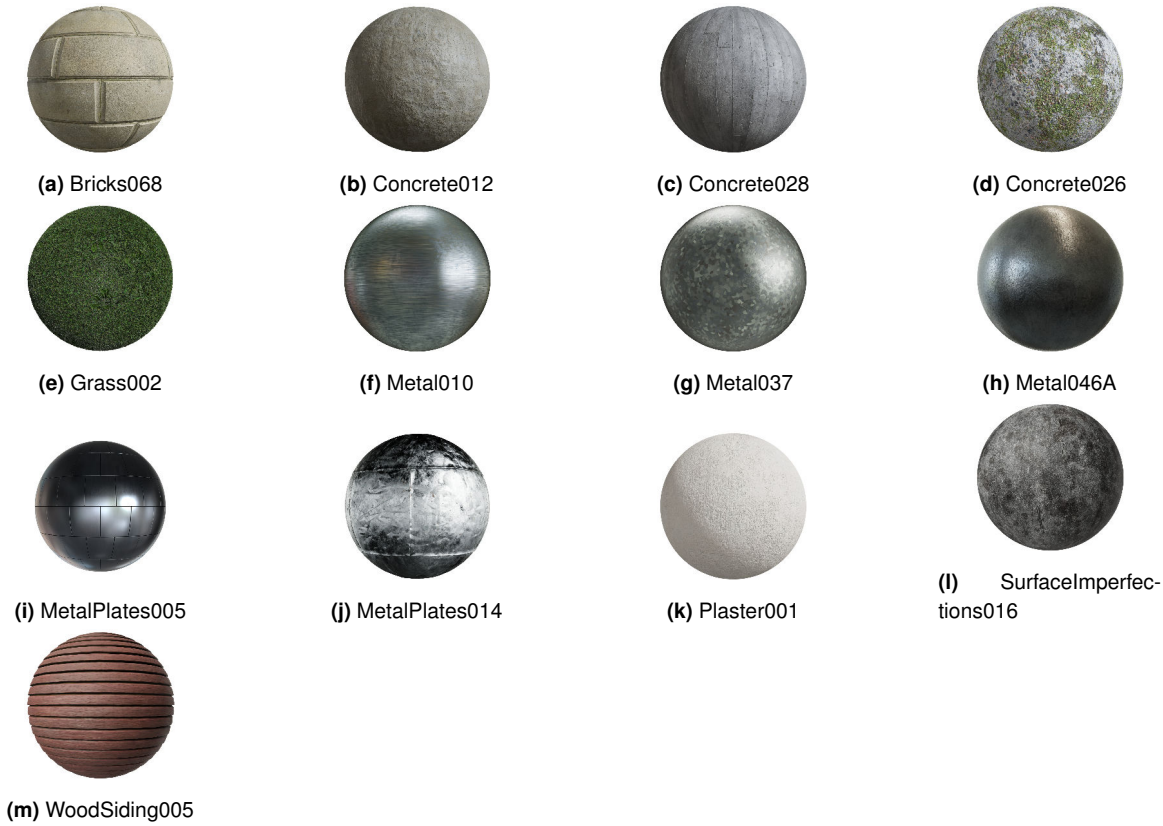


Figure A.7: The Microsoft Flight Simulator high-level architecture creates a digital twin of the real world [Benoit Patelout, CC BY-NC-SA 2.0].



(a) Bricks068 **(b)** Concrete012 **(c)** Concrete028 **(d)** Concrete026

(e) Grass002 **(f)** Metal010 **(g)** Metal037 **(h)** Metal046A

(i) MetalPlates005 **(j)** MetalPlates014 **(k)** Plaster001 **(l)** SurfaceImperfections016

(m) WoodSiding005

Figure A.8: Third-party textures used in the project

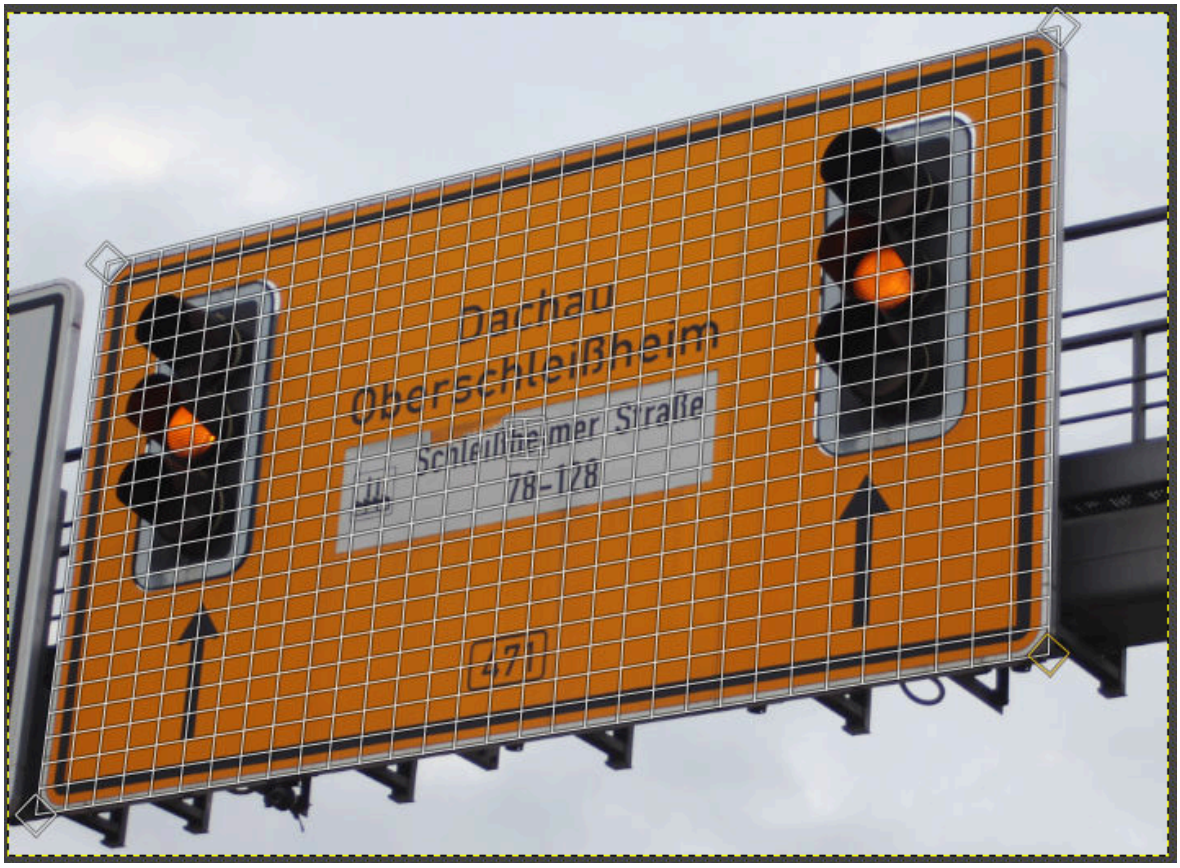
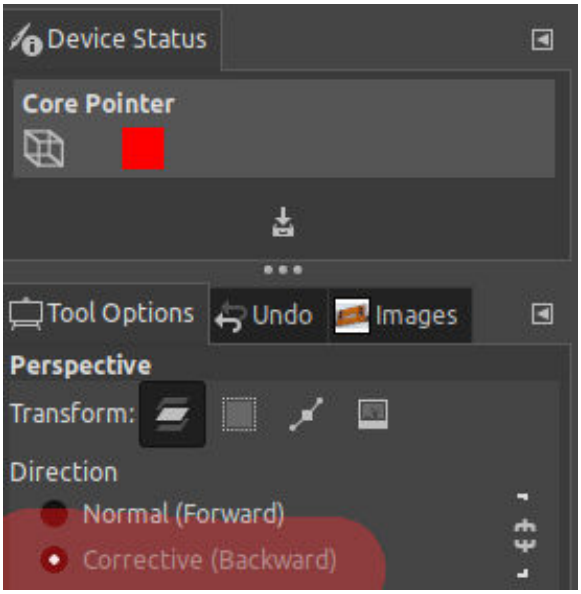
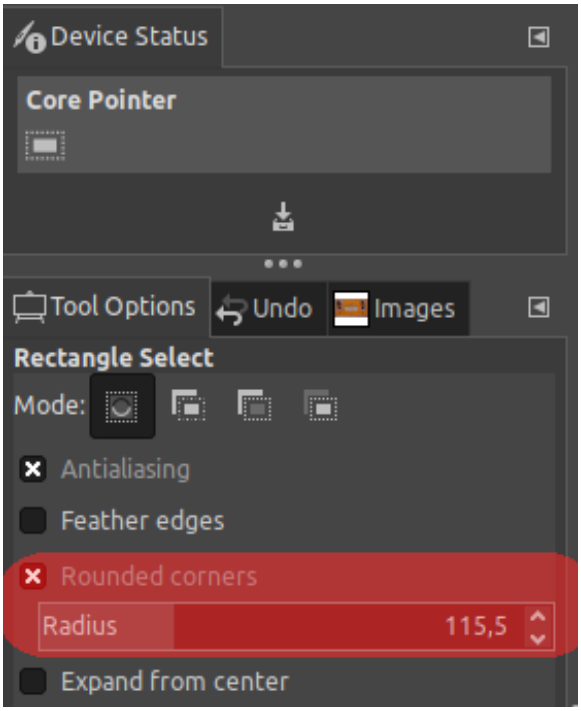


Figure A.9: Correct positioning of the perspective tool grid



(a) Perspective tool settings



(b) Example selection settings for cutting traffic light holes

Figure A.10: Details of the texture creation process

		detected color						
		Blue	Green	White	Red	Yellow	Gray	Black
labeled color	Blue	8	0	9	0	0	6	33
	Green	0	0	1	0	0	1	3
	White	9	0	76	2	0	41	43
	Red	2	0	3	13	0	3	19
	Yellow	0	0	2	0	0	9	2
	Gray	3	0	15	1	0	21	40
	Black	9	0	4	1	0	13	86

Table A.1: Full result of the color classification on the A9 color training set.

A.3 Changed Blueprints

This section is intended to give an overview of the changes made to the blueprint in CARLA, as these are very difficult to track using common version management systems.

A.3.1 VehicleFactory

The `VehicleFactory (/Game/Carla/Blueprints/Vehicles/VehicleFactory)` blueprint is in charge of spawning the vehicles, which is why several changes had to be taken here.

Spawn Logic

The first on is a new function within the same Blueprint with the name `Try Spawning` which replaces the default spawn logic within the `SpawnActor` function. The parameters `spawn_retries` and `spawn_ray_debug` are read from the actor's attributes and supplied to the function call. Within the function, the value of the `spawn_retries` parameter is checked. If is zero, the default spawn logic is used. Otherwise a line-trace is executed within the `DriveSurface` channel to get an initial estimate for the correct spawn position. Then, the `SpawnActor` is called withing a loop until the spawn succeeds or the maximum limit of iteration based on the `spawn_retries` parameter is reached.

Vehicle Scaling

Another function called `Set Size` was added to perform the resizing of the actor, but this one was added to the `Base Vehicle Pawn` class. It is called at the very and of the `SpawnActor` function. The parameters for the size are also read from the vehicle's attribute list. The function computes the scaling factors for each dimension based on the target size and the original bounding box size of the vehicle. After that, the physics is disables and the original rotation of the spawn is restored. This is necessary, because sometimes the physics system would already rotate the vehicle between spawning and disabling the physics.

A.3.2 WalkerFactory

Has the same `Try Spawning` block as for the vehicles. Changing size is currently not implemented, but could be similar to the implementation in the `VehicleFactory`

A.3.3 ProceduralActor

The `ProceduralActor` (/Game/Carla/Blueprints/ProceduralActor.ProceduralActor) is a new Blueprint added for the spawning of dynamic meshes, as needed by NeuralMLS integration. It has a `SetMesh` function which receives the points and faces from the custom `.obj` parser and creates mesh sections for the different parts of the vehicle.

A.3.4 PropFactory

The `PropFactory` (/Game/Carla/Blueprints/Props/PropFactory) is in charge of spawning props and is also used for the newly added dynamic mesh feature.

Resizing

Unlike vehicles, props consist of just a single static mesh, which is why the size can be determined before spawning. The result of the scaling factor calculation can then be directly included in the spawn transform of the `SpawnActor` block.

Dynamic Meshes

in case the mesh_file attribute is set, the normal spawn logic will be bypassed and the actual static mesh of the actor will be ignored. instead, the mesh file will be loaded, parsed and the result will be passed to the `SetMesh` function mentioned above.

List of Figures

List of Tables

Bibliography

[Aga23] Agafonkin, V. *SunCalc*. original-date: 2011-08-25T11:40:01Z. Jan. 2023. URL: https://github.com/mourner/suncalc (visited on 01/29/2023).

[Azf+22] Azfar, T., Weidner, J., Raheem, A., Ke, R., and Cheu, R. L. "Efficient Procedure of Building University Campus Models for Digital Twin Simulation". In: *IEEE Journal of Radio Frequency Identification* 6 (2022). Conference Name: IEEE Journal of Radio Frequency Identification, pp. 769–773. ISSN: 2469-7281. DOI: 10.1109/JRFID.2022.3212957.

[Bar] Barron, K. *suncalc: A fast, vectorized Python port of suncalc.js*. URL: https://github.com/kylebarron/suncalc-py (visited on 01/29/2023).

[BLS16] Biljecki, F., Ledoux, H., and Stoter, J. "An improved LOD specification for 3D building models". en. In: *Computers, Environment and Urban Systems* 59 (Sept. 2016), pp. 25–37. ISSN: 0198-9715. DOI: 10.1016/j.compenvurbsys.2016.04.005. URL: https://www.sciencedirect.com/science/article/pii/S0198971516300436 (visited on 01/09/2023).

[BZH22] Brase, R., Zhuge, W., and Höhl, W. "Nykus Exploration – Open Geodata in a Survival Game". 2022. URL: https://wiki.tum.de/display/infar/Nykus+Exploration.

[Bra14] Bratfisch, S. "LoD3-Gebäudemodelle in Bayern – alternative technische Lösungsansätze". de. In: (July 2014), p. 101.

[Cao+22] Cao, W., Zhou, L., Huang, Y., and Knoll, A. *Autonomous Driving Simulator based on Neurorobotics Platform*. arXiv:2301.00089 [cs]. Dec. 2022. DOI: 10.48550/arXiv.2301.00089. URL: http://arxiv.org/abs/2301.00089 (visited on 01/08/2023).

[Chi+20] Chivriga, C., Wiberg, B., Shentu, Y., and Loser, M. *BUFF - Bounding unstructured radiance volumes for free view synthesis*. Oct. 2020. URL: https://github.com/qway/nerfmeshes/.

[Coe+21] Coenen, T., Walravens, N., Vannieuwenhuyze, J., Lefever, S., Michiels, P., Otjacques, B., and Degreef, G. "Open Urban Digital Twins -insights in the current state of play". In: (May 2021).

[Cre+22] Creß, C., Zimmer, W., Strand, L., Lakshminarasimhan, V., Fortkord, M., Dai, S., and Knoll, A. *A9-Dataset: Multi-Sensor Infrastructure-Based Dataset for Mobility Research*. 2022. DOI: 10.48550/ARXIV.2204.06527. URL: https://arxiv.org/abs/2204.06527.

[Des+21] Deschaud, J.-E., Duque, D., Richa, J. P., Velasco-Forero, S., Marcotegui, B., and Goulette, F. "Paris-CARLA-3D: A Real and Synthetic Outdoor Point Cloud Dataset for Challenging Tasks in 3D Mapping". In: *CoRR* abs/2111.11348 (2021). arXiv: 2111.11348. URL: https://arxiv.org/abs/2111.11348.

[Dev21] Dev, M. G. *Microsoft Flight Simulator: The Future of Game Development*. en. July 2021. URL: https://developer.microsoft.com/en-us/games/blog/microsoft-flight-simulator-the-future-of-game-development/ (visited on 01/08/2023).

[Dos+17] Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., and Koltun, V. *CARLA: An Open Urban Driving Simulator*. arXiv:1711.03938 [cs]. Nov. 2017. DOI: 10.48550/arXiv.1711.03938. URL: http://arxiv.org/abs/1711.03938 (visited on 01/08/2023).

[Epi] Epic. *Unreal Engine*. URL: https://www.unrealengine.com/.

[Fou] Foundation, L. *Open 3D Engine*. URL: https://www.o3de.org/.

[Fue22] Fuentes, L. *Designing the Terrain System of Flight Simulator: Representing the Earth*. Mar. 2022. URL: https://www.asobostudio.com/news/asobo-gdc-2022 (visited on 01/08/2023).

[GNV21] Galazka, E., Niemirepo, T. T., and Vanne, J. "CiThruS2: Open-source Photorealistic 3D Framework for Driving and Traffic Simulation in Real Time". In: *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*. Sept. 2021, pp. 3284–3291. DOI: 10.1109/ITSC48978.2021.9564751.

[Gey22] Geyer, M. *On Track: Digitale Schiene Deutschland Building Digital Twin of Rail Network in NVIDIA Omniverse*. en-US. Sept. 2022. URL: https://blogs.nvidia.com/blog/2022/09/20/deutsche-bahn-railway-system-digital-twin/ (visited on 11/06/2022).

[Heg+22] Hegelsom, J. van, Mortel-Fronczak, J. M. van de, Moormann, L., Beek, D. A. van, and Rooda, J. E. *Development of a 3D Digital Twin of the Swalmen Tunnel in the Rijkswaterstaat Project*. arXiv:2107.12108 [cs, eess]. Feb. 2022. URL: http://arxiv.org/abs/2107.12108 (visited on 10/26/2022).

[JHH95] Jacobi, N., Husbands, P., and Harvey, I. "Noise and the Reality Gap: The Use of Simulation in Evolutionary Robotics". In: *Proceedings of the Third European Conference on Advances in Artificial Life*. Berlin, Heidelberg: Springer-Verlag, June 1995, pp. 704–720. ISBN: 978-3-540-59496-3. (Visited on 01/08/2023).

[Jon+20] Jones, D., Snider, C., Nassehi, A., Yon, J., and Hicks, B. "Characterising the Digital Twin: A systematic literature review". en. In: *CIRP Journal of Manufacturing Science and Technology* 29 (May 2020), pp. 36–52. ISSN: 1755-5817. DOI: 10.1016/j.cirpj.2020.02.002. URL: https://www.sciencedirect.com/science/article/pii/S1755581720300110 (visited on 01/08/2023).

[KH04] Koenig, N. and Howard, A. "Design and use paradigms for Gazebo, an open-source multi-robot simulator". In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*. Vol. 3. Sept. 2004, 2149–2154 vol.3. DOI: 10.1109/IROS.2004.1389727.

[Kra+08] Kraevoy, V., Sheffer, A., Shamir, A., and Cohen-Or, D. "Non-homogeneous Resizing of Complex Models". In: *ACM Trans. Graph.* 27 (Dec. 2008), p. 111. DOI: 10.1145/1457515.1409064.

[21] *Kraftfahrt-Bundesamt - Farbe - Neuzulassungen im Jahr 2021 nach Farben*. 2021. URL: https://www.kba.de/DE/Statistik/Fahrzeuge/Neuzulassungen/Farbe/2021/2021_n_farbe_kurzbericht.html?nn=3547524&fromStatistic=3547524&yearFilter=2021&fromStatistic=3547524&yearFilter=2021 (visited on 10/26/2022).

[Krä+19] Krämmer, A., Schöller, C., Gulati, D., Lakshminarasimhan, V., Kurz, F., Rosenbaum, D., Lenz, C., and Knoll, A. *Providentia – A Large-Scale Sensor System for the Assistance of Autonomous Vehicles and Its Evaluation*. 2019. DOI: 10.48550/ARXIV.1906.06789. URL: https://arxiv.org/abs/1906.06789.

[Liu+21] Liu, M., Fang, S., Dong, H., and Xu, C. "Review of digital twin about concepts, technologies, and industrial applications". en. In: *Journal of Manufacturing Systems*. Digital Twin towards Smart Manufacturing and Industry 4.0 58 (Jan. 2021), pp. 346–361. ISSN: 0278-6125. DOI: 10.1016/j.jmsy.2020.06.017. URL: https://www.sciencedirect.com/science/article/pii/S0278612520301072 (visited on 01/08/2023).

[Loc20] Lochbaum, J. "Erzeugung von Testdaten für automatisiertes Fahren auf Basis eines Open Source Fahrsimulators". ger. In: (Dec. 2020).

[Map20] MapChart. *A Quick Guide to Map Projections*. en-US. Sept. 2020. URL: https://blog.mapchart.net/misc/quick-guide-to-map-projections/ (visited on 01/12/2023).

[Mar+21] Martinez-Gonzalez, P., Oprea, S., Castro-Vargas, J. A., Garcia-Garcia, A., Orts-Escolano, S., Garcia-Rodriguez, J., and Vincze, M. *UnrealROX+: An Improved Tool for Acquiring Synthetic Data from Virtual 3D Environments*. arXiv:2104.11776 [cs]. Apr. 2021. DOI: 10.48550/arXiv.2104.11776. URL: http://arxiv.org/abs/2104.11776 (visited on 01/08/2023).

[Mül+22a] Müller, T., Evans, A., Schied, C., and Keller, A. "Instant Neural Graphics Primitives with a Multiresolution Hash Encoding". In: *ACM Trans. Graph.* 41.4 (July 2022). Place: New York, NY, USA Publisher: ACM, 102:1–102:15. DOI: 10.1145/3528223.3530127. URL: https://doi.org/10.1145/3528223.3530127.

[Mül+22b] Müller, T., Evans, A., Schied, C., and Keller, A. "Instant Neural Graphics Primitives with a Multiresolution Hash Encoding". In: *ACM Trans. Graph.* 41.4 (July 2022). Place: New York, NY, USA Publisher: ACM, 102:1–102:15. DOI: 10.1145/3528223.3530127. URL: https://doi.org/10.1145/3528223.3530127.

[NVI] NVIDIA. *NVIDIA Omniverse*. URL: https://www.nvidia.com/en-us/omniverse/.

[OGR] OGRE. *OGRE3D*. URL: https://www.ogre3d.org/.

[Özl18] Özlü, A. *Color Recognition*. 2018. URL: https://github.com/ahmetozlu/color_recognition.

[Pet19] Petroff, M. A. "Pannellum: a lightweight web-based panorama viewer". In: *Journal of Open Source Software* 4.40 (2019). Publisher: The Open Journal, p. 1628. DOI: 10.21105/joss.01628. URL: https://doi.org/10.21105/joss.01628.

[pro] providentia. *Mobilität am Robotik-Lehrstuhl der TUM*. de-DE. URL: https://innovation-mobility.com/ (visited on 01/12/2023).

[QY16] Qiu, W. and Yuille, A. *UnrealCV: Connecting Computer Vision to Unreal Engine*. arXiv:1609.01326 [cs]. Sept. 2016. DOI: 10.48550/arXiv.1609.01326. URL: http://arxiv.org/abs/1609.01326 (visited on 01/08/2023).

[RAK21] Richter, S., AlHaija, H., and Koltun, V. *Enhancing Photorealism Enhancement*. May 2021.

[Ric+16] Richter, S., Vineet, V., Roth, S., and Koltun, V. "Playing for Data: Ground Truth from Computer Games". In: vol. 9906. Oct. 2016. ISBN: 978-3-319-46474-9. DOI: 10.1007/978-3-319-46475-6_7.

[Rob] Robotec.ai. *o3de-ros2-gem/ros2-gem.md at development · RobotecAI/o3de-ros2-gem*. en. URL: https://github.com/RobotecAI/o3de-ros2-gem (visited on 01/08/2023).

[SH20] Schrotter, G. and Hürzeler, C. "The Digital Twin of the City of Zurich for Urban Planning". en. In: *PFG – Journal of Photogrammetry, Remote Sensing and Geoinformation Science* 88.1 (Feb. 2020), pp. 99–112. ISSN: 2512-2819. DOI: 10.1007/s41064-020-00092-2. URL: https://doi.org/10.1007/s41064-020-00092-2 (visited on 01/08/2023).

[SCS12] SCS Software. *Euro Truck Simulator 2*. Oct. 2012. URL: https://eurotrucksimulator2.com/.

[Sha+17] Shah, S., Dey, D., Lovett, C., and Kapoor, A. *AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles*. arXiv:1705.05065 [cs]. July 2017. DOI: 10.48550/arXiv.1705.05065. URL: http://arxiv.org/abs/1705.05065 (visited on 01/08/2023).

[She+22] Shechter, M., Hanocka, R., Metzer, G., Giryes, R., and Cohen-Or, D. *NeuralMLS: Geometry-Aware Control Point Deformation*. arXiv:2201.01873 [cs]. June 2022. URL: http://arxiv.org/abs/2201.01873 (visited on 01/08/2023).

[Sta18] Stanford Artificial Intelligence Laboratory et al. *Robotic Operating System*. May 2018. URL: https://www.ros.org.

[Ste] Steam, E. *Steam Community :: Euro Truck Simulator 2*. en. URL: https://steamcommunity.com/app/227300/workshop/ (visited on 01/08/2023).

[Str21] Strous, L. *Astronomy Answers: Position of the Sun*. July 2021. URL: https://www.aa.quae.nl/en/reken/zonpositie.html (visited on 01/29/2023).

[To+18] To, T., Tremblay, J., McKay, D., Yamaguchi, Y., Leung, K., Balanon, A., Cheng, J., Hodge, W., and Birchfield, S. *NDDS: NVIDIA Deep Learning Dataset Synthesizer*. 2018.

[UNI] UNIGINE. *UNIGINE*. URL: https://unigine.com/.

[Uni] Unity. *Unity*. URL: https://unity.com/.

[VM21] VanDerHorn, E. and Mahadevan, S. "Digital Twin: Generalization, characterization and implementation". In: *Decision Support Systems* 145 (2021), p. 113524. ISSN: 0167-9236. DOI: https://doi.org/10.1016/j.dss.2021.113524. URL: https://www.sciencedirect.com/science/article/pii/S0167923621000348.

[Vie22] ViewApp. *CityDriver*. 2022. URL: https://www.city-driver.com/.

[WBE22] Wagener, N., Beckmann, J., and Eckstein, L. "Efficient Creation of 3D-Virtual Environments for Driving Simulators". In: *2022 International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*. 2022, pp. 1–6. DOI: 10.1109/ICECCME55909.2022.9988421.

[Wor22] Wortmann, A. *Digital Twins*. 2022. URL: https://awortmann.github.io/research/digital_twins/ (visited on 10/26/2022).

[Zhe+22] Zheng, O., Abdel-Aty, M., Yue, L., Abdelraouf, A., Wang, Z., and Mahmoud, N. *CitySim: A Drone-Based Vehicle Trajectory Dataset for Safety Oriented Research and Digital Twins*. arXiv:2208.11036 [cs, stat]. Aug. 2022. DOI: 10.48550/arXiv.2208.11036. URL: http://arxiv.org/abs/2208.11036 (visited on 01/08/2023).