Eskil Torland
Trym Jørgensen

Automated creation of digital twins

Master's thesis in Computer Science
Supervisor: Gabriel Kiss
Co-supervisor: Frank Lindseth, Mamoona Birkhez Shami
June 2023

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Eskil Torland
Trym Jørgensen

Automated creation of digital twins

Master's thesis in Computer Science
Supervisor: Gabriel Kiss
Co-supervisor: Frank Lindseth, Mamoona Birkhez Shami
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Automated creation of digital twins

TDT4900 - Computer Science, Master Thesis

Eskil Torland
Trym Jørgensen

Supervisor: Gabriel Kiss

Co-supervisor: Frank Lindseth
Mamoona Birkhez Shami

June 14, 2023

Abstract

A digital twin is a virtual representation of a physical asset or object that is linked to its real-world counterpart through a data connection. In recent years, there has been a rapid increase in interest and investments, mainly due to the arrival of Industry 4.0. This emerging technology offers alternative solutions to conventional challenges, which hold significant relevance to modern businesses.

This master thesis revolves around the automatic generation of digital twins and the seamless integration of real-time data to facilitate dynamic updates. To accomplish these objectives Python scripts and Omniverse Extensions were developed, which enabled the construction of digital twins using diverse data sources. In addition to this, major research into different data sources and techniques has been carried out, resulting in a deeper understanding of the field.

The results of this thesis show that the automatic generation of digital twins is highly possible, which surpasses the capabilities of conventional software currently available. This advancement is manifested through notable improvement in the quality of the textures and terrains created, as well as giving a more flexible model that can be further developed with other sources and for different applications. Furthermore, the successful incorporation of real-time data was achieved by collecting sensor data and data from other relevant sources, which subsequently were sent to a database before getting integrated into the digital twin scene.

Sammendrag

En digital tvilling er en virtuell representasjon av en fysisk eiendel eller et objekt som er knyttet til sin virkelige motpart gjennom en dataforbindelse. De siste årene har det vært en rask økning i interesse og investeringer i denne teknologien, hovedsakelig på grunn av ankomsten av Industri 4.0. Denne nye teknologien tilbyr alternative løsninger på ordinære utfordringer, som har betydelig relevans for moderne bedrifter.

Denne masteroppgaven handler om automatisk generering av digitale tvillinger og problemfri integrering av sanntids datastrømmer for å muliggjøre dynamiske oppdateringer. For å oppnå disse målene ble Python-skript og Omniverse Extensions utviklet, noe som gjorde det mulig å konstruere digitale tvillinger ved hjelp av ulike datakilder. I tillegg til dette har det blitt gjennomført omfattende forskning på flere datakilder og teknikker, noe som har resulterer i en dypere forståelse av feltet.

Resultatene av denne master oppgaven viser at automatisk generering av digitale tvillinger er svært mulig, som også overgår de mer tradisjonelle programvarene som er tilgjengelig for øyeblikket. Denne fremgangen viser seg gjennom bemerkelsesverdig forbedring i kvaliteten på teksturene og terrengene som er laget, samt å gi en mer fleksibel modell som kan videreutvikles med andre kilder og for andre brukstilfeller. Videre ble integrering av sanntidsdata oppnådd ved å samle sensordata og data fra andre relevante kilder. Disse ble deretter sendt til en database før de ble integrert i den digitale tvillingscenen.

Preface

This thesis was written as part of the Master's thesis at the Norwegian University of Science and Technology (NTNU), at the Department of Computer Science (IDI).

We would like to extend a big thank you to our supervisors Gabriel Kiss and Frank Lindseth, who have given us good feedback and guidance throughout the project. They have also given us access to resources that have improved the result and enabled us to create a product that we are proud to present.

In addition to our supervisors, we would like to thank Mamoona Birkhez Shami for helping us with some of the technical aspects of our project that have overlapped with her doctorate.

Lastly, we would like to thank Alf Andreas Høiseth for providing us access to highly detailed textures through GeoNorge whenever needed.

Contents

Figures

List of Listings

Acronyms

API Application Programming Interface. 22, 23, 26, 27, 30, 33, 48, 63, 70, 72

AR Augmented Reality. 11

CRS Coordinate Reference System. 13

CSV Comma-Seperated Values. 36, 56

DNN Deep neural network. 12, 59, 70

DT Digital twin. 20

GEO Geostationary Earth Orbit. 16

GIS Geographic information system. 3

GLTF GL Transmission Format. 11, 56, 68, 69

GNSS Global Navigation Satellite System. xiii, xiv, 15, 16, 57, 61, 62

GPS Global Positioning System. 14, 16

GUI Graphical User Interface. 50

IBM International Business Machines. 1

IDE Integrated Development Environment. 11

IDI Department of Computer Science. vii

LEO Low Earth Orbit. 16

MEO Medium Earth Orbit. 16

NASA North American Space Association. 1

NIBIO Norwegian Institute of Bioeconomy Research. 20

NMA Norwegian Mapping Authority. 20

NRF National Research Foundation of Singapore. 7

NTNU Norwegian University of Science and Technology. vii, 17, 35, 53

NVDB Nasjonal vegdatabank. 3, 22, 23, 27, 30, 33, 37, 49, 50, 56–59, 67, 70, 72

OBJ Object. 69

OSM Open Steet Map. 23, 46, 48, 63–65

ROS Robot Operating System. 35, 36, 56, 61

SLA Singapore Land Authority. 7

UTM Universal Transverse Mercator. 14, 15, 26, 48, 55

VR Virtual Reality. 11, 16, 55, 56, 69

VTK Visualization ToolKit. 13

WGS84 World Geodetic System 1984. 14, 26, 48

XR Extended Reality. 11

Glossary

CARLA description . 10, 72

GeoNorge National website for any map or geolocated data in Norway. vii, 20, 63

Industry 4.0 The fourth industrial revolution, manufacturers implementing cloud computing, analytics, and AI in production. iii, 5

LIDAR LIDAR, which stands for light detection and ranging, is a method to measure distances by sending light rays out which are then bounced back. Using data retrieved from these rays a distance can be measured. 6, 9

OpenDrive OpenDrive is a format for describing a road network's logic. 10, 72

Ortophoto Aerial images taken from planes, drones, or satellites that has been georeferenced. 3, 20, 58, 63

USD Universal scene description, most commonly reffered to as USD, is Pixar's open-source file format used for 3D objects and scenes.. 10, 11, 17, 33, 42, 50, 54, 56, 68, 69

Chapter 1

Introduction

Digital twins have become essential to many projects such as autonomous vehicles, city planning, and simulations. According to IBM, *a digital twin is a virtual representation of an object or system that spans its lifecycle, is updated from real-time data, and uses simulation, machine learning, and reasoning to help decision making [1].* The concept of digital twins dates back at least 50 years to the launch of Apollo 13. When one of the oxygen tanks ruptured during the flight, NASA used telecommunications to connect the physical object to their simulators to reflect the real-world asset. NASA managed to repair the damage using these simulators, and the mission was deemed a successful failure. However, a digital twin should replicate the physical behaviors of an object in real-time data. In contrast, NASA had 15 different simulators for separate tasks that could be tweaked but required a good deal of manual work [2]. A digital twin also consists of several different parts and models that work together, much like the simulators of NASA.

1.1 Motivation

The process of creating a digital twin can be an extremely time-consuming task that involves constructing multiple 3D models of different parts of a physical entity and assembling these into a single virtual representation. In addition to the construction process, extensive data acquisition, generation and preprocessing are necessary to build a digital twin. The current software and tools used to create a digital twin either give a lackluster result or require significant manual work. The motivation for this project was to delve into the intricacies of the digital twin technology. We wished to not only comprehend its complexities but also to contribute meaningfully to this burgeoning field. To achieve this, we have focused on designing a unique, programmatic method to simplify the creation process of digital twins. Our aspiration is not confined merely to understanding or contribution; we are also striving to broaden the accessibility of this technology. By devising an efficient and streamlined creation pipeline, we hope to facilitate greater usage of digital twins across various sectors.

1.2 Goal and Research Questions

This master thesis aims to contribute to the understanding and creation process of digital twins while advancing the generation of digital twins programmatically. Another goal for this project is to add and test dynamic updates of objects within a digital twin with a stream of real-time data from a moving agent or other sources. To guide the study, the following research questions have been formulated:

RQ.1 What parts of a digital twin can be automatically generated?

RQ.2 What parts of the digital twin are most important to give it a realistic feel?

RQ.3 Can real-time data be streamed from a moving agent for dynamic updates in the digital twin?

RQ.4 Can digital twins be used to store and visualize meta-data received from a smart sensor?

1.3 Research Method

We have mainly adopted design and creation research methodology for this research but combined design science research methods where required. Design and creation research methods are also known as "research through design". This approach treats designing or creating an artifact as a form of research. Here, the design process is seen as a method for generating knowledge. Following this method, we created prototypes, deployed them in a real-world context, and studied their use to learn more about the problem space, the design solution, and interactions. On the other hand, in design science research, the researcher often goes through a process that includes identifying a problem, designing and creating an artifact (like a software application or a process), and evaluating that artifact's effectiveness in addressing the issue. The knowledge generated through this process can then be used to guide future design efforts. We have produced a rigorous, scientifically sound contribution to theoretical knowledge following this research method. Our research method was iterative and reflexive, involving cycles of design, implementation, testing, and refinement.

1.4 Contribution

This thesis has contributed to and experimented with the generation of digital twins programmatically. The technology is based on Python scripts that take in various data sources to create a dynamic and fully functional digital twin. The software is also built in a way that makes it easy to add other sources and build upon the foundation created in this project. By creating this foundation, we also open up the possibility of introducing other technologies and opportunities to use

digital twins. The model created delivered great results regarding the details of the mesh created and the quality of the textures added.

The major contributions of this project are as follows:

1. Simplifying the Digital Twin Pipeline so future work can be built upon it
2. Automating the process of building a 3D digital twin given the GIS coordinates of a place
3. Adding higher resolution Ortophoto in Omniverse
4. Adding higher quality elevation data to terrain
5. Real-time communication with the digital twin and a real world agent
6. Case study for real-time data stream: Car position in relation to road lines
7. Case study for Digital Twin: Validation of Traffic signs' position and data in NVDB
8. Case study for Digital Twin: Road Surface Monitoring
9. Case Study for Digital Twin: Key elements to create a realistic and immersive digital twin

1.5 Thesis Structure

The paper is divided into these chapters:

1. **Introduction**
   The introduction gives a short introduction to the overall subject of the thesis and covers the research method, motivation, and previous studies in the field.
2. **Background**
   The background chapter gives insight into potential use cases for digital twins, software, and methods currently used for creating digital twins and 3D objects.
3. **Method**
   The methods chapter goes into deeper details about which methods were developed and utilized to get the final results and how to reproduce these results. This chapter is divided into two major parts, with the first part focusing more on our programmatic approach to digital twin creation. The second part focuses more on the process of integrating live data streams into the digital twin.
4. **Results**
   The results chapter showcases our findings related to the research questions. In addition to this chapter, several Youtube videos will present the results from the programmatically generated digital twin and the real-time data being streamed to it.
5. **Discussion**
   The discussion chapter relates our results to our research questions and delves into an in-depth analysis of some of the challenges encountered.
6. **Conclusion**

The conclusion chapter summarizes our project and presents the main findings and looks at possible use cases of a digital twin and future work.

Chapter 2

Background

This chapter will give an introduction to core concepts used in digital twin creation and give examples of the current and potential uses of digital twins.

2.1 Digital twins

The rapid improvement of digital technologies in recent years has led organizations and companies to optimize their operations and drive innovations. One of the leading technological concepts that have gained significant attention because of this is digital twins. A digital twin is a virtual representation of a physical asset or object that is linked to its real-world counterpart through a data connection.

The concept idea of digital twins, as mentioned in chapter 1, has roots that can be traced back to the Apollo 13 mission. However, the term digital twin was first coined in 2002 when Dr. Michael Grieves applied the concept of digital twins in manufacturing [1]. Since then, the concept of digital twins has evolved significantly and the fundamental idea of a digital twin today is to create a comprehensive and dynamic model that mimics the behavior, characteristics, and functionalities of a physical asset. This virtual representation can either replicate the physical representation directly or incorporate it with some modifications to individual factors. Digital twins are often used in simulations and planning to assess potential adjustments and predict their impact. Digital twins can enhance cost and time efficiency by introducing new ways of testing and planning [3].

2.2 Potential use of Digital twins

As mentioned, digital twins are a virtual representation of an object or a place that can be used for simulations, optimizations, and decision-making. The potential use of these virtual representations has significantly increased with the arrival of Industry 4.0. Different industries, such as manufacturing, healthcare, and urban planning use this for training, maintenance, and planning.

2.2.1 Stadt Zurich - Digital twin

In recent years, population growth in Zurich has posed several new challenges such as population density, and it could lead to environmental issues, which are also seen in other places such as Singapore. To handle this issue the city of Zurich has tried to introduce a more adaptable solution with the use of digital twins. The terrain of the digital twin is created using LIDAR images from 2014. These images have a resolution of 50cm and are manually adjusted in some places to include additional information. The building models are based on building floor plans and are divided into sub-parts to ensure that each floor is accurately represented by a prism with its intended height and shape [4].

An important aspect of the Zurich digital twin lies in its emphasis on facilitating public access to the data. This can enable private parties and partners to create applications to enhance the use of the digital twin [5]. Several applications have been made from this public database to address some of the issues with density and climate change as previously stated. One of these applications can be seen in Figure 2.1, where an open-source application initially developed for New York City was adapted to include building mass from Zurich. This application can be used to improve urban development and living spaces [6].

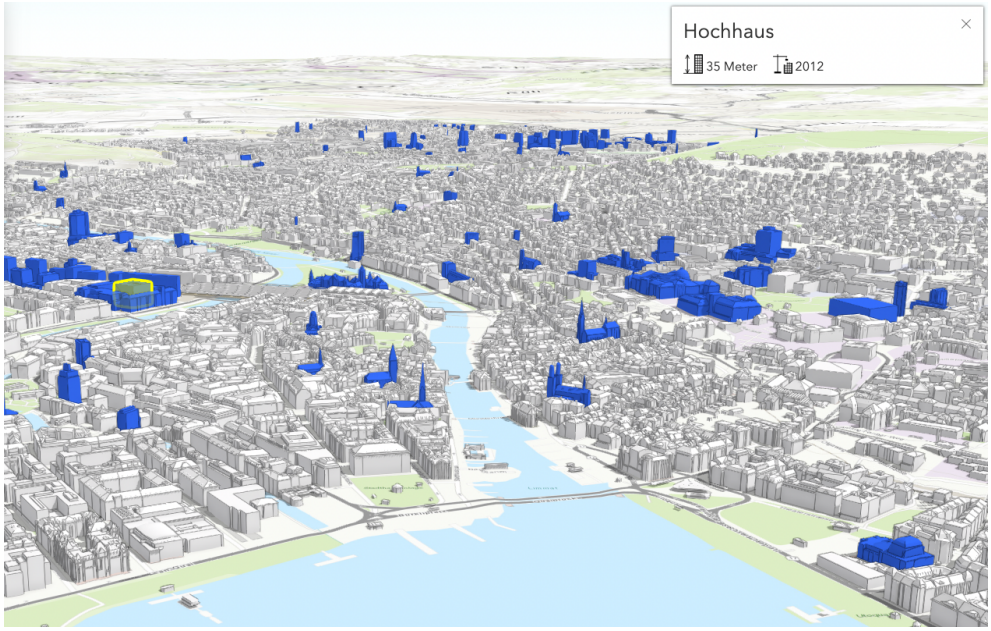


Figure 2.1: High rise building planner of Zurich
Source: https://link.springer.com/article/10.1007/s41064-020-00092-2/figures/10

WingtraOne

In 2021, WingtraOne, a Zurich-based company, tried creating a digital twin using data collected from a drone equipped with a Sony A6100 mapping camera. The drone required 6 hours of flight time to collect data from the whole city, which resulted in a digital twin with a resolution of 3cm per pixel for the textures [7] [8]. The benefit of using this technology is high-resolution information, and the drawback is the large amount of data needed and the cost of such hardware-heavy technology.

2.2.2 Virtual Singapore

Like Stadt Zurich's Digital Twin, Virtual Singapore is a dynamic 3D city model. The project, seen in Figure 2.2, is a collaboration between the National Research Foundation of Singapore (NRF) and the Singapore Land Authority (SLA). NRF will be taking the lead in developing the digital twin, while the SLA will provide topographical data and other relevant information useful for the project. Other public parties will also contribute to the process in ways such as modeling and simulations [9].

Virtual Singapore is primarily intended to tackle various problems that have surfaced in recent years. The most significant of these problems is climate change. The digital twin is designed to empower stakeholders from diverse sectors. They can create prediction tools and simulations using this technology, which can then be used to devise efficient strategies for different scenarios. Another use for the digital twin is to tap into new technologies, creating new job opportunities for the people of Singapore and generating value [10].

Figure 2.2: Virtual Singapore's digital twin
Source: `https://www.geospatialworld.net/prime/case-study/national-mapping/virtual-singapore-building-a-3d-empowered-smart-nation/`

2.2.3 Trondheim city model

The Trondheim city model is created by Trondheim municipality in an effort to enhance knowledge of how development areas or proposed individual projects will impact the neighborhood, district, or the entire city of Trondheim. The project utilizes a combination of software tools, including the Unity game development program and 3d modeling software 3ds Max, to generate an interactive representation of Trondheim [11]. In Figure 2.3 you can see the Unity model created for this project.

Figure 2.3: Trondheim city model presented in Unity
Source: `https://youtu.be/03ikDhb1O9Q`

The city model created serves as a great resource for different stakeholders involved with the city. Property designers can leverage this model to assess the visual implications of their projects, enabling them to make more informed decisions. Furthermore, by making the model accessible to the public, it serves as an educational and information tool, offering valuable insights to the general populace. However, it is worth noting that there is currently a lack of user-friendly interfaces for viewing the models created. Consequently, prior knowledge of 3D tools is needed to examine these models. Interested individuals can access the models from the following link: `https://kart.trondheim.kommune.no/3d_bymodell/`, where both the 3ds Max and Unity models are available.

2.2.4 Oppdrag Mjøsa

Oppdrag Mjøsa is another ongoing project aimed at exploring the application of digital twin technology. The project was initiated following the discovery of a new shipwreck in Mjøsa during the autumn of 2022. This shipwreck, estimated to be in

between 300-700 years old and measures approximately 10 meters in length and 2.4 meters in width, is believed to be the first discovery of many to come. Given the implications of climate change and changing rainfall patterns, it has become important to address the restoration of these cultural heritage sites and manage the lake in a sustainable manner [12].

Work has therefore been initiated to develop a digital twin of Mjøsa that will receive and make available data from various sensors. The goal is to be able to understand the lake's progressive development and proactively anticipate changes earlier. Initially, researchers will focus on developing a detailed terrain model of Mjøsa's lake bed. Satellite data, historical data, and other measurements and research data will eventually be incorporated into the digital twin [13].

2.3 3D object theory

3d modeling encompasses the process of creating a 3d representation of an object from a set of points that can be viewed or interacted with using specialized 3d software. This technology is used in many different fields of study and business.

The first step of 3d modeling involves the acquiring of data points, the process varies based on what the user is trying to model. The biggest difference appears when comparing the process of reconstructing an object versus creating an entirely new model. In the case of creating a new model, the user needs to manually place points and sculpt the object by hand. However, when it comes to reconstruction, the user can get their points from many different sources. LIDAR is a common tool used to gather point data of real-world geometry. The necessary equipment can be placed on moving vehicles on the ground or in the air, gathering millions of points over large areas.

When working with a set of vertices in 3D modeling, users have numerous options available to them. However, no single method can be universally applied to create a mesh from these vertices. The field of surface reconstruction and mesh optimization offers various algorithms and methods that cater to different requirements. Each method involves adjusting different variables to get the most precise mesh for a given purpose. To streamline the process of creating 3D models, several libraries exist that simplify the implementation details of 3D modeling.

2.4 Software

The field of digital twin creation involves the integration of large amounts of data and the use of advanced software. This section will provide an overview of software that is currently being used in the field of digital twin creation. By understanding the current software landscape, researchers can make informed decisions about which software and methods to utilize for their specific use case and process.

2.4.1 ArcGis Pro

ArcGIS Pro is a geographic information system application that allows the user to visualize, edit, and share geographic data. ArcGIS Pro supports several different data sources and formats which make it a versatile tool for both professionals and passionate amateurs[14].

2.4.2 CityEngine

CityEngine is a 3D modeling software developed by Esri, which allows the user to create interactive city scenes. These scenes are generated using real-world geographical information that is gathered from several different sources including Esri and OpenStreetMap. Using these interactive scenes, users can easily modify and manipulate a variety of elements within the scene such as buildings, roads, and other urban features [15]. CityEngine is an efficient software that quickly gives the user access to the models needed. CityEngine also offers exportation of this model to several other platforms such as Unreal Engine and Nvidia Omniverse. Despite its ease of use, CityEngine also comes with some limitations, notably the lack of details in the model generated. While having procedurally generated textures for buildings that gives the scene a realistic feel, the low resolution of the ground texture negatively impacts the overall outlook.

2.4.3 Roadrunner

Roadrunner is a 3d scene editor tool developed by Mathworks that allows for simulating and testing automated driving systems. It offers the ability to create, import, and export OpenDrive road networks, which can be used in other applications such as CARLA description and Nvidia Drivesim. In addition, Roadrunner provides access to a 3D asset library, allowing users to easily utilize large amounts of objects to create realistic scenes [16]. While Roadrunner gives realistic roads, the drawback of using this is the amount of manual work needed for creating accurate road networks.

2.4.4 Nvidia Omniverse

Nvidia Omniverse is 3D design and simulation software that revolutionizes real-time collaboration for teams. Nvidia Omniverse can seamlessly connect various 3D tools like Maya and Blender with Nvidia's simulation technologies, such as PhysX 5 and RTX-accelerated rendering, through flexible bi- or uni-directional connectors [17] [18]. One of Omniverse's strengths is its utilization of the Universal scene description (USD) file format. This makes the process of combining multiple assets from different sources into a single model and scene easier. Omniverse is designed to face challenges from multiple different industries such as film production and engineering and has therefore developed purpose-specific apps, with notable ones

being Code, Create, CreateXR, and Nucleus. These apps serve specific functions to allow collaboration, creation of scenes, and simulations.

Omniverse Code

Omniverse code is an Integrated Development Environment (IDE) designed for developers to create extensions and microservices for scenes and objects within Omniverse. It provides support for Python and C scripts, along with various Omniverse packages, enabling developers to seamlessly integrate data sources and additional functionalities into their scenes or worlds. There are two primary ways of integrating data into Omniverse. The first way is by creating an Omniverse Code extension, which seamlessly integrates with the Omniverse Code app and modifies its behavior. This enables users to design custom modules that expand the capabilities of Omniverse, changing it to fit their specific requirements. The second method involves utilizing Omniverse Code Sample as a starting point to create a connector. By doing this you create a bridge program that can interact with your Omniverse scenes the same way you can in Code, but from an external context. This allows for data transfer between any external application to the Omniverse environment. Both methods offer powerful means to extend the capabilities of Omniverse and enable seamless integration with external data sources.

Omniverse Create / CreateXR

Omniverse Create is an application created to accelerate scene composition and world-building. Its purpose is to accelerate the process of simulating and rendering large scenes with several different models and objects. The application supports multiple file formats such as obj, GL Transmission Format (GLTF), and USD, allowing the user to work with a wide range of assets [19]. Omniverse Create offers a wide range of different features and applications to enhance different tasks. For instance, it utilizes ray tracing which can give a scene more realistic reflections and lighting. Create also support animations and different simulators, such as for sun study or physics.

Omniverse CreateXR provides users with the capability to visualize their USD scenes using Virtual Reality (VR) and Augmented Reality (AR) technologies. This platform allows for an immersive exploration of models at a human scale. CreateXR is powered by NVIDIA RTX technology which enables fully ray-traced VR scenes, resulting in enhanced visual effects such as accurate reflections, shadows, and lights [20]. Despite CreateXR being in early beta, Omniverse claims that any Create scene can be seamlessly integrated with XR.

Omniverse Nucleus

Omniverse Nucleus is the database and collaboration tool developed by Omniverse. By establishing a Nucleus infrastructure, teams are able to connect to the database, which can facilitate simultaneous collaboration on shared assets [21].

This enables teams from different geographical locations to engage in parallel work and real-time collaboration, which can result in improved productivity and better results [22].

Nvidia DriveSim

Nvidia DriveSim is a simulation platform developed by NVIDIA. DriveSim is specifically designed for testing and validating autonomous driving systems. It provides a realistic virtual environment where autonomous vehicles can be simulated and trained. Drive Sim provides several tools to the user for generating datasets that can be used to train Deep neural network (DNN) for autonomous vehicle perception. Some of these tools can be seen in Figure 2.4. DriveSim leverages high-performance computing capabilities, including NVIDIA GPUs, to create detailed and immersive virtual worlds. These virtual environments replicate real-world scenarios, such as city streets, highways, and various weather conditions. The simulated environments also include dynamic objects like pedestrians, other vehicles, and traffic signals, allowing developers to test their autonomous driving algorithms and systems in a wide range of situations.

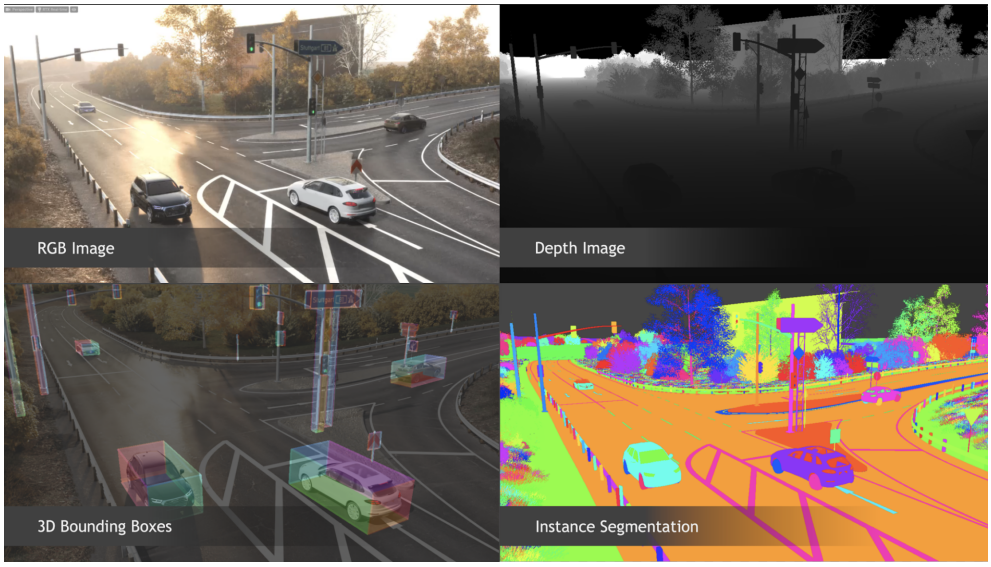


Figure 2.4: Different Nvidia Drive Sim tools
Source: `https://blogs.nvidia.com/blog/2021/04/12/nvidia-drive-sim-omniverse-early-access/`

2.4.5 Visualization Toolkit

Visualization ToolKit (VTK) is an open-source software for 3d graphics and modeling. Originally it was created as accompanying software for a textbook in 1993. Over the years the software grew and became popular with researchers and engineers alike, and is used in scientific research, medical imaging, geospatial analysis, and more.

2.4.6 Pyvista

PyVista is an open-source Python library for 3D visualization and analysis. It provides a simple, intuitive interface to VTK. PyVista is essentially a Pythonic wrapper around VTK that makes 3D visualization and analysis more accessible and straightforward. PyVista supports a wide range of 3D visualization features, including volume rendering, surface shading, mesh plotting, and more. It also supports a variety of data formats, including structured and unstructured grids, polygonal data, and point clouds.

2.5 Geomatics

2.5.1 Coordinate refrence systems

When creating digital twins of real-life locations it is important to have an accurate representation of the model in space. This is done by using a Coordinate Reference System (CRS). A geographic coordinate system is based on a spheroid that approximates the shape of the Earth and uses angular units(i.e. degrees) to reference a point on Earth. Since Earth is not a perfectly smooth sphere there are many different geographic coordinate systems for different parts of the world. These systems use different datums, which define what spheroid model is used to represent the earth, as well as deciding where the center is positioned [23].

Another type of CRS is projected coordinate systems. These systems use different mathematical transformations to project a spherical earth onto a flat plane, as seen in Figure 2.6. These systems use a geographic coordinate system to reference the original earth model being projected and use linear units to represent points in space(i.e. meters).

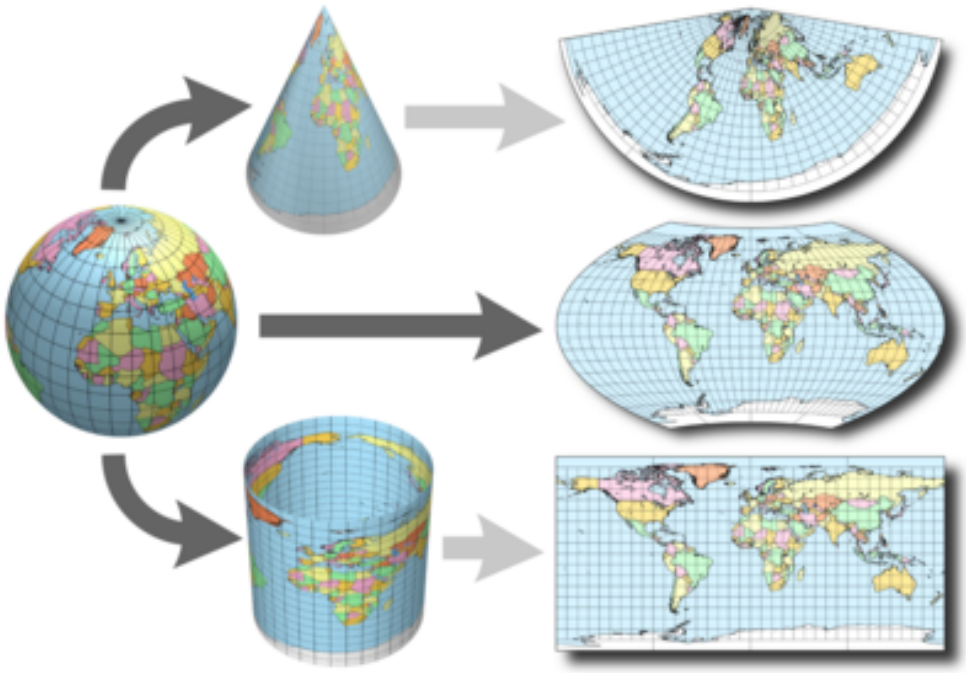


Figure 2.5: Projecting a geographical representation to a flat plane
Source: `https://www.earthdatascience.org/courses/use-data-open-source-python/intro-vector-data-python/spatial-data-vector-shapefiles/geographic-vs-projected-coordinate-reference-systems-python/`

Within these two different ways to do coordinate referencing the most used ones are World Geodetic System 1984 (WGS84) for geographic coordinate systems, and Universal Transverse Mercator (UTM) for projected coordinate systems. WGS84 is the datum that GPS uses to represent locations on Earth and is the one you will see used most for geographic coordinate systems. It uses the poles as 90 degrees north and south for latitude and 180 degrees east and west from the Greenwich meridian for longitude. With the latitude and longitude, you can represent a position in two ways, either as decimals of a whole degree, or as degrees, arcminutes, and arcseconds.

The UTM projection divides the earth up into 60 different zones such that each zone is 6 degrees in width. The zones stretch from 80 degrees south to 84 degrees north. The points north and south of this use another projection called UPS. The UTM belts can also be further divided by dividing them horizontally to create a grid. Each square of the grid is then classified by letters ranging from C to X, excluding I and O. This makes each square 6 by 8 degrees, except for X, which is 12 degrees. The letters A, B, Y, and Z are reserved in the aforementioned UPS projection. UTM uses meters as units, and each band has a central meridian that is considered to be 500 000 meters east. This is done to avoid any negative values. The position of the equator is 0 meters and increases as you go north if the position is in the northern hemisphere. If the position is on the southern

hemisphere equator is considered to be 10 000 000 meters north, decreasing the further south you get. This is also done to avoid any negative values. Due to how the northening values are set it means there may exist points with the same value north and south of the equator. To avoid confusing the points the letter of the latitude band is used[24]. A visualization of the UTM grid system can be seen in Figure 2.6.

Figure 2.6: Figure of UTM bands with grid letters
Source: `https://no.m.wikipedia.org/wiki/Fil:LA2-Europe-UTM-zones.png`

2.5.2 GNSS

Global Navigation Satellite System, most commonly referred to as GNSS, is a satellite-based system for navigation and positioning that enables users to de-

termine their geographical position precisely at anytime and anywhere on Earth. It consists of four major global navigation systems: the Global Positioning System (GPS), operated by the United States Space Force; Galileo, created and operated by the European Union; Glonass, operated by the Russian Military; and BeiDou, China's satellite system [25] [26] [27] [28]. Global Positioning System (GPS), Galileo, and Glonass employ a MEO, whereas BeiDou utilizes a Geostationary Earth Orbit (GEO) configuration. The difference in these arises from the variance in orbital altitude. MEO orbits at an altitude of between 2,000 and 36,000 km above the Earth's surface, while GEO satellites are positioned at an altitude of around 36,000km. Additionally, it is worth noting the existence of Low Earth Orbit (LEO) satellites, which traverse at altitudes ranging from 160 to 2,000 km and are often used in imaging applications [29]. The different orbital configurations utilized in GNSS can be viewed in Figure 2.7.

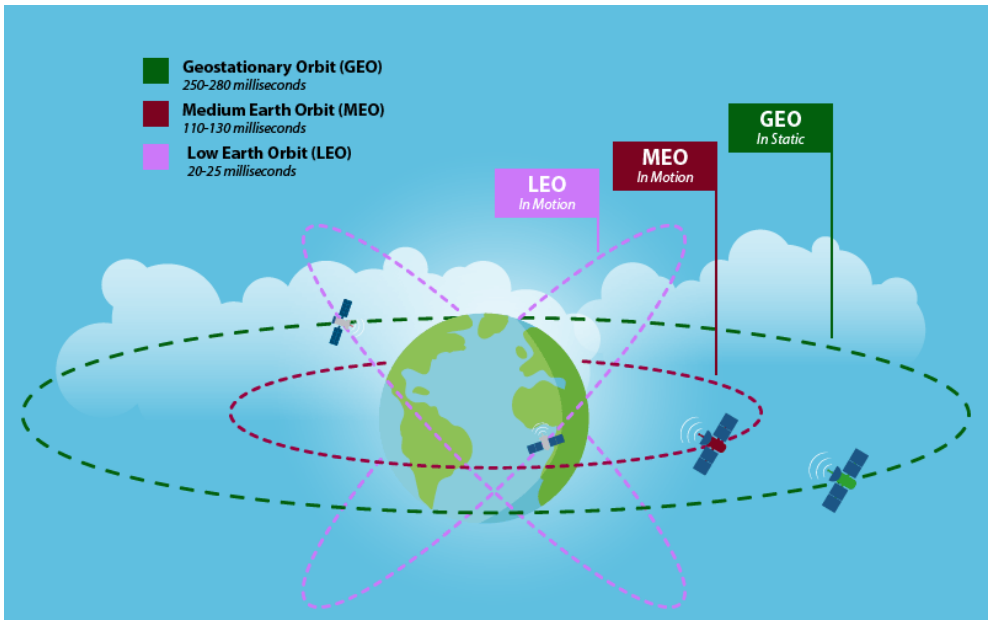


Figure 2.7: Orbital Configurations of GNSS Satellites
https://iasgyan.in/daily-current-affairs/geo-vs-leo-vs-meo

2.6 Virtual Reality

Virtual Reality (VR) is a technological concept that facilitates user immersion within a simulated environment through the utilization of a specialized headset or device. These environments typically consist of generated models within a 3d scene, which can be interacted with through movement and inputs. Virtual reality usually consists of a head-mounted device, often called a VR headset, and two motion joysticks. These joysticks are used to track the user's physical movement, thereby enabling their representation and manipulation within the virtual

domain. Meta Quest 2, one of the most common Virtual reality headsets, can be seen in Figure 2.8 accompanied by its joysticks.

Figure 2.8: The Meta Quest 2
Source: `https://www.complex.com/pop-culture/meta-quest-2-virtual-reality-headset-review`

2.7 Previous works

This project will be based on research conducted in two previous master theses. The first thesis, authored by Rune Strøm Brekke, aimed to create NTNU's Gløshaugen campus as a USD model. As with many other digital twin projects, Brekke's work necessitated a considerable amount of manual work done in City Engin [30]. The second thesis, authored by Aleksander Knutsen, transferred the digital twin to Nvidia Omniverse, and focused on wall texturing as well as automatic scene updates. Knutsen used Python scripts to process data retrieved from an autonomous vehicle. While a manual technique was employed for modifying the wall textures, an automatic approach was tested but demonstrated some insufficiency [31]. The digital twin model both these master theses were based upon was created by Rune Brekke, which is illustrated in Figure 2.9.

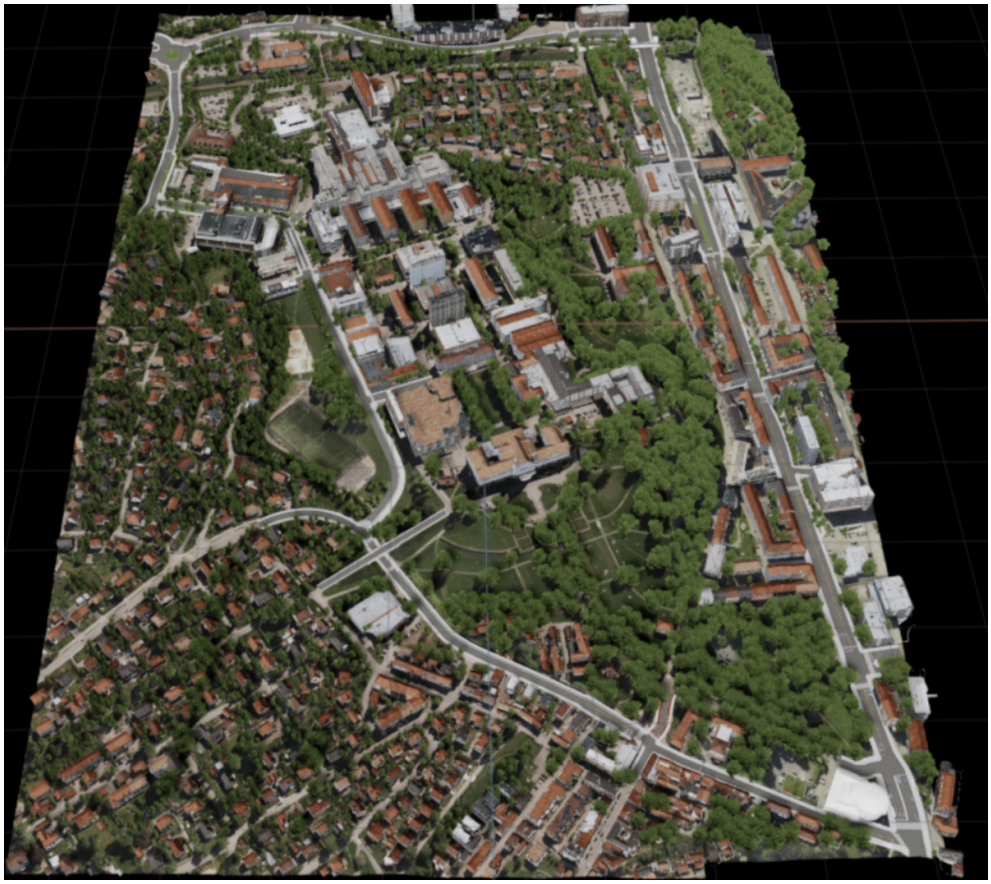


Figure 2.9: Digital twin model for Brekke and Knutsen's master theses
Source: `https://www.dropbox.com/s/vhp3kuwr0si57tn/Rune_Master-2021_DT-HDmap.pdf?dl=0`

Chapter 3

Methods

The process of creating a digital twin can be divided into two parts. The first is creating the digital representation of the object or place you want to digitalize. This includes data collection, data generation, and the creation of the digital representation. The result of this process is a static digital model replicating the physical entity at a specific time. However, it is important to note that this model might not represent the object in real time. The second task is to incorporate data in real-time by communicating with various sensors or other data sources into this digital model that converts this digital representation into a digital twin. This can include live images, live data, and other sensor data that static images can not capture. The traditional way of creating a digital twin can be time-consuming and challenging as it requires large amounts of data and data processing. In the coming chapter, we will present the methodology we used in our research in trying to create a high-resolution digital twin through programmatic generation. Our method of creating the digital twin makes it a good foundation and easily scalable by adding new data. The first section will introduce our data collection sources and programmatic approach to the problem, which includes the use of Python scripts to automate the generation of the digital twin. The second part will go into detail about the dynamic updates we added to our digital twin.

3.1 Digital twin creation

The process of creating a digital twin requires a large number of different types of data. The most important part is the topographic and orthographic data which is used to create the mesh and texture of the DT. In this section, we will show our methods of collecting data and what preprocessing we had to do before we ended up with our final model.

3.1.1 Data collection

Høydedata - Laserinnsyn

Høydedata is an online platform developed by the Norwegian Mapping Authority (NMA) that gives any user access to detailed heightmap datasets of Norway. NMA is responsible for producing, monitoring, and managing digital maps and geodata for Norway. The user interface of the platform allows users to define the area they wish to access and select the desired resolution for the heightmap, which can range from several points per meter, up to a single point every 50 meters.

However, there are some flaws with the system as it currently is. Firstly, the back-end processing can take anywhere from twenty minutes to 2 hours after a request was made by the user in the platform. This means the users are required to plan in more detail what is needed in advance. Secondly, the online platform offers a range of customizable settings that can significantly influence the output. Regrettably, we initially failed to recognize one crucial setting hidden within the program's advanced options. This led to us using ArcGIS Pro to cut the segment returned from Hoydedata into the smaller extent chosen from the start. In Figure 3.1 you can see the process of cutting the heightmap in ArcGis with the red square being our selected extent and the black box being the full segment. This process gave us valuable insight into different file formats and image processing techniques. However, we later discovered that the platform itself offered the capability to clip the output to the chosen extent directly, contributing to the use of additional tools being unnecessary.

Norge I Bilder

Norge I Bilder is an online platform developed through a collaboration between the Norwegian Public Road Administration, NMA, and Norwegian Institute of Bioeconomy Research (NIBIO). Through this online platform, you have the possibility to see orthographic photos of Norway that are made available by Norge Digitalt. With an account linked to Norge Digitalt and GeoNorge, you have the possibility of downloading these different Ortophotos. The resolution of the Ortophoto is determined by the establishment that has provided these to GeoNorge and may vary from 1 meter to 4 centimeters per pixel.

One of the drawbacks of using the platform occurs when a user tries to download images in their maximum resolution. Doing this will lead to the creation of

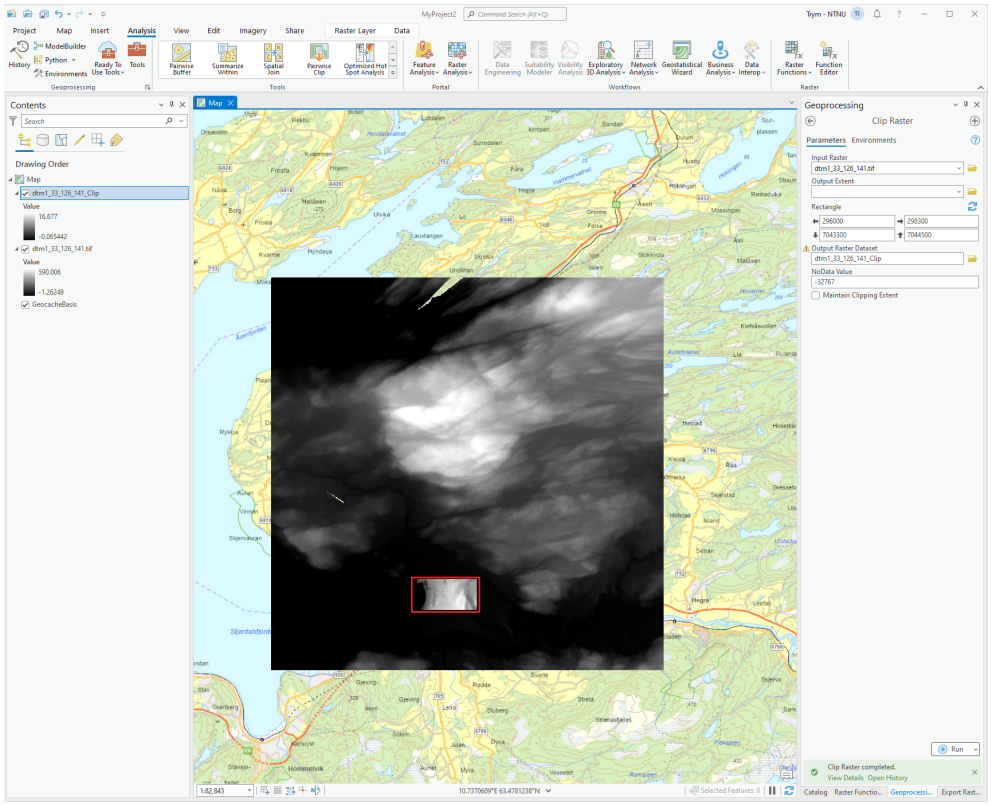


Figure 3.1: Clipping of heightmap in ArcGis

files that are too large for file transfers, causing "Norge i bilder" to segment these images into smaller images. As a part of doing this, the platform also includes a black border around the image. This is done because "Norge i bilder" uses a fixed size for their images and if one of the tiles from the extent is smaller than that size, the platform tries to adjust the images according to this fixed size. The addition of the black borders creates a new challenge for the creation of the digital twin, requiring us to trim the black borders prior to creating the textures and mesh. To accomplish this, we obtained the bounding box of the heightmap and the texture image, and obtained the points where these intersected. Using these points we could create a new image that excluded the black borders. The code snippet in Code 1 shows how this was done. Once completed, the new image was stored as a GeoTIFF dataset and used as a texture for the mesh created.

```
# Removes NorgeIBilder's black border on images
def trim(image, heightmap):

    heightmap_bounds = shapely.geometry.box(*heightmap.bounds)
    image_bounds = shapely.geometry.box(*image.bounds)
```

```
intersection = heightmap_bounds.intersection(image_bounds)
mask1 = shapely.geometry.mapping(intersection)
trimmed_image, transform = mask.mask(
    dataset=image,
    shapes=[mask1],
    crop=True,
)
```

Listing 1: Trimming of the black border from texture images

Nasjonal vegdatabank - Statens vegvesen

Nasjonal vegdatabank (NVDB) is a public database that includes information about the road networks in Norway. This includes roads and road objects, such as signs and speed bumps.NVDB is operated by the Norwegian Public Roads Administration and is used to implement and ensure good road quality. In this project,NVDB was used to collect precise positional and geometric data regarding signs and roads. This data was then processed and integrated into the digital twin.

To fetch the data from the database a GET request from the Python package *request* was used. The request was set up to retrieve specific types of data using different query parameters. In Code 2 you can see one way of doing this. The base API url, *'https://nvdbapiles-v3.atlas.vegvesen.no'*, directs the GET request to the database. The *'vegobjekter'* redirects you to the specified end-point, granting access to different types of road objects. In this example, we used '/96' to specify that we wanted the road signs. When doing this for the roads instead of the signs we used *'/vegnett/veglenkesekvenser/segmentert'* to specify the end-point. The parameters for the query can be set up to define different road types or to use a specified extent. For our project, we retrieved the bounding box from the heightmap provided by the user and used this to set the extent. This ensured that we received all the necessary objects for the digital twin.

```
def get_data(link:str,start:str):
    params = {'srid':'4326',
              'vegsystemreferanse':'EV14',
              'alle_versjoner':'true', 'inkluder': 'alle',
              'start': start}
    resp = requests.get(link,
                        params=params)
    return resp.json()

BASE_API_URL = 'https://nvdbapiles-v3.atlas.vegvesen.no'
signal_pts = BASE_API_URL + '/vegobjekter/96' #96 for traffic signs
respjson = get_data(link=signal_pts,start=None)
```

Listing 2: NVDB GET request

A shortcoming of NVDB seemed to happen when we wanted to fetch state-, county-, and municipal roads. When combining the three only the municipal roads and road objects were returned. This meant we had to divide the requests into 2 different queries. The first one included the parameters for the municipal roads and the second included the parameters for the rest, this can be seen in Code 3.

```
def set_params(heightmap):
    min_x, min_y, max_x, max_y = [*heightmap.bounds]
    bbox = f"{min_x},{min_y},{max_x},{max_y}"
    print(bbox)

    params_main = {'srid':'5973',
            'start': None,
            'vegsystemreferanse':'E,F,R',
            'kartutsnitt':f'{bbox}'
            }

    params_k = {'srid':'5973',
            'start': None,
            'vegsystemreferanse':'K',
            'kartutsnitt':f'{bbox}'
            }
    return params_main, params_k
```

Listing 3: NVDB GET request for multiple road types

Open street map

Open Steet Map (OSM) is a collaborative project that aims to create a free and open map of the world. Unlike other similar platforms, OSM is not controlled by a single company or entity. Instead, it is maintained by a community of volunteers who add and update data on roads, buildings, landmarks, public services, and more. This makes OSM a valuable resource for individuals and organizations in need of accurate and up-to-date map data. Organizations like Esri and QGIS use and pull data from OSM.

Overpass API is one of the developer tools that OSM offers. It is a read-only API that allows users to query OSM data and extract only the information they need. By using the API, users can filter data by, location, tags, types, and many other criteria. This makes it easier to work with the data and use it in custom applications.

The query in Code 4 illustrates how the data is gathered for our digital twin. In our case, we want to retrieve all the buildings within a designated area called "bbox". A "way" in this case is just an ordered sequence of nodes that are accompanied by information tags. Meanwhile, relations are a group of members, such as ways, which can store more complex features of the map, such as building

with holes. The specific information that our code attempts to retrieve are building parts and buildings. Building parts are complex features divided into several simpler features. The "out center" and "out geom" tags specify the desired information to be included in the output. In this case, the query will return the geographic center of the "way", as well as the geographic location of every node in the "way"

```
[out:json][timeout:25];
  // gather results
(
  // query part for: "building:part"
  way["building:part"]({bbox});
  relation["building:part"]({bbox});
  // query part for: "building"
  way["building"]({bbox});
  relation["building"]({bbox});
);
// print results
out center;
out geom;
```

Listing 4: Overpass API GET query for buildings and building parts

3.1.2 Implementation

There are several steps from gathering our data to having a complete model in Nvidia Omniverse. These steps are similar and implement similar technologies across all the elements of our digital twin.

Omniverse extension

To create the static digital twin we use the Omniverse extension template that follows with Omniverse code. The template includes boilerplate code, dependencies, its own Python environment, and more. This allows for fast integration of additional functionality to the Omniverse platform.

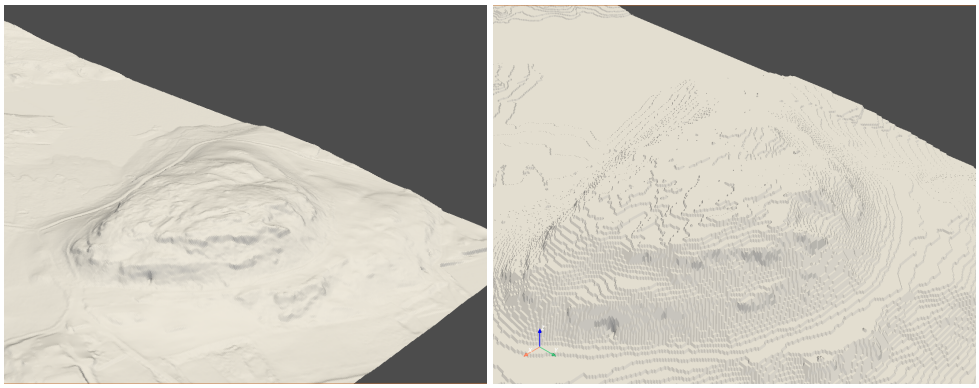
Terrain

The process of generating terrain is a process where reading and processing the data can be a difficult task. For reading the data we used the Rasterio library in Python. This library allows for reading geospatial raster data in the form of .tif files. One of the challenges when working with our terrain is the size of our rasters, as they require a sizable amount of memory. To address this issue, a generator function has been developed to only load one texture into memory at once.

There are several steps of data processing needed to be done before a mesh can be created. The first step is turning our heightmap image into the terrain model

we want. The NumPy function meshgrid is used to generate two 2D arrays that represent a grid of points in the x and y coordinates. The resulting arrays provide a structured grid where each point corresponds to a specific combination of x and y coordinates.

The heightmap from Kartverket is provided in a format with a specific number of points per meter. The meshgrid function returns arrays representing the indices of each value. In our case, we consider a one-unit increase to be one meter. This means in cases where the elevation data is not one point per meter, the model would not be geographically accurate. To address this issue we calculate a spacing factor that divides all the values in the arrays. This approach also incidentally solves another problem. When creating a structured grid mesh in Pyvista with integer values for the x and y coordinates, it implicitly converts the elevation values to integers. This is not the case when the values are converted to floats. A visualization of this difference can be seen in Figure 3.2 and the final result is a blank terrain model shown in Figure 3.3.



(a) Terrain created with float values **(b)** Terrain created with integer values

Figure 3.2: Comparison of data types in terrain generation

Figure 3.3: Terrain model before adding textures etc.

The initial step, shown in Code 1, involves trimming any black border from the texture. This ensures that the texture correctly aligns with the heightmap. The second step is to circumvent the 8000 by 8000-pixel limit Omniverse has on texture sizes. To do this we find the number of rows and columns the texture

can be divided into that keeps the size under this limit, as well as retaining the dimensions as integer values.

To divide our textures into smaller images we use two functions. The first one calculates the necessary data, and the second uses that data to create a new texture.

To generate the necessary data we first compute a Cartesian product based on the number of rows and columns to get all the necessary offsets. Using these offsets we create a window of appropriate size for each sub-image. The dimensions of this sub-image are determined by dividing the original image's height and width by the number of columns and rows.

Once the window has been created it is intersected with the original image to get the necessary extent data. The resulting window and the extent are then returned to the first function. Next, we select the pixels for our new sub-image before adding both the extent and sub-image to their respective lists. These steps are repeated for all our sub-images.

To extract the sub-meshes a function is made that takes the in a texture and the respective extent. The extent is transformed to match the one of our terrain. This value is used to extract a subset of the terrain mesh and to position the texture on the mesh.

Buildings

The buildings are created from data gathered through a request to Overpass API. This request gives us a list of buildings, which we then convert to SimpleNameSpace objects. SimpleNameSpace converts each JSON object to key-value pairs. This conversion enables the creation of objects that can be accessed using dot notation and is a more simple form of the standard class. Making it easy to work with the data from the request.

After retrieving the building data we iterate over the list of buildings. A function is then called that takes a "building" and the heightmap as inputs. This function generates the points that will be used for creating building meshes.

The buildings can be classified into two types, "multipolygon" or "way". The difference between the two is that a multipolygon consists of one inner and one outer set of nodes. We want to store the points in different lists so we can process them differently. We retrieve the longitude and latitude for each point and convert it from WGS84 to UTM33, which is the projection used in our project. For each point, we sample the corresponding elevation data from the heightmap using the UTM coordinates. We store the smallest height value for each building to ensure the building does not float above the terrain in the model. This way we can treat the polygon of points as a 2d footprint, simplifying the mesh creation process. We perform additional checks on the list of points, such as removing duplicate points if they exist at the start and end, and determining if the points are in clockwise order. This is useful when we are generating our mesh as we use an algorithm called Delaunay triangulation, which is affected by the order of the points.

In our workflow, we utilize Pyvista to create a 2D representation of a building's footprint based on the generated points. This involves constructing a Polydata object by combining the points and providing an initial approximation of the mesh's faces.

For buildings categorized as "multipolygonal," both the inner and outer points are included in the same Polydata object. However, it's crucial to note that the ordering of the inner and outer points differs. This distinction becomes important when applying Delaunay triangulation to optimize the constructed mesh. Specifically, the outer points are in clockwise order, while the inner points are in counterclockwise order. The orientation of the points influences the triangulation process, as well as the positioning of the mesh's edges. Ultimately, the resulting footprint is extruded to match the building's height, finalizing its representation.

Road Lines

To create the road lines for our digital twin we use data fromNVDB's API. We represent the lines in Omniverse as 3d models.

To get the necessary data we use the request shown in Code 2 with the parameters set in Code 3. An additional call is also sent to a different endpoint in the API to fetch the width of the roads. The response to both queries is stored in separate pandas data frames. A function was created that aimed to associate the road width information with the corresponding road based on their shared id. The road width in the database can vary depending on several factors, such as if it includes a road shoulder. Consequently, the function finds the smallest value as it is presumed to represent the edge of the road.

To get the coordinates of the edges of our road, we first iterate through our list of points for each road segment. The list is ordered as a polygonal chain, and we select the current index and the next index for each iteration to select two points. By calculating the angle between these points, we can use trigonometry functions and the road width to find the coordinates of the edges at the beginning and end of each line in our polygonal chain.

$$atan2(y,x) = \begin{cases} arctan\left(\frac{y}{x}\right) & \text{if } x > 0 \\ arctan\left(\frac{y}{x}\right) + \pi & \text{if } x < 0 \text{ and } y \geq 0 \\ arctan\left(\frac{y}{x}\right) - \pi & \text{if } x < 0 \text{ and } y < 0 \\ +\frac{\pi}{2} & \text{if } x = 0 \text{ and } y > 0 \\ -\frac{\pi}{2} & \text{if } x = 0 \text{ and } y < 0 \\ \text{undefined} & \text{if } x = 0 \text{ and } y = 0 \end{cases} \tag{3.1}$$

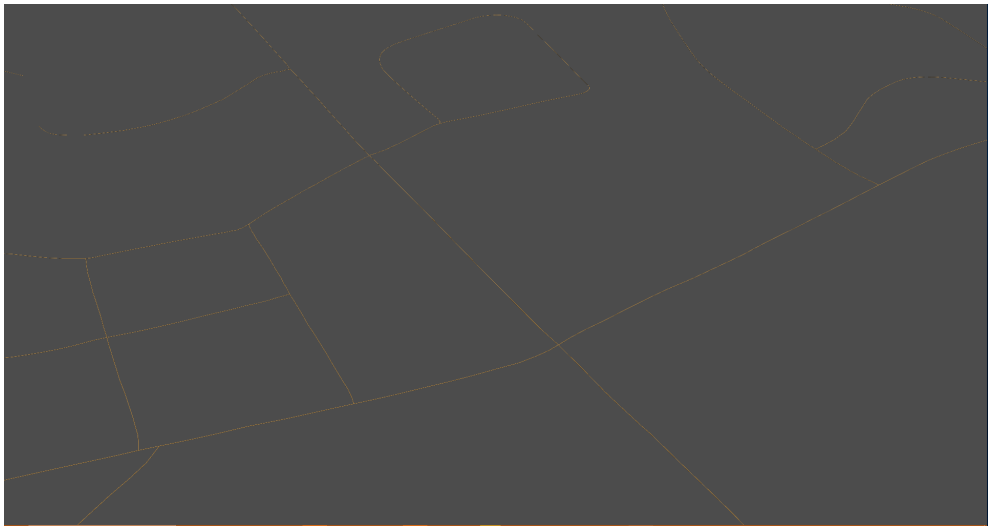
In our code, we use the atan2 function and calculate the angle between two points, p1 and p2. The calculation utilizes both the x and y values for these points as inputs.

$$angle = atan2(p2[y] - p1[y], p2[x] - p1[x]) \tag{3.2}$$

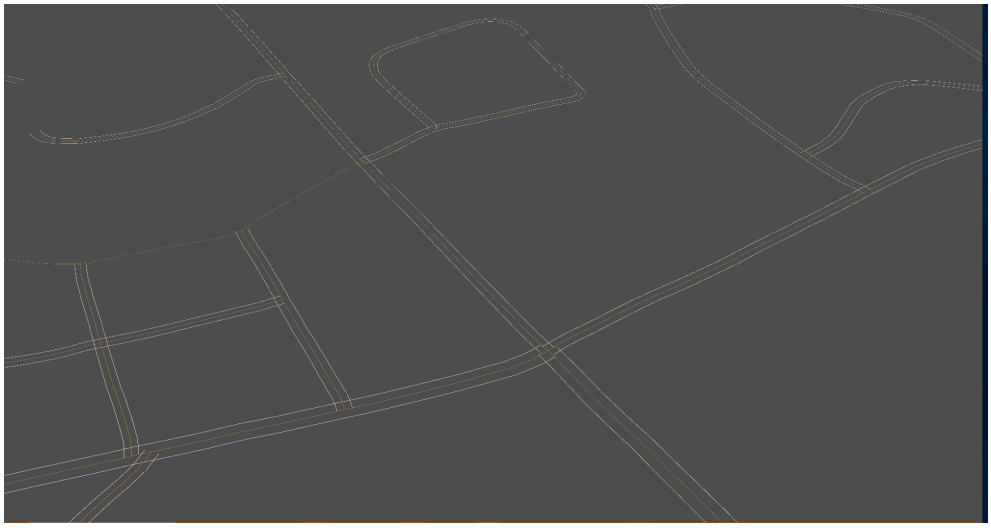
We create a new point by using the angle calculated, the original point p, and the distance from the center line. We assume the height is the same for the edges and center line. To create a point for each side of the road, the value of the new point is modified by either adding or subtracting from the original value.

$$point = \begin{cases} x = p[x] \pm \frac{roadwidth}{2} * sin(angle) \\ y = p[y] \pm \frac{roadwidth}{2} * cos(angle) \\ z = p[z] \end{cases} \tag{3.3}$$

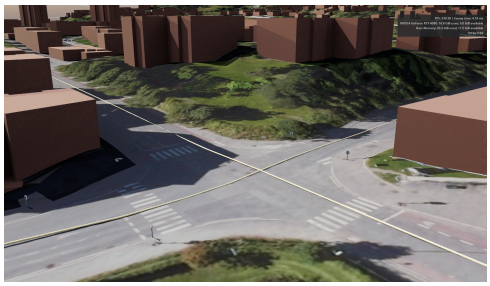
To be able to visualize this in Omniverse we create 3d meshes for the center line of the roads, as well as for the edges of either side. The roads with only the center and with edges are visualized in Figure 3.4



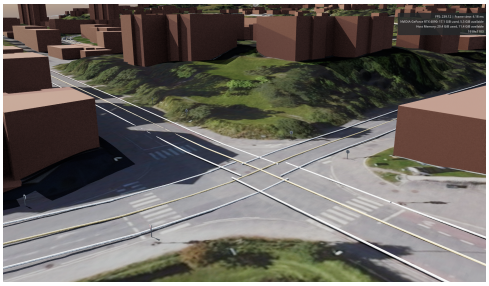
(a) Roads generated with only center line



(b) Roads generated with side lines



(c) Road Center in our Digital Twin



(d) Road with sidelines in our Digital Twin

Figure 3.4: Comparison of roads with and without side lines

Signs

TheNVDB database contains positional data for all types of public signs on Norwegian roads. The database was used to obtain the sign positions within the area of the digital twin. This was done by extracting the boundaries of the digital twin and using this as the parameters to an API call, following a similar approach as the one we did with the road lines. The data returned from the database contained several layers of information such as metadata, geometry, and location. The positional data and the sign information were then processed from this data and subsequently stored in a Pandas data frame.

Leveraging the information obtained from theNVDB database made it possible to automatically incorporate the signs into the Omniverse scene. This was accomplished by using the sign information to loop through a map in Python, which contained the different types of sign assets we had. SinceNVDB contain hundreds of different signs we made the decision to classify them into 4 different types: hazard signs, give way sign, prohibition signs, and mandatory signs. This can be seen in Code 5. In this asset map, the ranges of sign IDs are associated with corresponding asset filenames. Additionally, a default asset named 'cube.usd' is specified for cases where a sign's ID does not fall within any of the defined ranges.

```
asset_map = {
        range(100, 157): 'hazard-sign.usd',
        range(202, 215): 'give-way-sign.usd',
        range(302, 379): 'prohibition-sign.usd',
        range(402, 409): 'mandatory-sign.usd'
    }
    default_asset = 'cube.usd'
```

Listing 5: Asset map

3.2 Pipeline for automatic generation of static digital twins

The process of creating a digital twin can be divided into 5 steps for the user. A step-by-step guide for this can be seen in section 3.2. The first step involves the collection of the data used in the generation of the digital twin, specifically the height map and the textures. We recommend using `NorgeIBilder.no` and `Hoydedata.no`, it is also possible to other sources by ensuring the GeoTiff file format is used for the heightmap and textures, and textures ordered from top left to bottom right.

The second step is to link the software we have created with Omniverse. After doing this, it is possible to boot the program through the terminal by running the following command: *app\omni.code.bat –ext-folder exts –enable omni.hello.world*. The third and fourth steps involve selecting the various inputs collected in step number 1 and setting the output for where you want the digital twin to be saved. The final step involves starting the program by reviewing the input variables and

clicking the button. As mentioned all of these steps are described further in the step-by-step guide provided in section 3.2.

Digital Twin Creation Guide

Introduction:

The software that has been created has an ulterior motive of making high-quality digital twins more easily accessible to different user groups. This guide will walk you through using the software and what it takes to get a digital twin generated. First, a simple step-by-step tutorial will show you the fundamental steps of the software and after this it will go into more detail for each point.

Step by step tutorial:

Step 1	Step 2	Step 3	Step 4	Step 5
Collect Images	**Start Software**	**Choose Input**	**Choose Output**	**Generate DT**
• Collect the desired textures and heightmap for your digital twin. • NOTE: We recommend using GeoNorge and Hoydedata.no for this, but any GeoTiff file works	• Link up your extension to Omniverse Code • Boot up the Digital Twin creation software by opening the script in any IDE and typing in the terminal: • app\omni.code.bat --ext-folder exts --enable omni.hello.world	• When the software has booted up enter the collected images from step one into the input fields in the GUI. • NOTE: If there are more than one texture please enter the folder these are stored in.	• After you have entered the input variables choose the output path to where you want to store the digital twin in your Omniverse Nucleus server.	• Start generating you digital twin by hitting the "Create Mesh" button. • After this is done you cant sit back and wait while the digital twin is being processed.

Detailed description:

Step 1:

In the image to the left you can see our recommended settings for retrieving the heightmap from hoydedata.no. Here we have the "hjornekoordinater" as the extent of our digital twin.

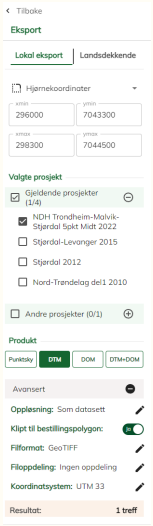
In the next box, named "valgte prosjekt", you can choose how detailed you want your terrain to be. We recommend using the best possible, which for this example is the 5pkt.

In the "avansert" tab, it is important that you choose the right resolution in "oppløsning" and that the box for "klipt til bestillingspolygon" are checked and "filoppdeling" is set to none, as this returns the image as one file.

NOTE:
Remember to use your coordinate system, for this project we used UTM33.

It is important that the images are GeoTiffs as the bounds and internal coordinate system are used for the creation of the digital twin.

After you have retrieved the heightmap from Hoydedata.no you can do the same process for Norgeibilder.no



Step 2:

The first thing necessary to start the program is to link the source code to Omniverse Code. This can be done by going into the extensions tab in the Omniverse and adding the new path.

NOTE: More info about this can be found on Omniverse Code's pages and at this link:

https://github.com/NVIDIA-Omniverse/kit-extension-template

To boot up the program you then need to open the source code in an IDE, we and Omniverse recommend using VS Code. Once you have the code in front of you you can open a new terminal window and run the following command:

app\omni.code.bat --ext-folder exts --enable omni.hello.world

Change "omni.hello.world" to what package your omniverse extension is set to.

A new Omniverse Code window should appear in front of you.

Step 3 and 4:

As seen in the image below you will be shown a simple GUI in your Omniverse Code window after finishing step 2. Here you will need to choose the correct input files, i.e the folder where your texture files are stored and the path to where your heightmap.tif is stored locally.

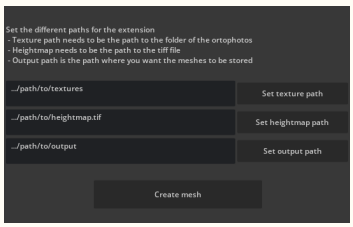
After you have set the different input variables needed to create your digital twin, you will need to set a path to where the digital twin will be stored on your Nucleus server. Here we recommend creating a new folder for your project.

The path text in the input fields should change as you edit them so you can see that the right paths are entered.

Step 5:

After you have gone through all the previous steps and you are sure that all the variables are correct, you can click on the "create mesh" button. The software will then create your digital twin and fill it with building meshes and more.

NOTE:
The time it takes to generate your digital twin will depend on several factors such as hardware quality and size of your textures and heightmap.



After the user has taken the necessary steps to start our program, they start creating the digital twin. As of now, the process is divided into 2 main subprocesses, the generation of the terrain and the generation of additional data. The most crucial part of the digital twin is the terrain model. To generate this, the software takes in the elevation map and textures provided by the user and divides this down into smaller tiles that Omniverse is able to interpret as 3D models. After the tiles have been generated they are uploaded to the Nucleus server and entered into the USD scene in Omniverse.

The next processes involve adding additional and significant data to the digital twin. For this project, the building meshes and road network were the major focus areas as this has a greater impact on our use case, autonomous vehicle training, and visualization. To do this, we extracted the boundaries from the elevation map and used this to extract data from different data sources, mainlyNVDB and Overpass API. After the data was collected, it was processed as discussed earlier in the chapter and then added to the USD scene. A flowchart of the software processes can be viewed in Figure 3.5.

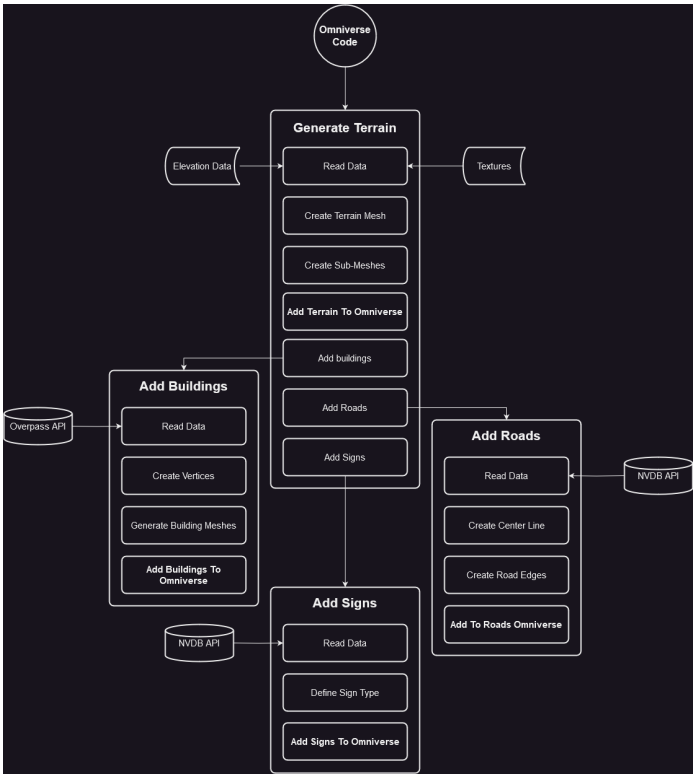


Figure 3.5: High-level flowchart of our system

3.3 Dynamic Digital Twin

A digital twin is a virtual replica of a physical object or system that allows for real-time monitoring and analysis. For this to be possible it is important for the digital twin to get real-time data from the physical object or system. Such data can be obtained from different sources such as sensors, IoT devices, and cameras. This information can be processed and fed into the digital twin to continuously update and modify the digital twin. The benefits of using live sources are numerous and the main point for our case is that it is easier for real-time monitoring of the autonomous vehicle. In addition to this, the vehicle provides other data that can be useful for our digital twin such as images and additional road information. Overall, the integration of real-time data sources is a critical aspect of developing an effective digital twin. In this section, we will address how we managed to address research question 3, which explores the possibility of streaming real-time data from an autonomous vehicle for dynamic updates.

3.3.1 Technologies

The coming section will explore the fundamental technologies employed that allowed us to integrate live updates in our digital twin, with the main concept being the PostgreSQL database.

PostgreSQL and TimescaleDB

Setting up the PostgreSQL database was a crucial point for enabling us to incorporate real-time data into our digital twin. The database consists of 3 distinct tables that store the geographical coordinates from the vehicle, images taken from the car, and information about potholes and cracks from the different roads that were traversed. Once the data is stored in one of these database tables it is processed through different Python scripts, which either leads to the information getting added into the digital twin or modify already existing data. Since most of the data that is captured are time-series data we thought it would be suited to introduce Timescale db into our database, as it would provide better performance to the overall product. Timescale is designed to handle high-frequency time-series data better by reducing the amount of data being stored and dividing the data into many smaller chunks. This could be crucial in our case where the geographical coordinates from the autonomous vehicle are sent 10 times a second and could lead to a large amount of data being stored. Timescale is easily added to any project using time data and does not remove any features from Postgres. Timescale offers additional features, such as gap filling, which can prove valuable for geographical data when there is a data outage or irregularities.

Omniverse connector

An Omniverse connector is used to enable communication from external software to Omniverse. The sample comes with boilerplate code, and examples to help the user. Our experience is that the code is unnecessarily nested and complicated, with poor or no documentation of the functions provided. After some refactoring and trial and error, we were able to use data stored within a database to alter our scene in Omniverse using this connector.

To enhance accessibility and facilitate future use, we have organized the classes used in the Omniverse Live example into separate files for improved reusability. The two files created are: "LiveSessionInfo.py" and "LiveSession.py".

The purpose of "LiveSessionInfo.py" is to collect all logic used and store the Omniverse Live Session created. While "LiveSession.py" contains the code responsible for creating, merging, and ending an Omniverse Live Session.

Robot Operating System

The Robot Operating System (ROS) is a free and open-source software that allows different components of a robot to communicate with each other without being tightly coupled together. ROS uses a publisher-subscriber architecture, meaning that a publisher can send information and messages on a certain topic where the subscribers of this topic will receive it.

When NTNU received the autonomous vehicle software, the developers created separate data classes to assist communication within the car. These classes needed to be integrated into our client script to establish a connection between the data obtained from the car and the database. To accomplish this, a Rospy client was developed to handle this transmission. The Rospy client utilized psycopg2, a PostgreSQL adapter that enables the execution of insertions and updates on a database. With psycopg2, a connection to the database was established. For each callback event, the Rospy client would perform an insertion operation, which included the data retrieved from the car into the database. The snippet of the callback function can be seen in Code 6.

```
def callback(data):
    global conn
    if conn is None:
       return
    cursor = conn.cursor()
    insert_query = "INSERT INTO car_position(time, position) "
                "VALUES (%s, %s)"

    time_stamp_since_epoch = data.header.stamp.secs
                + data.header.stamp.nsecs*1e-9
    date = get_date(time_stamp_since_epoch)
    position = str(data.utm_easting) + "," + str(data.utm_northing)
```

```
        cursor.execute(insert_query, (date, position))
        conn.commit()
```

Listing 6: Callback function for Rospy client

In addition to the Rospy client created to retrieve real-time data from the car, the use of rosbags was also incorporated. Rosbag is a file format and a playback tool from ROS that lets a user record every message sent on a topic. This was used to rerun the data, which could help support performance analysis and testing. By using the command *rostopic echo -b file.bag -p /topic* we could easily create CSV files from the data generated from the driving sessions. These CSV files were later processed using scripts and fed into the PostgreSQL database or used to analyze the data. With the Rospy client created we could also rerun the rosbag as if it was coming from the car by using the command *rosbag play recorded1.bag –topics /UTMPosition*. By executing this command, we could visualize and simulate real-time updates in our digital twin, mimicking the data stream as if it came from the vehicle.

3.3.2 Live updates from moving agent

Integration of real-time data from a moving agent into a digital twin was one of our main goals for this project. This integration would allow for easier monitoring of the vehicle and could lead to analyzing its patterns and driving capabilities. This was done by sending information to the PostgreSQL server described in section 3.3.1, where it would then be processed before modifying the digital twin in real-time.

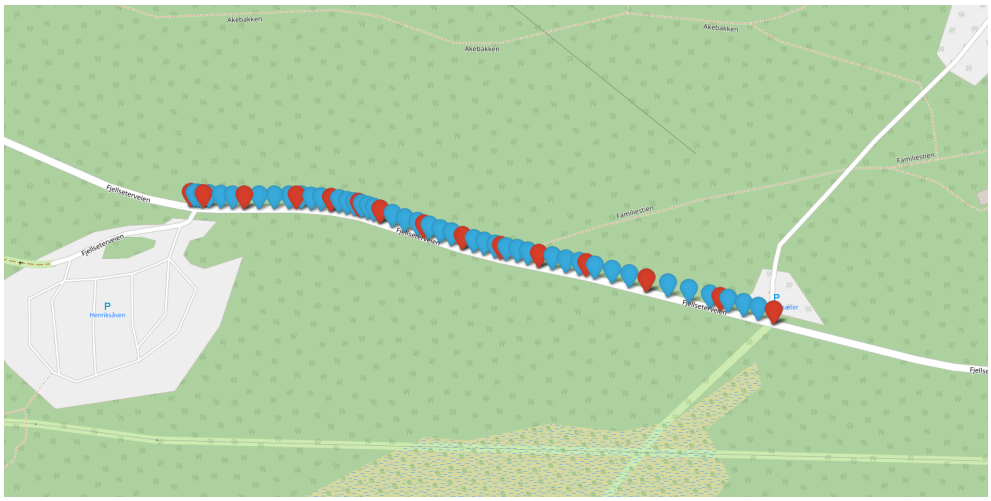
Modifying a digital twin in real-time with Omniverse necessitates setting up a live connection through an OmniLive session. An OmniLive session enables users from different platforms to collaborate on the same assets in real-time. Utilizing non-destructive edits makes it possible for the users to edit a scene without altering the base file. Once the live session is complete, the users can merge these modifications into the base file [32]. To enable the database to function with these OmniLive sessions, we created an Omniverse Extension with Python that could alter these sessions directly from a script that reads data from the database. Omniverse provides basic sample codes from some different use cases, one of these being altering a scene live. We employed this sample code to create a class to store the live session and a script that could modify the car in the digital twin with the new positions communicated to the database. The live session class created can be reused to create other features that can alter the models in the digital twin.

3.3.3 Analysis of driving data

The driving data obtained from the rosbags generated from the trips were used to analyze and examine the driving patterns. The road network data acquired

fromNVDB contained two or more points, mainly the starting and ending point of the current road segment. To overcome this limitation in the data, we used an interpolation method from the NumPy library called linspace. This allowed us to generate linearly spaced points in between the points gathered for the road. An example of this can be seen in Figure 3.6 where a road segment from Skistua was retrieved and filled with additional data points. With these points, the distance from the car to the center line was calculated and subjected to further analysis.

(a) Roads segment without linearly spaced points



(b) Road segment with added linearly spaced points

Figure 3.6: Example of additional linearly spaced points

3.3.4 Potholes and cracks

Using the data acquired from the cameras connected to the vehicle gives access to several new possibilities and opportunities. One possibility that was explored was

the use of a deep learning algorithm to extract more information about road conditions and integrate that into the digital twin. With the help of Mamoona Birkhez Shami, a Ph.D. student, a deep learning model was created to analyze the acquired camera footage to identify road defects. The data was stored in the PostgreSQL database and fed into the digital twin. The integration of this data could help with the assessment of road networks, leading to more proactive maintenance of the road and easier monitoring.

3.4 Development process

Throughout the development of this project, an iterative development process has been utilized, which was compromised into three distinct stages. A visual representation of this process can be seen in Figure 3.7. The initial stage involved the implementation process, where the necessary additions were integrated into the project. This primarily included further development of the previously mentioned Omniverse extensions discussed in section 3.1.2, or by looking at the integration of new data.

The second stage was the testing and evaluation part, where we studied and evaluated the results of our implemented features. This process incorporated comparative analyses against earlier versions, as well as utilizing research acquired by other developers or projects to evaluate against. By incorporating a test and evaluation phase, we were able to engage in meaningful discussions about the shortcomings and improvements of our product thus far. This process provided valuable insights into each other's work, enabling constructive conversations on how to enhance the product further.

In the final phase of the iterative approach, we brought in expert feedback to further process our results and decide on future goals for the next iteration. This feedback could come from our supervisors and other stakeholders.

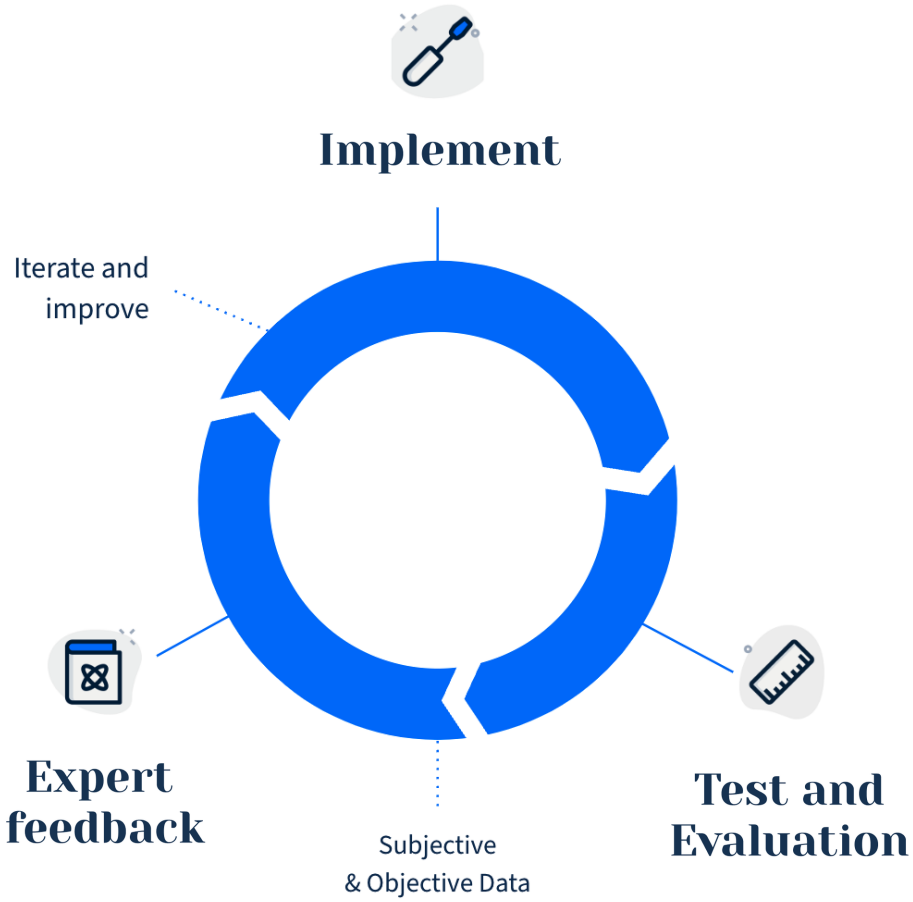


Figure 3.7: Development process loop

Chapter 4

Results

The objective of this research was to advance the programmatic and automated generation of digital twins. In this chapter, we will present the results we were able to achieve, which are divided into two primary sections: the results from the mesh and digital twin generation and the results we were able to gain from our live sources.

4.1 Presenting the Power of Automatic Digital Twin Generation

As aforementioned, the goal of this project was to automate the generation of digital twins. This was done to enhance access to high-resolution digital twins. Through many different iterations and development, we have achieved significant advancement in this field. The result of this is an Omniverse extension, which has the capability of creating a high-resolution digital twin given the right input variables.

To visually showcase our progress, we have created a video featuring several digital twin models created with the help of the software developed through this project. You can find the video from this link `https://youtu.be/uK-7a0N-g94` or in the box below, where we explore these advanced models and witness the power of automatic generation in action.

Overview of Automatic Digital Twin Creation:
`https://youtu.be/uK-7a0N-g94`

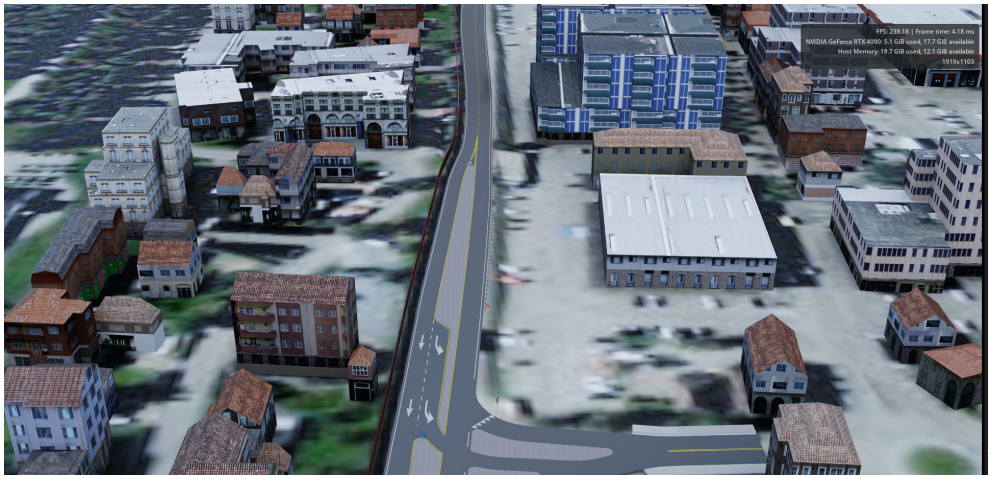
4.2 Exploratory process using CityEngine and RoadRunner

During our project, we went through several iterations for the digital twin. The first one we tested was a CityEngine model, which was generated entirely with their program. CityEngine is a powerful urban planning tool that generates 3d city models directly. This seemed like a suitable starting point for generating data and gaining initial insights into the project.

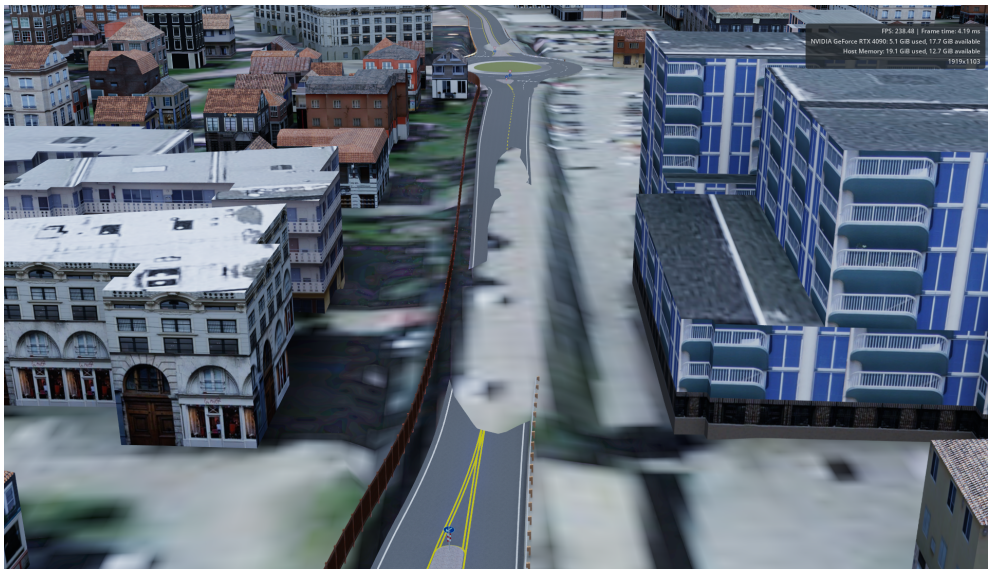
The CityEngine model offers several different customizable settings that allow the user to determine which component should be generated, as well as the quality of the model you want to create. In this initial iteration, we experimented with these settings to generate building meshes and road networks, as well as testing their vegetation model. This phase provided valuable insights into the area of digital twin creation and helped us scope out the requirements for our own program.

From our experience with CityEngine, we identified several key points. Firstly, we appreciated the ease of creating new models and the ability to generate terrain of different qualities. The second point we took from these tests was the different types of models we could integrate into our own solution such as vegetation and buildings. However, there were also some disadvantages to using CityEngine, mainly the quality of the terrain and the flexibility of the model. Even though you can alter the terrain quality to some extent, CityEngine is not specifically designed to deliver high-resolution terrains. Meaning that the standard did not live up to our expectations. Additionally, there were some flaws with the road network model created by CityEngine, mostly occurring in roundabouts and crossroads.

To find a solution for the flaws found in the road network created by CityEngine, we experimented with creating a road model using Matlab's program Roadrunner. This road model was very detailed and had several other elements that led to a good product for the digital twin, such as signs and guardrails. However, we found it difficult to include this model in the digital twin scene after exporting it as a USD object. This occurred due to the differences in elevation data used. Roadrunner divides the roads into tiles and uses the same height data for the entire tile even though it might contain several different points. This could lead to the road either being buried in the ground or floating on top of it. This can be seen in Figure 4.1, where the detailed road model is shown in two different places. The first image shows a position where the model fits perfectly into the scene, leading to a good addition to the digital twin. The second image shows when the model does not fit as well and therefore passes beneath the terrain in some places, leading to an inferior model.



(a) Roadrunner model fits perfectly to the scene



(b) Roadrunner model passes beneath the terrain

Figure 4.1: Showcase of two different outcomes for the Roadrunner models

4.3 Advancing towards an automatic solution

Using the points we learned from the first iteration, we started to develop our own model. For the initial iteration of our own model, we aimed to combine textures obtained from NorgeIbilder with heightmap data from HoydeData. However, we encountered some restrictions within Omniverse when we tried to create one big model. Specifically its maximum resolution for textures, which is limited to 8000x8000 pixels. Consequently, the model we developed couldn't load into our scene.

To overcome this constraint, we changed our approach to split the texture and

heightmap into smaller tiles that would fit within Omniverse's limitations. This was done by calculating a common divisor for the height map and texture that caused each tile to be under 8000x8000 pixels. Implementing this tile-based division introduced a more complex code, resulting in multiple locations in the code that could lead to errors. This can be seen in Figure 4.2, where the division system failed to align the heightmap and the textures properly, leading to considerably large gaps in the texture.

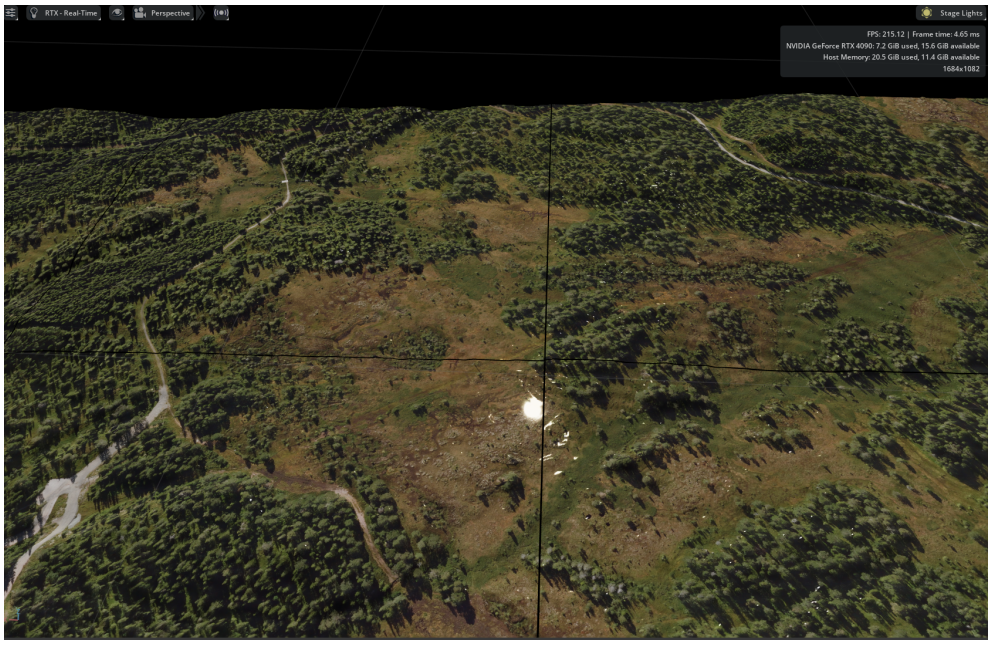


Figure 4.2: Gaps in the textures caused by complex code

Several modifications were made in the process of terrain generation from our initial model to the final version. In this initial model, the height map was interpreted by Pyvista as containing integer values instead of floating points, due to the utilization of integer values for the coordinates. As a consequence, the terrain had to undergo a smoothing process after being generated, resulting in an inferior product. The outcome of this smoothing procedure can be observed in Figure 4.3a, wherein the road does not display a consistently level surface throughout its entire path.

(a) First iteration of terrain model

(b) Improved terrain for a later iteration

Figure 4.3: Comparison of terrain model from first and second iteration

Furthermore, in this initial iteration of our model, we deliberately chose not to generate any extra data such as buildings and road outlines. Our intentions for this iteration were solely on testing if such large texture files could be implemented into Omniverse and get this up and running. However, in the subsequent

testing and evaluation stage, we observed that the stage felt empty and was missing some elements to create an immersive scene. This emptiness within the scene highlighted the importance of additional assets such as buildings for the digital twin model. Consequently, for the subsequent iteration, we elected to prioritize the incorporation of additional assets, mainly building meshes. Additionally, we aimed to investigate deeper into the problem we encountered with the gaps in the generated textures.

4.4 Creating a populated scene

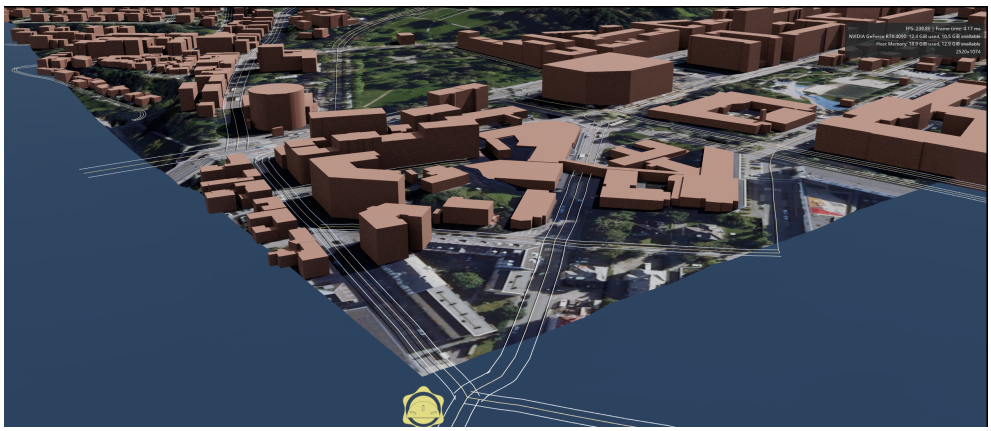
For this third iteration of creating the digital twin, our main goal was to populate the environment with diverse assets in order to address the identified emptiness observed during the testing and evaluation phase of the previous iteration. As a result, we collectively concluded that incorporating signs, road network outlines, and buildings into the digital twin would greatly enhance the experience, particularly from the perspective of autonomous cars, which has been our primary focus since the early stages of the project.

4.4.1 Building meshes

Figure 4.4 illustrates the building meshes create for our model. To make the scene more realistic and increase the feeling of depth we modeled the building mass of the selected area. The buildings are made from data retrieved from OSM's database, modeled, and placed in the scene. OSM's database contains detailed information on the buildings such as height and position, allowing us to create a model that is visually accurate, properly placed, and on a true-to-life scale.

The buildings still have a solid texture as a placeholder due to time constraints for creating a texture-generating function. The color can be changed in our script to the desired color that the user wants. It is also possible to generate roofs through OSM's data, although the function would need to be complex as the roofs can be of 16 different types.

Figure 4.4: Building meshes shown in Skistua digital twin



(a) Missing buildings near Gløshaugen

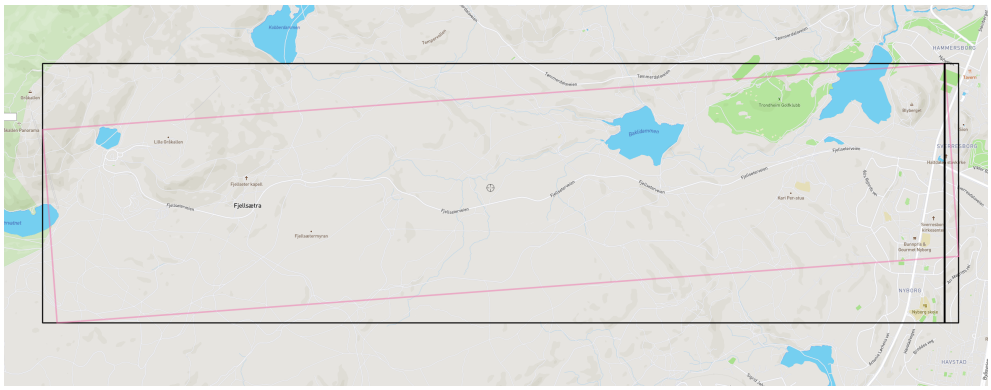
(b) Missing buildings near Skistua

Figure 4.5: Missing buildings

In our model, some of the buildings are missing along the edges of the model, as seen in Figure 4.5. This occurs due to the different coordinate systems used by Kartverket and OSM. OSM is using the geographic coordinate system WGS84 for selecting an area, while Kartverket is using the projected UTM coordinate system. Kartverket is also using a different datum called EUREF89. According to GeoData, this datum difference should only account for a maximum of 40 cm difference. They state that transforming between these is unnecessary as the transformation algorithm has an inaccuracy of +/- 1 meter[33].

That means our points should be in the right place geographically. This seems to occur due to how Kartverkets UTM based map is rotated compared to OSM's WGS84 based map. This can be seen in Figure 4.6a and Figure 4.6b. The pink extent in Figure 4.6b is Kartverkets extent on OSM's map. The perfectly horizontal extent has been skewed when represented in another coordinate system. It seems that the Overpass API creates a horizontal square in its own map projection. This interaction ends up creating an extent that does not include all the data in our model.

(a) Extent from Kartverket



(b) Extent from Kartverket compared to OSM

Figure 4.6: Visualization of how projection affects the look of our extent

4.4.2 Road network outlines

The road network is a fundamental component of the digital twin, especially when considering its perspective from a vehicle's point of view. A possibility to do this manually was mentioned in section 4.2, where we created a road network with Roadrunner and later imported it into the scene. However, we wanted to try to automate this process to improve the result and easier integrate it into our scene. This section showcases our investigation of automating the road network creation process, as well as presenting our findings.

The road network created for the digital twin has its basis from the NVDB where we retrieved the center line for each road in the specific extent obtained from the boundaries of the model. With this center line, we calculate the position of the parallel lines, which would represent the edge marking of the road, based on the width of the road. In the digital twin, the edge markings of the road network are visually represented by white-colored lines. whereas the center lines are represented in yellow. This color scheme was chosen to differentiate between the two elements in the road network and make it easier to interpret the road layout.

An example of this is shown in Figure 4.7, where a segment of our digital twin of Stjørdal is shown with the highlighted road markings for easier visibility. While the accuracy of these points is mostly right, there are some places where the positioning is slightly off, particularly when the road is divided into an odd number of lanes in each driving direction.

Figure 4.7: Road network visualized in Stjørdal Digital twin

4.4.3 NVDB signs

Much like with the road network, we felt that including road signs would improve the overall product. Furthermore, we aimed to assess the accuracy of the NVDB in providing reliable data. A way of doing this was to compare different sign positions in our digital twin against their real-world position. This is also further described in Case Study 3, subsection 4.9.2.

In order to add signs to our USD scene, we utilized the different numerical values assigned to each sign retrieved from the NVDB database. These numerical values are organized in ranges, an example of this could be that sign numbers in the range of 100 to 156 are hazard signs. By using these ranges we were able to categorize and create assets for each sign group, which could eventually be for each sign number instead of group.

In Figure 4.8, you can see the sign placement in our USD scene compared to the real world. This comparison shows the accuracy that the NVDB database can have in terms of positioning. The signs are placed in the same position as the real world, albeit with some minor offsets. As you move further from the main roads and city centers these offsets increase in some places. Since the items in the database are separated, it does not take into account items that are placed on top of each other, like in this case with the give way sign on top of the traffic light.

4.5 The final product

The final model we ended up with for this study is best characterized as a semi-automatic model. The model needs manual work for the process of retrieving the heightmap and textures from Hoydedata and NorgeIBilder. Once the necessary data was obtained, the user can input it into our programs Graphical User Interface and get a high-resolution virtual representation of any given location in a short amount of time. The representation includes building textures, road network outlines, and signs retrieved from the NVDB. An Omniverse USD scene was created from this data, making it easy to modify in later stages by incorporating real-time data. This scene holds potential for utilization in diverse applications such as simulations

In Figure 4.9, a visual representation of our model is presented, viewed from a top-down perspective. From afar, there are no big differences between the CityEngine-generated model and ours, but with a closer look you can see the details that come out when using our model and the textures from Norgeibilder, this is presented in more detail in subsection 4.5.1.

(a) Custom model view of sign

(b) Real world picture for comparison

Figure 4.8: Comparison of real-world and digital twin sign location

Figure 4.9: Custom model from top-down view

4.5.1 Mesh generation

Following the issues encountered with the textures during the previous iterations, we conducted an in-depth analysis of our code. As a result, we made the decision to abandon our previous approach and reprogram the texture generation process. This time with a deeper knowledge of the field and with a better understanding of the capabilities of Pyvista. In the final iteration, we adopted a new methodology where a single mesh was generated from the heightmap. Subsequently, we calculated the number of tiles needed to divide the texture into segments of 8000x8000 pixels. By identifying the intersections between these texture tiles and the heightmap mesh, we were able to accurately cut the mesh at these points. Resulting in a terrain mesh with a seamless texture, avoiding any gaps.

The mesh generated with our software is a more detailed model than what is created by most other digital twin tools. This is because our model is created with consideration of it suiting well for autonomous vehicle training, where details in terrain and textures are more crucial. A possibility with our model is the option to adjust the resolution for other purposes. This is possible due to the input data being completely customizable by the user with the data they choose from Norgeibilder or other sources. This adaptability makes our model a good foundation for many different use cases.

Figure 4.10: Custom model textures

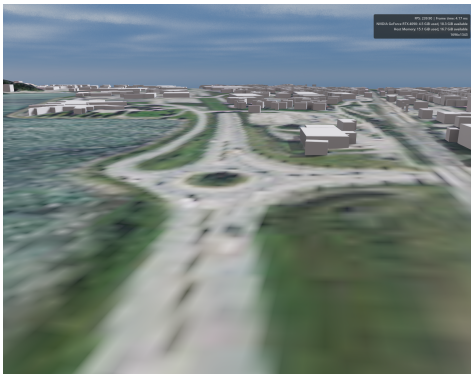


Figure 4.11: CityEngine textures

Figure 4.12: Side by side view of texture differences

As shown in Figure 4.12, a side-by-side view of the texturing of our digital twin compared to the one of CityEngine is presented. Our digital twin uses a texture that has a resolution of 8 cm per pixel, whereas CityEngine uses a resolution of 50 cm per pixel. For autonomous vehicle simulations, the degree of accuracy in a terrain model is a crucial determinant. Therefore, we utilized a point density of up to 5 points per meter for our models. This gives the terrain the closest representation to the real world, which leads to a more realistic model.

4.5.2 A visual journey through our final model

A video of the digital twin created with this final model can be found in the box below or at `https://youtu.be/EC5e0n1B5AQ`. The first part of the video shows a larger area outside Orkanger called Hemnekjølen. The area has a size of 4.8 x 2.4 kilometers and consists of a longer stretch of the E39 and slightly elevated terrain. The second area shown in the video is Gløshaugen in Trondheim. Here, a digital twin of 1.2 x 1.3 kilometers has been created that includes several larger building models such as NTNU Gløshaugen, "Studentersamfunnet", and Lerkendal stadium. There are also several road areas that are outlined with signs and road lines.

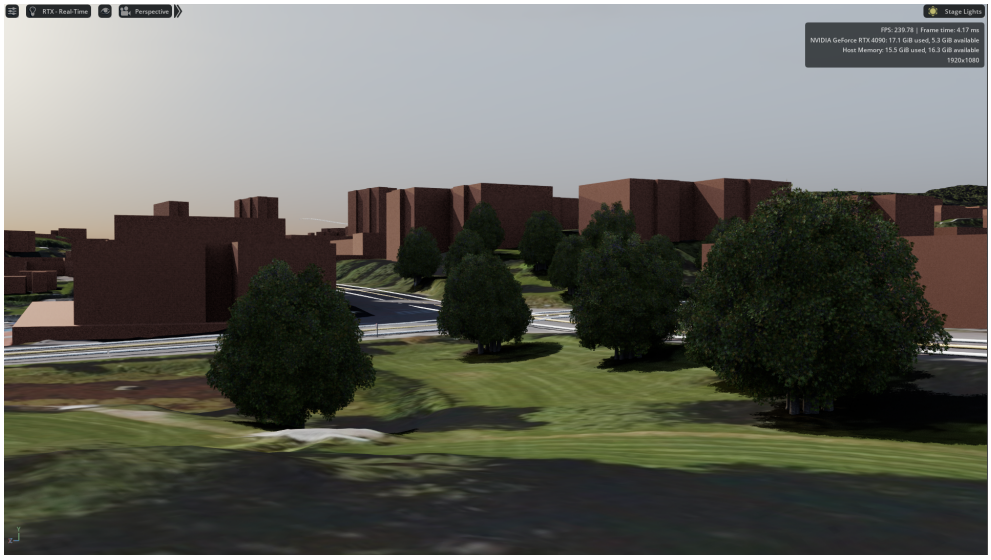
Digital twin - Hemnekjølen and Gløshaugen:
`https://youtu.be/EC5e0n1B5AQ`

4.6 Manual work added to the final model

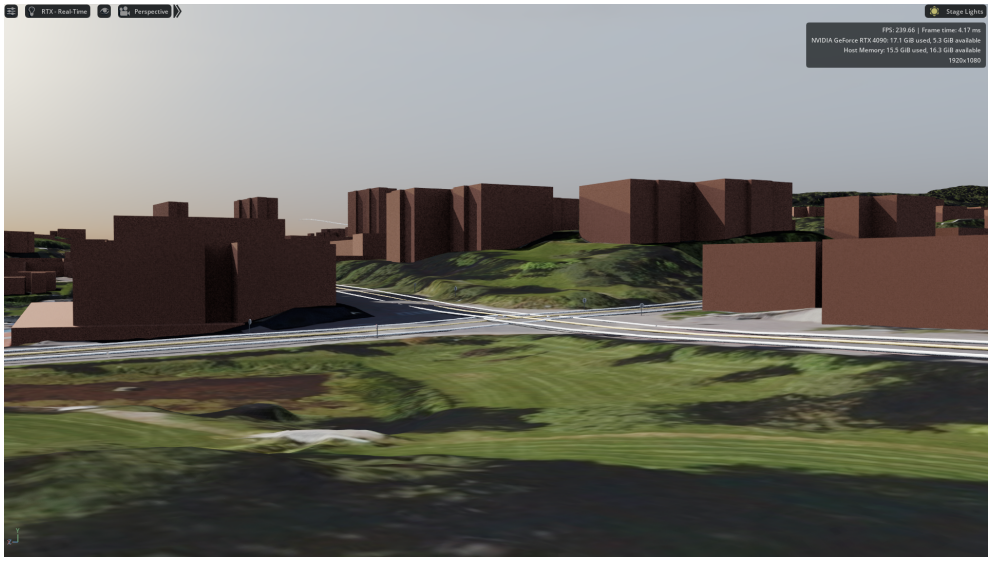
In our project, we experimented with using additional 3D models to enhance the final model's appearance. One of the key tools we used for this was the Omniverse Create paint-fill tool. Using this tool, we could select areas within our scene and fill them with various predefined and self-made assets. Another tool we employed was the Omniverse paintbrush, which enabled us to fill specific areas in our scene with these assets. We used this tool to include high-resolution assets such as trees and grass, making our scene more realistic. However, it is important to note that this process and these tools involve manual work and additional time to complete.

In Figure 4.13, it is possible to see the result this tool had on our scene. The left image shows the scene with the addition of different tree assets, creating a more realistic and immersive scene, while the right image lacks these assets, resulting in a less complete scene. All things considered, we believe that these extra assets are worth the time invested in integrating them into the scene as the rest of the model is automatically generated. We also believe that not every extra asset added needs to have a 1 to 1 correlation with a physical object as it is almost impossible to keep it consistent with the real-world model.

Another tool we used to increase the lifelike feel of our scene was to include an Omniverse environment. These environments can provide different lighting and backgrounds to any scene. Using these contributed to the overall quality of our scene significantly, making it more captivating for the viewer.



(a) Digital twin model with tree models



(b) Digital twin model without tree models

Figure 4.13: Comparison of digital twin with and without trees

4.7 From Static to Dynamic: the Impact of Dynamic Updates on Our Digital Twin

The second objective of this master's thesis was to include live sources and dynamic updates to our digital twin. This objective was successfully achieved by integrating a real-time stream of positional data obtained from a moving agent with our USD scene.

Utilizing the rospy client mentioned in section 3.3.1, we had the possibility of integrating this real-time data from the moving agent. This data was streamed from a vehicle with an interval of 0.1 seconds and included UTM32 coordinates for easting, northing, and height data. As this data was streamed to the database, it is processed through Python scripts which connected the data together with Omniverse. This is done by the Python script continuously reading data that is sent to the database. The script then converts the positional data from UTM32 to UTM33 as this was the preferred coordinate system used when creating our models. Thereafter, the angle of the car is calculated based on the previous positions retrieved and smoothed using an exponential moving average. The car object is then extracted from the Omniverse scene so the position and rotation can be updated.

Digital twin with live data from Skistua:
`https://youtu.be/_siQFzXiUvE`

Digital twin with live data from Stjørdal:
`https://youtu.be/lhuj_DFT3mc`

To visualize the integrations of real-time data in our digital twin we have created 2 videos from two different places. Video 1.1, which can be found at `https://youtu.be/_siQFzXiUvE`, shows an animation from the Skistua in Trondheim. After the video is finished with the overview, it transitions to a display of real-time data stream from a moving agent connected to Omniverse. The second video, which can be found at `https://youtu.be/lhuj_DFT3mc`, from this experiment visualizes Stjørdal Center much like the Skistua animation and finishes with a drive through the city. Both of the videoes can also be found in the box above.

4.8 Immersive Integration: Transforming our Scene into Virtual Reality

In our pursuit of enhancing our model and exploring new possibilities with our digital twin, we successfully integrated our scene into the virtual world. Through the smooth integration of VR through Omniverse's CreateXR application, it is now possible to walk around in our digital twin to see surfaces, models, and other aspects of our model. The immersive experience offers the user another point of view which in turn gave us information and insight to use to improve the overall product.

By following this link here `https://youtu.be/wUvBY6wMixg` or in the box below, you can witness our integration visually. The video takes you on a quick tour

through Holtermanns veg in Trondheim and gives an insight into the actual size of the models created. Additionally, the VR perspective highlights a noteworthy observation regarding the difference in heightdata retrieved from NVDB versus the terrain generated from our model. Specifically, the small gap between the road lines and the surface of our digital twin.

Experiencing the digital twin through Virtual reality
https://youtu.be/wUvBY6wMixg

That being said, it is important to acknowledge that there are some downsides to the use of Omniverse CreateXR. Firstly, in order to ensure a seamless user experience, the scene requires compressing the models used. Resulting in textures and visuals that are not as good when compared to the standards of Omniverse Create. Another limitation is that CreateXR is only compatible with USD files. As we faced challenges finding Python packages that supported the USD file format, we resorted to using GLTF files. To get our digital twin into the virtual world, you have to manually convert the assets to USD files, which can be time-consuming.

In conclusion, the integration of VR has expanded the use cases of our digital twin model and given us useful information. However, there are still some improvements that need to be made to our model and the CreateXR application for this to be beneficial.

4.9 Case studies

4.9.1 Case Study 1: Analysis of real-time car position data for in-depth insights

The first case study was conducted to see if it would be possible to interpret something more from the positional data collected from the moving agent. As mentioned in section 4.7, the ROS client gathers positional data every 0.1 seconds. If it had been possible to use this data for analyzing the previous trip, it would have given access to large amounts of data that could give a better result and improved interpretation. For this study, we utilized the ROSbags collected from the trips through Skistua and Stjørdal and converted these to CSV files. The CSV files contained approximately 4000 rows of data each.

After retrieving the data from the autonomous vehicle, we employed it to conduct an analysis of the driving patterns of the preceding trip. One way we did this involved assessing the vehicle's placement in relation to the center line of the roads traversed. By doing this, we gathered insightful data into the quality of the driving, which made it possible to draw further conclusions about driving conditions and the data collected.

The data collected gave an average distance from the center line of 1.74 meters when traversing Stjørdal center. This could indicate that the driver was reasonably placed in the middle of the roadway as an average lane is between 3 to 3.5 meters wide. It also shows that there may be opportunities to use the center and outer markings for other goals, such as being a contributing factor in autonomous driving. Additionally, to these measurements retrieved from Stjørdal, we averaged 3.09 meters in the same parameter over Skistua. The reason why the average was substantially larger in Skistua may be because of the inaccuracy in the data due to the dense foliage around the car blocking a good signal to the GNSS. This element is further discussed in subsection 5.1.1.

The results we got from this analysis do not show any direct outcome. However, they offer an indication of the potential for utilizing the acquired data in a broader context. One plausible application of this analysis and the road lines created lies in the use of it within a controlled environment, such as a factory or a construction site, where the lines and analysis could be employed to establish an autonomous transportation system. The reason these sites would suit this better arises from the fact that you can remove variables such as pedestrians and other cars.

A drawback of using this approach is the variation in the data based on if the road is a two-lane roadway or a single-lane road. The optimal way of doing this analysis would be to use the center point of each lane instead of the center line of the road. However, due to limitations in the level of detail provided by NVDB, the database we retrieve the road network from, this option is not possible. Nevertheless, NVDB offers very detailed information regarding the center lines, which enabled us to stay consistent with the data used across all locations. A top-down view of the car between the road lines created from NVDB is presented in Figure 4.14. The image showcases the accuracy of the database and our positional data retrieved from the moving agent, which strengthens our theory of the use of the markings for other purposes than only visually.

Figure 4.14: Top-down view of the car in between the center and outer road markings

4.9.2 Case Study 2 Digital Twin: Validation of Traffic signs' position and data in NVDB

The purpose of this case study was to take a closer look at the accuracy of the geographical coordinates and data provided by the NVDB. This was done by comparing the positions given by the NVDB against the real positions. The real positions from the sign were taken from multiple data sources such as Google Street View, our own Ortophoto, and footage retrieved from the car when recording data for real-time streams.

One thing we noticed in our case study of pinpointing the sign's geographical positions from the NVDB, was that the small offsets caused by the database had a greater impact on the scene as more signs accumulated around the same area. The signs also gradually shifted closer to the road's center point and, in some instances, onto it. This could cause them to look somewhat out of place when viewed from other angles than from the top-down view. This can be seen from the video in section 4.7 (`https://youtu.be/lhuj_DFT3mc?t=74`), where some of the signs appear out of place, especially in the roundabout and the main road later on. Besides this, a portion of the places we looked into, showed that the signs were positioned accurately with a few inches offset. This can be seen in the Figure 4.8 that was shown earlier from Skistua.

Figure 4.15: Start of the tunnel in Stjørdal with misplaced signs on top

Another thing you see missing when collecting data for signs from the NVDB is the height data and the orientation of the sign. As a consequence of this missing data, it is difficult to determine what rotation values the different signs should be assigned in Omniverse. The same thing applies to the height of each sign. This is especially evident for signs that were supposed to be underground in tunnels or elsewhere. An example of this can be seen in Figure 4.15 where the tunnel on E6 in Stjørdal begins. The signs that normally should be inside the tunnel, in the image marked by an orange outline, are placed on top of it since we retrieve the

height data from our heightmap which return the same value as the terrain.

All things considered, the concept of automatic sign generation through NVDB has great value, but the inaccuracy both in the placement and some missing data, leads to this solution not being good enough for the time being.

4.9.3 Case Study 3 Digital Twin: Road Surface Monitoring

This study aimed to investigate the feasibility of enhancing the digital twin by incorporating road damage information. To do this, a Deep neural network (DNN) model was created by Mamoona Birkhez Shami, which processes images obtained from a vehicle and outputs if road damage is discovered. The processed data were then sent to a dedicated database designed for this project.

To integrate this functionality into the digital twin, a Python script was developed. This script retrieves and processes data from the database, subsequently incorporating the newly identified positions generated by the DNN model.

The results of this study were the creation of a foundation that can be used to update and read information retrieved from different roads. Although a comprehensive test of the entire pipeline has not yet been conducted, it is expected that the results will be good as there is a substantial overlap in the foundation used for this and the code used for updating the moving vehicle shown earlier in section 4.7. A potential visualization of potholes and cracks in our digital twin is shown in Figure 4.16. Although these assets need a bit more fine-tuning and thorough testing, they would bring additional value to our digital twin.

Figure 4.16: Potential visualization of cracked road in our digital twin

4.9.4 Case Study 4 Digital Twin: Key elements to create a realistic and immersive digital twin

The final case study we did was to inspect closer what elements had the most impact to ensure a realistic and immersive feel for our digital twin. This study

was carried out throughout the whole process of creating the digital twin from the early iteration with CityEngine to our final model.

The first variable we noticed had a big influence on how realistic the model felt and looked when creating the digital twin was the terrain model. This is because the terrain takes part in both the full 3d model, but also in all additional assets that are added and changed after the creation of the scene. For this variable, we iterated over several different types of elevation maps and smoothing algorithms to ensure the best possible outcome. From our testing, we learned two important points. Firstly, it is best to utilize a highly detailed heightmap for optimal results. Secondly, it is crucial to ensure that PyVista interprets the provided values as floats.

A second thing we noticed played a big part in the digital twin is the addition of detailed objects. After we had created our digital twin, it was easy to see that it seemed empty when it was not populated with additional objects. To change this, we chose to automatically generate building meshes which improved the scene drastically. However, by conducting further research after including some objects we saw that the building meshes mostly affected city centers and not places like Hemnekjølen that only consist of some scattered houses throughout the scene. This is also possible to see from the real-time data stream from Skistua when moving away from the main road. An easy fix for this would be to utilize the Omniverse Paint Brush mentioned in section 4.6 to include detailed tree, leaf, and grass models.

Chapter 5

Discussion

This chapter will discuss challenges and other experiences we encountered during the project. To do this we will enlighten our research questions, which were established in section 1.2, and compare them against our work to discuss our findings.

5.1 Challenges and limitations

5.1.1 Real-time data stream

One of the major discoveries made in this project is the integration of real-time data from sensors and other objects into Omniverse. During the integration of this data, it was observed that the position data acquired from the vehicle had some flaws. The first issue arises due to the fact that the flow of data obtained with ROS uses a queuing system that is iterated through continuously. This data then needs to be processed and written to a database to be stored, this takes an amount of time for each iteration and leads to the digital twin being slightly behind where the physical agent is.

The second point is about the precision of the data being sent. This point has several factors that play into the accuracy, such as the measurements from the car and the surrounding landscape. The surrounding landscape affects how good the connection to the database is and whether it is possible to maintain a continuous flow of data. If this is not possible, there will be jumps in the data that will look unnatural in the virtual model. The measurements from the car are also a factor that determines the precision of each point. The position data measured from the car is taken from navigation satellites where the quality of the data can vary widely. This can lead to some unnatural movements that both affect the position of the car and the rotation it gets.

Aside from this aforementioned deficiency in the data, there are some other locations such as under bridges and behind thick foliage where the GNSS can not establish a connection to the satellite. Consequently, it will transmit a 0,0,0 positional reading to the database. These outliers caused by the inferior quality can be seen in a side-by-side view with the normal readings in Figure 5.1. This can

also be seen in the Youtube clip provided here: `https://youtu.be/u6zd2zNnL0E` where the car drives under a bridge in Stjørdal.

A way to fix these outliers caused by inferior readings from the car is hard to come up with as it is hard to tell which ones come from bad data and which ones come from actual bad driving. However, it is easier to find a solution to the problem that arises when the car loses connection to the satellite. Here, extrapolation can be used to predict the data points that do not fit in or interpolation to correct afterward.

(a) Outliers from the GNSS readings

(b) Normal values from the GNSS readings

Figure 5.1: Comparison of GNSS data

5.1.2 Limits of automation

Our method of creating a digital twin is not fully automatic. The user has to get their hands on Ortophotos and elevation data in the form of .tif files. Optimally this step would be something our software would take care of. This could be done by letting the user only type in a geographical extent, or select an area on a map, and the software would do the rest. Our solution right now works optimally for areas within Norway, which limits our solution due to challenges in accessing data from Kartverket and GeoNorge. The Ortophotos from GeoNorge require the user to have an account, and both services deliver the data through email. Furthermore, there is no direct accessibility to this data through an API. Instead, the user has to manually select the area in their GUI and undergo a delivery process that can last up to two hours. Unfortunately, there is a shortage of alternative data sources, however, we were fortunate to secure access to GeoNorge, along with its Ortophotos, which played a pivotal role in this project.

5.1.3 Buildings

The best comparison for our buildings is GeoDatas clip and ship service. When looking at the buildings that they offer in their service there is a significant difference in quality in the details. Unfortunately, this is not a free service and comes with the same caveat that the Ortophoto and heightdata have. To get the data you get sent an email with a download link, making it suboptimal for the creation of our automatic digital twin.

Building details

One of the things missing from our buildings is detail. Right now we generate the buildings from a footprint of the outline. This footprint then gets extruded to some height given. This works well, as the two-dimensional points in OSM capture the shape of the building very well. However, it falls short when looking at details like roofs, in our current model all the roofs are flat. For our areas chosen in Norway, this is seldom the case. A comparison from the same area can be seen in Figure 5.2. Here we see the difference of how the roof shape and building details affect the realism of our Digital Twin.

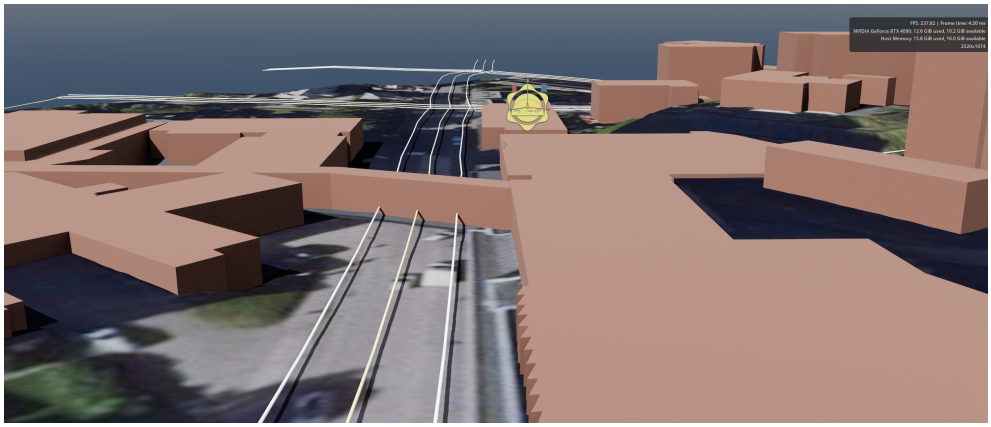
(a) Buildings from our Digital Twin

(b) Buildings from GeoData showing the difference in detail

Figure 5.2: Comparison of roof and detail differences

For some buildings or building parts, OSM includes the roof type of the building, as well as the roof height. This data could be used to implement a solution to create different roof shapes for some of the buildings.

Another detail that our model gets wrong is elevated walkways connecting buildings. This happens because we gather two-dimensional points and assign their ground height based on our terrain model. Consequently, when we extract the building footprints, these elevated walkways are also retrieved meaning there is no separation between the different parts. This leads to instances where building meshes inaccurately cover areas that are not occupied in real life. An example is shown in Figure 5.3.



(a) Footpath going over the road in our model

(b) The same footpath taken from google streetview

Figure 5.3: Side-by-side of how footpaths get created in our Digital Twin

Building heights

In our current model, we get the building height from OSM. This data can be extracted in three different versions. The first one being in meters, where we utilize this value to set the height of the building. The second version is the number of floors a building has. With this value, we calculate the approximate height of a building by multiplying the number of floors with a constant height value. This can lead to a small offset due to the variations in floor heights, however, it generally provides an acceptable approximation. The last value possible is the "None" value, indicating that there are no recorded values for the building. There are a noticeable amount of buildings with no height data, resulting in a less accurate representation of the building's shape. Figure 5.4 is an example of this "None"

value, which is taken from the gløshaugen campus in our model.

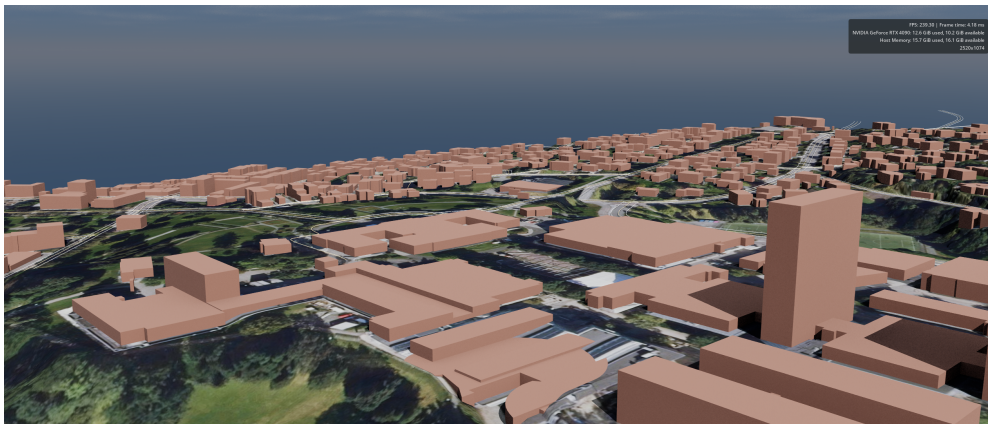


Figure 5.4: Height of buildings at Gløshaugen campus

When looking at sources like GeoData clip and ship we see that they have a more realistic model when it comes to building heights. This can be seen in Figure 5.5. GeoData has documentation on all the attributes they use for all of their buildings in thislink. In the documentation, there are several fields pertaining to height data, especially when it comes to the number of floors. This data is gathered from Eiendomsregisteret and is updated almost daily. This could be one source they use for their building height data.

Figure 5.5: Height of buildings in GeoData Clip and Ship

Another way to get the building height data is through Kartverket. Along with the elevation data we have used for our terrain model they also offer another form of elevation data. This is a surface model that includes elevation data for buildings, trees, etc. This can be used to set the height data of buildings.

Building textures

The biggest difference in our buildings from the real world is the lack of textures. One solution to this would be to adopt the same approach utilized by CityEngine, where textures are procedurally generated from a set pool. While this improves the looks of the model it is just an approximation of real life. The ideal solution, however, would be to use photogrammetry techniques to retrieve the real-life textures and integrate these with the corresponding building mesh in our scene.

5.1.4 NVDB assets

Signs

A current limitation of the integration of the signs is their unidirectional orientation, which fails to accurately represent real-world conditions. This inconsistency arises due to the incorporation of predefined assets into the digital twin without specifying the rotation. To address this issue, a potential solution could be to retrieve the road object from NVDB and link it up with the corresponding road segment to see whether the sign aligns or opposes the direction of road traffic. However, at present, we have not found a method to retrieve the cardinal direction of the road segment obtained from NVDB, making a comprehensive calculation impossible.

Another problem we encountered when implementing the signs into the digital twin arises from the positioning of some of the objects in the database. This can be seen in Figure 5.6, where some of the signs are slightly misplaced out to the side of the road. This occurred to objects that were further away from the center of the road or that were less accessible to the measurements that have been entered into the NVDB.

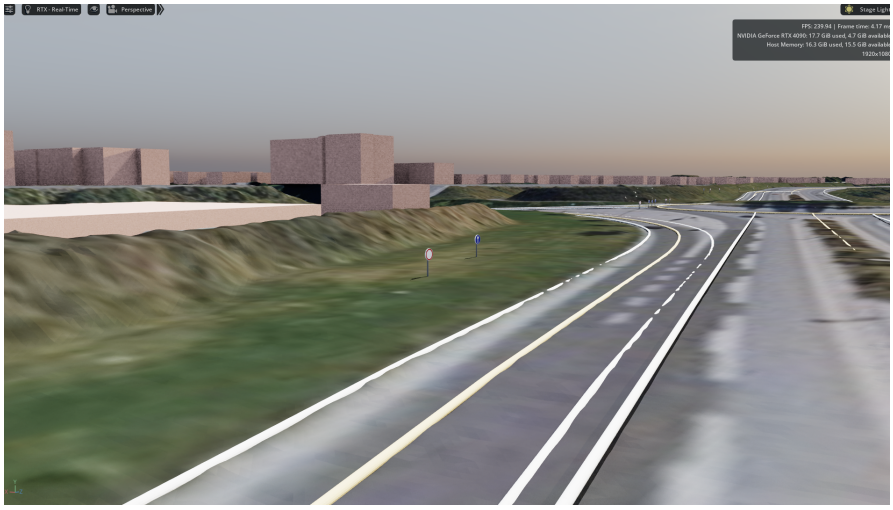


Figure 5.6: Misplaced signs in Stjørdal

5.1.5 Terrain and texture

There are several elements of the textures of the digital twin that can be discussed. The first one is how good the resolution of the texture must be to get a reasonable outcome for a digital twin. The second element is about the variables in Omniverse that affect how good the result of these textures will be. Both of these elements play an important role in answering RQ2.

Firstly, by using the 4-centimeter image resolution provided by `https://www.norgeibilder.no/`, you get a highly realistic model that looks good both on closer inspection, but also when viewed from afar. The downside with this is both the storage capacity that these 3d meshes need and the run time of our program. The USD file created is not that big since it just references other files. However, since the texture needs to be stored locally and split into smaller tiles for it to work with Omniverse, leads to us creating multiple GLTF's that are approximately 1 gigabyte. We recommend that the user looks into their specific use cases of the digital twin and review whether or not a 4cm resolution is needed. If the user is going to create a digital twin for larger areas than 1.5km x 1.5km we believe it is enough to use image resolution of 8-10cm per pixel. In addition to the storage capacity needed, the run time of our program greatly increases with such big files as it has to split up and generate meshes of large amounts of data. This is fine if you have plenty of time, but if you want a digital twin generated quickly, some quality will have to be forfeited.

Secondly, we need to discuss a big factor that plays into the texture. Through Omniverse Code's Python commands, it is possible to change variables for materials and other factors such as positional data for objects. This provides various opportunities to modify your scenes so that they seem more natural and realistic. This was done for the project on several factors, but when we tried to change the Index of refraction (IOR), the value that says how much reflection there should be in a material, it did not change. IOR is one of the most important factors for the texture of the digital twin because by default Omniverse's value provides far too high of a reflection that makes the texture look glossy. If you want to change this, you have to go in manually for each tile that the digital twin is split into and change the value in Omniverse UI. Because of this error with Omniverse, you have to perform a decent amount of manual work on the stage to get the best possible and most realistic result.

5.1.6 Load time and runtime

Throughout the project, a considerable amount of time has been used to look into the runtime and load time of our project. This is because we have seen a trend of these increasing, which makes sense as we have drastically improved the quality of both the terrain and the textures used, making the files created larger. As previously highlighted in the discussion, we believe it is important for the user to look at the use case of their digital twin before selecting the resolution. On average, the runtime for generating a digital twin with an area of 1.5x1.5

kilometers was 3 to 5 minutes. However, this increased drastically as the area increased. in the most extreme cases, it could take up to 35 minutes to create large digital twins. It is worth noting that these areas were so large they were only meant for testing purposes, but it shows the importance of considering the data used.

The load time of the model also increased when the models made were increased in size. A solution to this was found when we tested the digital twin in VR. Omniverse CreateXR only supports the USD file format, and when we manually converted our models to this file format, we saw a considerable change in the load time of the digital twin. Due to not finding any packages that support USD in Python, we chose to stick with GLTF and OBJ. However, it may be worth looking into Pixar and other packages in the future to optimize the loading time and storage of these objects.

5.2 Research Questions

In this section, we will go deeper into each of the research questions to answer them and showcase our findings.

RQ.1 - What parts of a digital twin can be automatically generated?
As described in section 4.4, there are several parts that we were able to automatically generate into our digital twin. Our research has made a significant contribution to the field by introducing the ability to automatically generate digital twins featuring varying resolutions in both textures and terrain. As a result, the digital twins generated through our software can be used for many different use cases, such as urban planning and simulations. In addition to this, our research has shown that it is possible to automatically generate building meshes, road networks, and semi-automatically generate road signs.

RQ.2 - What parts of the digital twin are most important to give it a realistic feel?
There are several parts of a digital twin that affect how realistic the final model looks. This is something we thought about during the entire process of creating our model, all the way from our CityEngine model, up until our finished product. The quality of the terrain model used was one main factor that affected how realistic the model looks and feel. The reason for that comes from using an inferior terrain model leads to transitions in the terrain that does not appear in the real world, making it look off and wrong to the viewer. Another factor that influences how realistic a model looks comes from our application of the digital twin, which is focused on autonomous vehicle training. This is particularly obvious when observing the road from a closer point of view, where smaller objects such as trees, signs, and railings are more noticeable. Using higher quality assets for these objects and focusing on placement of these can result in a better product.

RQ.3 - Can real-time data be streamed from a moving agent for dynamic updates in the digital twin?
In section 4.7, we showed the results we were able to achieve when it comes to

streaming real-time data from a moving agent. This showed that it is possible to set up a data stream from different data sources to Omniverse and thus dynamically update our scene with new data. Another feature that was experimented with was setting up a similar stream as the one used for the moving agent to update our scene with information about potholes collected from the roads traversed. By using the same method used to move the car in the digital twin, there should be a possibility to add new assets in real-time such as potholes.

RQ.4 - Can digital twins be used to store and visualize meta-data received from a smart sensor?
Through this project, we have shown that it is possible to integrate and visualize meta-data received from various types of smart sensors. Our initial data source was the NVDB, which provided comprehensive information on different road objects. Furthermore, we explored integrating data from smart sensors by employing a DNN to augment the information retrieved. This demonstrates that we can also store other data, like traffic lights, guard rails, and other objects in the digital twin. The data can be obtained from different sources, such as through an API or from other local software. It is also possible to visualize live data using an Omniverse Connector as well, as we have proven with our live car data.

5.3 Reflection

Throughout the semester, we utilized an iterative approach to continually improve our product. While this methodology proved effective for us, we believe that allocating more time to in-depth research at the beginning of the semester would have expedited the decision-making process. This is particularly evident in our acquisition of heightmaps. We selected the first method we came across, assuming it to be the best from Hoydedata. Consequently, we overlooked several settings that could have accelerated the creation of our digital twin.

For instance, we missed the opportunity to clip the heightmap to the extent we wanted. Instead, we got the entire map of the surrounding area, causing us to spend a lot of time using ArcGIS Pro to clip the heightmap to our selected extent. We also missed the highest-quality height map with our approach, spending a lot of time working with a lower-quality heightmap. When figuring out there was a better heightmap available this also caused us to change the code, as we could no longer assume all of the heightmaps were one pixel per meter.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The goal of this master's thesis was to enhance knowledge in the field of digital twin creation to create easier access to high-quality digital twins. In addition to this, there was a goal to integrate real-time data into the digital twin using smart sensors and other data sources to make dynamic changes to the model.

A way to automatically generate digital twins using Python scripts was developed to make the process easier. The results this program ended up having, compared to more traditional approaches, were remarkably good both in terms of visuals and terrain. That being said, the improved models led to a slightly longer run-time when opening and creating the digital twin as it incorporates larger and more detailed data. Generating the digital twins using custom scripts, allowed for more freedom in what could be added and what parts of the digital twin we wanted to dedicate more focus towards.

A method to include dynamic updates on the digital twin has been developed. Here, Omniverse's LiveSessions were used with Python scripts that read the data sent from a moving car to a database. Subsequently, the data were processed and used to alter the scene and move the car object in our digital twin. Additionally to this, there were performed tests for integrating real-time data from other sources.

All in all, our project has helped create a good foundation for improved, but also more accessible digital twins. The model can easily be further developed to integrate other data sources for generating other objects, as well as to improve the dynamic and real-time updates

6.2 Future work

As this master thesis has focused on integrating live data into the digital twin, we would say this would be a good way to proceed. Since the project showed that it is possible to integrate real-time data with Omniverse, a way forward would be to look at other sources that could provide value to the digital twin similarly. Examples of this could be the use of weather data to alter the digital twin or the use of other smart sensors. This would lead to a more dynamic digital twin that reflects the real-world object better.

A way to improve the digital twin scene to feel more realistic would be to include other 3D assets. As mentioned this project contributed to digital twin creation by providing a good foundation by creating a detailed terrain model automatically as well as integrating some 3D assets. But when viewing this from the driver's point of view, particularly in the Stjørdal video, you become alert to the missing 3D models seen in the real-world video. Adding guardrails and getting the signs to be facing the right direction can quickly make the model better. Another way to do this could be to take maps that are categorized into different segments in the terrain and use the Omniverse paint brush to automatically generate 3D assets into the scene.

Another part that would be worth looking into is using NVDB to generate OpenDrive road networks that can be used in simulations with CARLA description and other products. This would make generating sample data used for training autonomous vehicle driving agents easier and more accessible. In late September 2022, Omniverse and CARLA description entered into a collaboration, this has yet to show much, but this could lead to easier simulations and will certainly be worth keeping an eye on for further work.

There are also some improvements that can be made to our code. The first thing would be to rewrite the code but use asynchronous methods, which would help reduce the wait time we discussed earlier and improve responsiveness. The second thing that would be worth looking into would be finding an API to gather the heightmap and texture data. This would turn our model from semi-automatic to fully automatic making it easier for the user.

Bibliography

[1] IBM. 'What is a digital twin?' (2022), [Online]. Available: https://www.ibm.com/topics/what-is-a-digital-twin. last accessed: 10.05.2023.

[2] S. Ferguson. 'Apollo 13: The first digital twin.' (2020), [Online]. Available: https://blogs.sw.siemens.com/simcenter/apollo-13-the-first-digital-twin/. last accessed: 10.05.2023.

[3] A. D. E. I. Committee, 'Digital twin:definition value,' pp. 5–6, 2020. [Online]. Available: https://www.aiaa.org/docs/default-source/uploadedfiles/issues-and-advocacy/policy-papers/digital-twin-institute-position-paper-(december-2020).pdf?sfvrsn=b8a6cc93_2.

[4] G. Schrotter and C. Hürzeler, 'The digital twin of the city of zurich for urban planning,' *PFG–Journal of Photogrammetry, Remote Sensing and Geoinformation Science*, vol. 88, no. 1, pp. 99–112, 2020.

[5] S. Zürich. 'Open data zürich.' (2023), [Online]. Available: https://www.stadtzuerich.ch/opendata.secure.html#. last accessed: 03.05.2023.

[6] S. Zürich. 'Hochhaeuser.' (2022), [Online]. Available: https://hochhaeuser.stadt-zuerich.ch/. last accessed: 03.05.2023.

[7] Wingtra. 'Wingtra creates stunning digital twin of zurich.' (2021), [Online]. Available: https://wingtra.com/case_studies/wingtra-creates-stunning-digital-twin-of-zurich/. last accessed: 03.05.2023.

[8] Wingtra. 'Wingtraone gen ii.' (2021), [Online]. Available: https://wingtra.com/mapping-drone-wingtraone/gen-ii/. last accessed: 03.05.2023.

[9] G. of Singapore. 'Virtual singapore.' (2021), [Online]. Available: https://web.archive.org/web/20230308062728/https://www.nrf.gov.sg/programmes/virtual-singapore. last accessed: 08.03.2023.

[10] S. L. Authority. 'Virtual singapore.' (2023), [Online]. Available: https://www.sla.gov.sg/geospatial/gw/virtual-singapore. last accessed: 20.03.2023.

[11] M. Mauland. 'Ny tredimensjonal modell av trondheim gjør det enklere å planlegge framtidas by.' (2015), [Online]. Available: https://trondheim2030.no/2015/11/09/ny-tredimensjonal-modell-av-trondheim-gjor-det-enklere-a-planlegge-framtidas-by/. last accessed: 07.06.2023.

[12] L. Oftedahl. 'Oppdrag mjøsa får internasjonal oppmerksomhet.' (2022), [Online]. Available: https://nyheter.ntnu.no/oppdrag-mjosa-far-internasjonal-oppmerksomhet/. last accessed: 13.06.2023.

[13] M. Wang-Svendsen. 'Ny video gir innsikt om mjøsa-vraket.' (2023), [Online]. Available: https://gemini.no/2023/05/video-gir-innsikt-om-mjosa-vraket/. last accessed: 07.06.2023.

[14] ESRI. 'Arcgis pro.' (2023), [Online]. Available: https://www.esri.com/en-us/arcgis/products/arcgis-pro/overview. last accessed: 13.05.2023.

[15] ESRI. 'Arcgis cityengine.' (2023), [Online]. Available: https://www.esri.com/en-us/arcgis/products/arcgis-cityengine/overview. last accessed: 09.05.2023.

[16] I. The MathWorks. 'Roadrunner - design 3d scenes for automated driving simulation.' (2023), [Online]. Available: https://se.mathworks.com/products/roadrunner.html. last accessed: 09.05.2023.

[17] Nvidia. 'Nvidia omniverse.' (2023), [Online]. Available: https://www.nvidia.com/en-us/omniverse/. last accessed: 16.05.2023.

[18] Nvidia. 'Welcome to omniverse!' (2023), [Online]. Available: https://developer.nvidia.com/nvidia-omniverse-news-first-launch. last accessed: 16.05.2023.

[19] Nvidia. 'What is omniverse create?' (2023), [Online]. Available: https://docs.omniverse.nvidia.com/app_create/app_create/overview.html#what-is-omniverse-create. last accessed: 16.05.2023.

[20] Nvidia. 'Omniverse create xr.' (2023), [Online]. Available: https://docs.omniverse.nvidia.com/app_omniverse-xr/app_omniverse-xr/overview.html. last accessed: 07.06.2023.

[21] Nvidia. 'Nucleus - introduction.' (2023), [Online]. Available: https://docs.omniverse.nvidia.com/prod_nucleus/prod_nucleus/overview.html. last accessed: 07.06.2023.

[22] Nvidia. 'Nucleus - features and benefits.' (2023), [Online]. Available: https://docs.omniverse.nvidia.com/prod_nucleus/prod_nucleus/features.html. last accessed: 07.06.2023.

[23] H. Smith. 'Coordinate systems: What's the difference?' (2020), [Online]. Available: https://www.esri.com/arcgis-blog/products/arcgis-pro/mapping/coordinate-systems-difference/. last accessed: 05.06.2023.

[24] MapTools. 'More details about utm grid zones.' (), [Online]. Available: https://www.maptools.com/tutorials/grid_zone_details. last accessed: 12.06.2023.

[25] T. V. Do. 'Beidou navigation satellite system.' (2021), [Online]. Available: https://www.autopi.io/blog/beidou-gps-navigation-satellite-system/. last accessed: 13.06.2023.

[26] U. S. Force. 'What is gps?' (2021), [Online]. Available: https://www.gps.gov/systems/gps/. last accessed: 13.06.2023.

[27] ESA. 'What is galileo?' (), [Online]. Available: https://www.esa.int/Applications/Navigation/Galileo/What_is_Galileo. last accessed: 13.06.2023.

[28] Claudia.Prajanu. 'Glonass general introduction.' (2018), [Online]. Available: https://gssc.esa.int/navipedia/index.php/GLONASS_General_Introduction. last accessed: 13.06.2023.

[29] C. Heukelman. 'Leo vs. meo vs. geo satellites: What's the difference?' (2018), [Online]. Available: https://www.symmetryelectronics.com/blog/leo-vs-meo-vs-geo-satellites-what-s-the-difference-symmetry-blog/. last accessed: 08.06.2023.

[30] R. S. Brekke, 'Creating models of the real world in universal scene description,' M.S. thesis, NTNU, Jun. 2021.

[31] A. Knutsen, 'Gløshaugen's digital twin,' M.S. thesis, NTNU, Jun. 2022.

[32] NVIDIA. 'Omnilive.' (2023), [Online]. Available: https://docs.omniverse.nvidia.com/app_create/prod_extensions/ext_live.html. last accessed: 11.05.2023.

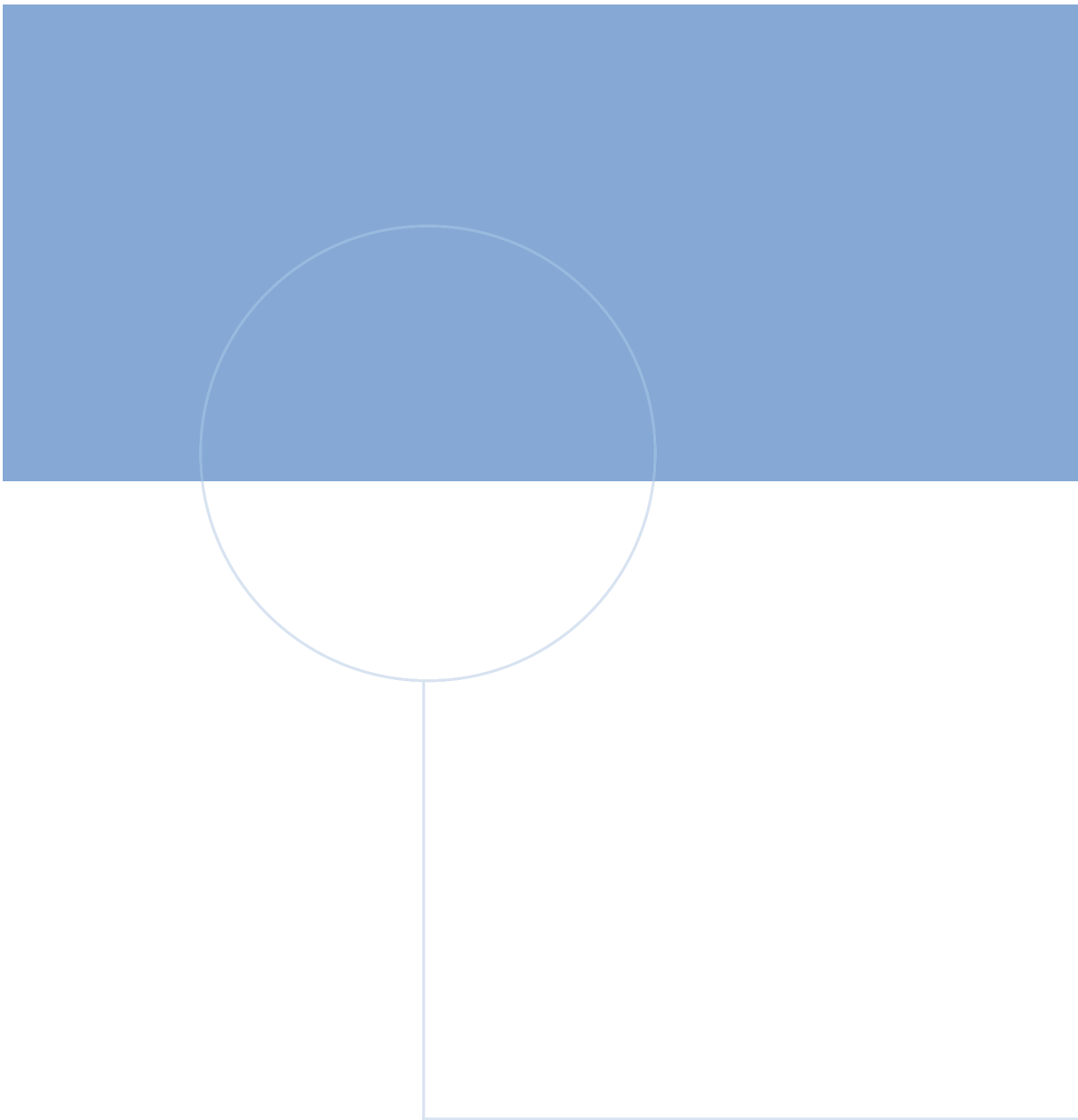
[33] GeoData. 'Datumtransformasjon mellom wgs84 og etrs89 (euref89) i arcgis for desktop.' (), [Online]. Available: https://geodata.no/guider/datumtransformasjon-mellom-wgs84-og-etrs89-euref89-i-arcgis-for-desktop. last accessed: 06.06.2023.

Appendix A

Additional Material

A.1 Code

Our source code can be found here `https://github.com/orgs/OmniverseDT/repositories`. Here we have divided the source code into two parts, DT_Creation and DTLive. DT_Creation contains the source code for the generation of the digital twin and DTLive has the source code for the real-time data stream connected to Omniverse. Note that the code needs to be connected towards either the connect sample from Omniverse or an Omniverse Code extension. More information about this can be found in the repositories or on Omniverse's pages.



NTNU

Norwegian University of
Science and Technology