TU WIEN Informatics

Automated Digital Content Creation from Point Clouds and Image Data

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Georg Schenzel, BSc
Matrikelnummer 01633078

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Mag. Dr.techn. Peter Kán
Mitwirkung: Univ.Prof. Mag.rer.nat. Dr.techn. Hannes Kaufmann

Wien, 14. Oktober 2024

Georg Schenzel	Peter Kán

TU WIEN Informatics

Automated Digital Content Creation from Point Clouds and Image Data

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Georg Schenzel, BSc
Registration Number 01633078

to the Faculty of Informatics

at the TU Wien

Advisor: Mag. Dr.techn. Peter Kán
Assistance: Univ.Prof. Mag.rer.nat. Dr.techn. Hannes Kaufmann

Vienna, October 14, 2024

Georg Schenzel	Peter Kán

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Georg Schenzel, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 14. Oktober 2024

Georg Schenzel

Danksagung

Ich möchte diese Gelegenheit nutzen, um allen zu danken, die mich während dieser Diplomarbeit unterstützt haben.

Als aller erstes, meine tiefste Dankbarkeit geht an meinen Betreuer Peter Kán, der mir von Anfang an wertvolle Unterstützung gegeben hat. Er hat mir geholfen, meine Ideen zu entwickeln, und sein ständiges konstruktives und detailliertes Feedback hat wesentlich zur Qualität dieser Arbeit beigetragen.

Ich möchte auch meinem Co-Betreuer Hannes Kaufmann meinen Dank aussprechen. Er hat mir vor allem beim Start dieses Projektes geholfen und die Zusammenarbeit mit Magna ermöglicht.

Danke an alle Mitarbeiter von Magna, mit denen ich zusammenarbeiten durfte: Daniel Schleicher, Banu Bueyueker und Pavlo Tkachenko. Sie haben mich in die richtige Richtung gelenkt und mir wertvolles Feedback gegeben.

Danke an Peter Widmer von NavVis. Ich weiß es sehr zu schätzen, dass er nach Wien gekommen ist und mir den VLX-Laserscanner zur verfügung gestellt hat, der es mir ermöglicht hat hochwertige daten zu generieren, die für die erzielten Ergebnisse unerlässlich waren.

Abschließend möchte ich mich bei meiner Familie und meinen Freunden herzlichst bedanken. Ihre Unterstützung und ihr Ansporn haben mir geholfen, während all der Monate die ich mit Entwickeln, Experimentieren und Schreiben dieser Arbeit verbracht habe, motiviert und konzentriert zu bleiben.

Acknowledgements

I would like to take this opportunity to thank everyone who supported me throughout the process of writing this thesis.

First and foremost, my deepest gratitude goes to my supervisor, Peter Kán, for providing invaluable guidance from the very beginning. He helped me develop my ideas, and his continuous, constructive, and detailed feedback greatly contributed to the quality of this work.

I would also like to extend my thanks to my co-supervisor, Hannes Kaufmann, for his help in getting this project started and for facilitating the collaboration with Magna.

Thanks to everyone from Magna with whom I had the pleasure of working: Daniel Schleicher, Banu Bueyueker, and Pavlo Tkachenko. They helped guide me in the right direction and provided valuable feedback.

Thanks to Peter Widmer from NavVis. I highly appreciate that he came to Vienna and let me use the VLX laser scanner, which allowed me to generate the high-quality input data essential for the results I achieved.

Lastly, I want to express my heartfelt thanks to my family and friends. Their support and encouragement helped me stay motivated and focused throughout all the months I spent developing, experimenting, and writing this thesis.

Kurzfassung

Diese Arbeit stellt eine Pipeline vor, die reale Daten wie Punktwolken und Bilder nutzt, um digitale Zwillinge für autonome Fahrsimulationen zu erstellen. Simulationen spielen eine entscheidende Rolle bei der Entwicklung sicherer autonomer Fahrsysteme, da sie kostengünstige und risikofreie Tests ermöglichen. Um die Zuverlässigkeit dieser Simulationen zu gewährleisten, müssen die virtuellen Umgebungen der realen Welt sehr ähnlich sein. Unsere Pipeline erzeugt aus den Eingabedaten hochwertige 3D-Meshes mit fotorealistischen Texturen. Zusätzlich wird eine semantische Segmentierung des rekonstruierten 3D-Modells durchgeführt, die als Grundlage für nachfolgende Simulationsanwendungen dient. Diese semantische Segmentierung wird durch einen Virtual-View-Ansatz erreicht, bei dem 2D-Renderings der Szene mit einem vortrainierten Modell segmentiert werden und die Ergebnisse dann auf die 3D-Szene zurückprojiziert werden. Informationen über den Straßenverlauf und die Fahrspuren werden aus OpenStreetMap bezogen und mit dem 3D-Modell überlagert. Schlussendlich wird das Ergebnis der Pipeline zur Erstellung einer virtuellen Umgebung im Fahrsimulator CARLA verwendet.

Wir haben Bild- und Punktwolkendaten von drei verschiedenen Orten gesammelt und die Pipeline mit diesen Daten getestet. Wir haben die Unterschiede in den Rekonstruktionen aus beiden Eingabemodalitäten verglichen und ihre Effektivität für praktische Anwendungen evaluiert. Die Rekonstruktionen der Szenen wurden manuell semantisch annotiert, um Referenzwerte für die quantitative Evaluierung des 3D semantischen Segmentationsalgorithmus zu erhalten.

Die Pipeline wurde in Python mit dem Ziel eines hohen Automatisierungsgrades implementiert. Sie ist in der Lage innerhalb weniger Stunden einen qualitativ hochwertigen digitalen Zwilling zu erstellen, wobei der/die Benutzer/in weniger als 20 Minuten für manuelle Tätigkeiten benötigt. Der semantische Segmentationsalgorithmus erreicht einen mIoU-Wert von 55,2 und einen F1-Wert von 67,1, was eine gute Leistung der Segmentierung der Gitterpunkte in unseren Datensätzen widerspiegelt. Dieser Ansatz ist ein Schritt nach vorne für eine sicherere und schnellere Entwicklung von automatisierten Fahrsystemen.

Abstract

This thesis presents a pipeline that leverages real-world data, such as point clouds and images, to create digital twins for autonomous driving simulations. Simulations play a crucial role in the development of safe automated driving systems, as they enable cost-effective and risk-free testing. To ensure the reliability of these simulations, virtual environments must closely resemble the real world. Our pipeline generates high-quality 3D meshes with photorealistic textures from the input data. Additionally, a 3D semantic segmentation of the reconstructed mesh is performed, providing ground truth data for downstream simulation tasks. This semantic segmentation is achieved using a virtual-view approach, where 2D renderings of the scene are segmented with an off-the-shelf model, and the predictions are projected back into the 3D scene. Information about the road layout and lanes is obtained from OpenStreetMap and aligned with the mesh. Finally, the pipeline output is used to create a virtual map in the driving simulator CARLA.

We captured image and point cloud data from three locations and tested the pipeline using this input. We compared the differences in reconstructions from both input modalities, assessed their feasibility, and evaluated their effectiveness for practical applications. Reconstructions of the scenes were manually semantically annotated to provide ground truth for quantitative evaluation of the 3D semantic segmentation algorithm.

The pipeline was implemented in Python with the goal of achieving a high degree of automation. It can produce a high-quality digital twin in a matter of hours, requiring minimal user intervention of under 20 minutes. The semantic segmentation algorithm achieves an mIoU of 55.2 and an F1 score of 67.1, reflecting a good performance for labeling the vertices of our datasets. This streamlined approach is a step forward for safer and faster development of automated driving systems.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**
1.1 Goals of the Thesis . . . 1
1.2 Methodology . . . 2
1.3 Structure of the Thesis . . . 3

2 Related Work **5**
2.1 Autonomous Driving . . . 5
2.2 Driving Simulations . . . 6
2.3 Mesh Reconstruction . . . 7
2.4 3D Semantic Segmentation . . . 8
2.5 Datasets . . . 9

3 Pipeline **11**
3.1 Requirements . . . 11
3.2 Pipeline Overview . . . 14
3.3 Input Data . . . 16
3.4 Mesh Reconstruction . . . 18
3.5 Semantic Segmentation . . . 21
3.6 Georeferencing . . . 24
3.7 Roads . . . 25
3.8 Manual Actions . . . 26

4 Implementation **31**
4.1 Dependencies . . . 31
4.2 Pipeline . . . 32
4.3 Reconstruction . . . 36
4.4 Semantic Segmentation . . . 38
4.5 Road Networks . . . 41

4.6 Mesh Splitting 42

5 Evaluation **43**
5.1 Data 44
5.2 Reconstruction 53
5.3 Segmentation 70
5.4 Automation 77

6 Conclusion **79**
6.1 Limitations 80
6.2 Future Work 81

Overview of Generative AI Tools Used **83**

List of Figures **85**

List of Tables **87**

Bibliography **89**

CHAPTER 1

Introduction

Advancements in computer vision, vehicle dynamics, and the availability of better sensor modalities are making automated driving systems (ADSs) increasingly relevant. Simulations play a crucial role in the development and testing of ADSs. These simulations require high-quality virtual environments to ensure realistic and effective testing scenarios.

Digital twins of real-world scenes are valuable for simulating edge cases or identifying issues in specific scenarios. However, creating such a digital twin is both time-consuming and expensive. These scenes require high-quality, accurate meshes for physical driving simulations, watertight and artifact-free surfaces, and high-quality textures for photo-realistic rendering. Additionally, the models should include semantic labels to provide ground truth for downstream tasks, as well as information about the road network and traffic signs to enable automatic navigation and simulation of vehicles and pedestrians.

Instead of manually creating such scenes from scratch, data from laser scans or images can be used to aid their creation. Many tools can produce 3D reconstructions from this data, but they typically only produce a 3D mesh of the scene. Performing a semantic segmentation of the mesh and identifying the road metadata is a complex task with no out-of-the-box solution.

1.1 Goals of the Thesis

This thesis aims to produce a (semi-) automatic pipeline to aid in creating simulation-ready virtual scenes for the simulation platform CARLA [14]. CARLA requires two pieces of data to create a virtual world: a semantically segmented 3D mesh and a road network specification in the form of an OpenDrive [3] file. The mesh reconstruction should use either point clouds or images captured at the real-world scene as input and produce a mesh with high-quality textures. CARLA requires the meshes to be split into

sub-meshes depending on their semantic class. The whole process is desired to be as automatic as possible, requiring minimal manual intervention.

The next goal was to explore the differences between the two input modalities, evaluate the resulting reconstructions, and assess the feasibility of using such data from real-world scenes. Doing so requires image and point-cloud input datasets and ground truth semantically annotated reconstructions. The last goal was to define requirements for the input data and produce guidelines on how to properly capture images and point clouds.

1.2 Methodology

1.2.1 Establishing Requirements

This thesis was done on behalf of Magna Engineering Center Steyr GmbH & Co KG[1]. We engaged with experts in autonomous driving simulations to understand their use cases, requirements, and data acquisition options. This collaborative approach ensured that the research was aligned with industry needs and relevant to the challenges faced in autonomous driving simulations.

1.2.2 Literature Research

This thesis spans multiple different research fields. In an extensive literature review, these fields were covered. This included the research of use cases for autonomous driving simulations to understand how the output data of this thesis is used in the development and testing of ADSs. Furthermore, the state-of-the-art in urban 3D datasets, 3D reconstruction from point clouds and images, and 3D semantic segmentation of meshes was researched.

1.2.3 Pipeline Implementation

Initially, we tested the whole process with a proof of concept partial implementation of the pipeline to assess the feasibility of our approach. We captured a small image-based dataset, manually performed a mesh reconstruction with RealityCapture, and then performed a semantic segmentation of the mesh using the preliminary implementation of the segmentation algorithm. This initial assessment showed promising results. The mesh and semantic segmentation showed potential, and the initial driving tests in CARLA were successful.

Afterward, a modular and flexible pipeline was built to automate the various steps required for the task. This pipeline allowed for experimentation with different approaches and implementations, which made it possible to improve the produced results iteratively. This pipeline was adjusted based on feedback from Magna to ensure it met their specific needs. Two different reconstruction strategies were implemented with the pipeline: reconstruction

[1] https://www.magna.com/company/company-information/magna-groups/magna-powertrain

from images and reconstruction from point clouds. A virtual view semantic segmentation approach was implemented to split the mesh into semantic sub-meshes. With this method we could leverage pre-trained 2D segmentation models to project labeled 2D renderings of the scene back onto the mesh.

1.2.4 Data Aquisition

Data acquisition involved capturing image-based and point cloud-based datasets. For the image-based data, we used a handheld DSLR camera to capture various intersections with hundreds to a few thousand images each. The point cloud data was collected using the VLX mobile mapping device, which was generously provided by NavVis[2].

1.2.5 Dataset Creation

Following the pipeline development and data acquisition, the next step was to create datasets to allow for the evaluation of the pipeline. This involved generating 3D reconstructions from both the captured images and point clouds. These reconstructions were then cleaned and manually annotated to provide a ground truth to test the segmentation algorithm. Creating these datasets was labor-intensive but essential for producing reliable results in the evaluation phase.

1.2.6 Evaluation

The evaluation phase involved quantitative assessments and descriptive analysis of the pipeline and its reconstruction and segmentation results. We analyzed the segmentation quality with the help of our manually annotated 3D dataset. Reconstructions from both data sources, images, and point clouds were compared against each other. Additionally, we examined the impact of various parameters on the pipeline performance to optimize the outcomes and better understand the strengths and limitations of different approaches.

1.3 Structure of the Thesis

In Chapter 2, the results of our extensive literature review are presented, highlighting state-of-the-art methods and technologies relevant to the different tasks at hand. In Chapter 3 the overall design of the pipeline is described. This includes coverage of our defined requirements, a high-level overview of the pipeline, and a detailed breakdown of each individual step performed. Chapter 4 goes into more detail about the implementation of various different tasks that are performed by the pipeline. In Chapter 5, our created dataset, the pipeline outputs, and our evaluation are presented. Chapter 6 concludes this work by giving a brief summary, highlighting some limitations of the implementation, and proposing related topics for future works.

[2] https://www.navvis.com/

CHAPTER 2

Related Work

2.1 Autonomous Driving

The field of autonomous driving is vast, with numerous approaches and emerging technologies for automation at varying levels [60]. Many advancements are due to progress in deep learning and artificial intelligence, with deep learning being applied to challenges such as scene perception, localization, and path planning [23, 27].

Convolutional Neural Networks (CNNs) and Recurrent Neural Networks' (RNNs) ability to act as object detectors makes them suitable for use in autonomous driving systems [27]. Deep learning is also used to tackle the problem of localization and mapping. It offers a data-driven alternative to traditional model-based methods like SLAM. This is enabled by the increasing amount of data and computational power available [27].

Chib et al. [11] categorize autonomous driving systems using deep learning into modular and end-to-end architectures. Modular architectures divide the problem into different sub-tasks. Such sub-tasks include object detection, localization, semantic segmentation, path planning, and vehicle control. It relies on the sensor data and multiple algorithms and models to produce control outputs. This approach provides a few challenges: First, errors that occur in the output of a sub-task can be propagated to later tasks. For example, a misclassified object might negatively affect route planning, leading to dangerous situations. Second, managing the different modules increases the complexity of such systems. Third, these systems can be inefficient due to duplicate and unnecessary calculations.

End-to-end architectures try to tackle these challenges. In an end-to-end system, sensor inputs are mapped directly to control outputs, and only a single model is trained. This eliminates the chance of error propagation and provides a more efficient system. Due to recent advancements in this field, such systems are no longer a black box. They can generate auxiliary outputs, attention maps, and interpretable maps that can help to reason about the system and identify sources of errors [11].

Self-driving cars have many advantages. For users, it provides a stress-free way of transportation with potentially faster commute times. They can assist governments in traffic enforcement, enhancing road capacities, and reducing the number of accidents by reducing distracted or drunk driving. Furthermore, self-driving cars are a greener mode of transportation. By reducing car ownership, less parking space is needed, and optimal fuel consumption can be ensured. With shared access to self-driving cars, this can still be an efficient, personalized, and reliable way of transportation [27].

2.2 Driving Simulations

Testing and validating ADSs before deployment on the road is crucial to ensure the safety of all traffic participants. Performing these tests is expensive, highly regulated, and not risk-free. Simulations can be used to check how ADSs behave in various different and controlled virtual scenarios [31, 60]. The simulations are run on platforms built on modern game engines [14] or even in commercial video games [60]. This thesis focuses on simulations running inside CARLA [14].

CARLA[1] (Car Learning to Act) is an open-source simulator for autonomous driving research. It is built on Unreal Engine 4[2], which offers high-fidelity graphics and physics simulations. CARLA is built in a client-server architecture, where the server handles the simulation, including physics and graphics. The client communicates with the server to interact with the simulation. This includes controlling vehicles, receiving sensor data, or changing the world. The client is implemented as a Python API [14].

To declare road networks used in the simulations OpenDrive [3] is used. OpenDrive is a specification and file format containing information about road geometry, lanes, road markings, and additional features like signs and traffic lights.

CARLA is used by various state-of-the-art autonomous driving systems. Osiński et al. [47] trained a reinforcement-learning-based model with simulations in CARLA and showed a successful sim-to-real policy transfer. Their model receives only RGB images and 2D semantic segmentation maps rendered by CARLA as input. It then outputs the vehicle controls. They tested their system in different driving scenarios, where the vehicle had to follow a route from a given list of checkpoints. For domain randomization, they use ten different weather conditions in CARLA, different simulation qualities, cama settings, and image augmentations.

Gutiérrez-Moreno et al. [28] trained and tested another deep reinforcement learning model that handles difficult intersections with CARLA. Since CARLA is very computationally intensive, the model was trained using a simpler simulation platform before being refined using CARLA.

To provide a suite of scenarios for training and testing autonomous driving systems, CARLA Real Traffic Scenarios (CRTS) has been built. It provides a suite of scenarios

[1] https://carla.org/
[2] https://www.unrealengine.com

based on real-world traffic conditions and contains tactical tasks that last several seconds [46].

Gomez et al. [22] used CARLA to evaluate their fully autonomous driving architecture using another suite of challenging driving scenarios. These scenarios include stop signs, adaptive cruise control, pedestrians crossing, and pedestrians unexpectedly jumping on the street. CARLA is also being used to create synthetic datasets by rendering the different sensors and various scenarios [45], lowering the cost of development of such systems [47].

2.3 Mesh Reconstruction

Several methods exist for reconstructing 3D meshes from real-world measurements. Depending on the input modality, different solutions are used.

Point Clouds

Point clouds can be obtained from LiDAR scanners. Reconstructing meshes from point clouds is a well-established problem. Poisson surface reconstruction aims to reconstruct a smooth surface from a set of points and their normals. The goal is to find an implicit function that is zero at each point and its gradient equal to the corresponding normal. This function can be obtained by utilizing Poisson's equation. From this function, a mesh is created. Poisson surface reconstruction considers all points at once, creates watertight models, and produces smooth surfaces that approximate noisy data [37, 38].

The Ball Pivoting Algorithm reconstructs surfaces by rolling a virtual ball over the point cloud. The ball is rotated around edges until it hits a third vertex, forming a new triangle. This method is effective for point clouds with an even distribution. Water tightness of the resulting model is only guaranteed if the sampled points are denser than the radius of the ball [6].

Images

Images can be obtained manually with a handheld camera, from cameras mounted on a capture device [12] (e.g., a car or backpack), or from UAVs [39]. Reconstructing meshes from a set of images is a more complex task, solved with photogrammetry in software like Meshroom [24] or RealityCapture[3].

Photogrammetry is a process that encompasses many different steps. It starts with feature extraction, where distinctive points in the input images are detected. For robustness, these features must be scale and rotation invariant. The next step is to find pairs of images that overlap. This is done by comparing and matching the features of all images together. Once the images are matched together, the features of pairs of images must be matched. From such a matching, the 3D motion between camera positions can be

[3] https://www.capturingreality.com/

calculated. Using Structure from Motion [8, 51], these matched features are turned into candidate points in 3D space space. A depth map from each image is calculated and then all depth maps are merged to form a point cloud from the whole scene. This point cloud is then meshed and textured by projecting the images onto the mesh [25].

2.4 3D Semantic Segmentation

3D semantic segmentation is an active research field with various solutions for different domains and input modalities. We can differentiate between the segmentation of point clouds and meshes.

Deep neural networks (DNNs) are a prevalent way to tackle this problem. Initially, the research focused on segmenting point clouds. Some models directly consume point clouds [9, 48, 55]. Other models use a view-based approach, where 2D views of the point cloud are fed into a convolutional neural network [41, 7]. 3D point cloud models can be improved by fusing the points with features of 2D views (images) of the scene [35, 21]. Self-attention can be used to enable scalability to scenes with millions of points [63].

Since meshes have some advantages over point clouds for scene representation [2, 17, 58, 29], research shifted to also perform semantic segmentation on 3D meshes. Meshes are more efficient than point clouds for storing large flat surfaces, requiring fewer data points and less memory. Furthermore, they allow for detailed texture storage. Designing a Deep Neural Network (DNN) to work on meshes presents the challenge of having to process their unstructured format while also implementing learning operations such as pooling, convolution, and feature aggregation [2, 29].

Hanock et al. [29] solved this by applying convolutions on edges and the four edges of their incident triangles. Pooling is applied as an edge collapse operation that retains surface topology. Graph-based neural networks are often used for deep learning tasks involving meshes. Some models use a center of gravity (CoG) approach, where each face is considered as a node in the graph. Additionally, features are extracted from the local texture and fused with the nodes [58, 62, 61]. Gao et al. also use a graph-based neural network. Instead of CoGs, they perform a prior over-segmentation that looks at the planarity of the mesh and then use a graph convolutional network on a graph with the segments as nodes [19]. Another approach is to sample points on the mesh, segmenting the resulting point cloud, and then interpolating the labels back to faces, vertices, or texels [26].

Similarly to the segmentation of point clouds, a view-based approach can be used on meshes, where multiple renderings of the 3D scene are used as the input for the neural network [40, 1, 53]. A view-based method poses the advantage of being able to use 2D models for the segmentation task, for which much larger and more diverse datasets exist. Off-the-shelf 2D semantic segmentation models [64, 57, 59, 10] can be used for this. MMSegmentation[4] is a semantic segmentation toolbox written in Python that provides

[4] https://github.com/open-mmlab/mmsegmentation

various models pre-trained on a diverse range of datasets. Especially interesting for this thesis are models pre-trained on CityScapes [12].

2.5 Datasets

Semantically labeled 3D datasets of urban scenes exist for both point clouds [5, 16, 30, 39, 17] and meshes [58, 42, 39, 18, 17]. However, we need to also consider the scale and resolution of these datasets. Existing urban-scale datasets lack the granularity of classes required for the task at hand. While most of them contain labels like building, car, and street, they often do not provide labels for traffic signs and sidewalks, which are of high interest to identify in this project. A few point cloud-based datasets provide this desired granularity [5, 16].

CHAPTER 3

Pipeline

This chapter outlines the design of the pipeline. We start by discussing the data, software, and usability requirements. We provide a high-level overview of the various pipeline configurations based on the different input modalities. Then, each step the pipeline performs, and the corresponding design decisions are described in detail. The key topics covered include mesh reconstruction, semantic segmentation, georeferencing, road network generation, and manual operations.

3.1 Requirements

In this section, the pipeline's requirements regarding its usage, input data, and output data are presented. These requirements were established through discussions with Magna, aligning with their intended use cases and specifications. While some requirements were defined at the start of the thesis, they evolved during the development process as new insights emerged and access to more data was available. Additionally, insights were gained from preliminary manual tests involving mesh reconstructions and imports into CARLA.

We define the requirements regarding the pipeline's outputs, which are a 3D mesh, a semantic segmentation of said mesh, and aligned road network information. Furthermore, we define requirements for the software itself, including automation, flexibility, and traceability.

3.1.1 Mesh Output

The mesh output of the pipeline is used for two different tasks: the rendering of virtual scenes and 3D vehicular physics simulations. These virtual scenes should be photorealistic and accurately represent their real-world counterpart.

To achieve photorealistic rendering, the mesh requires high-resolution textures. Road markings, traffic signs, and text should be sharp and identifiable as such. This is important for downstream tasks, such as testing autonomous driving systems. Ideally, such textures would be extracted directly from the input data.

The mesh should be smooth, manifold, and contain no holes. Artifacts such as cut-off objects, dents in the road, or added geometry should be reduced to a minimum. This is crucial to allow cars to drive in the scene during the physics simulation.

The final mesh must be provided in the `fbx` format to be usable in the CARLA import process. This mesh must be split into sub-meshes based on their semantic classes. These sub-meshes should also be named appropriately, conforming to the naming convention CARLA uses for automatically assigning the correct class labels to the meshes.

3.1.2 Semantic Segmentation

To perform a semantic segmentation of the mesh, we first need to define the desired classes. The required granularity of the semantic segmentation comes from the target downstream tasks. One of them is being able to move, swap, or remove traffic signs and cars. To do so, we require classes for these objects. Knowing where the sidewalk is can help in performing pedestrian simulations. Autonomous driving simulations require further knowledge about the road and sidewalk. These use cases leave us with the required semantic classes of *street, sidewalk, car,* and *traffic signs.*

Many downstream semantic segmentation tasks are trained on the CityScapes [12] dataset, which contains the classes mentioned above and classes of similar granularity. This is why classes defined in the CityCscapes are a very good fit for this thesis.

3.1.3 Road Network

CARLA uses an OpenDrive file (`xodr`) to generate its internal representation of the road network. A graph is built from each node and connection defined in the file. This graph contains nodes for each lane in the street. The `xodr` file uses a local coordinate system that is then used by CARLA. Global positioning of the whole map is provided in this file to allow georeferencing.

The road graph should be aligned with the mesh. To do so, the local coordinate systems of the mesh and the OpenDrive file must be aligned. The individual nodes of the road graph can be manually moved and corrected in CARLA, so a perfect alignment is not required. The level of detail of roads and lanes available on OpenStreetMap[1] is sufficient for this task.

[1] `https://www.openstreetmap.org`

3.1.4 The Pipeline

Automation The primary goal of the pipeline is to provide a tool that significantly reduces the cost of creating digital twins of street scenes. Automation of each step, as well as execution of all steps together, is crucial in this cost reduction. While full automation is not a strict requirement, the integration of manual steps should be minimized and, where possible, consolidated to optimize efficiency. These manual steps can sometimes offer a more efficient solution for tasks beyond this thesis's scope. These tasks include the selection of a reconstruction region, selecting sampling points on the roads, fixing errors in the segmentation, and fine-tuning the alignment of the road and mesh. Ideally, these manual steps would be bundled together, thus reducing the need for constant monitoring and intervention throughout the pipeline's operation.

Flexibility The pipeline is designed to serve as a robust platform for experimentation with various approaches and algorithms to address the diverse challenges involved. This flexibility allows for the evaluation of the performance of different strategies in this thesis. Furthermore, the pipeline may provide a foundation for future research and development. To achieve these objectives, the layout of the full pipeline should be easily modifiable. There must be a clearly defined way for managing the in- and outputs of the different steps. Additionally, each step should be capable of exposing a set of parameters, allowing tuning of the behavior of their operation.

Traceability Traceability is an important requirement, ensuring that each modification and process within the pipeline is observable and accountable. This involves making the output files of each step viewable. This is important for debugging purposes as well as the evaluation of the individual steps. To further increase the transparency of the pipeline, additional diagnostic data should be stored in the intermediate files. If the user identifies an error in the output, reviewing the intermediate files and identifying where it occurred must be possible. The issue could be addressed by adjusting the parameters or manually correcting the error in an external tool. Afterward, it must be possible to rerun the pipeline from this location.

Change Detection The pipeline should have the capability to detect changes in intermediate files, serving two primary purposes. Firstly, it allows the pipeline to automatically skip steps that have already been completed, which is useful for recovering from interruptions or crashes. Secondly, it allows for the manual correction of intermediate files, as mentioned above. The pipeline should only run steps where the inputs are newer than the outputs. This avoids redundant processing and enhances the efficiency of the pipeline.

3.2 Pipeline Overview

Our proposed pipeline can work on different input modalities, each of which requires very different processing steps. Two pipeline layouts have been designed for this thesis. A high-level overview of both variants is given in Fig. 3.1.

First, the input data is fed through a set of steps dependent on the chosen input modality. Two strategies are implemented: one for processing images using photogrammetry and another for handling point clouds. Each input-specific strategy produces two pieces of data: a reconstructed and textured 3D mesh and georeferencing information of said mesh.

From then on, the pipeline is mostly independent of the input modality. The pipeline continues by running the same set of steps. The remaining part handles the 3D semantic segmentation, `xodr` file generation, further processing of meshes and metadata, and the final import into CARLA.

3.2.1 Configuration

Running the pipeline requires many parameters to be set. This includes the choice of input modality, the project name, and potentially the geographic base point. Furthermore, the user may need to set the paths to some executables on which the pipeline depends. These parameters are passed in a `yaml` configuration file.

The following executables can be configured:

- **carla**: The path to the directory of the CARLA repo, which must be built from source.
- **VCVars64.bat**: Launching CARLA requires the Microsoft C++ toolset, which is shipped with a Visual Studio installation. Default config value: default installation location of Visual Studio 2019 Community Edition.
- **CloudCompare.exe**: Default config value: default installation location of CloudCompare.
- **Blender.exe**: Default config value: default installation location of Blender 4.1.
- **RealityCapture.exe**: Default config value: default installation location of RealityCapture.

In addition to these required input arguments, each step of the pipeline exposes a multitude of parameters to allow fine-tuning its behavior. When running the pipeline, these parameters can be passed via the command line interface or be set in the config file for easier use and reproducibility. Sensible default values have been set for each of them. These values have been established by the experiments performed in Chapter 5.

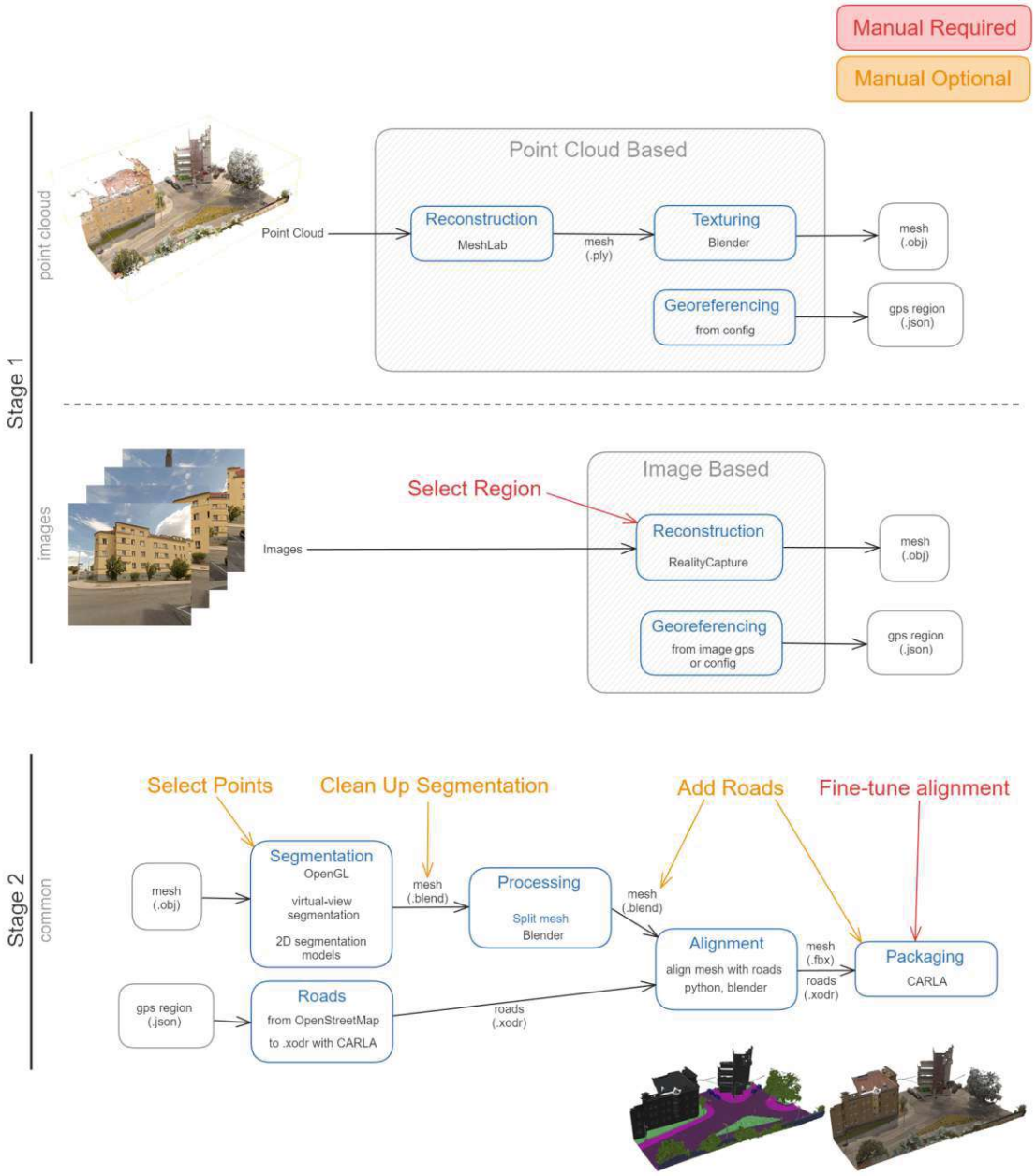


Figure 3.1: An overview of the pipeline. Showing the steps performed for reconstructions from point clouds (top) and images (middle). On the bottom, the common subsequent steps are shown. Manual steps that are required (red) or optional (orange) are highlighted at the specific point of the pipeline where they occur.

3.3 Input Data

This section discusses the properties and requirements of the two types of input data: images and point clouds.

3.3.1 Images

Images are used for the pipeline's photogrammetry strategy. There are many factors to consider when selecting an appropriate camera, configuring it, and capturing the images to get good results in this process. This thesis uses RealityCapture for the photogrammetry step. Since it is a closed-source application, we do not know what algorithms are used. We assume that with their implementation, we have requirements similar to those for structure from motion (SfM). We derive the following guidelines from the literature about camera considerations for SfM [44, 34] and recommendations by the RealityCapture documentation [49].

Camera To extract as many features for alignment as possible, the camera needs to preserve a lot of detail, have a high dynamic range, and produce images with low noise. Using a high-resolution sensor for this is crucial. A resolution of at least 16 MP is recommended. Mosbrucker et al. [44] recommend a sensor size of at least $300\,\mathrm{mm}^2$ to reduce noise. A bit-depth of at least 14 bits allows information to be captured with a high dynamic range. Modern DSLR cameras fit these criteria.

Settings The camera must be configured such that it provides a proper exposure of the images. Exposure refers to the amount of light captured by the sensor. The change in exposure is measured in exposure value (EV). An increase of 1 EV means twice as much light being captured. Good exposure in this context means preserving information in the scene's dark and bright areas.

To achieve good exposure, three camera parameters need to be adjusted: Aperture, shutter speed, and ISO. Aperture refers to the size of the opening in the lens. It is measured in f-stops, which is inversely proportional to the aperture. A larger aperture results in more light being captured at the cost of a shallow depth of field, meaning only a narrow area of the scene is in focus. Reducing the aperture allows more of the scene to be in focus. Shutter speed is the duration the sensor is capturing light. A shutter speed that is too low can result in blurry images, especially for handheld or mounted cameras. ISO is the absolute sensitivity of the sensor. Twice the ISO value results in twice the amount of light captured. However, with higher ISO values, the images get increasingly noisy.

The optimal aperture is 1-3 f-stop values larger than the lens's minimum aperture. This reduces artifacts such as chromatic aberration and vignetting. Furthermore, it ensures more of the scene is in focus. Shutter speed should be chosen so that motion blur is minimized. This depends on how and where the camera is mounted, but generally, for lenses with a focal length of 50mm or lower a value of 1/100s to 1/400s is recommended for

this application. ISO should be chosen as low as possible to reduce noise. The optimization of these parameters can only be done in bright lighting conditions. Disabling autofocus, sensor-based stabilization systems, distortion correction, and noise reduction filters are recommended.

Capturing Once the camera and its settings are dialed in, we need to consider when and how to capture the images. Ideal lighting conditions are on bright, overcast days. This results in smooth lighting with fewer hard shadows and enough light for proper exposure. How the scene is captured is crucial for the photogrammetry process of aligning cameras and detecting features. Each part of the scene must be captured by at least three images to be reconstructed. Following the *coarse-to-fine* rule and covering the scene at different scales is required for optimal results.

This is done by first taking pictures of the whole scene at a greater distance, trying to capture a lot of context. Then, moving closer to capture more details. It is important to move closer gradually to allow the images at the different scales to be matched.

Another crucial rule is to always move around the scene and not just pivot at one spot. SfM relies on the parallax of the movement for its reconstruction. Only pivoting results in a panorama, where no depth information can be extracted. Furthermore, when moving around an object, it is beneficial to close loops around it.

Some of these guidelines are easily applied when capturing a single object. However, capturing urban scenes gets a bit more complicated, as there are many different objects of interest and limited options for moving around the scene. Ideally, each important object in the scene, such as cars and traffic lights, is covered by orbiting around them at different heights. Wide context shots can be covered by walking around the sidewalk and street. Furthermore, it is important also to capture detail shots facing towards the ground, as street surfaces often have a low textural variety, more data is needed for a proper reconstruction. A more detailed explanation of capturing images for photogrammetry in an urban setting can be seen in the sections 5.1.1 and 5.1.3.

GPS Locations embedded as EXIF metadata can be used for georeferencing. While some cameras have a GPS sensor built-in, this is usually not the case. While capturing the images, a mobile phone can track the GPS positions and store them in a `gpx` file. Software like GeoSetter[2] can then be used to set the GPS location of each image by comparing the timestamp with the entries in the `gpx` file. The camera's internal clock must be calibrated precisely.

Format The images should be captured in a `RAW` format, ensuring that the sensors capture the images with the highest possible quality. To avoid artifacts due to compression, the images should be converted to either `TIFF` or `PNG`.

[2] `https://geosetter.de/`

3.3.2 Point Clouds

When using a point cloud as the input to the pipeline, its quality directly influences the reconstruction quality. A single point cloud in the form of an `E57` file [33] is the basis for further processing. While an organized point cloud, such as one generated by a single sweep of a LiDAR sensor, may be used, it is insufficient for comprehensive scene coverage due to its limited perspective. Therefore, an unorganized point cloud that encompasses the entire scene is required.

The mesh reconstruction requires that per-point normals are available. Although normals can be calculated post-capture, in our tests this failed to produce correct normals for thin objects like traffic signs. Thus, it is advantageous if normals are generated during the capture or initial processing of the point cloud. The point cloud must also include RGB color information for each point, as the texturing step relies on this data. The maximum possible texture resolution depends on the point cloud density. Thus, the point cloud should be dense enough to preserve enough texture information.

Artifacts such as moving vehicles or pedestrians should be removed from the point cloud to avoid appearing in the reconstructed mesh. Alternatively, these artifacts can be removed from the reconstructed mesh by modifying the pipeline's intermediate files. Furthermore, the point cloud should be georeferenced. This can be done by providing the pipeline with a global base point, which aligns the local origin of the point cloud with its global geographical position.

3.4 Mesh Reconstruction

3.4.1 From Images

Performing mesh reconstruction with photogrammetry can be done with many commercial and non-commercial tools. In this thesis, RealityCapture[3] was used and evaluated for this task. RealityCapture can handle lots of input data. Thousands of images can be processed in a reasonable time of a few hours. Since RealityCapture is closed-source, we cannot go into detail about the exact algorithms used in their implementation.

The whole reconstruction process can run completely automatically. RealityCapture is executed over the command line, and multiple commands are chained together. The images are loaded into the program and aligned. At this point, a point cloud of the scene is can be seen.

RealityCapture can estimate the ground plane and reconstruction region, but sometimes this can fail to produce correct results. A manual step is required here to define these two properties correctly. A misaligned ground plane will hurt the performance of downstream steps. Furthermore, constraining the reconstruction region allows us to only reconstruct the area of interest. Areas outside the scene often produce more artifacts, so it is beneficial to cut them out.

[3] `https://www.capturingreality.com/`

After this manual step, we continue by performing the remaining operations in RealityCapture. The mesh is reconstructed using the aligned images. RealityCapture produces a mesh with millions of polygons, so a simplification step is performed to get this to a reasonable level before calculating the textures. Finally, RealityCapture textures the simplified mesh by projecting the input images into the scene.

In addition to a mesh output, RealityCapture can also export the camera registration as extrinsic camera parameters. This information can be used in downstream steps for georeferencing or semantic segmentation.

For a good reconstruction, it is important that RealityCapture can align as many images as possible. Experiments with reconstructions in RealityCapture from multiple datasets captured at different locations revealed what influences RealityCapture's ability to create high-quality results. While the alignment can be improved by fine-tuning different parameters or manually aligning detected components, this is not very economical. RealityCapture scales incredibly well. Often, the solution was to provide more pictures.

3.4.2 From Point Clouds

Performing mesh reconstruction from point clouds consists of 3 steps: Firstly, the point cloud is pre-processed. This is an optional step to ensure that normals are calculated, should they not already be present. Then, a mesh is reconstructed directly from the point cloud using screened Poisson surface reconstruction [38]. Lastly, the mesh is further post-processed to reduce artifacts that may emerge from the reconstruction process.

Such artifacts are often visible as a "bubbling" effect. This happens in regions where the point cloud coverage is low, there are occlusions, or the normals are noisy, such as the top of trees or traffic signs. It also may happen at the border of the scene. Examples of these artifacts can be seen in Fig. 3.2. To fix the artifacts, the reconstructed mesh is compared to the input point cloud. For each vertex, the distance to the closest point of the point cloud is calculated. Vertices further away than a certain threshold are removed from the mesh.

The spatial resolution of the reconstructed mesh is controlled by the depth of the Poisson surface reconstruction, which controls the level of detail by specifying the depth of the octree used by the algorithm. This depth defines how many voxels are used. The size of a voxel is given by $\frac{s}{2^d}$ where d is the octree depth, and s is the size of the scene. This fraction gives the distance between two sampled points. Table 3.1 shows the spatial resolution for different scene sizes and octree depths. To accurately depict small objects such as poles and traffic signs, a reconstruction depth of 12-13 is required, depending on the scene size. A more detailed comparison of different reconstruction depths is done in Section 5.2.2.

The reconstructed mesh has a polycount in the millions. This is way higher than necessary and would significantly slow down downstream tasks. Depending on the scene size and complexity, the mesh is simplified to a more efficient polycount of $100,000$ to $1,000,000$ faces.

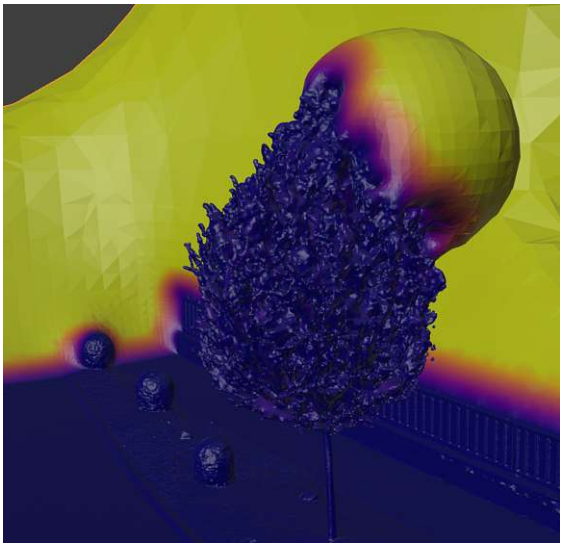


Figure 3.2: A mesh reconstructed with Poisson surface reconstruction. Showing artifacts occurring in occluded areas. Yellow areas are more than a meter away from the closest point in the point cloud.

Octree depth	**Spatial resolution**	**Voxel size (mm) for a 50m scene**	**Voxel size (mm) for a 100m scene**
8	256^3	192	384
9	512^3	96	192
10	1024^3	48	96
11	2048^3	24	48
12	4096^3	12	24
13	8192^3	6	12

Table 3.1: Reconstruction resolution at different reconstruction depths for Poisson surface reconstruction.

Color information from the point cloud is only preserved as vertex color attributes. The simplified mesh does not have a vertex density that is high enough to provide the required radiometric resolution. Texture baking is performed in Blender to get the texture information from the high-resolution mesh to the simplified mesh. UV unwrapping of the mesh is performed, and then the colors of the high-resolution mesh are baked to a texture on the low-resolution mesh. Since the high-resolution mesh is the source of the color information, the reconstruction depth does not only specify the spatial resolution but also the maximum texture resolution of the final mesh.

3.5 Semantic Segmentation

3.5.1 3D Semantic Segmentation

There are different approaches for performing 3D semantic segmentation. In order to choose an appropriate technique, the first question we have to answer is whether to segment the mesh or the point cloud. The segmentation of point clouds is a problem that is more mature in the literature than the segmentation of meshes. However, there are two arguments for doing the segmentation on the mesh. First, we desire a mesh as the output of the pipeline. Creating a segmentation of the point cloud would require the transfer of point cloud labels to the mesh, which in itself could be a source of errors. Second, meshes have many advantages over point clouds in this context [2, 17, 58, 29].

Meshes are better than point clouds for representing the geometry of scenes [2, 17]. They can represent flat surfaces and sharp edges more easily with fewer data points [58]. They can describe intricate details while disambiguating from other close surfaces [29]. Furthermore, meshes consume less memory [17, 58] since they are more efficient in storing large, flat surfaces. Another advantage of meshes is that they can contain high-frequency color information via textures. This is crucial for identifying objects with a non-distinctive shape (like traffic signs on a wall or areas separated only by road markings).

Many solutions for both domains are based on DNNs. Therefore, another important aspect to consider is the availability of datasets in the domain of urban scenes for each modality. Unfortunately, datasets in this domain are scarce, especially when considering the granularity of classes we require for the task at hand. There are two point cloud datasets available [16, 5] with semantic labels from the CityScapes dataset [12]. However, they do not quite match the previously defined requirements for point clouds. They do not have a high enough density, and RGB color information is not directly available.

To the best of our knowledge, there is no mesh-based dataset that fits the requirements. The datasets are either limited in scale, don't cover many different types of scenes, or have a low semantic richness [17]. This could be because mesh-based datasets are often created for large-scale urban applications, such as smart cities or urban planning [17], which has interests of a larger scale and does not require that high of a granularity.

3.5.2 View Based Semantic Segmentation

This leaves us with another approach: View-based 3D semantic segmentation. This technique generates 2D views of the scene, which are subsequently semantically segmented using a 2D model. The resulting 2D labels are then projected back into the 3D scene. Doing so for many different views allows for covering each part of the scene and accumulating predictions on the vertices of the mesh. This approach has been performed on point clouds [35, 41, 21, 7] and meshes [53, 40, 1].

Meshes are a better fit than point clouds for creating 2D views. Firstly, because they

describe surfaces, there are no holes in the views and occlusion works properly. Secondly, they can provide a much higher radiometric frequency.

Coincidently, the dataset from which we derive our class labels, CitiyScapes [12], is a fitting domain for this task. It is a large and diverse urban 2D dataset with pixel-level semantic labels. It covers 50 cities containing 5,000 fully annotated and 20,000 partially annotated images. The field of 2D semantic segmentation is rich with great-performing state-of-the-art models [10, 64, 57, 59]. We can use a model pre-trained on CityScapes for the 2D segmentation task. For a comparison of different 2D models used with this 3D segmentation technique, see Section 5.3.3.

3.5.3 Virtual View Rendering

The next question is, how to obtain 2D views of the 3D mesh. We can use the original images with their estimated registrations to perform the segmentation task when using the photogrammetry strategy. This has the advantage of feeding real images to the 2D semantic segmentation model and ensures that each part of the scene is covered. However, since the registrations are only estimates, any error will result in the bleeding of semantic labels to more distant surfaces when performing the back projection (see Fig. 3.3). Furthermore, this approach would not work for the point cloud strategy.

Kundu et al. [40] show that using virtual views (renderings) of the scene not only produces good results in a view-based segmentation but outperforms the usage of real images with estimated registrations. This is because they can have a smarter view selection, a higher field of view, look from behind walls, and have an exact back projection since the camera parameters are known precisely. Furthermore, they show that similar performance can be obtained with as little as 12 virtual views versus 1700 original views for an indoor scene. These arguments lead us to the usage of a virtual-view-based 3D semantic segmentation approach using 2D semantic segmentation models pre-trained on CityScapes.

3.5.4 Sampling Virtual Views

The selection of virtual views is critical for the segmentation performance [40]. Each part of the mesh needs to be covered by at least one view. However, just naively sampling the views does not produce optimal results. Since a pre-trained model is used, we need to consider the domain it was trained on. The CityScapes dataset is captured using a camera mounted on top of a vehicle driving on streets. Consequently, models trained on this dataset perform best with images taken from a similar perspective. Virtual views taken from the sidewalk often misclassify the ground below as *road* instead of *sidewalk*. We show this with the experiments done in Section 5.3.2.

The best-performing sampling method uses only samples from a street perspective. Since we cannot automatically sample the road points before the semantic segmentation, we require the user to manually select points on the street. The direction is sampled by positioning the camera horizontally and rotating it uniformly around the up-axis in 8

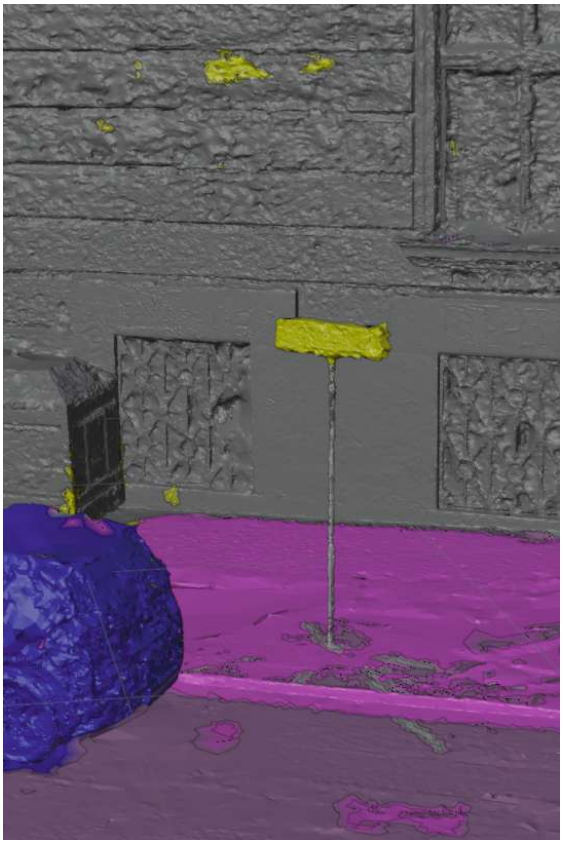


Figure 3.3: Projection bleeding. The label "traffic sign" (yellow) is labeled on the wall due to inaccuracies in projecting it into the scene.

steps. These manually selected points are used to generate samples that are similar to that of the 2D domain.

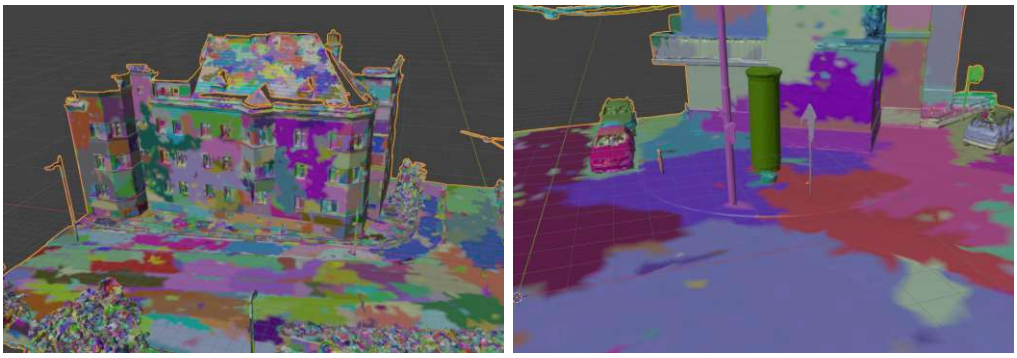
3.5.5 Over-Segmentation

The above-described segmentation process assigns a semantic label to each vertex. This only considers a projected 2D RGB view of the 3D scene. The topological properties of the mesh are lost in this projection. However, the topology of the mesh contains a lot of valuable information.

Object boundaries can be deducted from the geometric properties of a mesh. They often align with the boundaries of planar regions [19]. Furthermore, we can look at the local curvature. Concave edges often form boundaries between objects, while convex areas usually belong to the same object [36].

To leverage these properties, the Felzenszwalb algorithm [15] can be adapted to segment 3D meshes [13, 36]. Other approaches train a random forest to estimate the planarity based on a multitude of features [19] or use region-growing approaches [50].

The Felzenszwalb algorithm is used in this thesis due to its simplicity and efficiency. It segments based on the difference of adjacent vertex normals. The smaller the difference between two normals the more likely they belong to the same segment. Fig. 3.4a shows how such an over-segmentation looks.



(a) Over-segmentation

(b) Under-segmentation issue. Components span across object boundaries.

Figure 3.4: Examples of over-segmentation and under-segmentation

One challenge when doing this over-segmentation is that we need to be careful not to under-segment the mesh, meaning that we get too few segments, with segments spanning multiple semantic regions (see Fig. 3.4b). This issue arises when semantic boundaries are not obvious from just the topology (i.e., when a sidewalk is separated by a different texture but has the same elevation as the street). To alleviate this issue, the color difference of the vertices is added to the weight to help form boundaries in these regions. However, since we only look at vertex colors, this becomes a sampling problem, and texture changes that do not fall exactly on the vertices might still be missed.

Overall, this over-segmentation helps reduce noise and bleeding while forming smoother boundaries. However, the maximum segment size needs to be limited to avoid the issues mentioned above, reducing the effectiveness of this step. See Section 5.3.1 for a more detailed breakdown of this step and its parameters.

3.6 Georeferencing

Georeferencing of the mesh is required for extracting and aligning with road network metadata. The way this data is obtained depends on the input modality.

Images When using images as the input, the GPS information may be obtained by the GPS location embedded in the EXIF metadata. GPS tracking provided by consumer-grade devices such as cameras or smartphones is not accurate enough for precise georeferencing. They have an accuracy of only 7-13m in urban environments [43]. However, this data may still be used to obtain the scene's rough geographical bounding box.

The estimated camera registrations in the local space of the mesh, in conjunction with the corresponding global GPS positions, can be used to find a transformation from local space to global space. This matrix should only contain a translation, rotation, and scaling, as we do not want to deform the mesh. The translation part of the matrix is the base point of the mesh in relation to the global coordinate system. The rotation part can be applied to the mesh to align the rotation around the up axis.

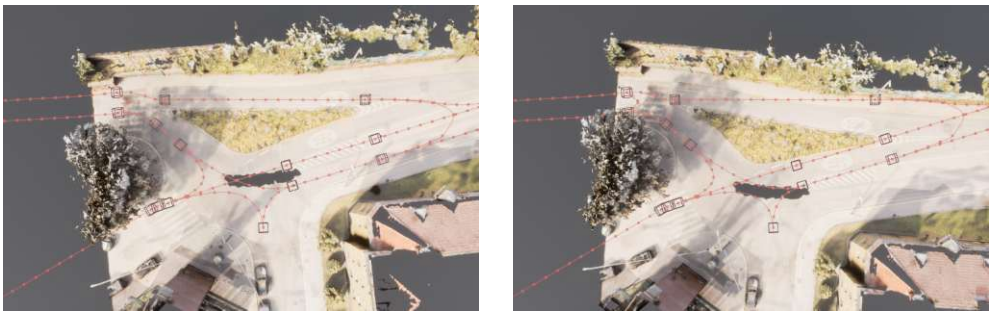
Point Cloud The point cloud strategy requires the base point to be known beforehand. The user must provide it when running the pipeline.

Manually Should the images not contain GPS metadata, the georeferencing can be overridden manually by specifying the base point when running the pipeline. In this case, there is no way of knowing the correct heading or alignment of the scene. This only gives a very rough alignment.

3.7 Roads

CARLA needs information about the road network in the scene as an `xodr` file. Creating such a file from just the 3D mesh reconstruction is a challenging task. Therefore, we utilized the publicly available source of road networks OpenStreetMap. The API can be queried with a geographical bounding box to obtain information about a region as an `osm` file. This file can be converted to `xodr` using CARLA. The data from OpenStreetMap contains information about where the roads and lanes run and how they are connected.

This data is good enough but not perfect. Fig. 3.5a shows a road network aligned with the mesh. It can be observed that the lanes follow the actual lanes roughly but an exact alignment is not possible without having to deform the graph.



(a) A good alignment after manually moving the mesh. Note how lanes still do not perfectly align with the geometry.

(b) The initial alignment before manual correction.

Figure 3.5: A reconstructed mesh with the road graph overlayed in CARLA

3.8 Manual Actions

Some of the steps described in the previous sections require or can be improved by performing manual actions. Manual actions allow the user to correct the errors before the pipeline continues processing the data. These manual actions aim to be as simple and fast as possible. The goal is still to be efficient and produce results with minimal manual work. The manual actions only take a fraction of the work that it would take to create the whole digital twin manually.

It is also important to consider when these actions need to be performed. Preferably, manual steps can be performed back to back or even at the start or end of the pipeline. We want to minimize the time a user has to wait until the next manual step needs to be performed.

Furthermore, all intermediate files are exposed and can be modified should unexpected issues arise. The following sections describe common manual actions that are expected to be performed.

3.8.1 Selecting Street Points

As described in Section 3.5.4, the 3D semantic segmentation performs best when knowledge about the road positions is available. To provide this information, the user must select a few points on the street that will be used as the sampling positions. Ideally, the points are placed spaced evenly and on all lanes.

The process is very straightforward. The user opens the file in CloudCompare[4]. Then, using the point-picking tool, the user clicks on the scene to place the sample positions as they see fit. When done, the point list must be exported as a `txt` file containing a list of x, y, and z positions, which are used as the input for the pipeline. The whole process only takes a few minutes to perform. In Fig. 3.6, a scene opened in CloudCompare with a few street points selected can be seen.

This action can be performed before running the pipeline when using a point cloud as the input. If using the photogrammetry strategy, the user must wait until the 3D reconstruction is complete, as this uses a new local coordinate system.

3.8.2 Setting the Reconstruction Region and Ground Plane

This action is only needed for the photogrammetry strategy. At the edge of the scene, there might be less data available, resulting in more artifacts in the reconstruction. Manually setting a reconstruction region also ensures that the boundary of the mesh is a straight line. Furthermore, the user might not want to reconstruct the whole scene or want to be in control of exactly what region is reconstructed. While RealityCapture can automatically choose the reconstruction region, in our tests performed, this region was often larger than required and included some artifacts. Fig. 3.7 shows a reconstruction

[4] `https://www.danielgm.net/cc/`

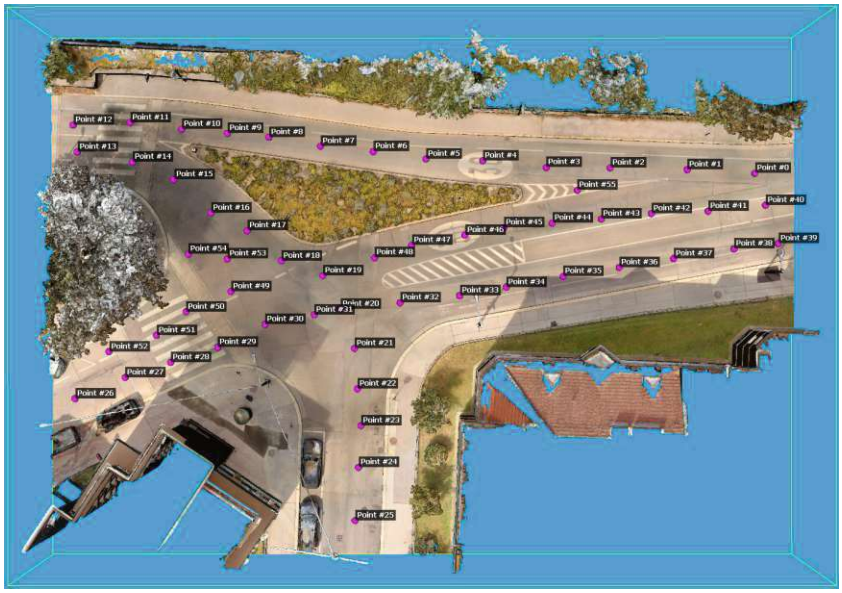


Figure 3.6: Points picked on the street in CloudCompare

region configured inside RealityCapture. RealityCapture might also fail to level the ground plane correctly. This step provides an opportunity for the user to fix this issue.

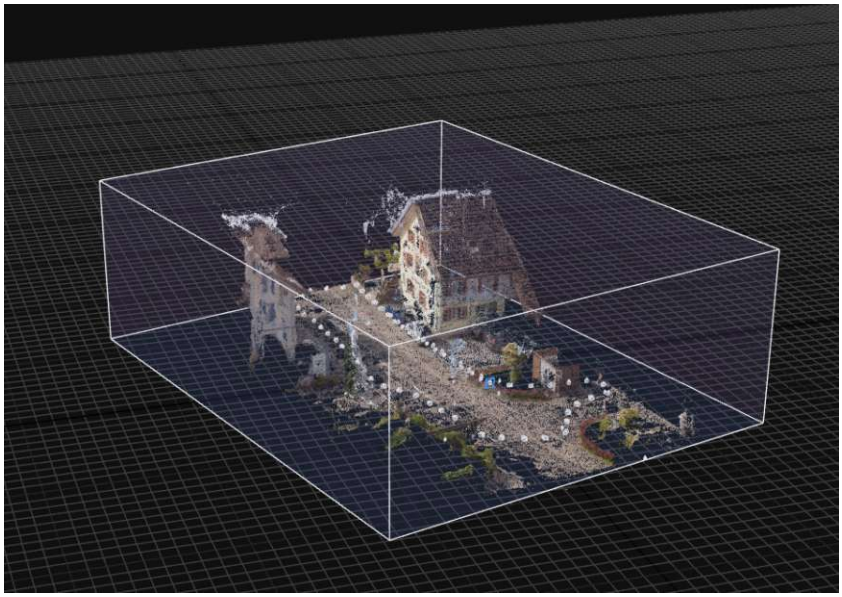


Figure 3.7: The reconstruction region shown in RealityCapture

When the pipeline reaches the step where this action must be taken, RealityCapture is opened automatically. The user must set the reconstruction region and ground plane and close RealityCapture. The pipeline will continue running afterward automatically.

This is also only a matter of minutes. However, the user has to wait until the pipeline reaches this step, which might take longer than an hour.

3.8.3 Adding Connecting Roads

The pipeline is designed to reconstruct individual street segments and intersections. Some downstream tasks that require driving simulations benefit from a larger drivable area. This allows for acceleration into the intersection and more freedom for creating different scenarios.

There are different ways to extend the drivable surface. This depends on the requirements for the scene and the user's preference. In this thesis, we use Blender for the task. First, the scene is embedded in a large flat plane, and the elevation is roughly matched with the outside of the scene. Then, the road mesh is connected to this plane to ensure cars can move smoothly between the two surfaces (see Fig. 3.8). This step is completely optional. It might be performed after the semantic segmentation or even after the full pipeline is complete and the scene is imported into CARLA.

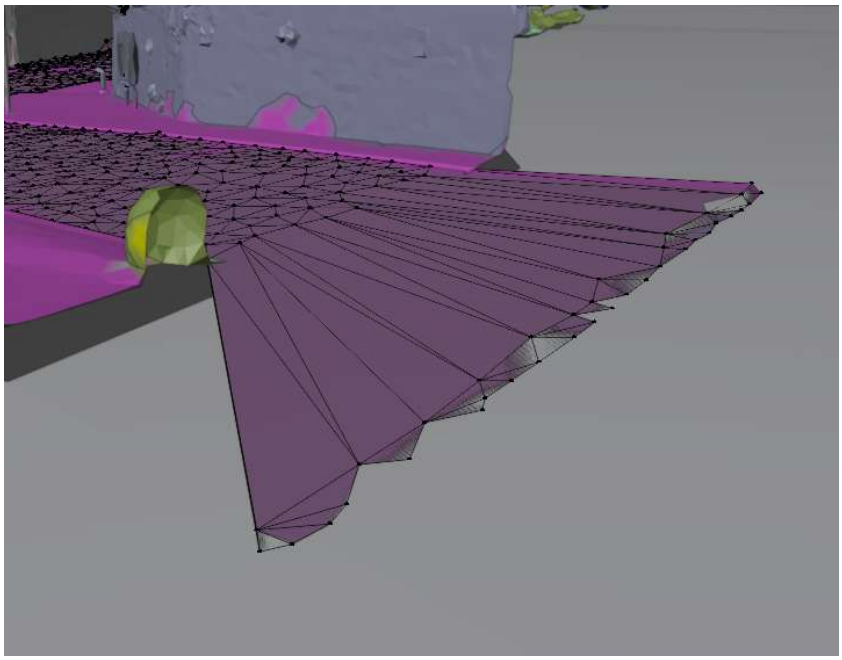


Figure 3.8: A reconstructed mesh with an extended drivable area created in Blender

3.8.4 Correcting Semantic Segmentation

The semantic segmentation will have some miss-labeled regions. When semantic segmentation is required for downstream tasks, it is important to be able to correct such errors. This correction can be performed in Blender with the aid of a custom addon.

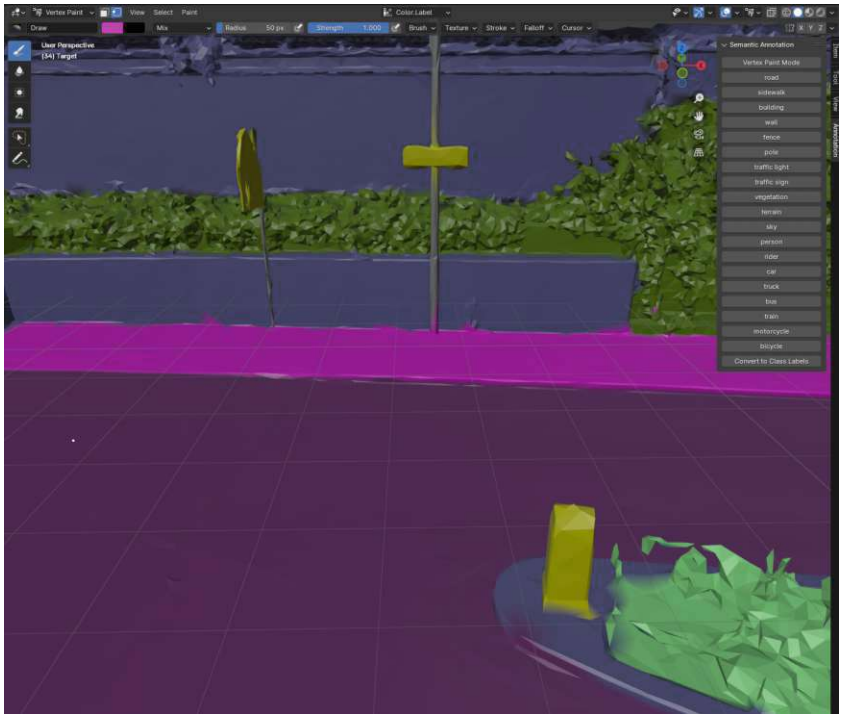


Figure 3.9: A semantic segmentation of a reconstruction. Showing the manual correction tool in Blender. Vertex labels can adjusted using the vertex painting tool.

The automatic semantic segmentation is displayed on the mesh as per-vertex color information. It uses the common color scheme of the CityScapes dataset. The semantic segmentation can then be changed with the vertex painting tool. The addon helps the user by providing a color palette to easily change the brush's color to the desired class. After correcting the segmentation, the RGB colors are converted back to class labels and exported for further processing by the pipeline. In Fig. 3.9 a screenshot of this step can be seen.

3.8.5 Road Alignment

The automatic alignment of the road network data with the mesh is not perfect. The reasons for this are that the raw geographic input data is not accurate enough, and that the data from OpenStreetMap is just an approximation. This alignment can be fine-tuned in Unreal Engine after the scene has been imported into CARLA. The meshes can simply be moved to better align with the road network. Individual nodes of the road graph can be moved to improve the alignment further. Fig. 3.5 shows how the scene looks before and after manually fixing the alignment.

CHAPTER 4

Implementation

4.1 Dependencies

The pipeline consists of various different tasks in various fields. It relies on numerous Python packages and external tools. The following section summarizes the most essential tools and libraries the pipeline utilizes.

Python The pipeline itself is written in Python[1]. The Python version has been chosen to be compatible with all dependencies. The biggest limiting factor is CARLA, which is only compatible with Python up to version 3.8. All other Python dependencies are compatible with this version, so 3.8 is the version used.

Besides a set of Python packages that can be installed with pip, the installation of two packages requires special attention. PyTorch and mmSegmentation[2] require matching versions and a compatible CUDA installation.

CARLA The largest dependency is CARLA[3]. To create custom CARLA maps from `fbx` and `xodr` files, CARLA and Unreal Engine 4 need to be compiled from source. This compilation process is resource-intensive, it requires over 165 GB of disk space and takes several hours to run.

Blender Blender 4.1[4] is used for mesh processing, visualization, and executing manual actions via a custom addon. Its scripting capabilities allow for the automation of different tasks in the pipeline, such as UV unwrapping, texture baking, mesh splitting, and file conversion.

[1] `https://www.python.org/`
[2] `https://github.com/open-mmlab/mmsegmentation`
[3] `https://carla.org/`
[4] `https://www.blender.org/download/releases/4-1/`

CloudCompare CloudCompare[5] is used for the manual step of picking street points. Additionally, CloudCompare is a valuable tool for visualizing and debugging the input point cloud before processing.

MeshLab MeshLab[6] is another tool utilized for mesh processing. It implements many algorithms and helper tools for working with meshes and point clouds. We use it to run Poisson surface reconstruction [38], clean up, and simplify the meshes.

RealityCapture RealityCapture[7] is a commercial tool used for the photogrammetry step. It generates high-quality 3D models from photographic data and scales well with many images. Despite being a commercial tool, its integration into the pipeline is justified by its superior performance and output quality compared to open-source alternatives.

4.2 Pipeline

The pipeline is designed to handle complex workflows by coordinating various tasks and tools efficiently. The overall implementation can be divided into two main components: the individual steps of the pipeline and the orchestration mechanism that ties these steps together.

4.2.1 Steps

Each step of the pipeline is implemented as a distinct Python function. These functions are the building blocks of the pipeline, processing inputs and generating outputs based on the given requirements.

The individual steps receive inputs given as file paths or configuration objects. File paths are used to specify the location of data files that need to be processed or the destination where output files should be saved. Configuration objects provide additional parameters required for the execution of each step. These objects are instantiated with the parameters provided by the configuration file. This provides a way to adjust the steps' behavior without having to alter the codebase directly.

These steps are implemented either directly in Python or by utilizing external tools like RealityCapture, Blender, or CARLA, which are launched as subprocesses. For Blender, a Python file is passed and executed in the Blender environment. RealityCapture and CARLA's behavior is controlled via CLI arguments.

4.2.2 Orchestration

The requirements for the pipeline established in Section 3.1.4 are automation, flexibility, traceability, and change detection. To achieve traceability and change detection, we

[5] https://www.danielgm.net/cc/
[6] https://www.meshlab.net/
[7] https://www.capturingreality.com/

require the data passed between each step to be stored on the filesystem. Specifying the locations of these files and connecting the outputs to the inputs between steps is the responsibility of the pipeline. For flexibility, the definition of the pipeline must be simple and easily modifiable. It should be clear what steps run in what order.

A general-purpose pipeline system has been implemented to address these requirements. The entry point for this system is the Pipeline class. This class orchestrates the execution of various steps, ensuring that the outputs of one step can be utilized as inputs for subsequent steps. The core components of the pipeline include the `Artifact`, `PipelineData`, `PipelineStep`, and `StepGroup` classes.

The `Artifact` class represents a piece of data produced or consumed by a pipeline step. Each `Artifact` is associated with a file path. This class exposes the filename of the corresponding file, forming the basis for output-to-input connection.

`PipelineData` manages the collection of `Artifacts` and the initial arguments passed to the pipeline. It provides mechanisms for registering new artifacts and finding the latest artifact based on its file name. This ensures that data from the latest possible step is used.

The `PipelineStep` class encapsulates a single step within the pipeline, including its identifier, name, a callable function, input artifacts, and output artifacts. It also receives the arguments for running the callback as a dictionary, which contains the paths to input and output artifacts and further arguments mapped to their corresponding function parameters. It includes methods for running the step, checking if the inputs are newer than the outputs, and verifying the existence of inputs and outputs. This allows steps to only be executed when necessary.

Steps are grouped into `StepGroup` instances, which help organize related steps logically and hierarchically. Each group has a unique identifier, name, and a directory for storing its sub-steps. Steps can be added to a group using the `add_step` method, which automatically parses the callable function's parameters to determine the necessary inputs and outputs. This parsing ensures that the correct artifacts can be mapped to this step. Artifacts are linked to their parameters by comparing the artifact name to the parameter name.

The main `Pipeline` class orchestrates the entire process. It initializes with a root directory and a set of arguments. This class is responsible for adding and finding input files, adding steps, and executing the pipeline. The run method determines which steps need to be executed based on the existence and age of their inputs and outputs. Furthermore, it provides a mechanism to forcefully run specific steps if required. This method also ensures that each step is executed in the correct order, respecting dependencies and the defined sequence.

The mapping of input and output artifacts to function arguments happens by a naming convention. For each step, the callable function is analyzed. Parameters that start with the `out_` prefix are considered outputs. For each such argument, an artifact is created

with the file path in the step's directory. The file name is derived from the argument name. Expecting an argument name of the form `out_some_file_name_suffix` will set the file path to `<root>/<group>/<step>/some_file_name.suffix`. Similarly, files with the `in_` prefix are considered inputs. The filename is derived in the same way. The pipeline looks for an artifact produced in a previous step with the same name and links it to this argument. Any remaining arguments are filled from the pipeline arguments. These are used to provide configuration parameters for the individual steps. They are mapped by their type, so it is recommended to use classes holding configuration data. Multiple steps can produce artifacts with the same name. In this case, the pipeline ensures that the artifact from the latest step possible is used.

The pipeline takes care of the directory structure. Inputs for the pipeline are expected to be in a `_in` directory inside the project's root directory. Alternatively, an alternate path for this input directory can be specified in the configuration. For each `StepGroup`, the pipeline creates a directory, in which each `PipelineStep` gets its directory. These directories and subdirectories are numbered in their execution order. The directory structures created by the pipeline can be seen in Fig. 4.1.

```
Point Cloud Project
├──_in
├──1 reconstruction
│   ├──1.1 reconstruct
│   ├──1.2 cleanup
│   ├──1.3 simplify
│   └──1.4 texturize
├──2 roads
│   ├──2.1 extract geo region
│   ├──2.2 fetch osm
│   └──2.3 convert to xodr
├──3 MANUAL select samples
├──4 segmentation
│   ├──4.1 prior
│   ├──4.2 semantic segmentation
│   ├──4.3 MANUAL adjust segmentation
│   ├──4.4 update segmentation
│   └──4.5 apply_segmentation
└──5 finalize
    ├──5.1 align
    ├──5.2 convert to fbx
    └──5.3 create carla package
```

(a) Point Cloud

```
Photogrammetry Project
├──_in
├──1 reconstruction
│   ├──1.1 rc align
│   ├──1.2 MANUAL set reconstruction region
│   ├──1.3 rc reconstruct
│   └──1.4 rc export
├──2 roads
│   ├──2.1 extract geo region
│   ├──2.2 fetch osm
│   └──2.3 convert to xodr
├──3 MANUAL select samples
├──4 segmentation
│   ├──4.1 prior
│   ├──4.2 semantic segmentation
│   ├──4.3 MANUAL adjust segmentation
│   ├──4.4 update segmentation
│   └──4.5 apply_segmentation
└──5 finalize
    ├──5.1 align
    ├──5.2 convert to fbx
    └──5.3 create carla package
```

(b) Photogrammetry

Figure 4.1: The folder structures created after running the pipeline

The Python code in Fig. 4.2 shows how the pipeline can be used. A new pipeline with a single group "`reconstruction`" is created. This group contains three steps: `pointcloud_reconstruct`, `simplify_mesh`, and `texturize_mesh` that will be

```
cfg_pointcloud = PointCloudConfig(...)
cfg_reconstruction = ReconstructionConfig(...)

args = [cfg_pointcloud, cfg_reconstruction]
pipeline = Pipeline(Path("path/to/project/dir"), arguments=args)
pipeline.add_input("pointcloud.e57")

reconstruction = pipeline.add_group("reconstruction")
reconstruction.add_step("reconstruct", pointcloud_reconstruct)
reconstruction.add_step("simplify", simplify_mesh)
reconstruction.add_step("texturize", texturize_mesh)
```

Figure 4.2: Example pipeline usage. Demonstrating the creation of a new pipeline with a point cloud input, two config arguments, and a single group with 3 steps.

```
def pointcloud_reconstruct(
        in_pointcloud_e57: Path,
        out_scene_high_res_ply: Path,
        cfg: PointCloudConfig):
    ...

def simplify_mesh(
        in_scene_high_res_ply: Path,
        out_scene_ply: Path,
        cfg: ReconstructionConfig):
    ...

def texturize_mesh(
        in_scene_ply: Path,
        in_scene_high_res_ply: Path,
        out_scene_obj: Path,
        cfg: ReconstructionConfig):
    ...
```

Figure 4.3: Example step definitions. When executing the pipeline, the function arguments are filled in automatically. Input and output paths are derived from the parameter names.

executed in order. An input to the pipeline is added, and the pipeline expects the file `<root>/_in/pointcloud.e57` to exist. An artifact is created for this file, which is used as the input for the first step.

The function definitions for the three steps are shown in Fig. 4.3. The first function has one input argument, which is mapped to the existing pipeline input `pointcloud.e57`. It produces a `scene_high_res.ply` file as an output artifact. The next step takes

this artifact as input, the file path of the artifact created by the previous step will be passed as an argument by the pipeline. This step, in turn, produces a `scene.ply` file. The last step takes both artifacts and produces a `scene.obj` file. Note that each of the steps takes another argument. This argument provides configuration parameters used by the step. These arguments are passed when constructing the pipeline.

4.3 Reconstruction

This section covers the implementation of the mesh reconstruction from both input modalities: point clouds and images.

4.3.1 From Point Clouds

The reconstruction process of point clouds is done using Screened Poisson surface reconstruction [38]. The input for this stage is an `e57` file, which is processed using MeshLab with the Python library. Per-vertex normals are required. If they are not present in the input data, they are optionally computed at this stage.

This reconstruction might introduce errors in the form of additional geometry away from the input points. To clean this up, each vertex's distance to the closest point in the input point cloud is calculated, and vertices that exceed a certain distance threshold are removed. Non-manifold edges, which can cause issues in subsequent processing steps, are repaired using MeshLab by removing faces until all edges are manifold. Additionally, MeshLab is used to close small holes in the mesh.

After generating the high-resolution mesh, the next step is simplification, again using MeshLab with the quadratic edge collapse function [20]. This simplification process involves using a very low planar weight to favor the reduction of polygons in flat regions. This approach helps maintain important geometric details while reducing the overall complexity of the mesh. The simplified mesh is then saved for further processing. Optionally, for meshes in the size of millions of vertices, cluster decimation [8] can be used. While potentially introducing some artifacts, this algorithm runs significantly faster. In our tests on a 60 million vertex mesh, this algorithm took only 20 minutes, while the quadratic edge collapse algorithm had to be stopped after 12 hours.

The next step is texturizing the mesh, which uses Blender as a subprocess. Blender receives a Python script that implements the texturizing process, with file paths passed via the command line. The goal is to transfer the vertex color information of the denser high-resolution mesh to a texture on the low-resolution mesh. This is achieved using *Render Baking*.

Render baking is a process where the lighting information is pre-computed and stored in texture maps. We can tell Blender to perform this baking from one object to another.

[8] `https://pymeshlab.readthedocs.io/en/latest/filter_list.html#meshing_decimation_clustering`

Rays are cast inwards from the low-resolution object onto the high-resolution object. A cage is used so that the rays originate outside the object to ensure they do not miss the target object. In this process, the Cycles renderer is set to bake only the diffuse lighting pass, which captures only the color information without additional effects.

To create the texture, the high-resolution and low-resolution meshes are loaded into Blender. A material and shader are created to render the high-resolution mesh from vertex colors. Then the low-resolution mesh is UV unwrapped and a new material and texture are created. Render baking is then performed from the high-resolution to the low-resolution mesh.

Finally, the texturized mesh is exported as an `obj` file. Additionally, a `blend` file is saved, which retains all the Blender-specific settings and materials, making it easier to debug this process.

4.3.2 From Images

The reconstruction from images is done with photogrammetry in RealityCapture. This process involves multiple steps, some requiring custom parameters to be set. RealityCapture provides a command-line interface that allows multiple commands and arguments to be chained together. The arguments for these commands are stored in `xml` files, which are maintained and distributed with the Python project. To allow dynamic arguments, these `xml` files are copied to a temporary location, and relevant parameters such as texture resolution and face count are changed.

The photogrammetry process begins by aligning the images, a step where the software matches corresponding points in different photos to determine the positions and orientations of the cameras. During this phase, RealityCapture creates components, which are groups of images that have been successfully aligned. These components are then tried to be merged, which can potentially form a larger component containing more aligned cameras. GPS information is not used during alignment due to the insufficient accuracy of consumer-grade devices.

Once the initial alignment is complete, manual intervention is required. The user must select the reconstruction region, which ensures that the reconstruction efforts focus on the scene's most relevant parts. Additionally, the user may need to adjust the ground plane to ensure the model's accurate orientation.

Following the manual adjustments, RealityCapture computes the 3D model using the default configuration. This model is then simplified to reduce its complexity while preserving details, making it more manageable for further processing and visualization. Texturizing the model is the next step, which involves UV unwrapping the mesh and then mapping high-resolution image data onto its surface.

Finally, the model and the camera registrations are exported. The model export includes all the geometric and texture information, while the camera registrations provide the positions and orientations of the cameras used in the photogrammetry process.

4.4 Semantic Segmentation

The 3D semantic segmentation step uses a virtual view approach as described in section 3.5. In this approach, 2D views of the scene are rendered, semantically segmented, and the resulting per-pixel labels are projected back onto the mesh. To leverage the topology of the mesh, a prior over-segmentation is performed.

4.4.1 Over-Segmentation

The first step in the segmentation process is performing an over-segmentation using the Felzenszwalb algorithm adapted for mesh segmentation. The Felzenszwalb segmentation algorithm is a graph-based method. It is designed to segment images by creating a graph where each node is a pixel and edges are obtained from pixel adjacency. The edges are weighted based on the color difference of the pixels. The Felzenszwalb algorithm creates a partition of the graph such that nodes within the same segment are more similar to each other than to nodes in other segments. The algorithm achieves this by iteratively merging nodes according to edge weights. The Felzenszwalb algorithm can naturally be used with the graph-like structure of meshes. The mesh with it's vertices and edges is directly used as the input graph. Edge weights are based on local properties of the adjoining vertices.

This implementation of the Felzenszwalb segmentation algorithm is adapted from the C++ implementation used in the ScanNet [13] project. It has been rewritten and optimized using Python and NumPy to vectorize operations where possible. The algorithm produces a mapping of each vertex in the mesh to a corresponding component ID.

The `Universe` class is a data structure used to manage the segmentation process. It initializes each vertex as its own segment and provides methods to find and merge segments efficiently. This helps track which vertices belong to which segments as they are merged.

Initially, a weight is calculated for each edge in the input graph. A larger weight means a larger dissimilarity between the connecting vertices. The edge weights are influenced by two properties:

1. The first one considers the difference in vertex normals. The calculation is based on their dot product. Let $d = n_1 \cdot n_2$ be the dot product between the two normals. The dot product ranges from -1 for parallel vectors facing away to 0 for perpendicular vectors to 1 for parallel vectors facing the same direction. The edge weight is set to $w_{\text{geo}} = 1 - d$, resulting in normals facing the same direction having the lowest weight. We want concave regions to have a higher weight than convex regions. We square the weight if the normals face away from each other, which decreases the weight for regions that are only slightly convex.

2. The next factor is calculated from the two vertices' color difference. This is simply the Euclidean distance of the RGB color values. The two weights are then added together. The influence of each factor can be controlled in the config file.

Then, the edges are sorted by weight and processed in order. The algorithm looks at the two segments connected by each edge, merging segments if the edge weight is below a certain threshold. This threshold is dynamically adjusted to ensure meaningful segments. When merging the components, a maximum size can be configured to reduce potential undersegmentation. After the initial segmentation, it merges small segments to meet size requirements and renumbers segments to ensure they are consecutive.

4.4.2 View Sampling

A good selection of virtual views is critical for a good segmentation performance. A virtual view of the scene is given by the virtual camera's position, orientation, aspect ratio, and field of view. These parameters are stored together in a 4x4 transformation matrix. Each part of the scene must be covered. Furthermore, having perspectives similar to those used in training the 2D semantic segmentation model further improves the performance.

The following approaches for sampling views have been implemented:

Uniform Uniformly sampling the scene by selecting points of the scene on a grid with even spacing. Per location, multiple orientations are obtained by rotating the camera at each point. The problem with this sampling technique is that these grid points often fall within geometry or are too close to objects. This makes many of the sampled views unusable.

Random Randomly sampling the scene is done by randomly selecting a vertex and then moving a certain distance away from it along its normal. This decreases the chance of the camera being too close to the mesh. However, in certain conditions, this problem can still occur.

Manual (from the street) For the street sampling approach, the user has to specify a set of points on the road manually. Then, the camera is positioned at different heights above these points. The camera is rotated by 45° steps for each point to ensure each view is covered.

4.4.3 Rendering

The next step is rendering the scene from each view. This is done using OpenGL, specifically the Python wrapper ModernGL. The scene mesh Wavefront `obj` is imported from the previous stages. It may contain multiple textures stored in the same directory. To render the scene, a shader receives the geometry, view matrix, projection matrix, and

texture. The rendering is very simple, without any lighting effects. The value of the sampled texture is directly written as the fragment color output. Before rendering, the color buffer is cleared with a light blue to simulate a simple sky background. This proved to produce better segmentation performance than using a transparent background (see Section 5.3.2).

After rendering, the color and depth buffer are read back into Numpy arrays. The depth buffer is then analyzed. Pixels where the depth buffer equals 1 are regions where no geometry was rendered. The ratio between the number of these pixels and the total number of pixels in the image is calculated. Renderings where more than a certain percentage of pixels are empty will be thrown away, as they do not provide sufficient information to perform meaningful 2D semantic segmentation.

4.4.4 2D Segmentation

Performing 2D semantic segmentation is done using models pre-trained on CityScapes. mmSegmentation is a segmentation toolbox written in Python. It provides access to many state-of-the-art models pre-trained on various different datasets. The model used can be changed in the configuration file. By default, mask2former [10] is used. The renderings obtained from OpenGL are fed into the model, which returns a 2D array containing the predicted class for each pixel.

4.4.5 Back Projection

The final step involves projecting the per-pixel class labels back onto the 3D mesh, respecting the prior over-segmentation. We accumulate predictions on the segments by looking up the closest vertex for each predicted pixel and fetching its segment. Then, after all predictions are accumulated, we choose the best class for each segment and assign the label to all its vertices.

A naive approach using raycasting for each pixel is very inefficient. Since we know the exact camera parameters, we can leverage OpenGL to create a mapping from pixels to vertex IDs. For this, we need to render the scene with each fragment being set to the vertex ID of the closest vertex. Since the vertex IDs are not available in the fragment shader, we need to use a geometry shader to pass them through. This geometry shader re-emits each triangle. For each emitted triangle, the shader assigns vertex IDs and barycentric coordinates to each triangle vertex. This approach ensures that each triangle has its own vertices, allowing us to store the three vertex IDs and barycentric coordinates for each vertex. This setup is necessary because we need access to all three vertices within the fragment shader to find the id of the closest vertex. In the fragment shader, barycentric coordinates are automatically interpolated. By examining these coordinates, we can determine the closest vertex to each fragment. The shader reads the corresponding vertex ID and writes it to the shader output. This method efficiently determines the nearest vertex for each fragment.

The buffer output from the fragment shader is read as a Numpy array, mapping each pixel to the closest vertex of the mesh. We index the over-segmentation array using this mapping to create a pixel-to-segment mapping.

The predicted 2D classes can be projected onto the mesh using this mapping. An array of size `(segments, classes)` stores the segmentation results. For each pixel in each 2D semantic segmentation, the corresponding segment's class counter is incremented by one. To improve efficiency, instead of iterating over all pixels, the arrays are converted to tensors. Let's call the flattened pixel-to-segment mapping `seg` and the flattened predicted segmentation `pred`. This gives us two tensors of equal length, where the i-th entry corresponds to a prediction of the class `pred[i]` for the segment `seg[i]`. The `index_put_` function from PyTorch allows us to index the accumulation tensor by `(seg, pred)` and add one to each entry of `accumulation[seg[i], pred[i]]`. The method allows for duplicates and runs in parallel, which significantly speeds up the process.

Once all views are processed, we are left with the accumulation array containing all sampled classifications for each component. To get per vertex class labels the accumulation array is indexed by the component mapping. Resulting in an array of size `(vertices, classes)`, where each entry represents the count of samples recorded for each class-vertex combination.

Weights can be applied to the class labels as an optional improvement. For example, if a particular class is frequently misclassified, applying a larger weight to said class can make positive identifications more impactful.

To determine the final class label for each vertex, we select the class with the highest (weighted) sample count. This is accomplished by taking the array's `argmax`, which identifies the class with the maximum value.

Since this approach increments the sample count for each pixel, larger triangles on the image contribute more samples to each of their vertices than smaller ones. This in turn makes regions that are closer to the camera more impactful.

4.5 Road Networks

The source for road network information is the OpenStreetMap[9] (OSM) API. This API can be queried with a geographical bounding box to return map data as an `osm` file. As described in Section 3.6, georeferencing data comes either from the data source or user input.

An important aspect to consider when working with geographical positions is the usage of different spatial reference systems. These systems define how locations on the Earth's surface are measured. These coordinate systems are often given as an EPSG code, which refers to an entry in the Geodetic Parameter Dataset created by the European Petroleum

[9] `https://www.openstreetmap.org`

Survey Group[10]. OpenStreetMap uses the WGS-84 coordinate system (EPSG:4326), which is typically used for GPS applications and commonly found in EXIF metadata. For example, the Cumberlandstraße dataset located in Vienna (see Section 5.1.3) has its origin at 48.19212, 16.2955 in EPSG:4326.

Conversion is necessary to support datasets recorded in different coordinate systems. This conversion can be performed using PyProj[11], which creates a transformer between two coordinate systems. It ensures the georeferencing is given in the correct system required for the OpenStreetMap API.

Pyproj is also used to align the mesh and the OpenDrive file, which have different local coordinate systems. For each, a global offset that places them within a specific region of a larger map is given. This offset allows the local coordinates (x, y) to be translated into global coordinates (latitude, longitude). The alignment is performed by translating the mesh. This is done by first transforming the mesh coordinate system into EPSG:4326. Then, the EPSG:4326 transformation is applied to convert this offset from lat/long to x/y in a local coordinate system. This is exactly the system in which the offsets in the OpenDrive file are given. Adding these offsets gives a 2D translation between the two local coordinate systems. Applying this offset to the mesh completes the alignment.

4.6 Mesh Splitting

After obtaining the semantic segmentation for the mesh, the mesh needs to be split into individual sub-meshes, one for each class. CARLA requires each mesh to be separated to use the semantic information. The pipeline uses Blender to perform this splitting.

The semantic segmentation is provided as a per-vertex class label in a NumPy array stored as a `npy` file. A Blender script receives the path to the mesh and the segmentation. The script then stores the class labels as vertex attributes and assigns vertex colors based on the standard Cityscapes color palette. The class labels must be stored as vertex attributes to preserve them when modifying the mesh, as the vertex ids change after each splitting operation. Additionally, another layer of vertex colors is added to store debugging metrics.

The script iterates over each class label, selecting all vertices belonging to that class. It then uses the Blender "separate" operation to split the selection into a new mesh. The resulting meshes are named according to their class. CARLA can automatically detect some classes based on their mesh names, so they are assigned names that CARLA recognizes. The names that overlap with our defined classes are `Road_Road` for *road* and `Road_Sidewalk` for *sidewalk*. The remaining classes must be assigned manually inside CARLA if required. This can be done by moving the meshes into the corresponding sub-directory for each class.

[10] `https://epsg.io/`
[11] `https://pyproj4.github.io/pyproj/stable/`

CHAPTER 5

Evaluation

This thesis aims to build a pipeline that can create a semantically labeled digital twin of streets and intersections. This digital twin should be photorealistic, have an accurate mesh, and have high-resolution textures. The semantic segmentation should accurately separate the mesh into relevant regions, such as roads, sidewalks, and buildings. The pipeline should automate as many steps as possible, requiring no or minimal user intervention. The pipeline's running time is not a critical factor. If the tool has to run for many hours, it is acceptable if this can be done offline without relying on user input too often.

To evaluate how well the created pipeline satisfies these requirements, various experiments on the individual steps and the pipeline itself were performed. These experiments were done through comparative analyses and controlled tests with quantitative metrics.

The experiments were designed to answer the following research questions:

- Does the pipeline produce high-quality meshes and textures from *image inputs*?
- Does the pipeline produce high-quality meshes and textures from *point cloud inputs*?
- How does the reconstruction quality differ between *both input modalities*?
- What are *good parameters* for the mesh reconstruction from a point cloud?
- What is an appropriate *mesh resolution* for the usage in ADSs?
- What is an appropriate *texture resolution* for the usage in ADSs?
- What influence do different *image-capturing strategies* have on the reconstructed mesh?
- Can *virtual view semantic segmentation* produce an adequate semantic segmentation of the reconstructed mesh?

- How do the *parameters* of the segmentation process (over-segmentation, sampling strategy, 2D segmentation model) influence the segmentation quality?
- How *long* does the pipeline and each step run?
- To what degree is the pipeline *automated*?

The following steps were performed to answer these research questions: Point cloud and image datasets were captured using different hardware and strategies. The pipeline was used to create reconstructions of these datasets. Some of the reconstructions were further cleaned up and annotated manually to provide a ground truth dataset for benchmarking.

Then, we demonstrate the visual quality of the reconstructions by analyzing the meshes and textures. By having point cloud and image data from the same scenes captured simultaneously, a direct comparison between reconstructions of both modalities could be performed. The effect of different parameters for the mesh reconstruction was observed by highlighting differences in the reconstructed scenes.

The semantic segmentation performance was tested with the ground truth datasets, and the impact of different parameters and approaches was analyzed. Finally, we measured the running time of each step and analyzed the degree of automation of the pipeline.

5.1 Data

This section discusses the data used to build and evaluate the pipeline. The quality of the input data is crucial for the quality of the reconstruction of the digital twin. The pipeline needs either a high-density and precise point cloud with RGB and normal information or a set of high-quality images covering the whole scene from multiple perspectives. To the best of our knowledge, no image-based datasets within our desired domain and quality exist. There are publicly available point cloud datasets, however, they either lack the required point density [5, 16] or normal information [52]. While normals can be calculated, it does not produce accurate results, especially for thin geoemtry like traffic signs.

This thesis covers the entire process of creating a digital twin, starting from the data acquisition. We capture our own data, allowing us to investigate the feasibility of the different techniques and create guidelines for doing so with good results.

5.1.1 Handheld Camera

The first datasets created for the pipeline were image-based datasets of different locations in Vienna. Using a handheld camera allowed us to be very flexible in terms of what locations we could capture. It gave us direct control over how the images were captured and what perspectives were covered. We show, that when strategically capturing images with just a handheld camera, good reconstructions can be achieved.

Capturing

The images were captured with a DSLR, a Canon EOS 100D, which has an 18 MP crop sensor. An 18mm lens provided a broad field of view, ensuring an appropriate overlap between images. Capturing the first datasets quickly revealed problem areas that needed special attention. The photogrammetry pipeline had difficulty reconstructing low texture and reflective regions. In particular, roads and cars were not reconstructed well. Other areas were reconstructed much better, showing promise for the technique and leading us to pursue it further.

Creating usable image data for the pipeline was an iterative process. The quality of reconstructions was continuously improved by capturing new datasets with more images, better coverage, and more thought put into capturing the images. This revealed that many issues can be improved with more images covering problem areas from different viewpoints.

The process of capturing was the following: First, wide shots were taken to capture the context of the whole scene. This was done by walking a route around the scene multiple times. We walked on the sidewalk and the road, taking images roughly every meter. Four passes were performed, with the direction in which the images were taken being rotated by 90 degrees each time. This approach does not produce a full 360° coverage with our camera but is sufficient for aligning these images with more close-up shots and covering most parts of the scenes from multiple viewpoints.

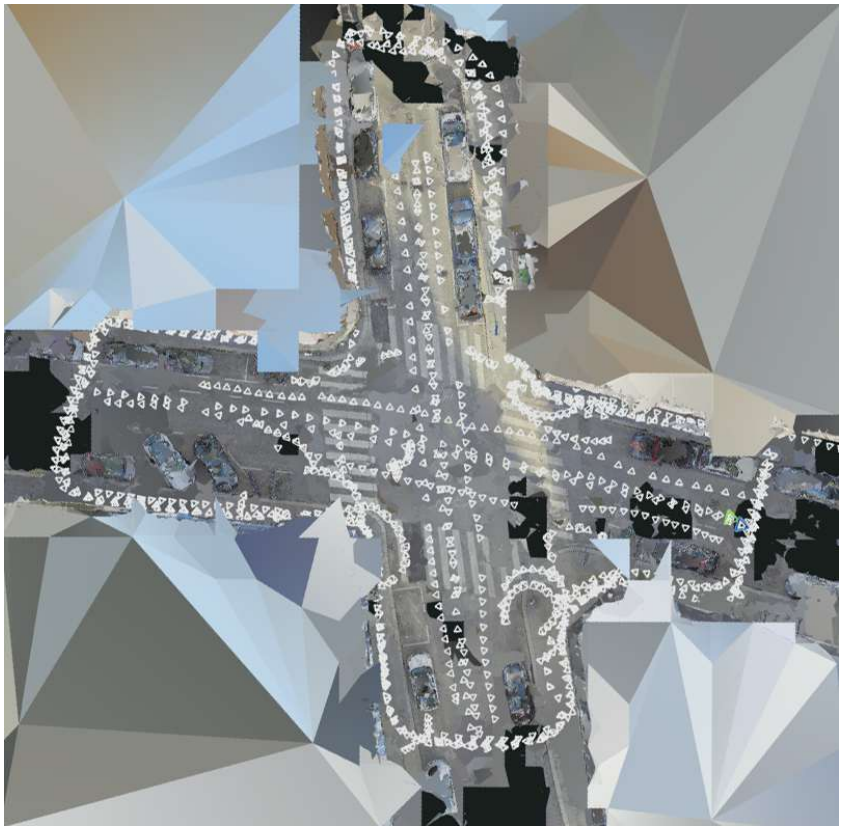
Next, more close-up shots were taken, especially focusing on problem areas. This included shots facing the ground, with the street covering more than half of the image. helping the photogrammetry process find more features in these low-textured regions which results in a better reconstruction of the floor.

Finally, detail shots were taken of smaller objects with more complex geometry. This mainly included traffic signs, poles, and cars. To capture them in full detail, images were taken by orbiting around each of them multiple times at different heights and distances. This ensures that there is plenty of data available for a detailed reconstruction. Example images and the distribution of views can be seen in Fig. 5.1.

These datasets consist of 500 to 1200 images per scene covering 1000 to 2000 m^2. Images were captured in a `raw` format and processed to even out the lighting. This was done by lifting the shadows and reducing the highlights, which reduces the hardness of shadows and creates more uniform lighting. Fig. 5.2 shows a comparison between a raw and edited image.

Capturing these datasets took about 30 minutes each. Most of the time spent was due to having to walk the loop four times while trying to avoid capturing pedestrians and cars, as moving objects can negatively impact the alignment process.

(a) Input images. These shots are specifically done to capture more features of the road surface



(b) Camera registrations in RealityCapture

Figure 5.1: Example input images and their registraions in the scene. The strategies used to capture the images can be seen here: walking a path around the scene on the sidewalk, walking on the road, and orbiting around traffic signs.

(a) Raw (b) Edited

Figure 5.2: A comparison of a captured image before and after processing.

5.1.2 NavVis VLX

The next device used for data acquisition was the NavVis VLX (as seen in Fig. 5.3). This mobile mapping device features two LiDAR scanners and four 18MP cameras. The device uses simultaneous localization and mapping (SLAM) and loop-closing algorithms to create a complete point cloud of the scene. The registration of the scans is done only with the IMU and LiDAR data. The images are stitched together to form a 360° panorama, which is then used to color the points. Offline processing further improves the results, filtering out moving objects such as cars and pedestrians, uniformly sampling the point cloud, and calculating normals. This process produces a very dense and high-precision point cloud.

The usage of the VLX is simple. The device is worn on the shoulders and can be controlled with a small display in the front. While recording, a live preview of the captured point cloud centered around the device is displayed. LiDAR data is captured automatically while the user walks around the scene. Images need to be captured manually. This is done by pressing a trigger button on the device.

This device allowed us to capture LiDAR and image data of the same scene at the same time, allowing for a direct comparison of the two strategies of the pipeline. Capturing images with a 360° view makes capturing image-based datasets much easier and faster, eliminating the need to walk over each location multiple times. The images captured

with the device replace the wide, establishing shots described in the previous section. This data was augmented by images captured with a handheld DSLR, focusing on traffic signs, the road surface, and cars.

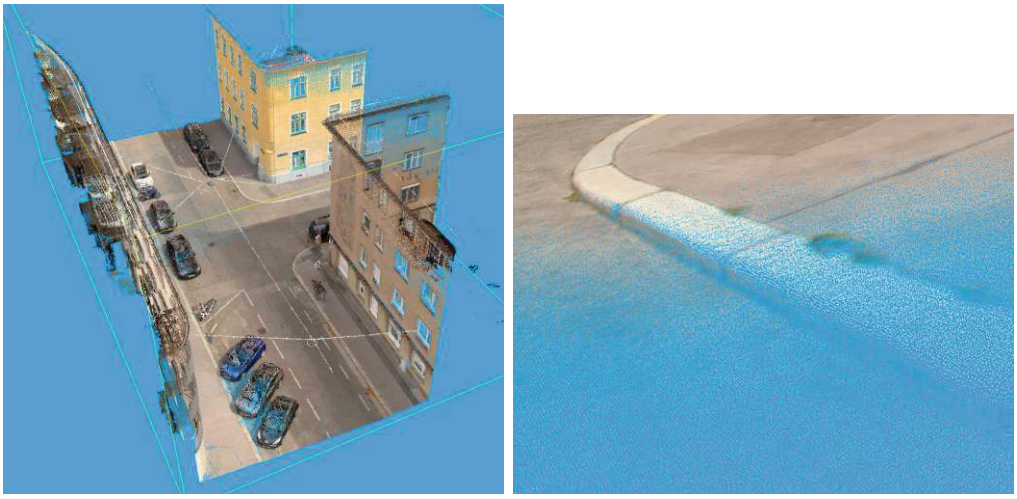
Figure 5.3: The NavVis VLX mobile mapping device

Point Cloud

The point cloud obtained after processing by NavVis can be seen in Fig. 5.4. The close-up render shows the density of the point cloud and how well it captures textural information. The geometry is very precise, with noise-free flat surfaces and detailed small objects such as poles. The captured datasets contain roughly 50 million to 150 million points for scenes of the size of 800 m^2 to 2000 m^2.

Images

The images captured by this device are embedded in the `e57` point cloud as panorama images. They have a resolution of 8192 by 4096 pixels. To use them in the photogrammetry pipeline, they have been projected to a cube map, with each side stored as an individual file, with a resolution of 3072 by 3072 pixels. Images are captured manually by pressing a button on the device. This gives control of the exact position and frequency at which images are captured. We captured images roughly every two steps, giving us about 300



(a) (b)

Figure 5.4: A point cloud captured with the NavVis VLX

panorama images per scene, which results in 6 times as many unprojected images usable for the photogrammetry strategy.

5.1.3 Datasets

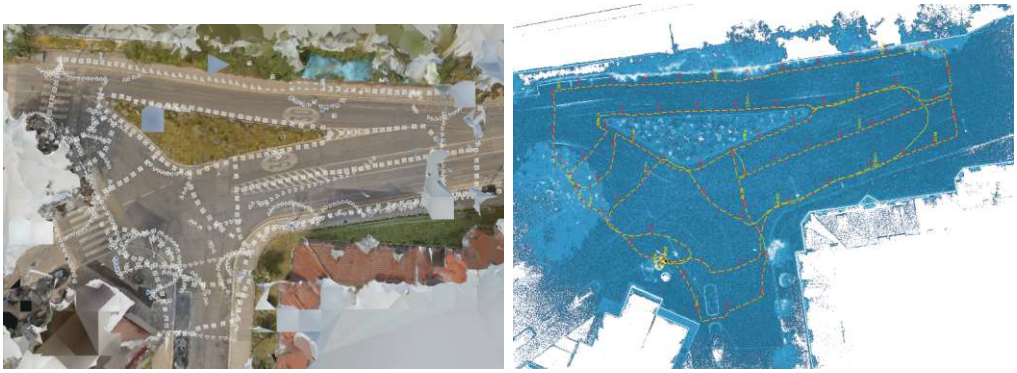
The methods presented in this thesis are evaluated on three different custom datasets. All three have point cloud data captured from the VLX, and two of the datasets contain images captured by the VLX and additionally a handheld DSLR. Table 5.1 gives an overview of the datasets, their area, and input data count.

Dataset	Ground Area	Points	Images from VLX	Images from DSLR
Cumberland	2000 m^2	133.6 million	1949	887
Jenullgasse	740 m^2	78.3 million	1307	1396
Mex	810 m^2	69.8 million	-	-

Table 5.1: Statistics of our captured datasets

Cumberlandstraße

The first dataset, *Cumberlandstraße*, spans an area of 2000 m^2 of an intersection in Vienna. It features a complex intersection with many traffic signs, road markings, lanes, and vegetation. The scene was captured with the laser scanner and the handheld camera. The path taken along the scene can be seen in Fig. 5.5. Images and laser scans were captured by walking on the sidewalk and road. Further images were captured by walking around complex objects such as traffic signs. The raw data contains 133.6 million points



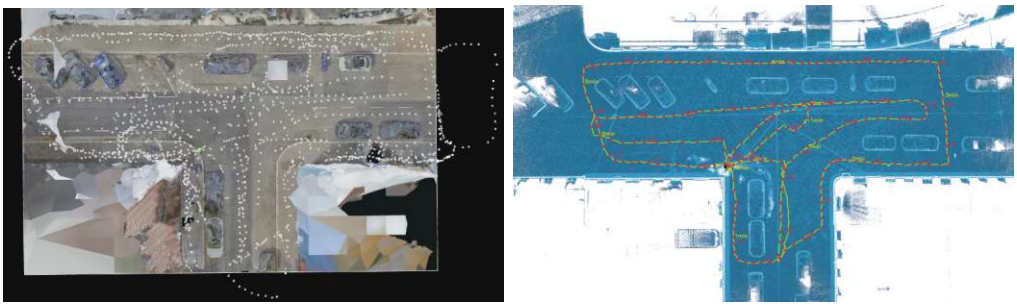
(a) (b)

Figure 5.5: Cumberlandstraße: camera registrations (a) and mapping path (b)

for the point cloud and 2836 images, 1949 of which come from the scanner, and the remaining 887 from the DSLR.

Jenullgasse

The next scene, *Jenullgasse*, was also captured in Vienna. Since reflective and transparent surfaces can be challenging for LiDAR scanners and the photogrammetry process, this location was specifically chosen, as it contains many parked cars. Again, the data was captured by walking on the sidewalk or road and by orbiting around objects of interest. The path can be seen in Fig. 5.6. We captured each car from many different perspectives, providing the photogrammetry pipeline with ample data for reconstructing these challenging objects. In total, 2703 images were captured, 1307 of which stem from the VLX. The point cloud has 78.3 million points with a scene size of 740 m^2.



(a) (b)

Figure 5.6: Jenullgasse: camera registrations (a) and mapping path (b)

Mex

The last scene is from an intersection in a town in Switzerland called *Mex*. This one was only captured by the VLX scanner. The point cloud has 69.8 million points, and it

spans an area of 810 m^2. This dataset was an example provided by NavVis, capturing this dataset, they did not focus on capturing many panorama images of the scene. Thus, the number of images is insufficient for a good reconstruction using photogrammetry.

5.1.4 Ground Truth Semantic Labels

Reconstructed meshes were manually annotated with semantic labels for all three datasets to provide a benchmark for the segmentation part of the pipeline. These meshes are reconstructed from the point clouds, as they have a smoother and more precise topology. To create the annotation, first, a semantic segmentation was performed with the pipeline to get most of the vertices annotated correctly. Then, the manual step of the pipeline, which allows correcting the segmentation, was used (see Section 3.8.4). In Blender, each vertex was inspected and set to the correct label, adhering to the class definitions from CityScapes [12]. Fig. 5.8 shows a top-down view of each scene with the corresponding annotations.

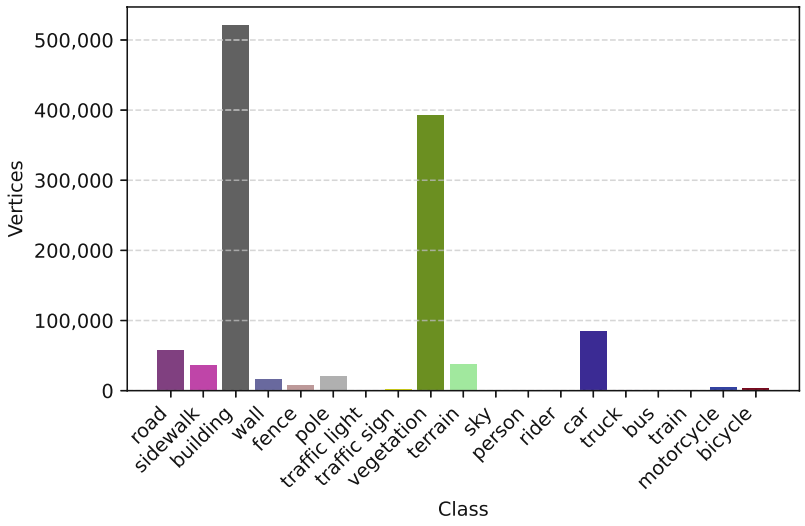


Figure 5.7: Class distribution of the ground truth annotations for all three datasets combined

In Fig. 5.7, a distribution of the labels can be seen. Due to the nature of the domain, the datasets do not have an even class distribution, and some of the classes are not present in our dataset. Predictions from these classes are filtered out by default and are not considered when calculating the evaluation metrics. Our annotated ground truth datasets use the following subset of the CityScapes classes: *road, sidewalk, building, wall, fence, pole, traffic sign, vegetation, terrain, car, motorcycle,* and *bicycle.* Creating the ground truth annotations took a total of 10 hours for all three datasets.

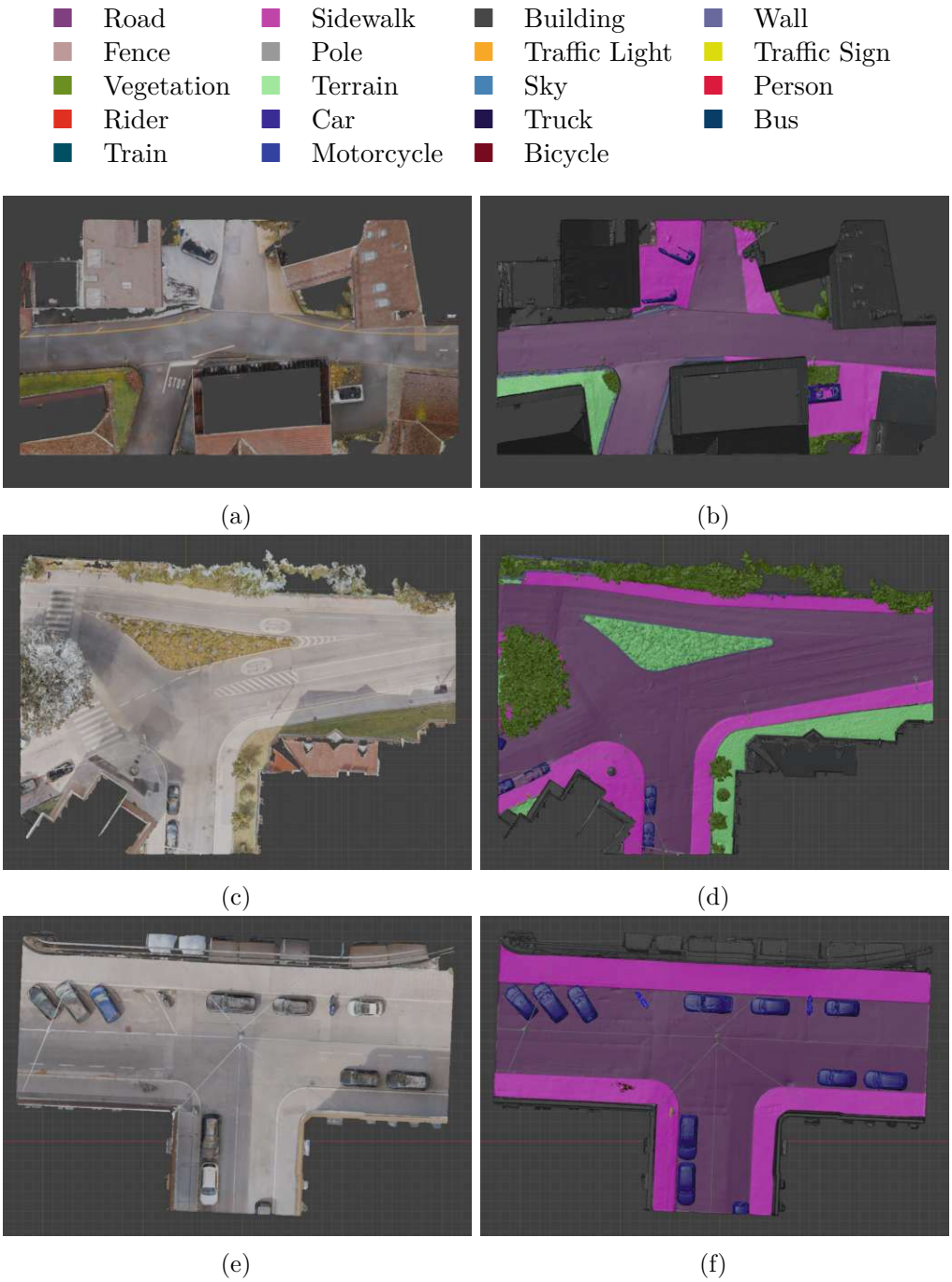


Figure 5.8: Ground truth annotations on Mex (a, b), Cumberlandstraße (c, d), and Jenullgasse (e, f)

5.2 Reconstruction

This section examines the reconstructions generated by the pipeline. It begins with a visual demonstration and description of reconstructed meshes from both input modalities. Next, four experiments were performed to investigate the influence of differences in input data and configurations. This was done using quantitative and descriptive analysis. In doing so, good baseline parameters for the pipeline configuration were defined.

5.2.1 Results

In this section, we showcase reconstructions from images and point clouds and then highlight their differences and challenges.

From Images

The image-based datasets described in Sections 5.1.1 and 5.1.2 produced usable reconstruction results, albeit not artifact-free. While most areas were reconstructed accurately, some scenes contain obvious issues. These issues are mainly visible on cars and the road. Fig. 5.9 highlights renderings of reconstructions from this data.

In the reconstruction process, not all of the input images could be aligned by reality capture. The reconstructions of *Cumberlandstraße* and *Jenullgasse* used 68.8% and 51.8% of the images respectively.

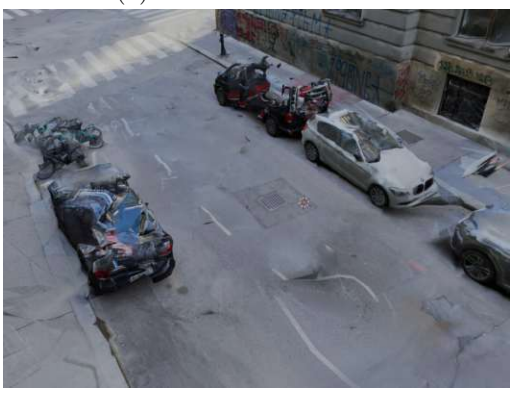
In Sub-figure 5.9a a wall with a lot of textural and geometric variation can be seen. The reconstruction here is very accurate, and the resulting texture is sharp and has a high resolution.

In Sub-figure 5.9b, a big hole in the street can be observed. This is a big issue, as this interferes with the physics simulation of vehicles. In some cases, it might be impossible to drive on these areas without manually fixing the mesh. This is because these regions have less textural variation, which results in fewer features usable by the photogrammetry software. This issue did not occur with our datasets, which contain more images focusing on capturing details on the ground. Furthermore, cars were most often reconstructed with many artifacts, especially where the windows are.

In Sub-figure 5.9c, two traffic signs can be seen. These signs were specifically targeted with orbiting detail shots. However, there are still artifacts in the form of holes, cut-off portions, and blurry textures. This shows how challenging the flat, uniform, and reflective surfaces of signs themselves are.

Another common issue with image-based reconstructions is the appearance of additional, often floating geometry. This happens especially at the outside and higher part of the mesh. The white/blue texture of these regions hints that they might be created due to RealityCapture wrongly estimating the distance of the sky and clouds. Fig. 5.10 shows these artifacts in a reconstructed mesh.

(a) Good reconstruction in areas with a high texture variation.



(b) Deformations on cars and the road.

(c) Traffic signs are sometimes only partially reconstructed.

Figure 5.9: Results of a reconstruction using an image based dataset.

Figure 5.10: A scene reconstructed from images, showcasing floating artifacts.

From Point Clouds

Reconstructions from point clouds produced very smooth and precise meshes. Fig. 5.11 highlights such a reconstructed mesh. The ground was fully reconstructed without any holes or major defects. Cars and bikes appear with detailed geometry, and there are only minor artifacts, such as their windows not being reconstructed. However, the texture of the reflective surfaces of cars does not accurately represent the car's true color, showing visible reflections of the surrounding area. This is an artifact that is propagated from the input data. Thin objects like traffic signs and poles are sometimes only partially reconstructed.

The generated textures are of a high quality, and road markings are clearly visible and sharp. This is made possible by the high-density input point cloud used for the reconstructions.

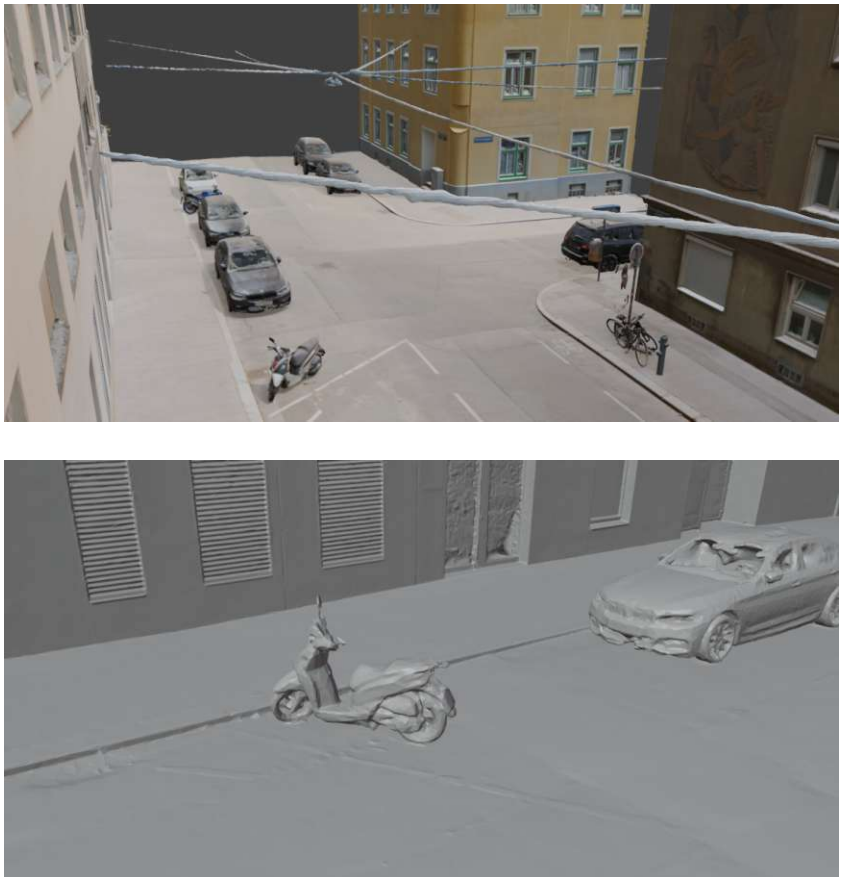


Figure 5.11: Jenullgasse reconstructed from a point cloud

Comparison

Having two datasets of the same scenes with both modalities allows for the direct comparison of the strategies. As seen in Fig. 5.12 and 5.13, both modalities produce accurate meshes with few striking visible artifacts when seen from far away.

(a) From a point cloud

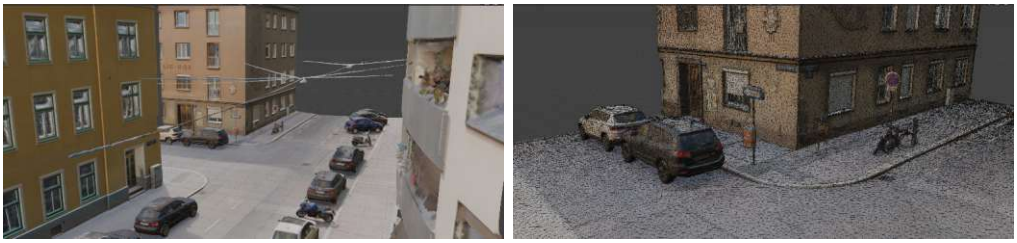
(b) From images

Figure 5.12: Cumberlandstraße reconstructed

With both strategies, the meshes contain no holes in the road, which is crucial for the driving simulation. The reconstruction from the point cloud produces a very uniform mesh, while the photogrammetry mesh is more noisy and jagged. However, the road surface is smooth enough and does not behave differently in manual simulated driving tests.

Both meshes have high-resolution textures, allowing the identification of road markings and traffic signs. However, the reconstructions are not perfect. Issues arise, especially with regard to traffic signs and cars. In Fig. 5.14, renderings of reconstructions of both data sources from the same perspective are shown. Using these images, we highlight the key differences between both results.

In both reconstructions, traffic signs do often have holes or are missing entirely. Overall, the signs from the photogrammetry strategy show such artifacts less frequently. However, since the raw point cloud contains these missing sections of the signs, the artifacts are possibly due to the geometry being too thin and sometimes being dismissed in the Poisson surface reconstruction [38]. The text on the traffic signs is sharper and more readable in



(a) From point cloud

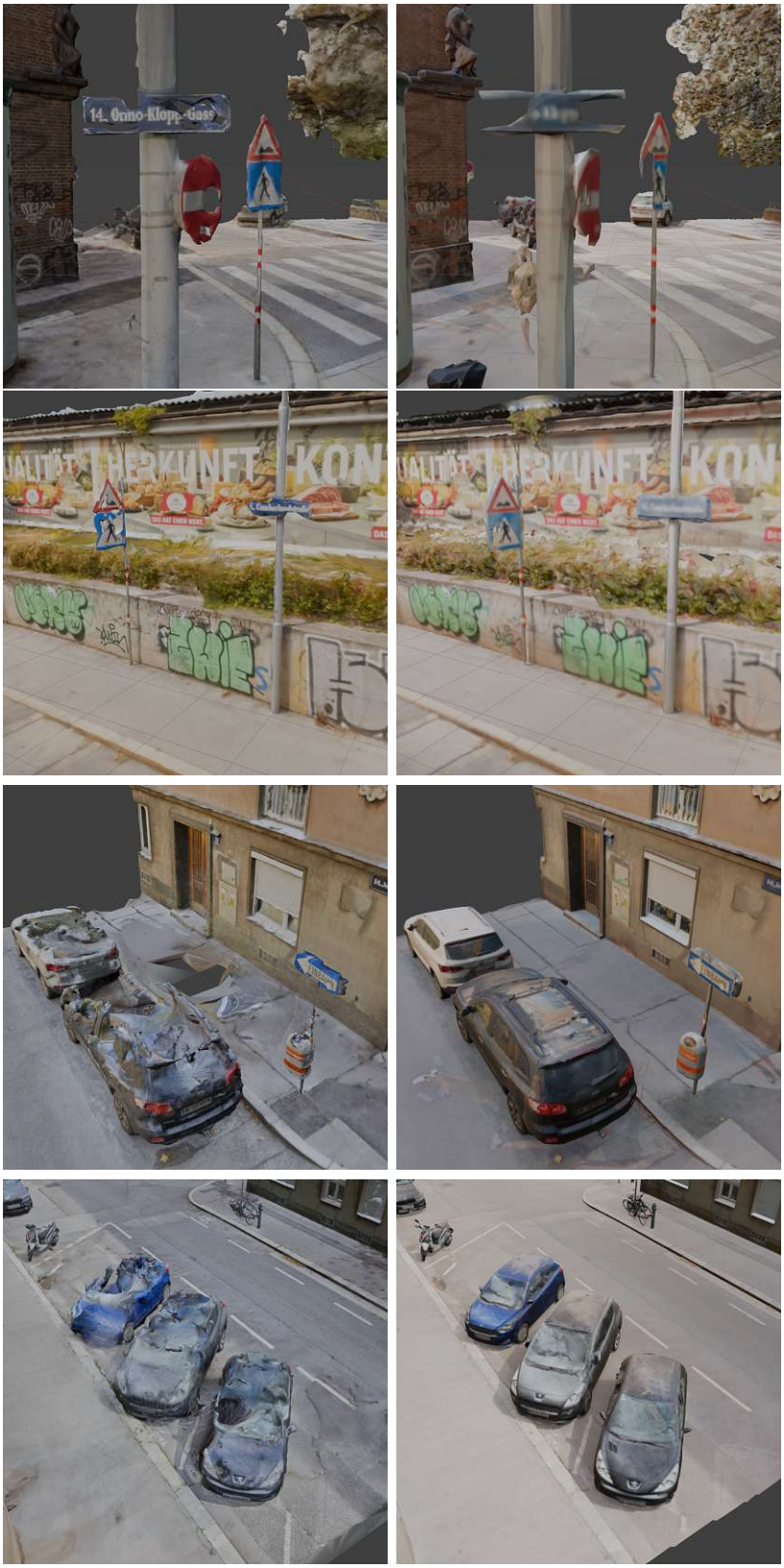
(b) From images

Figure 5.13: Jenullgasse reconstructed

the image-based reconstruction.

The cars reconstructed from the LiDAR data have a mostly accurate geometry. Geometric artifacts are mainly visible in the transparent windows. The texturing is not accurate, showing the reflections of the surrounding area. These reflections are visible in the raw input data of the point cloud and arise due to the way color information is mapped from RGB images to the points. The cars reconstructed from images show significant defects in any case. Due to the reflection and transparency, the photogrammetry process fails to find enough matching features on these surfaces.

Overall, the reconstructions from both modalities produce very similar high-quality results. Both strategies proved to be viable for the creation of digital twins of streets and intersections.



(a) From images (b) From point cloud

Figure 5.14: Direct comparison of mesh and texture reconstructions from both modalities

5.2.2 Poisson Reconstruction Depth

With this experiment, we explore the influence of the Poisson surface reconstruction [38] depth on the resulting reconstruction. The algorithm is only used with the point cloud strategy.

The choice of the Poisson surface reconstruction depth controls the depth of the underlying octree used in the Poisson surface reconstruction algorithm [38]. Each increase in depth doubles the spatial resolution in each axis. It significantly impacts processing time and the mesh resolution. This experiment aims to evaluate the differences in the resulting reconstruction at varying reconstruction depths. Different features become visible at different reconstruction depths. In particular, we are interested in traffic signs and road markings.

Methodology

For this experiment, the three point-cloud-based datasets obtained from the NavVis VLX were used (see Section 5.1.2). By running the first steps of the pipeline, the point cloud was first reconstructed to a mesh, then artifacts were cleaned up, and finally, the mesh was textured. This allowed us to measure processing and post-processing times. The scene was reconstructed at depths 10 to 13. The upper limit was set to 13 because reconstructions beyond that resulted in meshes of unmanageable size with hundreds of millions of faces.

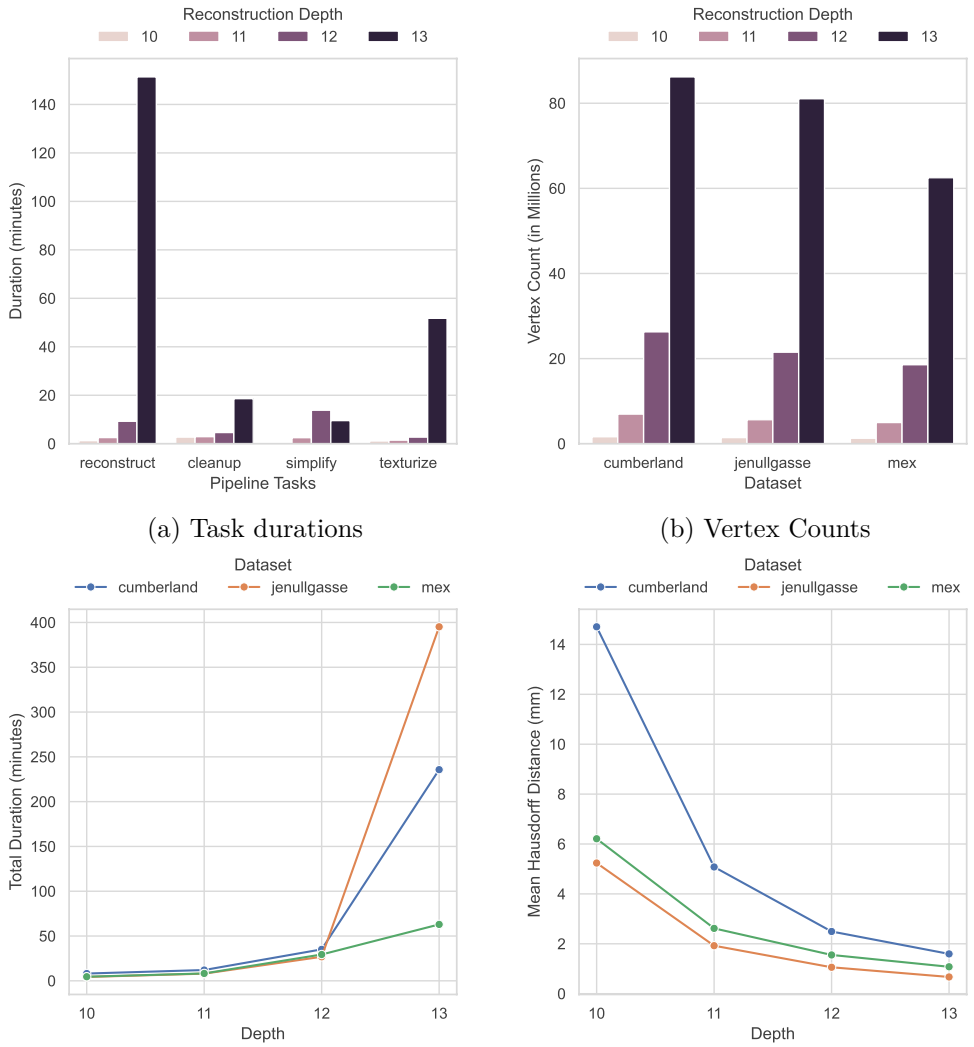
Critical areas of interest were highlighted and compared between each reconstruction depth. The reconstruction and further processing durations were recorded, and renderings of the scene were created for comparison.

To measure the difference between the point cloud and the reconstructed meshes, the Hausdorff distance [4] was used. The Hausdorf distance measures how far away two meshes are. Given two Meshes, A and B, the Hausdorff distance for a point on Mesh A is given by the distance to the closest point on Mesh B. We measured the distance by sampling points on the point cloud and calculating their Hausdorff distance to the reconstructed mesh. This direction was chosen to minimize errors due to the "bubbling" artifacts in the reconstruction.

Results

Distance Sub-figure 5.15d shows the measured Hausdorf distances from the point cloud to each reconstructed mesh. As the depth increases, the mean distances decrease from 6 mm to 14 mm at a depth of 10 to only 1 mm to 2 mm at a depth of 13.

Vertex Count Increasing the reconstruction depth greatly increases the vertex count of the resulting mesh (see Sub-figure 5.15b). For each increase in depth, the count increases by a factor of 3-4. At a depth of 13, this reaches 60 to 80 million vertices. Meshes of this size create new challenges for downstream processing.



(a) Task durations

(b) Vertex Counts

(c) Total reconstruction duration

(d) Mean Hausdorff distance from the input point cloud

Figure 5.15: Quantitative results of the experiments with varying Poisson surface reconstruction depths

Processing Time As seen in Sub-figures 5.15a and 5.15c, the reconstruction time, as well as the duration of the successive tasks increase with a higher depth. At a depth of 13, reconstruction and cleanup took over two hours, which is acceptable for our purposes. However, the mesh size became unmanageable with our initial simplification implementation. The simplification ran for over 12 hours before we had to cancel it. This increase in computing time could be attributed to Meshlab running out of RAM, which is limited to 32GB on our testing hardware. We used a different simplification method in this case to still be able to create reconstructions with this depth and evaluate them.

Instead of just using quadratic edge collapse [20], we first performed Clustering Decimation[1], which runs significantly faster, even on meshes with millions of vertices. However, this technique introduces unwanted artifacts, so it is only enabled when necessary. This approach was only used for reconstructions created with a reconstruction depth of 13.

While the measured Hausdorf distances, processing times, and vertex counts provide valuable quantitative data, they do not paint a full picture. Small errors in the mesh may only slightly influence the Hausdorff distance while potentially being detrimental to the use in downstream tasks. We evaluate areas of interest in the reconstructed meshes to provide a more comprehensive assessment. For this, various renderings of the meshes (as seen in Fig. 5.16) are used.

(a) Depth 10 (b) Depth 11 (c) Depth 12 (d) Depth 13

Figure 5.16: Reconstructions from a point cloud at different Poisson depths

Traffic Signs The first row of images shows a stop sign at all reconstruction depths. At depths of 10 to 11 the sign is not, or only partially visible. At depth 12, the sign becomes visible and identifiable as such. However, there are minor artifacts in the form

[1] https://pymeshlab.readthedocs.io/en/0.1.9/filter_list.html#simplification_clustering_decimation

of holes. Interestingly, these artifacts did not occur for axis-aligned signs, as seen in the second row. At a reconstruction depth of 13, the sign is visible in high quality and free of holes or major deformations.

Poles Poles are visible at all reconstruction depths. However, at lower depths, some deformations are visible. With higher depths, they become smoother and more detailed.

Color Resolution The color information is available only as a per-vertex color attribute at this pipeline stage. This color is later used to bake a texture on a lower-resolution mesh. Thus, the density of the vertices limits the final texture resolution. Features such as road markings are visible at every reconstruction depth. However, they are blurry at lower depths of 10 and 11. Increasing the depth to 12 or 13 results in noticeably sharper textures.

Discussion

From the above-presented results, we can conclude that the choice of reconstruction depth is a tradeoff between quality and performance. A value of 12 provides a good middle ground with small artifacts. Increasing the depth to 13 produces noticeable improvements in the quality of textures and the mesh at the cost of significantly larger files, longer processing times, higher system requirements, and an additional source of artifacts being introduced. At both resolutions, traffic signs and road markings are clearly visible. Depths beyond 13 are not worth considering, as the input point clouds for this experiment have roughly the same amount of points as the meshes reconstructed with a depth of 13 have vertices.

5.2.3 Mesh Resolution

The mesh reconstruction step produces meshes with millions of vertices and faces. Meshes of this size are inefficient and significantly increase the processing time and RAM usage of downstream tasks. With this experiment, we investigate how simplifying the reconstructed mesh to different face counts influences the reconstruction quality. This is relevant for both strategies, meshes from point clouds and photogrammetry.

Methodology

In this experiment, our three point-cloud-based datasets were used again. First, a mesh reconstruction was performed using a Poisson Depth of 12, ensuring high-quality meshes were created. The meshes were simplified to face counts of $100,000$, $500,000$, and $1,000,000$. The simplification was done using Quadric Edge Collapse Decimation [20], which allows a decimation to a specific vertex count. The Hausdorff distance from the input point cloud to the simplified mesh was calculated. We are interested in the mean of the Hausdorff distances of all points, giving us the average distance from the point cloud.

Furthermore, simplified reconstructions at the same face counts were created for the image-based reconstructions. This allowed us to evaluate the differences caused by varying face counts for both modalities. Areas of interest were compared for all different face counts. These areas include the quality of thin objects such as signs and poles and how efficiently flat areas are stored.

Results

Fig. 5.17 shows the Hausdorff distances measured from the point cloud to the reconstructed meshes for each point-cloud-based dataset. For comparison, we also calculated the Hausdorff distance to the non-simplified meshes, which have 16 to 52 million faces. The mean Hausdorff distance significantly improves with an increasing face count. At a face count of 1 000 000 million, the Hausdorff distance approaches the distance the full-resolution mesh achieves. When comparing the point counts of the datasets (see Table 5.1) with this figure, we observe that the Hausdorff distance correlates with the dataset size. This shows that larger scenes require a higher mesh resolution for a similar reconstruction quality.

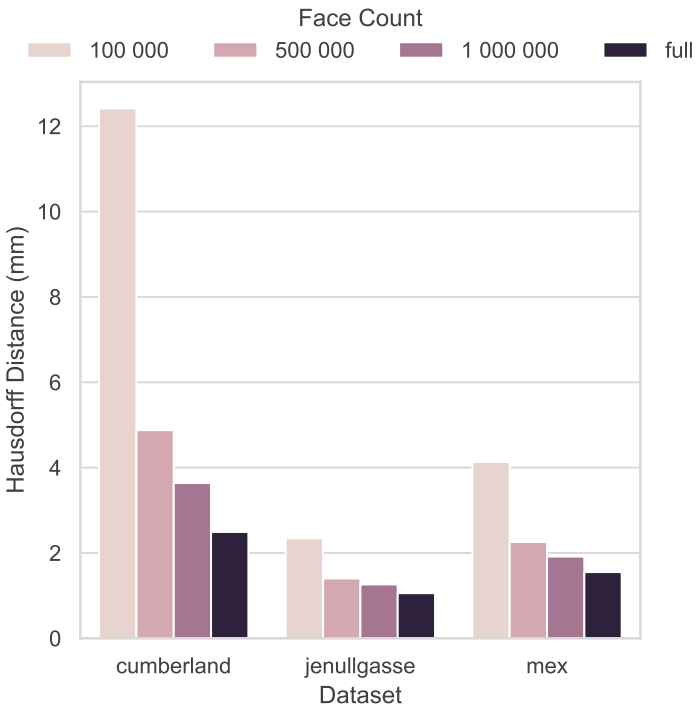
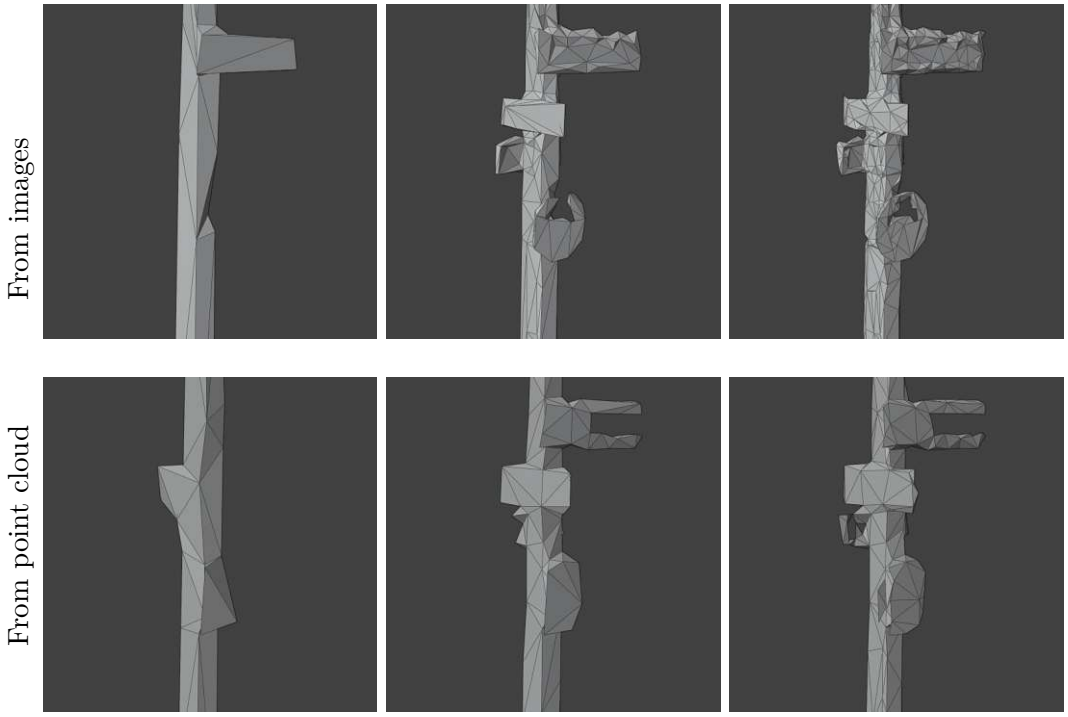


Figure 5.17: Hausdorff Distnace for different face counts

Fig. 5.18 shows a pole with four traffic signs for each reconstructed mesh. The reconstructions from images and point clouds behave similar for this object. A resolution of 100, 000 faces is insufficient, as some signs are completely absent in the reconstruction. Increasing the resolution to 500, 000 faces, the signs get reconstructed with a few polygons. This is

enough to identify them, but increasing the face count to 1,000,000 further increases the quality of the signs.

Fig. 5.19 shows a larger segment of the scene. Here we can observe the difference between the two input modalities. The meshes reconstructed from the point cloud have a more uniform face distribution.



(a) 100 000 faces (b) 500 000 faces (c) 1 000 000 faces

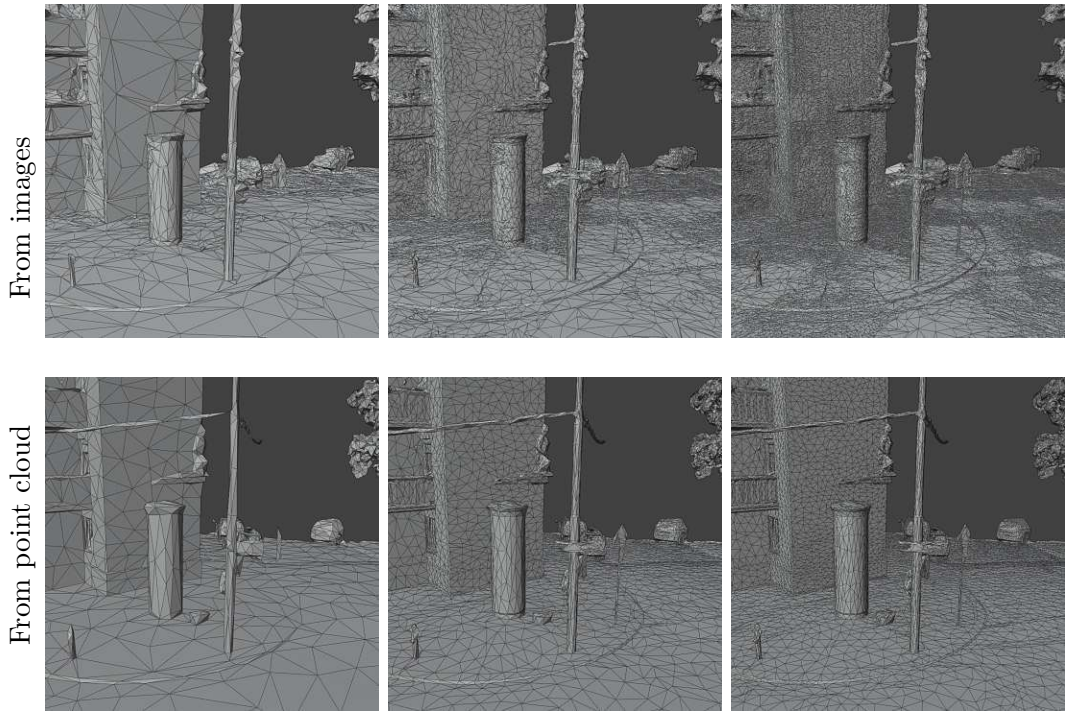
Figure 5.18: A pole with traffic signs simplified to different face counts for both modalities

Discussion

The reconstructions from both modalities produce similar results and artifacts at similar face counts. Higher resolutions increase the mesh quality while increasing file size and reducing performance.

Another factor to be considered is the topology of the mesh. In some cases, object boundaries might not align with the edges of the mesh. This is especially true for low sidewalks, which become flat with low mesh resolutions. When that happens, the segmentation step cannot produce accurate and clear boundaries. In some cases, the mesh resolution constrains the maximum possible segmentation quality.

Depending on the scene size and complexity, a face count of 500,000 to 1,000,000 is recommended. If the segmentation produces edges that are not aligned with object boundaries, increasing the face count might help.



(a) 100 000 faces (b) 500 000 faces (c) 1 000 000 faces

Figure 5.19: A reconstruction simplified to different face counts for both modalities

5.2.4 Texture Resolutions

This experiment aims to compare the quality of different texture resolutions for both input modalities.

Methodology

Image-based and point-cloud-based reconstructions of *Cumberlandstraße* were used for this experiment. After performing a mesh reconstruction, clean-up, and a simplification to 500,000 faces, the texturing step was performed at resolutions 4096 (4k), 8192 (8k), and 16384 (16k). The texturing from point clouds only supports a single texture, so only one texture was used with both modalities. The overall differences were highlighted, and the legibility and sharpness of road markings and text were compared between the different resolutions and modalities.

Results

Overall Fig. 5.20 shows a wall with a detailed texture and a foreground object. For the image-based approach, a resolution of 4k shows an erroneous-looking pattern on the bricks. At higher resolutions, this issue gets resolved. However, other artifacts are

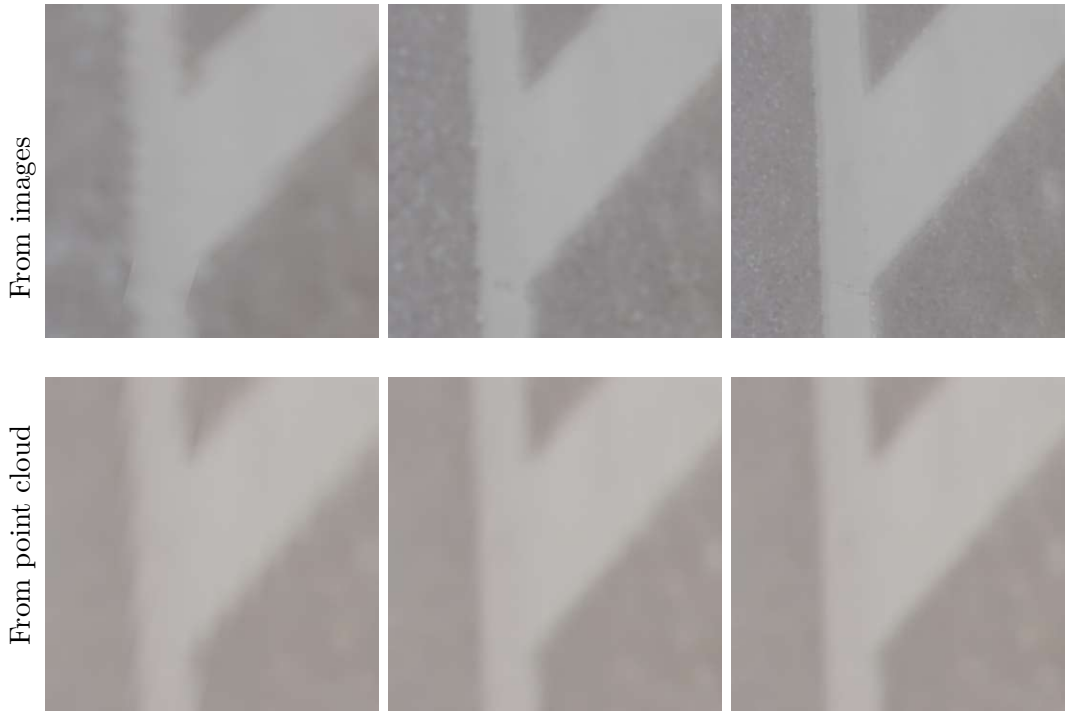
(a) 4K resolution (b) 8K resolution (c) 16K resolution

Figure 5.20: Comparison of different texture resolutions for both modalities

introduced on the sidewalk as darker-tinted blobs. The point-cloud-based approach shows no visible errors and gets noticeable sharper with increasing resolution.

Road Markings Fig. 5.21 highlights road markings with all created textures. For the image-based results with the lowest resolution of 4k, road markings are visibly blurry. At the next higher resolution of 8k, this is already greatly improved. The jump in quality from 4k to 8k is the most noticeable. Further increasing the resolution to 16k further improves the edge sharpness. With the point-cloud-based reconstruction, an increase in quality can also be observed. However, this improvement is not as significant as with the image-based approach. The image-based reconstruction benefits of a 16k texture in this case. The line boundary is clearer, and more details on the road surface can be observed. The point-cloud-based reconstructions lack detail on the road surface.

Text Fig. 5.22 shows a closeup of a sign containing text. For the image-based approach, this text becomes barely readable at a resolution of 8k, with a significant improvement at a resolution of 16k. In the case of the point-cloud-based approach, the text is not readable, even at the highest resolution.



(a) 4K resolution (b) 8K resolution (c) 16K resolution

Figure 5.21: Close-up of a road marking, comparing different texture resolutions for both modalities

Discussion

Depending on the total surface area of the scene, a texture resolution of 8k or 16k is sufficient. If performance is not a constraint, a resolution of 16k is best. The main difference in the textures of both modalities is their sharpness and consistency. Textures obtained from photogrammetry can have a potentially much higher resolution. However, due to slight errors in the camera alignment, the texture projection may not align perfectly, which can result in some artifacts. The textures obtained from the point-cloud-bassed strategy are overall more consistent and artifact-free at the cost of a lower maximum resolution and sharpness. In the photogrammetry process, multiple 16k textures could be used for an even higher fidelity.

(a) 4K resolution (b) 8K resolution (c) 16K resolution

Figure 5.22: Close-up of text, comparing different texture resolutions for both modalities

5.2.5 Image Subsets

The last experiment we performed investigated whether images captured from the NavVis VLX are enough for a good reconstruction and if complementing them with further images can improve the reconstruction quality.

Methodology

We created two reconstructions with different images from the image-based dataset of *Cumberlandstraße*. This dataset contains 2836 images. 1949 of these images were captured using the VLX, providing a broad coverage of the scene. The remaining 887 images were captured using a handheld camera and focused on capturing details on the ground and around traffic signs and poles. One of the reconstructions used all images, and the other used just the images captured with the VLX. The resulting meshes were compared with a descriptive analysis. The key differences are highlighted in this section.

(a) Subset, only the 1949 images captured with the VLX

(b) All 2836 images

Figure 5.23: Comparison of an image-based reconstruction using only a subset of all images (a) to a reconstruction using all images (b)

Results

Fig. 5.23 shows renderings of the reconstructions described above. The pole of the traffic sign failed to be reconstructed and the texture on the sign is very blurry when using only the images from the VLX. Adding the detail shots, the traffic sign gets reconstructed a lot better. The pole is fully visible and the sign is mostly reconstructed. However, it still contains a missing segment in the middle. The texture of the sign is also a lot more accurate.

The pole in the second row shows more artifacts that occur when not using all the images. The signs are only partially visible, and the pole contains visible deformations. When using all images, the signs are nearly fully reconstructed, the textures legible, and the pole is significantly smoother.

The last row shows the road without textures, which illustrates the roughness of the surface. When not using all images, the road is rougher. Using all images makes this surface smoother and reduces the amount of bumps.

Discussion

While just using images captured by the VLX produces usable results, they contain noticeable artifacts. Using close-up shots focusing on capturing more details of roads and objects of interest significantly improves the reconstruction results. Providing a smoother road surface, better-reconstructed meshes for detailed objects, and a better texture quality in these areas.

5.3 Segmentation

In this section, the results of the segmentation evaluation are presented. First, we show what the resulting segmented meshes look like. Then, we present the evaluation of the influence of different parameters on the segmentation performance.

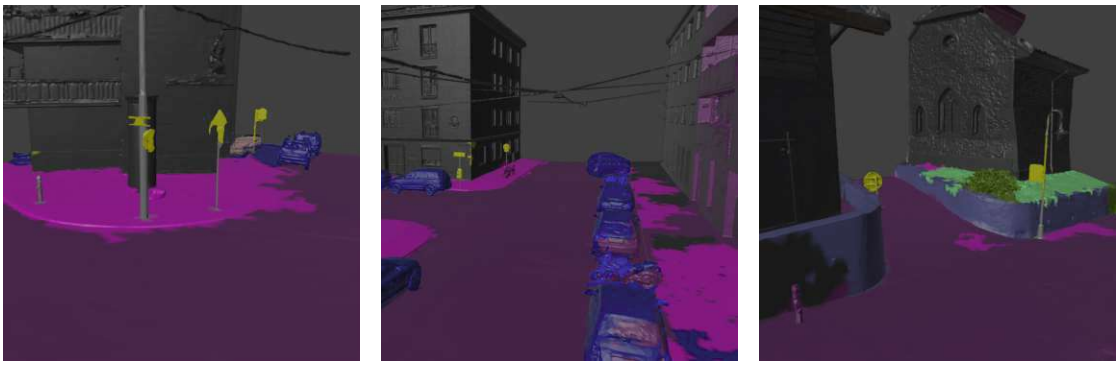
This evaluation was done by using the manually labeled ground truth datasets to measure different metrics. The overall performance was measured in mean per-class intersection over union (mIoU) and mean F1 score (mF1). Furthermore, for each class, the intersection over union (IoU), precision, recall, and F1 score were calculated. IoU measures the similarity between the set of predicted vertices and ground truth labeled vertices for each class. It is calculated by dividing the intersection of the sets with their union. mIoU refers to the mean IoU over all classes. The parameters in question are parameters related to the over-segmentation, sampling techniques, and 2D segmentation model selection.

Fig. 5.24 shows reconstructions of all three datasets segmented by the pipeline. Overall, the segmentations are mostly accurate. Roads and sidewalks are usually separated, traffic signs are identified as such, vegetation and terrain are recognized, and cars are identified. However, three types of issues can be observed:

1. **Fuzzy boundaries**: The border between classes sometimes does not follow the actual object boundaries. This often occurs with sidewalks, as there is little variation in the mesh and texture.

2. **Misslabeled regions**: Larger regions might be misslabeled. This happens especially in regions that are occluded from the virtual views used in the segmentation. This can also happen in regions that are misclassified in multiple views.

3. **Noise**: Smaller patches of objects may be fragmented into multiple classes. This can happen for classes that are very similar, such as *terrain* and *vegetation*, *road* and *sidewalk*, or *wall* and *building*.



(a) **Cumberlandstraße**
Signs and poles are detectet. Fuzzy boundary between the sidewalk and road.

(b) **Jenullgasse**
Cars are detected well. The occluded sidewalk is only partially labeled correctly. Incorrectly labeled part of the building.

(c) **Mex**
Misslabeled patches of sidewalk on the road.

Figure 5.24: Segmentations produced by the pipeline

5.3.1 Over-Segmentation

Methodology

First, to find suitable parameters for the over-segmentation, segmentations with different parameters influencing the over-segmentation were performed and checked against the ground truth. The parameters in question are:

k threshold The k threshold is a constant used in the Felzenzwalb algorithm [15]. It influences the difference between two components that is required for them to be considered a boundary. Higher values for k tend to produce larger-sized components.

min vertices Our algorithm implementation tries to make each segment at least this size if possible. This is done by naively merging small components without looking at the quality of the edge connecting them. Since this does not look at the local properties of the mesh, it might merge components that do not belong together, thus resulting in under-segmentation.

max vertices The maximum component size can be limited. This limits the possibility of under-segmentation, where a component spans a large region containing multiple classes.

Results

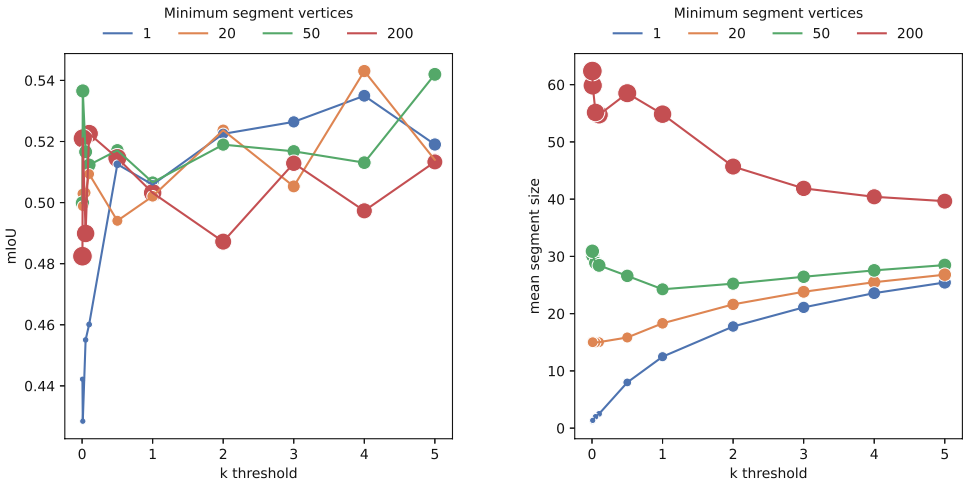
Fig. 5.25a shows the influence of the k threshold on mIoU across different minimum vertex counts. Observing the progression of the curve without a minimum segment size constraint (blue line), it is clear that performance improves as the k threshold increases. This is likely because a higher k threshold leads to larger segment sizes, which helps the algorithm reduce noise, form smoother boundaries, and label occluded areas.

The experiments with higher minimum vertex counts show unpredictable results for low k thresholds. This can be explained by the fact that with lower values, the segment sizes produced by the Felzenzwalb algorithm tend to be lower (see Fig. 5.25b). Since the minimum vertex size constraint is then applied through a naive approach without considering edge weights, errors may be introduced. The more vertices are merged, the higher the potential error.

Performance stabilizes across different minimum segment sizes at higher levels of the k threshold. With the approach that does not enforce a minimum vertex count producing more consistent results. In Fig. 5.25b it can be seen that the mean segment size is smaller than the minimum vertex size. This can be explained by the way the merging is implemented. The minimum vertex size is only a soft constraint, two adjacent components are only merged if both are below this threshold. This also explains why the segment size decreases with higher k thresholds when using a minimum segment size of 200. As the segments produced in the initial segmentation become larger, fewer segments get merged in the second step because these segments would become too large.

5.3.2 Virtual View Sampling

This experiment analyzes the influence of different sampling techniques for selecting virtual views. The virtual views are used to perform a 2D semantic segmentation on renderings of the reconstructed mesh, and the predicted labels are then projected back onto the mesh. The methods in question are random sampling, uniform sampling in a grid, and sampling from manually specified points on the street. The goal was to check the hypothesis that using perspectives similar to the ones the 2D segmentation model was trained on improves the segmentation performance. Furthermore, the influence of the chosen number of samples and the combinations of the techniques were investigated.



(a) Influence of k threshold on mIoU

(b) Influence of k threshold on the segment size

Figure 5.25: The effect of different k thresholds and minimum vertex counts

Methodology

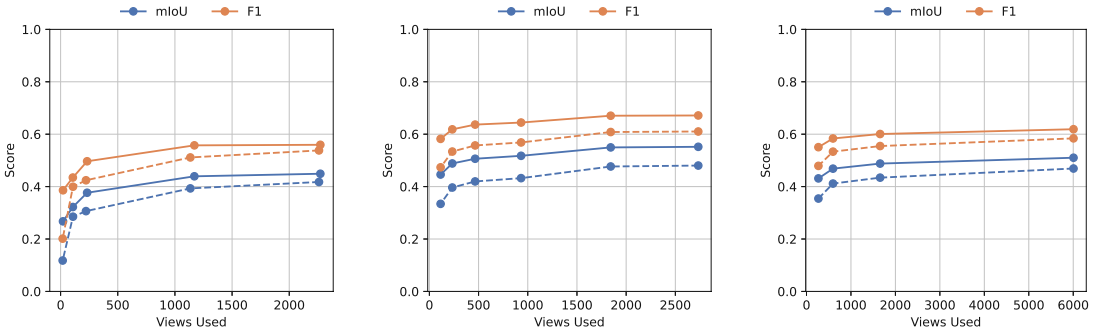
For all labeled datasets, points on the street were selected to provide street perspective sampling positions. Each technique was analyzed separately with varying sample counts to investigate how more sampled views influence the segmentation performance. Street perspective sampling used subsets of the selected points and duplicates at different heights. For each point, eight views were generated by rotating the camera around the up-axis. The uniform sampling count was adjusted by varying the grid size, which defines the distance between sampled points. The number of samples is inversely proportional to the grid size and depends on the dimensions of the scene. The random count was controlled directly. The resulting segmentations were then checked against the ground truth.

For the combined experiments, 40 different street sample points with vertical offsets of 1.5m, 3m, and 4.5m were used, resulting in 960 views. Additionally, uniform sampling was configured with a grid size of 4m, and 1000 random samples were used. Then, variations of this configuration were performed by disabling subsets of the sampling techniques.

Results

Fig. 5.26 shows the mIoU and mF1 scores against the chosen sampling counts. The experiments were performed with and without the prior over-segmentation. In all cases, more samples produced better results, with diminishing returns when approaching the respectively chosen maximum sample sizes.

Random Sampling Randomly sampling the scene produced the worst results, approaching an mIoU of 44.9 and an mF1 score of 55.9. After filtering views that show



(a) Random Sampling (b) Manual Sampling (c) Uniform Sampling

Figure 5.26: Comparison of different sampling methods. Showing metrics with over-segmentation (full line) and without (dotted line). The x-axes show the sum of views used (after filtering bad views) for benchmarking all three datasets together.

more than 60% of the image does not show any geometry, only 75% of the views were used in the segmentation.

Manual Sampling As expected, manual sampling from the street performs better, approaching an mIoU of 55.2 and an mF1 score of 67.1. 40 street points at three heights resulted in 960 views, of which 95% were usable.

Uniform Sampling Uniformly sampling the scene produces results comparable to those of street sampling. This is likely because the uniform sampling covers the views selected on the street. The number of generated samples depends on the scene size. For our datasets, a uniform grid size of 4 resulted in 3,000 to 8,000 views being considered. Only 40% of the views were used, which is not very resource-efficient.

Combined Table 5.2 shows the segmentation performance with different combinations of sampling techniques. The combined approach uses all views from the sampling strategies used. Using all sampling methods achieves a baseline mIoU of 53.3 and an mF1 score of 63.7. The best-performing configuration uses just street sampling, increasing the mIoU by 1.9 and the mF1 score by 3.5. A detailed evaluation per class is shown in table 5.3.

The IoU upper bound measures the maximal achievable IoU given the produced over-segmentation. This is calculated by finding an optimal assignment of components to classes and calculating the IoU of the resulting segmentation. The mean IoU upper bound is 89.3

Configuration	mIoU	Δ mIoU	F1	Δ F1
With all (baseline)	53.3	00.0	63.7	0.0
Without random sampling	53.3	-00.0	64.0	0.3
Without street sampling	52.0	-01.3	62.4	-1.3
Without uniform sampling	52.1	-01.2	63.3	-0.4
Without oversegmentation sampling	52.4	-00.9	62.8	-0.9
Only random sampling	44.9	-08.4	55.9	-7.7
Only street sampling	**55.2**	**+01.9**	**67.1**	**+03.5**
Only uniform sampling	51.0	-02.3	61.9	-1.8

Table 5.2: Benchmarks of different segmentation configurations. Mean intersection over union (mIoU, %) and F1 score (%) and their change against the baseline configuration are given.

Class	IoU	F1	precision	recall	mIoU Upper Bound
road	41.8	59.0	45.1	85.2	92.0
sidewalk	60.9	75.7	73.6	77.8	84.6
building	87.5	93.3	92.9	93.8	98.7
wall	20.9	34.5	54.4	25.3	70.8
fence	14.5	25.4	42.5	18.1	87.5
pole	15.4	26.7	79.0	16.0	94.9
traffic sign	62.3	76.8	74.6	79.2	72.9
vegetation	90.6	95.0	97.8	92.4	98.4
terrain	68.0	81.0	80.1	81.8	93.2
car	73.8	84.9	92.2	78.7	96.7
motorcycle	50.3	66.9	51.1	96.9	91.5
bicycle	76.3	86.6	85.8	87.3	90.3

Table 5.3: Per-class metrics of a benchmark with the best performing model. The metrics are intersection over union (IoU, %), F1 (%), precision (%), and recall (%). Also indicates the maximum possible mIoU given the calculated over-segmentation.

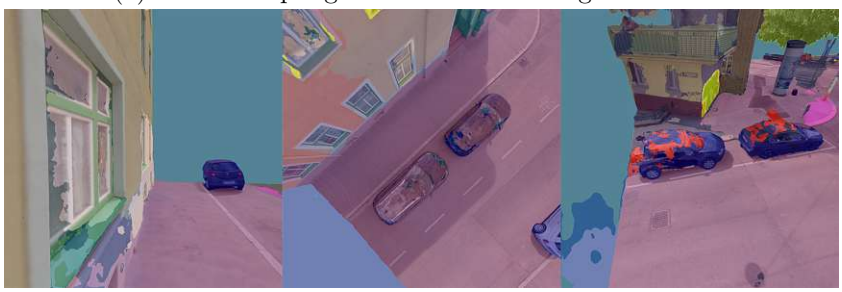
Discussion

Uniform sampling and street sampling showed a comparable performance. Since street sampling requires manually selecting the sample points, uniform sampling can be used instead for further pipeline automation. In all cases, using a prior over-segmentation increased the performance considerably.

Removing random and uniform views and using just manually selected street perspective views performs best. This indicates that it is more important to have good viewpoints than more viewpoints. Randomly and uniformly sampled viewpoints can be in unfavorable positions because they are either from within geometry, contain many occlusions, or have

perspectives not seen in the 2D domain the models were trained on. This results in an inaccurate 2D segmentation propagating to the 3D segmentation. The difference between good and bad sampling positions can be seen in Fig. 5.27.

(a) Street sampling with accurate 2D segmentations.



(b) Random sampling. Contains many misclassified areas. Especially *sidewalk* (pink) is often misclassified as *road* (purple).

Figure 5.27: Examples of 2D semantic segmentations. Rendered input images are overlayed with a semantic segmentation map generated with mask2former [10].

5.3.3 2D Segmentation Models

There are many 2D semantic segmentation models trained on the CityScapes [12] dataset available [10, 59, 57, 64, 56, 54, 32]. The 3D semantic segmentation performance was evaluated with different state-of-the-art 2D semantic segmentation models to find the best model to be used in the pipeline. The best-performing configuration, as established in the previous sections, was used for this experiment. All of the models are trained on the CityScapes dataset. They are provided by mmSegmentation[2]. The largest possible models that fit the available 8GB VRAM were chosen.

Table 5.4 shows the results of this experiment. The choice of 2D segmentation model is crucial for the 3D segmentation performance. Out of the tested 2D segmentation models, mask2former is the model that performs best on the CityScapes [12] dataset, achieving a mIoU of 81.71. Likewise, mask2former shows the best performance in our 3D segmentation benchmark.

[2] https://github.com/open-mmlab/mmsegmentation

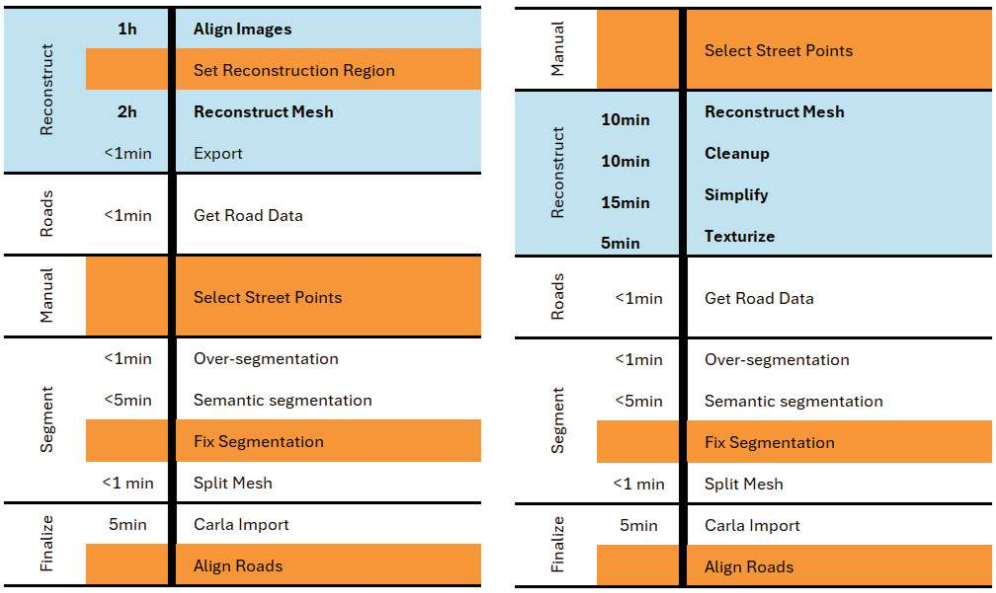
Model	mIoU	F1	CityScapes mIoU
mask2former [10]	**55.2**	**67.1**	**81.7**
segformer [56]	51.6	63.1	78.6
hrnet [54]	50.6	62.7	78.5
ddrnet [59]	45.7	57.8	80.0
pidnet [57]	41.0	52.2	80.9
pspnet [19]	38.9	49.1	79.5
isanet [32]	34.4	45.1	79.3

Table 5.4: Benchmark of the 3D segmentation performance using different 2D segmentation models. Also shows the respective mIoU on the CityScapes dataset.

5.4 Automation

5.4.1 Runnig Time

In this section, we investigate the degree of automation of the pipeline, including how long it and its steps run, when manual steps are necessary or beneficial, and how long they take.



(a) Image-based (b) Point-cloud-based

Figure 5.28: Timeline of the pipeline, showing key steps being performed for both strategies. Manual actions are highlighted in orange. The longest-running steps are highlighted in blue.

Fig. 5.28 shows a timeline of the steps performed by the pipeline for both branches, image-

based and point-cloud-based. The time measurements were taken from the reconstruction and segmentation of the three datasets described in section 5.1.3. The benchmark was performed on a Windows 11 PC with an AMD Ryzen 7 3700X 8-Core Processor, an NVIDIA RTX 2080, and 32 GB of RAM.

The indicated times are only rough indicators. The actual running time can vary depending on a number of factors, such as the number of input images, point cloud size, samples chosen while performing the 3D segmentation, and higher-resolution texture and mesh reconstructions. Still, this evaluation gives a good overview of the level of automation and running time of the whole pipeline and each step.

Most of the steps are fully automated, manual steps are highlighted in orange (Fig. 5.28). In both cases, the longest part of the pipeline is performing the mesh reconstruction, which can take several hours. The remaining parts run relatively quickly, completing in under 15 minutes. The point-cloud-based strategy has the advantage that the bulk of the processing time can be done without interruption. This is because the selection of the street points can be performed on the input point cloud itself, as it has the same coordinate system as the reconstructed mesh. In the image-based strategy, the user has to wait until the mesh is reconstructed to do so.

Another drawback of the image-based strategy is that a manual action must be performed in the middle of the mesh reconstruction. The user has to define the reconstruction region. For this strategy, the user has to wait twice, causing the pipeline to stall.

5.4.2 Time Savings

Some of the pipeline steps could be performed manually. Photogrammetry can be performed in the GUI of RealityCapture. Point clouds can be reconstructed and cleaned within MeshLab, and textures can then be baked in Blender. However, this involves many tedious and repetitive operations of preparing and importing input data, exporting to different formats, configuring the parameters and operations in all these applications, and waiting for long-running tasks to complete, only to start the next one. This manual overhead would make the manual creation of a digital twin take even longer.

The semantic segmentation could also be performed manually. However, this would be a time-consuming task. Our ground truth annotation process took several hours per scene, even though this only involved fixing an existing annotation done by the pipeline.

The pipeline takes care of all these steps. Only where strictly necessary the user has to perform a manual action. These actions are clearly defined, and the pipeline gives instructions on what to do and automatically watches the filesystem to detect when the action was performed so that it can continue immediately. This shows how much manual work the pipeline automates and allows for a much more efficient creation of a digital twin.

CHAPTER 6

Conclusion

This thesis has demonstrated the feasibility of creating high-quality digital twins from both images and point clouds in a semi-automated manner. Photogrammetry, using RealityCapture, proved to be an effective method for reconstructing 3D meshes from image data. By capturing between 500 and 2,000 strategically taken images, a high-quality 3D model can be generated within a few hours. We also provided detailed guidelines on camera hardware, configuration, and proper capturing techniques to ensure the input data is suitable for reconstruction.

Point clouds were also utilized as input data. For this modality, Poisson surface reconstruction is used to convert them into 3D meshes. When using high-quality point clouds, this process is both efficient and accurate. The input data requires RGB colors and normal information for each point. We captured this data using an existing mobile mapping unit that produces dense and precise point clouds.

A comparison between image-based and point cloud-based reconstructions revealed that both methods yield high-quality meshes and textures. Image-based inputs allow for higher-resolution textures, while the texture resolution from point clouds is limited by the density of the point cloud data. Reconstructions from point clouds showed a higher mesh accuracy with fewer geometric artifacts. Despite these differences, both methods produce reliable results with few artifacts.

The reconstructions were free of significant issues that could impede physics simulations. Roads were smooth and without holes, allowing simulated vehicles to navigate them without issue. However, some artifacts were observed, particularly in the reconstruction of cars and traffic signs, which occasionally exhibited holes or missing sections. This is caused by the reflective and low-texture surfaces, which are challenging for both the photogrammetry process and the capturing and coloring of point clouds.

The developed pipeline is mostly automated, running in a matter of hours with only a few well-defined manual interventions required, which take less than 20 minutes. This makes

it a highly cost-effective and efficient method for creating digital twins of intersections.

In addition to its efficiency, the pipeline is highly flexible. Intermediate files can be reviewed and corrected as needed, ensuring that any unexpected errors can be investigated and manually resolved. The pipeline's modularity also allows for easy modification and extension, opening up the project for future research and improvements.

This thesis also demonstrated that 3D semantic segmentation can be performed without the need for extensive ground truth data by leveraging pre-trained 2D segmentation models. This is done by segmenting 2D views of the 3D scene and back-projecting the predicted class labels onto the vertices of the mesh.

Additionally, we created a ground truth semantically annotated dataset. It contains reconstructed meshes of 3 intersections with per-vertex semantic labels. This dataset is valuable for evaluating semantic segmentation algorithms for 3D meshes in urban settings and was used to evaluate the performance of the pipeline's semantic segmentation algorithm. While our approach does not match the performance of state-of-the-art 3D semantic segmentation models trained on comparable domains, it still delivers highly usable results. This achievement is particularly significant given that we lacked sufficient ground truth data to train a model from scratch and instead had to rely on pre-trained 2D models for this task. Moreover, although the segmentation results are not perfect, the approach offers significant time savings for semantically segmenting a mesh for use in autonomous driving simulations compared to manually doing so — a key objective of this thesis.

6.1 Limitations

While the pipeline developed in this thesis is promising, there are several limitations that need to be addressed in future work:

Scene Size The current approach struggles with large scenes, which require longer processing times and have higher hardware requirements. This is especially true when point clouds are used as the input. A possible solution is to split the scene into smaller sections for processing and merge them afterward. Such an approach would allow the creation of digital twins covering way larger scenes, spanning multiple intersections and roads, or even large parts of cities.

Road Accuracy The alignment of road data with the 3D mesh is currently only an approximation. To improve this, a smarter alignment algorithm could be developed that more accurately aligns the road network with the road mesh by looking at the geometry. Alternatively, a method for creating the road network directly from the road mesh could be explored.

Fixing Artifacts Certain scene elements, such as signs and cars, often contain significant artifacts. These issues could be mitigated by detecting these elements and replacing

them with pre-made, high-quality meshes. This is especially important for traffic signs, as they play a crucial role in autonomous driving simulations.

6.2 Future Work

Building on the foundation laid in this thesis, several avenues for future research and development are apparent. Enhancing the pipeline's ability to handle larger and more complex scenes, improving the accuracy of road network creation, improving the 3D semantic segmentation performance, and refining artifact handling are key areas for further exploration.

Another possible direction for future work involves optimizing the data capture process. Developing dedicated capturing devices that streamline the acquisition of images, LiDAR data, or both could significantly enhance the input data quality and capturing efficiency. Currently, the pipeline handles these modalities as separate, independent branches, but fusing both data types presents an intriguing research opportunity. Such fusion could leverage the strengths and mitigate the weaknesses of each modality.

The generation of a broader variety of scenes can be beneficial for downstream tasks. This could involve developing tools to modify existing reconstructions or even create automatic variations easily. Additionally, exploring methods for procedurally generating scenes that adhere to specific constraints could allow for testing specific scenarios with many variations.

In conclusion, this thesis has made significant strides toward automating the creation of digital twins for autonomous driving simulations. While challenges remain, the work presented here lays a solid foundation for future advancements in this field.

Overview of Generative AI Tools Used

ChatGPT

GPT-4o mini and *GPT-4o* were used only to aid the writing process. The tool was used to help me improve the writing of a handful of paragraphs. The output of the tool was not used verbatim. Instead, I used it as a feedback and recommendation system. I looked at the altered output and hand-picked changes to apply to my text. The tool was not used to generate text from nothing. I always included text I had written in my own words in the prompt. Usually accompanied by prompts similar to: "Help me improve my writing, keep it similar to my own style."

List of Figures

3.1 Overview of the pipeline 15
3.2 Artifacts on a reconstruction 20
3.3 Projection bleeding 23
3.4 Examples of over-segmentation and under-segmentation 24
3.5 A reconstructed mesh with the road graph overlayed in CARLA 25
3.6 Points picked on the street in CloudCompare 27
3.7 The reconstruction region shown in RealityCapture 27
3.8 A reconstructed mesh with an extended drivable area created in Blender . 28
3.9 Manual correction addon in Blender 29

4.1 The folder structures created after running the pipeline 34
4.2 Example pipeline usage in Python 35
4.3 Example step definitions in Python 35

5.1 Example input images 46
5.2 A comparison of a captured image before and after processing. 47
5.3 The NavVis VLX mobile mapping device 48
5.4 A point cloud captured with the NavVis VLX 49
5.5 Cumberlandstraße: registrations and path 50
5.6 Jenullgasse: registrations and path 50
5.7 Class distribution of the ground truth annotations for all three datasets combined 51
5.8 Ground truth annotations 52
5.9 Results of a reconstruction using an image based dataset. 54
5.10 A scene reconstructed from images, showcasing floating artifacts. 54
5.11 Jenullgasse reconstructed from a point cloud 55
5.12 Cumberlandstraße reconstructed 56
5.13 Jenullgasse reconstructed 57
5.14 Direct comparison of mesh and texture reconstructions from both modalities 58
5.15 Quantitative results of the experiments with varying Poisson surface reconstruction depths 60
5.16 Reconstructions from a point cloud at different Poisson depths 61
5.17 Hausdorff Distnace for different face counts 63
5.18 A pole with traffic signs simplified to different face counts for both modalities 64

5.19 A reconstruction simplified to different face counts for both modalities . . 65
5.20 Comparison of different texture resolutions for both modalities 66
5.21 Close-up of a road marking, comparing different texture resolutions for both modalities 67
5.22 Close-up of text, comparing different texture resolutions for both modalities 68
5.23 Comparison of an image-based reconstruction using different images . . . 69
5.24 Segmentations produced by the pipeline 71
5.25 The effect of different k thresholds and minimum vertex counts 73
5.26 Comparison of different sampling methods 74
5.27 Examples of 2D semantic segmentations 76
5.28 Timeline of the pipeline 77

List of Tables

3.1 Reconstruction resolution at different reconstruction depths for Poisson surface reconstruction. 20

5.1 Statistics of our captured datasets 49
5.2 Benchmarks of different segmentation configurations 75
5.3 Per-class metrics of a benchmark with the best performing model 75
5.4 Benchmark of the 3D segmentation performance using different 2D segmentation models 77

Bibliography

[1] Ahmed Abdelreheem, Ivan Skorokhodov, Maks Ovsjanikov, and Peter Wonka. SATR: Zero-Shot Semantic Segmentation of 3D Shapes. In *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 15120–15133, Paris, France, October 2023. IEEE.

[2] Jibril Muhammad Adam, Weiquan Liu, Yu Zang, Muhammad Kamran Afzal, Saifullahi Aminu Bello, Abdullahi Uwaisu Muhammad, Cheng Wang, and Jonathan Li. Deep learning-based semantic segmentation of urban-scale 3D meshes in remote sensing: A survey. *International Journal of Applied Earth Observation and Geoinformation*, 121:103365, July 2023.

[3] ASAM e.V. Opendrive format specification. `https://www.asam.net/standards/detail/opendrive/`, January 2020. [Online; Accessed: 2024-07-14].

[4] N. Aspert, D. Santa-Cruz, and T. Ebrahimi. Mesh: measuring errors between surfaces using the hausdorff distance. In *Proceedings. IEEE International Conference on Multimedia and Expo*, volume 1, pages 705–708 vol.1, 2002.

[5] J. Behley, M. Garbade, A. Milioto, J. Quenzel, S. Behnke, C. Stachniss, and J. Gall. SemanticKITTI: A Dataset for Semantic Scene Understanding of LiDAR Sequences. In *Proc. of the IEEE/CVF International Conf. on Computer Vision (ICCV)*, pages 9297–9307, 2019.

[6] F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva, and G. Taubin. The ball-pivoting algorithm for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):349–359, October 1999.

[7] Alexandre Boulch, Joris Guerry, Bertrand Le Saux, and Nicolas Audebert. SnapNet: 3D point cloud semantic labeling with 2D deep segmentation networks. *Computers & Graphics*, 71:189–198, April 2018.

[8] M. Brown and D.G. Lowe. Unsupervised 3d object recognition and reconstruction in unordered datasets. In *Fifth International Conference on 3-D Digital Imaging and Modeling (3DIM'05)*, pages 56–63, 2005.

[9] R. Qi Charles, Hao Su, Mo Kaichun, and Leonidas J. Guibas. PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 77–85, Honolulu, HI, July 2017. IEEE.

[10] Bowen Cheng, Ishan Misra, Alexander G. Schwing, Alexander Kirillov, and Rohit Girdhar. Masked-attention Mask Transformer for Universal Image Segmentation. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1280–1289, New Orleans, LA, USA, June 2022. IEEE.

[11] Pranav Singh Chib and Pravendra Singh. Recent advancements in end-to-end autonomous driving using deep learning: A survey. *IEEE Transactions on Intelligent Vehicles*, 9(1):103–118, 2024.

[12] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The Cityscapes Dataset for Semantic Urban Scene Understanding. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3213–3223, Las Vegas, NV, USA, June 2016. IEEE.

[13] Angela Dai, Angel X Chang, Manolis Savva, Maciej Halber, Thomas Funkhouser, and Matthias Nießner. Scannet: Richly-annotated 3d reconstructions of indoor scenes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5828–5839, 2017.

[14] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In Sergey Levine, Vincent Vanhoucke, and Ken Goldberg, editors, *Proceedings of the 1st Annual Conference on Robot Learning*, volume 78 of *Proceedings of Machine Learning Research*, pages 1–16. PMLR, 13–15 Nov 2017.

[15] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Efficient Graph-Based Image Segmentation. *International Journal of Computer Vision*, 59(2):167–181, September 2004.

[16] Whye Kit Fong, Rohit Mohan, Juana Valeria Hurtado, Lubing Zhou, Holger Caesar, Oscar Beijbom, and Abhinav Valada. Panoptic nuscenes: A large-scale benchmark for lidar panoptic segmentation and tracking. *IEEE Robotics and Automation Letters*, 7(2):3795–3802, 2022.

[17] Lin Gao, Yu Liu, Xi Chen, Yuxiang Liu, Shen Yan, and Maojun Zhang. Cus3d: A new comprehensive urban-scale semantic-segmentation 3d benchmark dataset. *Remote Sensing*, 16(6):1079, 2024.

[18] Weixiao Gao, Liangliang Nan, Bas Boom, and Hugo Ledoux. SUM: A benchmark dataset of Semantic Urban Meshes. *ISPRS Journal of Photogrammetry and Remote Sensing*, 179:108–120, September 2021.

[19] Weixiao Gao, Liangliang Nan, Bas Boom, and Hugo Ledoux. PSSNet: Planarity-sensible Semantic Segmentation of large-scale urban meshes. *ISPRS Journal of Photogrammetry and Remote Sensing*, 196:32–44, February 2023.

[20] Michael Garland and Paul S Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 209–216, 1997.

[21] Kyle Genova, Xiaoqi Yin, Abhijit Kundu, Caroline Pantofaru, Forrester Cole, Avneesh Sud, Brian Brewington, Brian Shucker, and Thomas Funkhouser. Learning 3d semantic segmentation with only 2d image supervision. In *2021 International Conference on 3D Vision (3DV)*, pages 361–372, 2021.

[22] Carlos Gómez-Huélamo, Javier Del Egido, Luis M. Bergasa, Rafael Barea, Elena López-Guillén, Felipe Arango, Javier Araluce, and Joaquín López. Train here, drive there: Simulating real-world use cases with fully-autonomous driving architecture in carla simulator. In Luis M. Bergasa, Manuel Ocaña, Rafael Barea, Elena López-Guillén, and Pedro Revenga, editors, *Advances in Physical Agents II*, pages 44–59, Cham, 2021. Springer International Publishing.

[23] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A Survey of Deep Learning Techniques for Autonomous Driving. *Journal of Field Robotics*, 37(3):362–386, April 2020.

[24] Carsten Griwodz, Simone Gasparini, Lilian Calvet, Pierre Gurdjos, Fabien Castan, Benoit Maujean, Gregoire De Lillo, and Yann Lanthony. AliceVision Meshroom: An open-source 3D reconstruction pipeline. In *Proceedings of the 12th ACM Multimedia Systems Conference*, pages 241–247, Istanbul Turkey, June 2021. ACM.

[25] Carsten Griwodz, Simone Gasparini, Lilian Calvet, Pierre Gurdjos, Fabien Castan, Benoit Maujean, Gregoire De Lillo, and Yann Lanthony. Alicevision Meshroom: An open-source 3D reconstruction pipeline. In *Proc. 12th ACM Multimed. Syst. Conf. - MMSys '21*. ACM Press, 2021.

[26] Grégoire Grzeczkowicz and Bruno Vallet. Semantic Segmentation of Urban Textured Meshes Through Point Sampling. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, V-2-2022:177–184, May 2022.

[27] Abhishek Gupta, Alagan Anpalagan, Ling Guan, and Ahmed Shaharyar Khwaja. Deep learning for object detection and scene perception in self-driving cars: Survey, challenges, and open issues. *Array*, 10:100057, 2021.

[28] Rodrigo Gutiérrez-Moreno, Rafael Barea, Elena López-Guillén, Javier Araluce, and Luis M. Bergasa. Reinforcement learning-based autonomous driving at intersections in carla simulator. *Sensors*, 22(21), 2022.

[29] Rana Hanocka, Amir Hertz, Noa Fish, Raja Giryes, Shachar Fleishman, and Daniel Cohen-Or. MeshCNN: A Network with an Edge. *ACM Transactions on Graphics*, 38(4):1–12, August 2019.

[30] Qingyong Hu, Bo Yang, Sheikh Khalid, Wen Xiao, Niki Trigoni, and Andrew Markham. Towards semantic segmentation of urban-scale 3d point clouds: A dataset, benchmarks and challenges. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4977–4987, June 2021.

[31] Xuemin Hu, Shen Li, Tingyu Huang, Bo Tang, Rouxing Huai, and Long Chen. How simulation helps autonomous driving: A survey of sim2real, digital twins, and parallel intelligence. *IEEE Transactions on Intelligent Vehicles*, 9(1):593–612, 2024.

[32] L Huang, Y Yuan, J Guo, C Zhang, X Chen, and J Wang. Interlaced sparse self-attention for semantic segmentation. arxiv 2019. *arXiv preprint arXiv:1907.12273*, 2019.

[33] Daniel Huber. The astm e57 file format for 3d imaging data exchange. In *Proceedings of SPIE Electronics Imaging Science and Technology Conference (IS&T), 3D Imaging Metrology*, volume 7864, January 2011.

[34] Michael R James and Stuart Robson. Straightforward reconstruction of 3d surfaces and topography with a camera: Accuracy and geoscience application. *Journal of Geophysical Research: Earth Surface*, 117(F3), 2012.

[35] Maximilian Jaritz, Jiayuan Gu, and Hao Su. Multi-View PointNet for 3D Scene Understanding. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pages 3995–4003, Seoul, Korea (South), October 2019. IEEE.

[36] Andrej Karpathy, Stephen Miller, and Li Fei-Fei. Object discovery in 3D scenes via shape analysis. In *2013 IEEE International Conference on Robotics and Automation*, pages 2088–2095, Karlsruhe, Germany, May 2013. IEEE.

[37] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In *Proceedings of the fourth Eurographics symposium on Geometry processing*, volume 7, 2006.

[38] Michael Kazhdan and Hugues Hoppe. Screened poisson surface reconstruction. *ACM Transactions on Graphics*, 32(3):1–13, June 2013.

[39] Michael Kölle, Dominik Laupheimer, Stefan Schmohl, Norbert Haala, Franz Rottensteiner, Jan Dirk Wegner, and Hugo Ledoux. The Hessigheim 3D (H3D) Benchmark on Semantic Segmentation of High-Resolution 3D Point Clouds and Textured Meshes from UAV LiDAR and Multi-View-Stereo. *ISPRS Open Journal of Photogrammetry and Remote Sensing*, 1:100001, October 2021.

[40] Abhijit Kundu, Xiaoqi Yin, Alireza Fathi, David Ross, Brian Brewington, Thomas Funkhouser, and Caroline Pantofaru. Virtual multi-view fusion for 3d semantic segmentation. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision – ECCV 2020*, pages 518–535, Cham, 2020. Springer International Publishing.

[41] Felix Järemo Lawin, Martin Danelljan, Patrik Tosteberg, Goutam Bhat, Fahad Shahbaz Khan, and Michael Felsberg. Deep projective 3d semantic segmentation. In Michael Felsberg, Anders Heyden, and Norbert Krüger, editors, *Computer Analysis of Images and Patterns*, pages 95–107, Cham, 2017. Springer International Publishing.

[42] Liqiang Lin, Yilin Liu, Yue Hu, Xingguang Yan, Ke Xie, and Hui Huang. Capturing, Reconstructing, and Simulating: The UrbanScene3D Dataset. In Shai Avidan, Gabriel Brostow, Moustapha Cissé, Giovanni Maria Farinella, and Tal Hassner, editors, *Computer Vision – ECCV 2022*, volume 13668, pages 93–109. Springer Nature Switzerland, Cham, 2022.

[43] Krista Merry and Pete Bettinger. Smartphone gps accuracy study in an urban environment. *PLOS ONE*, 14(7):1–19, 07 2019.

[44] Adam R Mosbrucker, Jon J Major, Kurt R Spicer, and John Pitlick. Camera system considerations for geomorphic applications of sfm photogrammetry. *Earth Surface Processes and Landforms*, 42(6):969–986, 2017.

[45] D.R. Niranjan, B C VinayKarthik, and Mohana. Deep learning based object detection model for autonomous driving research using carla simulator. In *2021 2nd International Conference on Smart Electronics and Communication (ICOSEC)*, pages 1251–1258, 2021.

[46] Blazej Osinski, Piotr Milos, Adam Jakubowski, Pawel Ziecina, Michal Martyniak, Christopher Galias, Antonia Breuer, Silviu Homoceanu, and Henryk Michalewski. CARLA real traffic scenarios - novel training ground and benchmark for autonomous driving. *CoRR*, abs/2012.11329, 2020.

[47] Błażej Osiński, Adam Jakubowski, Paweł Zięcina, Piotr Miłoś, Christopher Galias, Silviu Homoceanu, and Henryk Michalewski. Simulation-based reinforcement learning for real-world autonomous driving. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6411–6418, 2020.

[48] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[49] Capturing Reality. How to take photographs. https://rchelp.capturingreality.com/en-US/tutorials/takingpictures.htm. [Accessed: 2024-10-03].

[50] Mohammad Rouhani, Florent Lafarge, and Pierre Alliez. Semantic segmentation of 3D textured meshes for urban scene analysis. *ISPRS Journal of Photogrammetry and Remote Sensing*, 123:124–139, January 2017.

[51] Rajvi Shah, Aditya Deshpande, and P.J. Narayanan. Multistage sfm: Revisiting incremental structure from motion. In *2014 2nd International Conference on 3D Vision*, volume 1, pages 417–424, 2014.

[52] Magistrat Wien Magistratsabteilung 41 Stadtvermessung. Kappazunder dataset, 2020. data retrieved from Geodatenviewer der Stadtvermessung Wien, https://www.wien.gv.at/geodatenviewer/portal/wien/.

[53] Abubakar Sulaiman Gezawa, Qicong Wang, Haruna Chiroma, and Yunqi Lei. A Deep Learning Approach to Mesh Segmentation. *Computer Modeling in Engineering & Sciences*, 135(2):1745–1763, 2023.

[54] Ke Sun, Bin Xiao, Dong Liu, and Jingdong Wang. Deep high-resolution representation learning for human pose estimation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[55] Hugues Thomas, Charles R. Qi, Jean-Emmanuel Deschaud, Beatriz Marcotegui, Francois Goulette, and Leonidas J. Guibas. Kpconv: Flexible and deformable convolution for point clouds. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[56] Enze Xie, Wenhai Wang, Zhiding Yu, Anima Anandkumar, Jose M. Alvarez, and Ping Luo. Segformer: Simple and efficient design for semantic segmentation with transformers. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 12077–12090. Curran Associates, Inc., 2021.

[57] Jiacong Xu, Zixiang Xiong, and Shankar P. Bhattacharyya. PIDNet: A Real-time Semantic Segmentation Network Inspired by PID Controllers. In *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 19529–19539, Vancouver, BC, Canada, June 2023. IEEE.

[58] Yetao Yang, Rongkui Tang, Mengjiao Xia, and Chen Zhang. A surface graph based deep learning framework for large-scale urban mesh semantic segmentation. *International Journal of Applied Earth Observation and Geoinformation*, 119:103322, May 2023.

[59] Puyuan Yi, Shengkun Tang, and Jian Yao. Ddr-net: Learning multi-stage multi-view stereo with dynamic depth range. *arXiv preprint arXiv:2103.14275*, 2021.

[60] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. A Survey of Autonomous Driving: Common Practices and Emerging Technologies. *IEEE Access*, 8:58443–58469, 2020.

[61] Guangyun Zhang and Rongting Zhang. MeshNet-SP: A Semantic Urban 3D Mesh Segmentation Network with Sparse Prior. *Remote Sensing*, 15(22):5324, November 2023.

[62] Rongting Zhang, Guangyun Zhang, Jihao Yin, Xiuping Jia, and Ajmal Mian. Mesh-Based DGCNN: Semantic Segmentation of Textured 3-D Urban Scenes. *IEEE Transactions on Geoscience and Remote Sensing*, 61:1–12, 2023.

[63] Hengshuang Zhao, Li Jiang, Jiaya Jia, Philip H.S. Torr, and Vladlen Koltun. Point transformer. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 16259–16268, October 2021.

[64] Jingchun Zhou, Mingliang Hao, Dehuan Zhang, Peiyu Zou, and Weishi Zhang. Fusion PSPnet Image Segmentation Based Method for Multi-Focus Image Fusion. *IEEE Photonics Journal*, 11(6):1–12, December 2019.