University of
Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study Program/Specialization: Master of Science in Computer Science Secure and Reliable Systems	Spring semester 2021 Open / ~~Confidential~~
Author: Berke Kağan Nohut	
Supervisor: Ass. Prof. Naeem Khademi	
Title of Master's Thesis:Digital Tunnel Twin Using Procedurally Made 3D Models	
ECTS: 30	
Keywords: Procedural 3D Models Digital Twins Sensor Communications	Number of Pages: 89 + code in repository Stavanger, 15 June 2021

Faculty of Science and Technology
Department of Electrical Engineering and Computer Science

Digital Tunnel Twin Using Procedurally Made 3D Models

Master's Thesis in Computer Science - Secure and Reliable Systems by

Berke Kağan Nohut

Supervisor

Ass. Prof. Naeem Khademi

June 15, 2021

Abstract

A digital twin is a dynamic information model that provides real-time monitoring and controlling functionalities by connecting the physical and virtual environments. This thesis focuses on the road tunnel use case for the digital twins and aims to create a tunnel twin framework for a given tunnel.

Extending a previous work that creates procedural 3D models of the Norwegian road tunnels from the NVDB public database, this thesis builds connections to link the static models with the related information.

The developed application connects to NVDB and an emulation server that creates sensor data in the absence of the actual and parses them to the usable format in a database for the virtual twin to use. The virtual twin utilizes this information to reflect the real-time changes in the physical environment.

The digital twin framework is observed to generate twins for the given tunnel, create solid connections to the given sources, and the virtual twin component can reflect the changes. However, the developed application was working slower than expected, and optimization in database utilization is undoubtedly needed.

Acknowledgements

I would like to thank my supervisor, Ass. Prof.Naeem Khademi for his invaluable guidance and feedback.

I also would like to thank my family for their continuous support.

Contents

Abstract ii

Acknowledgements iii

Contents iv

1 Introduction 1

1.1 Motivation 1

1.2 Objectives 2

1.3 Outline 3

1.4 Key Concepts 3

1.4.1 Sensor Communications 4

1.4.2 Building Information Model 7

2 Related Work 9

2.1 Digital Twin 9

2.1.1 Design 11

2.1.2 Implementation 12

3 Methodology **15**

3.1 Idea 15

3.1.1 Use Case 16

3.2 Methodology 16

4 Design **19**

4.1 Communication 19

4.2 Resource 22

4.3 Database 22

4.4 Proposed Architecture 23

5 Implementation **26**

5.1 Broker 26

5.1.1 Broker Server 27

5.1.2 Broker Client 27

5.2 OPC UA Emulation Server 28

5.3 Digital Twin Framework 30

5.3.1 Broker Client 30

5.3.2 Influx Client . . . 32
5.3.3 OPC UA Client . . . 34
5.3.4 NVDB Client . . . 34
5.3.5 Digital Twin . . . 36
5.4 Virtual Twin . . . 39
5.4.1 Object Management . . . 40
5.4.2 Broker Client . . . 42
5.4.3 NDVB Object . . . 43
5.4.4 Database Manager . . . 44
5.4.5 Procedural Model Generator . . . 46
5.4.6 Interfaces . . . 47

6 Evaluation **51**

6.1 Experimental Setup . . . 52
6.1.1 Requirements . . . 52
6.1.2 Test Execution . . . 53
6.2 Visualization . . . 55
6.3 Known Issues . . . 59
6.3.1 Object Interaction Issue . . . 59
6.3.2 Information Panel Wrapping Issue . . . 61

6.3.3 Emulation Server VTS Information 61

6.3.4 Broker Server Exception 62

7 Conclusion **63**

7.1 Future Work 64

7.1.1 Features 64

7.1.2 Improvements 65

Bibliography **67**

A Appendices **71**

A.1 Images From Tunnels 71

A.2 Setup and Run 76

A.2.1 Minimum System Requirements 76

A.2.2 Required Programs 76

A.2.3 Required Libraries 77

A.2.4 Configuration 77

A.2.5 Execution 78

List of Figures

1.1 An example 3D BIM 8

2.1 Digital Twin 10

2.2 Digital Twin Design 12

3.1 Gantt Chart 17

4.1 Publisher Subscriber with Broker 21

4.2 Data Flow 23

4.3 Component Diagram 25

5.1 Digital Twin Execution Flow 38

5.2 Virtual Twin Component Hierarchy 40

5.3 Twin Manager Links 42

5.4 Nvdb Object Command Sequence 44

5.5 History Tool 48

5.6 BIM Model in Virtual Twin 50

6.1 Tunnel with Traffic 56

6.2 Car Fire 57

6.3 During Tunnel Emergency 57

6.4 Information Panel of Sign 58

6.5 Manually Attached Object Collider 60

6.6 Generated Collider 60

Chapter 1

Introduction

1.1 Motivation

It is a challenge for the industries to keep pace up with the rapidly growing trends in digitalization. However, instead of seeing this as a liability, the emerging software technologies can be used as opportunities to optimize resources and boost productivity.

As one of the mentioned emerging technologies, digital twins offer a sophisticated way of integration between sensors on industrial components with digital software. By creating a connection between physical and virtual systems, a digital twin provides accurate time monitoring and controlling functionalities [1][22][23].

The concept of the digital twin has been known for almost two decades[1]. However, the research on the topic just started to accelerate. Moreover, many digital twin software today were produced to tackle specific problems, and this lack of standardization[2][21] puts obstacles in the concept's development.

Digital twins require underlying real-time source data providers. These can be databases, APIs or sensors. Moreover, in some cases, these systems are developed together with the virtual components of the twin to make the

integration smoother. However, the necessary data can be collected from the existing databases, APIs, and the sensor data can be emulated.

A valuable data collection on Norwegian road tunnels can be found in NVDB publicly[24]. This collection indeed enables the digitalization of the Norwegian tunnels because of the digital twins' dependency on well-defined data collections. However, there is a lack of a framework and standardization in this field. Having identified this problem, the motivation of this thesis work is to create a digital twin framework for the Norwegian road tunnels.

1.2 Objectives

The primary objective of this thesis work is to develop a framework that is capable of instantiating a digital twin for any given Norwegian road tunnel.

Hence the work can be divided into milestones as follows:

- Studying, analyzing and understanding the state of the art of the digital twins, including their relationship with BIM models and their interfaces with 3D models.
- Studying the common sensor communication standards and establishing communication with systems such as SCADA-based systems to collect and send data from physical objects inside tunnels.
- Producing an emulation mechanism for sensor mentioned above sensor data to use in the absence of real data.
- Producing a framework that instantiates digital twins for road tunnels using the above-mentioned communication and connecting it to the interactable low to medium fidelity models that have been developed before.
- Creating prototype tunnel twins and documentation for the thesis work.

1.3 Outline

The thesis is consists of multiple chapters. The chapters are going to be structured as follows:

- **Chapter 2** presents several similar or related work to this project, and by describing and comparing them, several previous work from both academy and industry are introduced.
- **Chapter 3** is the explanation of the idea behind the project and the steps taken during the idea, design, implementation and evaluation phases.
- **Chapter 4** is dedicated to the design of the project. Design-related challenges and the architectural choices to tackle them are presented in this chapter.
- **Chapter 5** consists of the implementation details of the digital twin framework with all the consisting applications and tools.
- **Chapter 6** evaluates the current state of this work by providing test results and discussion.
- **Chapter 7** concludes the work by explaining the results and discusses the future work.

1.4 Key Concepts

This section presents the vital background material, which is included or closely related to this thesis work. Both the sensor communications and the BIM are strongly tied to this thesis. Therefore, they are covered briefly in this chapter before advancing into further chapters.

1.4.1 Sensor Communications

The technologies covered in this chapter are closely related to the communication parts of the digital twin work. Although some might not be directly included in the implementation, it is still essential to understand the basic connections among them because of their close relationships. Moreover, as mentioned before, a digital twin is an information model, and any method of providing additional data to the model should improve it in theory. Therefore, covering more ways of communication should provide valuable information for potential future work.

SCADA

Supervisory control and data acquisition or SCADA is a system that extends to multiple computers and systems. It is a system that lies at the software level to enable data monitoring and acquisition through integrated human-machine interfaces (HMI)[3]. SCADA has interfaces to low-level machinery through the utilization of programmable logic controllers (PLC) and proportional-integral-derivatives (PID). High-level commands that are received through a network can be executed using this low-level integration.

SCADA is an event-driven system. That is, the system only sends data through a network if there is a change in the data. This behavior optimizes network usage, and when compared to a system that sends data regularly, it better utilizes the bandwidth. Similarly, any digital twin integrated into SCADA systems will automatically adapt to this mechanism because the twin will only receive the data SCADA sends.

SCADA system consists of client and server applications. While the clients are responsible for the data display, the server applications manage the communication and data storage. In addition, SCADA systems often use components called remote terminal units (RTU) to receive remote commands. The communication between SCADA and RTUs is managed by the master terminal unit (MTU). The server application can act like an MTU, and in this case, as an addition to managing internal communications, it also manages external communication. However, MTU can also be a different unit.

SCADA systems also communicate with external tools or third-party solutions. Thus, there exists a need for standardization. SCADA systems follow standards such as OPC and implement clients for external usage.

OPC

Open Platform Communications (OPC) is a standard for telecommunications for secure and reliable data transfer for industrial usage[4][5] and is developed and maintained by the OPC Foundation. The purpose behind having this standard is to create an interface and standardize the flow between PLC protocols and end users. OPC is a series of standards that sets the rules on how the clients and servers interact with each other. It also includes the standards for the server to server communications. OPC standards abstract protocols that lie in the PLC level, such as ModBus. That is, any system that requires exchange messages with those protocols can use OPC messages to send their requests. Moreover, real-time sensor data access, alarm messages and the historical data related to the systems are also covered in the standards.

OPC UA

OPC UA (Unified Architecture) is a framework that unifies the OPC standards and implementations by extending and integrating the functionalities under a service oriented architecture (SOA)[7][8]. It has a multi layered approach that adds security to the existing OPC functions while not losing any capability[9]. Also, its framework enables new features to be included without affecting the already existing capabilities. OPC UA is, like its predecessor, a platform independent architecture. As the primary purpose behind having a standard is to ensure communications among systems from multiple vendors, this is an essential aspect. Several open source APIs for OPC UA are developed in many different programming languages[6], making it easier to integrate it into any system or software.

OPC UA supports binary and web protocols for data transmission[10]. Transport Control Protocol (TCP) is used for the binary protocol, and Hypertext Transfer Protocol (HTTP) is used for the web protocol. While

HTTP offers easier usage as various tools are available for direct usage, the binary protocol is desirable for better performance and small resource usage. For example, it is known that OPC UA is used in the software delivered to vegvessen related to SCADA monitoring[11]. Therefore, when a road tunnel digital twin software is considered, OPC UA communication is essential.

ModBus

ModBus is an open source communication protocol that is widely used with industrial devices as a manner of intercommunication [12]. It is generally used to connect the SCADA monitoring systems with plant computers. Maintenance and development related to the ModBus standard have been conducted by the ModBus organization since 2004 [13][14].

ModBus has a master slave architecture, where every device has a unique address in the network. The master ModBus device issues the commands, and the slaves respond to these commands and execute them. The commands include the unique address of the slave device and can be a reading or writing command for the data from/to the registers or reading I/O values from device ports. This communication can be done over serial lines such as Ethernet connections or a network transport layer using TCP protocol (ModBus TCP/IP)[15]. There also exists an implementation that uses User Datagram Protocol (UDP) protocol, which reduces the overhead TCP brings[16].

MQTT

The Message Queueing Telemetry Transport or MQTT is a messaging protocol. It is widely used for device to device communications. It runs over the transport layer and depends on a reliable protocol that runs under it. Therefore, it generally uses TCP protocol. It is an open source standard that is preferred in networks with a limited capacity[19].

MQTT uses a publisher-subscriber architecture, where a broker (server) manages the communication among the clients. This logic is managed with

a concept names topics[19]. Clients subscribe to these topics. When the broker receives a message, it forwards it to the clients that subscribed to that particular topic. A topic message with no subscribers to its topic is discarded. However, when a new client subscribes to that topic after a topic message is discarded, the broker sends the latest discarded message.

MQTT has three message types. Connect and disconnect messages are used to connect or disconnect a client with the broker. Moreover, the publish message is used for publishing messages. The messages also have an identifier named Quality of Service (QoS)[20]. This is to adjust resource usage.

- **At most once** sends the message and forgets about it. In other words, no acknowledgement message is required or expected.
- **At least once** is to ensure the message is delivered to the clients; the broker sends the message until an acknowledgement is received.
- **Exactly once** is to ensure there exist no duplicate deliveries, which can happen with the other QoS. A two-way handshake is initiated with the recipient client, and the message is passed only after that.

MQTT can scale easily as it uses little resources on top of the TCP (or other alternatives). Also, it can be used as a reliable communication manner in a network with few or many clients. This scalability allows it to be used with the embedded devices or usage in the sensor networks[17]. Moreover, extended protocols aim to improve measures against security concerns[18], hence making it more secure to use within sensor networks.

1.4.2 Building Information Model

Building information modeling is the process of managing a building from the beginning of its lifetime to the end. In order to achieve this, building information models (BIM) are created by experts, and the model is fed information throughout the project's lifetime [27]. The BIM is a computer file, which can be in many different formats. These formats can be proprietary, like RVT or NWG, or non-proprietary (IFC format)[26].

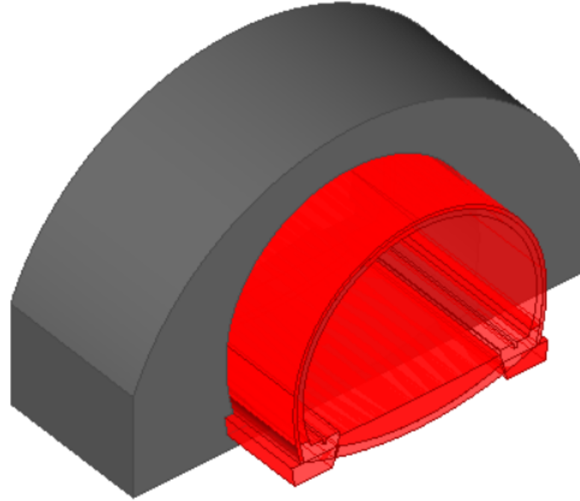


Figure 1.1: An example 3D BIM

A BIM model is usually represented with a dimension, and when presented, the dimension is noted. A BIM model can be 2D, 3D, 4D, 5D, or even 6D, where each dimension is stating a new information processing capability[28][29][30]. The meanings of these dimensions are explained below.

- **2D & 3D** Indicates the dimensions of the model.
- **4D** Time, events are represented on a timeline with gannt charts.
- **5D** Cost related information
- **6D** Building performance. For instance, sustainability, energy and safety aspects

Chapter 2

Related Work

This chapter will present an overview of the academic work about the digital twins and their relationships with sensor communication techniques and BIM. Research that was helpful for this thesis' progression or similar to this one will be discussed and presented throughout this chapter.

2.1 Digital Twin

The digital twin is a well-known and widely used concept. It has been in the literature for around 20 years, and there were many academic and industrial research and implementation[1]. However, with the growth in IoT interest and the increase in network speed and capacity, this already known concept has once again started to be popular.

A digital twin with the most exact words is an information model. It connects a physical entity to a virtual entity by storing relevant characteristics of the physical entity. This data binds the two entities together, hence the name: digital twin. The twins share the same characteristics, and they behave according to this data. That is, a change in any twin is reflected in the other, creating a compelling concept.

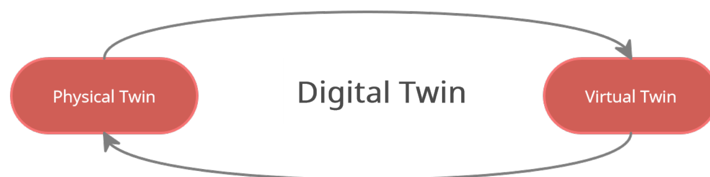


Figure 2.1: Digital Twin

The digital twin may have multiple connections to the physical object. It may communicate through sensor communications, or it may have connections to APIs or databases that can deliver relevant characteristics of the physical entity. Digital twin's capabilities are closely related to its ability to collect data. That is, the closer it is to collect data in real-time, the better it can represent the physical twin's behavior. Moreover, the design of the twin can be made to collect only the data that serves a specific use case, or it can simply collect all the data that can be collected.

The core features of a digital twin allow storing, monitoring and manipulating the physical entity using its virtual twin. It also enables its users to investigate the historical data of any characteristic stored. Therefore, there can be many use cases of a digital twin. The virtual twin can be used in any smart building or workplace to optimize production by closely monitoring the data. It can assist workers during construction or maintenance work. Also, as it reflects the real-time data, it is a natural environment for simulations. Particularly, this thesis is concerned about the road tunnels, and the above-stated use cases apply for the tunnels as well. In this case, the virtual twins can be used as safety rescue applications to assist their users in times of crisis during tunnel accidents. Moreover, digital twins can be used in simulations with different areas to focus on. Digital twins can be used to simulate the behaviors of any component in a system. They can be used as a testing workbench to support and verify the decisions at any point[33].

When it comes to the design and implementation of digital twins, there are no standards that directly cover digital twins[2]. However, some details are undoubtedly essential during a digital twin's design and implementation phases for optimization and scalability. Related literature and industrial work about the design and implementation of the digital twins will be dis-

cussed in the subchapters.

2.1.1 Design

The architecture of any software is designed to realize specific use cases, and when it comes to the design of a digital twin, the same scenario applies.

Digital twins are used for many different use cases[22]. Hence it makes it hard to discuss all the different design principles. Therefore, it is decided to proceed with the simulation use case; as in the following chapters, the proof of work and the evaluation of this thesis will proceed with simulation scenarios.

One of the most common use cases for digital twins is the simulations. In order to simulate every behavior and characteristic of a physical environment, the virtual twin requires vast amounts of data to work with. Although managing all the aspects of all the physical components in an environment is a hard job and might even be infeasible, designing an architecture to enhance the simulation aspect of a digital twin might be enough. In [32], it is discussed that the main principle of the digital twin is to use the information from the existing IT systems and use the collected data to make information models available for any simulation in the future.

Digital twin-based simulation, its usefulness, and its effects on reducing costs are mentioned [34]. It is once again implied that specific digital twin designs can allow the digital twins to work in simulation use cases.

The given examples and much more indicate that the digital twins' hierarchy in a product should be designed as individual software with solid connections to the existing systems.

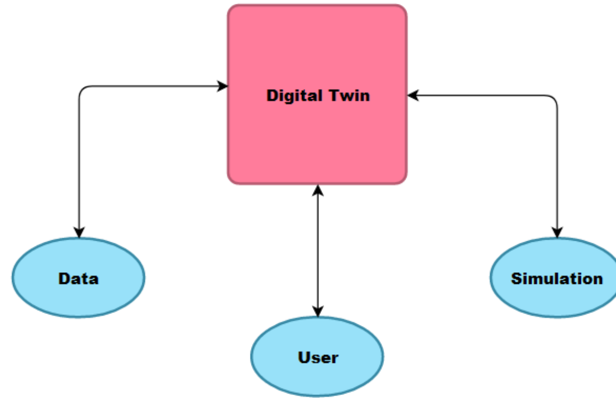


Figure 2.2: Digital Twin Design

In a simulation use case, the mentioned three aspects play significant roles in the design phase of a twin[35]. It is argued that the simulation component brings the analysis and optimization, while user interaction is needed for an accomplished implementation. As mentioned earlier, a digital twin relies on the underlying data and must be firmly connected to all the IT systems and the mentioned aspects.

2.1.2 Implementation

Some other use cases for the digital twins are maintenance, production and construction. Some of the existing digital twins for the road tunnels will be presented to present real-life examples of digital twins.

Tunnel Digitalization Center by SCAUT

The tunnel digitalization center offers a real scale representation of the road tunnels[38]. The approach they took is to model the existing structure, design BIM, and integrate the various sensors into the BIM. The digital twin contains all the information and models as well as the information

from their disorder identification algorithm, which uses an AI.

The work aims to contribute to tunnels' maintenance and construction phases by introducing predictive maintenance by analyzing the disorders beforehand and basically indicating the wrongs in the system.

Although the system they have contains similarities to this thesis' idea, the approach taken for this work when it comes to modeling is to model the existing structure manually. This project, however, relies on an existing procedural 3D modeling. Therefore, having the constraint of being procedural and dynamic on all sensor, information or object management. Moreover, the disorder identification AI is out of scope for the current work.

Tunnelware.io

Tunnelware utilizes VR to aid the construction of tunnels[37]. The system uses 3D models and sensor data to manage and analyze events that are happening in a tunnel. BIM is also used in their system to increase the control and visualization over the tunnel.

Similar to the Tunnel Digitalization Center, the aim is mainly to contribute to the construction and maintenance phases of a tunnel. Therefore, similarly to the comparison above, this thesis work differs from it, as its approach aims to be dynamic and procedural.

Koningstunnel Den Haag by Infranea

The project in [36] shows many resemblances with the current work. It combines parametric modeling and laser scanning to model objects in Autodesk Revit. BIM engineers create the specifications for installations, and the 3D models are placed. Moreover, VR technology is used to increase user interactivity and enhance visualization. Their solution not only aims to contribute to the planning and maintenance of tunnels, but it is also further developed to train administrators for emergency scenarios.

Still, the scope of this thesis is different as it aims to create a digital tunnel

twin for all the given Norwegian tunnels instead of explicitly designing for a particular one. Moreover, the used parametric modeling technique requires the utilization of laser scanners. This work only aims to use existing data sources like nvdb to create the twin models.

Chapter 3

Methodology

This chapter will present the idea behind this project and the steps taken to overcome the work needed. A potential use case will be defined, and it will be used as an example scenario in the following chapters.

3.1 Idea

BIM models are widely used to contribute to the construction phase of road tunnels. Moreover, there are BIM models that are used to keep track of the maintenance data. It allows its users to store and access current and historical data as well as to monitor the status of the tunnel.

Likewise, a digital twin can be used for the construction phase of a building or keep track of its maintenance status. Although these technologies are both similar in purpose, as they are both information models, a digital twin by definition has to have automated ways of updating its model. When developing digital twins, the general principle is to develop both physical and virtual twins together to ensure the compatibility of the twins. This might be applicable for the buildings that will be constructed, but having a digital twin for a tunnel that was constructed, is another case.

Depending on a previous work that builds procedural 3D models from

NPRA data in NVDB and another work that enhances the models by adding interactivity, this project aims to have automated ways to create and update information models of Norwegian tunnels. In other words, the idea is to have a framework that can create a digital twin for any Norwegian road tunnel.

3.1.1 Use Case

The road authorities are constantly monitoring roads for maintenance purposes. The SCADA systems combined with the sensors on objects in tunnels provide an information flow, and people evaluate this information to decide the actions regarding it. Although this flow is built, tested, and used for years, it mostly works on HMI screens and information panels. Therefore, it lacks a 3D perspective. On the other hand, the building information models are 3D models, some are even 4D, but they also lack the dynamic information flow.

Therefore, combining this information with the future possibility of a two way information flow, which is explained in the future work section, this thesis proposes a real-time road tunnel monitoring and simulations use case with the digital twins.

3.2 Methodology

The steps that will be followed to create the idea mentioned above will be explained in this section. As explained in chapter 1.2, the project has multiple milestones. A Gantt chart that shows the timeline of the work is presented below.

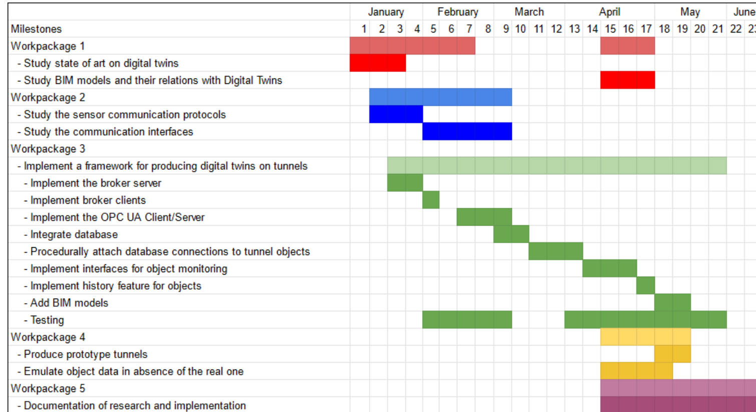


Figure 3.1: Gantt Chart

It can be observed that the mentioned milestones have been placed in work packages and given reasonable time slots each. The work will start by doing a series of research and then will be continued with related implementations. The framework for the digital twins is the main objective, and all the implementation is related to it. The architecture of the framework will be explained in detail in the next chapter. Testing is an integral part of any software product, and it should not be taken lightly. Hence, after almost every implementation step, a time slot will be allocated to test the implemented component and for regression testing. Finally, the report part of the project will mainly be conducted at the end. However, weekly reports will be produced through the project to ease the final report's workload.

The entire development was carried on by using Scrum methodology with two week sprints. Moreover, the progress was monitored, supervised and tracked from an issue board, in which the sub-components of the mentioned milestones can be found.

Finally, the project's main objective, the digital twin framework, can be examined in four components with the respective development order.

- Internal Communication
  - Broker Server
  - Broker Clients (Virtual Twin and Digital Twin)
- Sensor Communication
  - OPC UA Emulation Server
  - OPC UA Client
- Database
  - Influx Server
  - Influx Client (Virtual Twin and Digital Twin)
- Virtual Twin
  - Unity Project

Chapter 4

Design

The proposed idea suggests a framework for digital twins. That is, the final product should be able to instantiate multiple twins for different tunnels. In order to support this idea, it is crucial to assess the needs of instantiating one digital twin before coming up with an architecture.

This chapter will discuss the important architectural decisions and will present the project architecture.

4.1 Communication

The digital twin must connect to one to many APIs, databases or servers. However, getting the information from the mentioned sources passes through sending requests and receiving the desired information. Hence, there is not much to decide.

On the other hand, the communication between the digital twin and the virtual twin can be implemented using different strategies. One approach would be data polling, which is, in the simplest terms, regularly asking for the data. However, the virtual twin needs to be the real-time reflection of the sources, and it should send requests to fetch the data as close as to the real-time even though the data has not changed at all, which might

be unoptimized considering multiple tunnel twins with hundreds of tunnel object twins in each.

The second option is to have an event-oriented strategy. One example of this is the publisher-subscriber design. In which the subscribers wait for an event to fetch the data. That is, this design reduces the unnecessary update requests by notifying the subscribers when the data changes and telling them to fetch. Furthermore, in a publisher/subscriber pattern, there is no need for a publisher to know anything about its subscribers. Therefore, there can be multiple virtual twin instances that listen to the same digital twin. This creates room for some future work, which will be explained in a later chapter.

There are many ways to implement the publisher-subscriber pattern. However, when their topologies are compared, it is seen that two options are available. The first one is to rely on a communication broker to manage the traffic in between. On the other hand, the second one does not require a middleman because it uses IP multicasting strategies. Data distribution service, for instance, is an example of the latter[31].

The virtual twin needs to display all the changes, and considering the worst case scenario, brokerless option relies on multicast messages to be transmitted. However, this would require a wrapper over UDP to make it reliable. Instead of allocating resources to that, a message broker system that works over TCP was decided to be implemented. The publisher-subscriber design with a broker program in the middle allows the virtual twins to join and leave the notification channel and receive notifications.

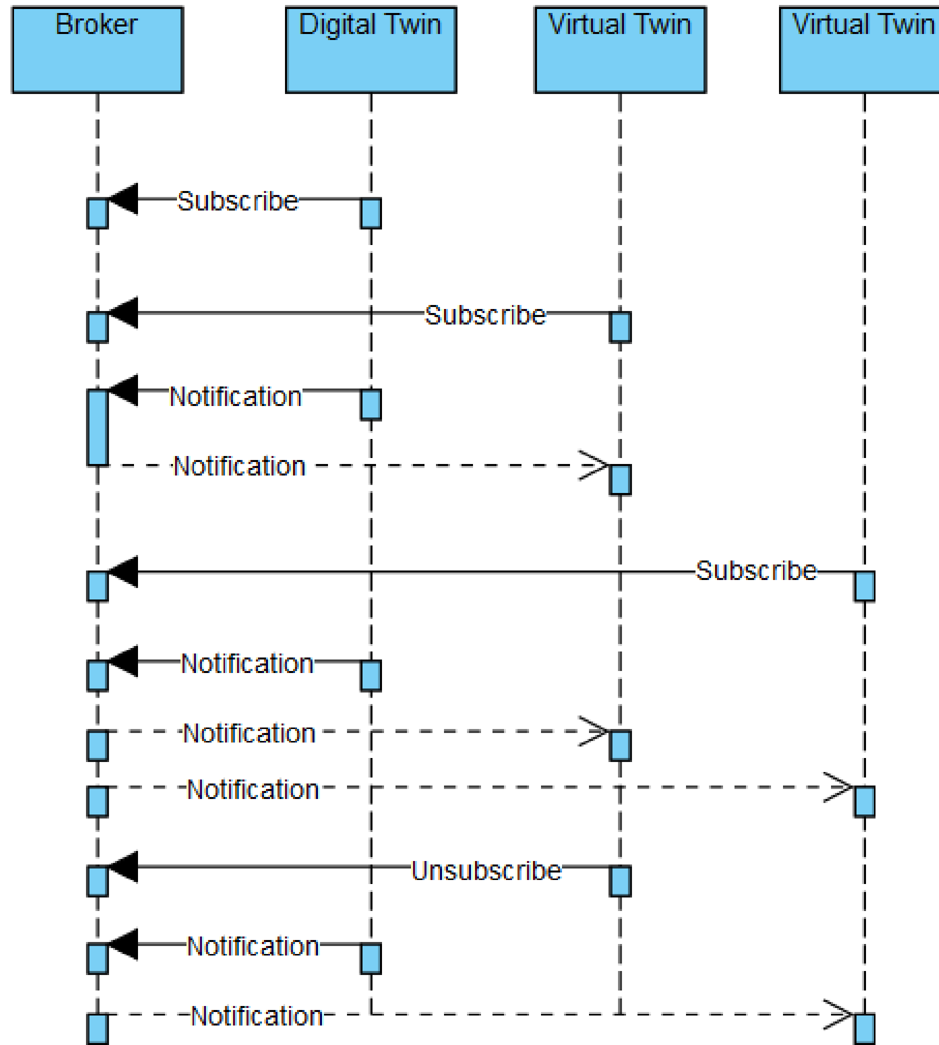


Figure 4.1: Publisher Subscriber with Broker

4.2 Resource

The virtual twin depends on the procedural models for the twin to display the collected data. Although generating the low to medium fidelity procedural models is not costly in terms of resource usage (GPU, CPU or storage), forcing multiple twins to instantiate on the same machine can slow the entire process for not only one but all the twins. Moreover, the procedural model generation depends on Maya 2020 software, which may not necessarily be accessible on the host machine. Also, the models can be generated beforehand, and in this case, it would be more optimized to store the models on a server machine rather than the host. Therefore, the correct approach would be to separate the framework from the model generator.

4.3 Database

Thirdly, it is vital to understand the database operations are always costly and is tend to be the bottleneck in many software. Also, due to the nature of the digital twins, the data must be written and read as close as in real-time. This will end up with the database receiving a series of operations. Moreover, once again, because of the nature of the digital twins, historical data must always be accessible. That is, the database has to store data that is constantly growing, as the twin will continually update its data with new insertions instead of updating the existing to keep the history. Therefore, it is essential to choose a database that is specialized for this purpose. Hence, a time-series database is a highly suitable option.

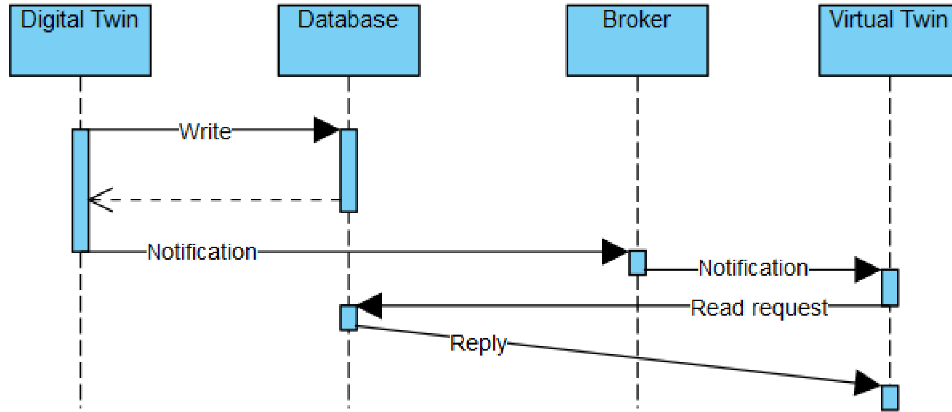


Figure 4.2: Data Flow

4.4 Proposed Architecture

When all the assessments are combined, it was decided that a distributed architecture would be suitable for this project. The framework should run either on one or in separate machines. It should also support running in multiple instances. As the broker server can handle the communications between publishers and subscribers in the same network, a project running on multiple devices would not create a communication issue. A future work that enhances this architecture will be explained in a later chapter.

The digital twin framework includes instances of internal client components. These clients are used for internal or external communications and are all have different purposes. For instance, the broker client is used exclusively for broker communications.

The framework initialization is done with respect to the tunnel data from NVDB. If no data found for a tunnel or the NVDB connection fails, the system cannot work as the unique IDs of tunnels or objects cannot be found or linked throughout the components. Therefore, a working nvdb client is

the system's dependency.

The project includes an emulation server that creates, updates and publishes sensor data information. The OPC UA client communicates directly to this server and regularly fetches relevant data. Although the real-time sensor data flow starts from this server, it is not a dependency of the system as the digital twin can successfully run without fetching sensor data.

A database client is used to handle database read/write operations. The entire purpose of the digital twins is to create an automated real-time information model, where a user can access data from any time during the twin's lifetime. Without having an active database connection, the very purpose of the project fails. Thus, the framework is dependent on an active influx connection.

The broker server, OPC emulation server, the procedural model generator and the virtual twin are the other components of the system. They can all sit in different machines in the same network. The only exclusion might be the virtual twin as the display or interaction related operations are executed there. Therefore the user should at least have access to its host machine. However, it still is designed as a different component from the framework and has no direct link or dependency to it.

The proposed architecture of the entire project is presented in the following component diagram.

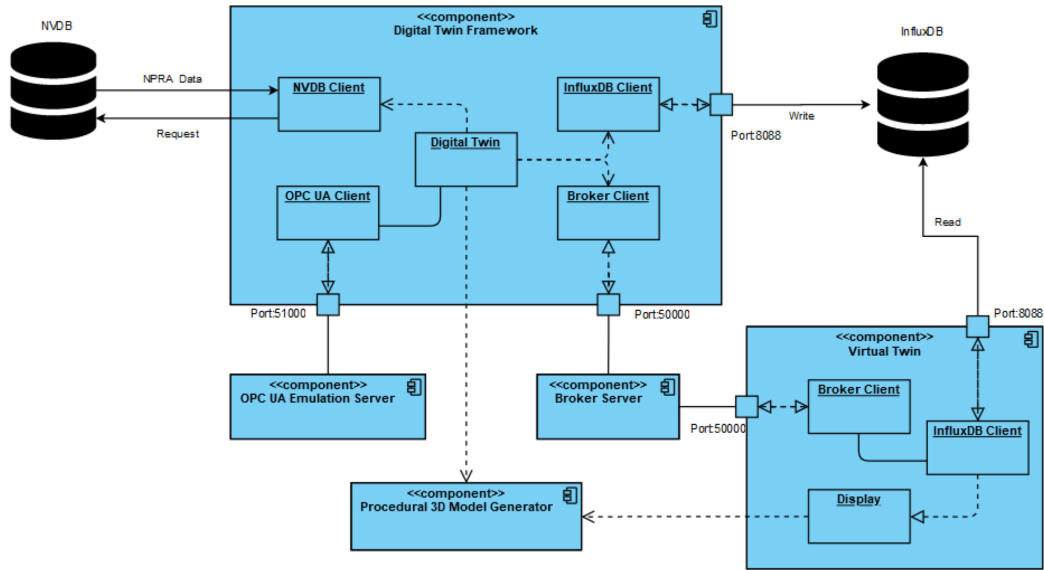


Figure 4.3: Component Diagram

The next chapter will discuss the steps that have been followed during the implementation phase of the project. Each of the previously presented components and their interactions among themselves will be explained in detail at the software level.

Chapter 5

Implementation

This chapter is dedicated to explaining the implementation of the entire project. The digital twin framework and the individual external components of the project will be presented in more detail. Diagrams, charts and code snippets are included in the chapter to provide a better understanding of the code and the flow of the project.

The chapter is divided into four sections, in which an individual component is presented.

5.1 Broker

In order to apply the previously discussed publisher-subscriber design pattern, there was a need to implement a program to handle the internal communications.

The broker program is divided into two components, the server and the clients. Although this section will present both, it must be noted that the clients are not individual components of the server and thus will be used in other components.

5.1.1 Broker Server

The broker server is a multi threaded TCP server that stores the subscription information of its subscribers. The subscription information contains the communication string of the client and a subscription channel id. The channel id is used as a key to group the subscribers and the publishers into the same channels. Therefore, these groups can be used to determine which subscribers should receive a notification from a publisher.

The broker server can be configured before launch to set its address and port, and there can be given a limit for the number of clients that it will serve.

It manages the connections by passing each client connection to a worker thread, which responds via three messages:

- Subscribe: It is used to subscribe to a channel. Other types of messages from a client will be discarded if it is not yet subscribed. Therefore, this must be the very first message to be sent by a client.
- Unsubscribe: This message is used to leave a channel.
- Notify: This message should be sent to a server by a publisher whenever it makes a change in the tunnel data. The notification will be forwarded to the subscribers of the same channel by the server.

5.1.2 Broker Client

Broker clients were implemented into the digital twin framework and to the virtual twin to enable their communication. Like the broker server, the broker clients are TCP clients that send the aforementioned messages to the broker and listen to the notification messages from the server. The entire communication flow is visualized in Figure 4.1. The details for each broker client will be presented later in their respective sub sections in the digital twin framework and virtual twin sections.

5.2 OPC UA Emulation Server

The digital twin framework is designed to be able to utilize OPC UA TCP protocols to communicate with the OPC UA Servers. However, the lack of an accessible OPC Server with actual tunnel data dictates the implementation of a mock server. The emulation server is designed and implemented to initialize an OPC UA server with configurable data. Moreover, in order to make it closer to the real-time data, a document explaining sensor data in Vegvesen[39] is used. Furthermore, knowing that the sensor data changes in the real world, time-based emulation on the data has been implemented. Lastly, as an improvement to the configuration of the sensor data, a config writer code has been implemented to create automated configurations for a particular tunnel and all the tunnel objects inside.

The configuration file contains multiple groups of fields. The first group is for the server configuration and contains the information for the server connection. Like any TCP server, the emulation server requires a host machine address and a port to bind to.

The second group controls the randomness and the occurrence of the emulation and contains three different variables with their values being between 0 to 100.

- **Emulation Frequency**: Controls the interval of emulation. An emulation frequency of 0 means there will be no emulation at all, and the data will be idle. A frequency of 1 will mean there will be an emulation every 100 seconds, and a frequency of 100 will mean the data will be emulated every second.
- **Data Change Frequency**: Controls the likelihood of an OPC node change. As there exist multiple objects that can be emulated in a tunnel, this variable is needed to create randomness of which object changes. That is, having a value of 100 means that all the objects will change at each emulation iteration, while a value of 1 means that every object will have 1/100 chance to be changed. Lastly, a value of 0 means there will be no changes at all.
- **Node Break Frequency**: Controls the likelihood of an OPC node to be broken. In the real world, there is a chance of a physical device

being broken and not receive further communications. It is desired to have such a case in the emulation, but its chances are separated from any other value to keep it more real.

The last group is to create OPC nodes and their data. It is represented in two variables in the config.

- Node: Stores the id of the object to be emulated.
- Variables: List of the variables that the node contains.

The OPC UA server contains an object node, which stores all the node data in a server. The emulation server creates a new namespace with every object id and puts that object's variables under that namespace. Later, the namespace is added to the object node.

Algorithm 1 Initializing Nodes in OPC UA Emulation Server

Require: $opcua.Server$, the OPC UA server
Require: $object_id$, tunnel object id from NVDB
1: Get $object_node = Server.Object_Node$
2: **for each** $Node \in Config$ **do**
3: $namespace \leftarrow server.register_namespace(Node.object_id)$
4: $object \leftarrow object_node.add(namespace, Node.object_id$
5: **for each** $var \in Node$ **do**
6: $object.add_variable(var)$
7: **end for**
8: **end for**

The stored nodes contain multiple variable fields. These are generated according to the configuration file. The config file provided with the project contains variables that are inspired by the actual data from Vegvesen. For instance, every node contains a heartbeat count variable, which stores an integer between 1 to 99 or a cycle variable that controls the interval for each heartbeat message. More variables can be added to these by modifying the config from the config writer script, which sends API calls to NVDB to retrieve object ids and their type ids. For instance, a tunnel lightning object can have any nine digit object id but must have the id 86 for its type[25].

Therefore, it is possible to detect the object types before generating a config for them. However, the new fields should still be added to the code and the emulator similarly to the existing ones for the current version of the code.

Finally, the emulation is controlled by the variables that have been explained, and timer threads execute the emulation. Every time an emulation iteration runs, the code discovers all the nodes individually and starts a separate thread for each group. For example, every lightning object is emulated by one thread or every sign is emulated by another one. Therefore, the emulation cycles are not dependent on each other. Moreover, the heartbeat counts are increased by one every cycle seconds, and these cycles can be different for each node. Hence, during the initialization of the heartbeat emulation thread, nodes are grouped by their cycles and each group is controlled by a distinct timer thread.

5.3 Digital Twin Framework

This thesis's primary motivation and objective is to create a digital twin framework that can instantiate digital twins from their ids in NVDB. Therefore, the following implementation has been done to enable the statement above.

The digital twin framework includes multiple components, which were initially presented in Figure 4.3 with the component diagram of the system. This section will address the implementation details of the internal components of the framework and their relationships.

5.3.1 Broker Client

Although it was mentioned in section 5.1.2, the implementation details were not presented as different implementations include some differences. This sub section will cover the implementation of the broker client implementation for the digital twin framework.

The broker client is a multi threaded TCP client. Therefore, it has the capa-

bility to connect to a server and send messages to it while actively listening to any responses from the server. This is a crucial capability for not only this but also any broker client in the entire system. Although the system does not emulate two way communication in the current implementation, it is an important future capability. Hence, the broker clients have to support it.

The broker client is created during the initialization of the digital twin framework. The framework reads the broker config file to learn the broker server's address. Then, it creates a broker client instance by providing a tunnel id. The broker client connects to the broker server and creates a dedicated thread to communicate to the server and, more importantly, not to block the framework.

An out message queue is used to send messages to the broker server. Moreover, in order to be able to detect and send these messages, the implementation uses a timeout. This allows the processing of the event queue.

Algorithm 2 Broker Client

Require: socket, Active TCP socket
1: Initialize $socket.connect(Broker)$
2: Set $socket.timeout \leftarrow \varepsilon$
3: **while** true **do**
4:     **if** $q \neq \{\}$ **then**
5:         Send $q.front$
6:         **if** timeout **then**
7:             Pass

The broker client uses the same messages with the broker server. These messages are passed to its out queue by the framework.

Subscribe is sent just after the instantiation with the provided tunnel id, and notification messages are sent whenever the database changes. Lastly, when the framework application terminates, it closes the TCP socket. Thus, the unsubscribe message is sent then.

5.3.2 Influx Client

The framework uses a database client to communicate with the influx database that the system uses. Similar to the broker client, it is created during initialization and destroyed before termination. Moreover, it is created according to its configuration file and with a tunnel, id to determine the database to be used.

The influx client creates a connection to the database server when instantiated. Secondly, it tries to create a database with the provided tunnel id if it does not exist already and switches to it. In other words, it starts to use the freshly created or already existing database.

```
1  def __init__(self, name, host, channel):
2      self.client = InfluxDBClient(host=host[0], port=host[1])
3      self.channel = channel
4      self.database_name = "Tunnel_" + channel
5      self.create_database()
6
7  def create_database(self):
8
9      databases = self.client.get_list_database()
10     if self.database_name not in databases:
11         self.client.create_database(self.database_name)
12
13     self.client.switch_database(self.database_name)
```

Influx uses measurements instead of tables, which are generally used in relational SQL. Moreover, it needs tags and fields to store a value in a measurement. They are stored as JSON objects. For instance, the length of the tunnel can be stored as:

```
1  {
2      "measurement": tunnel_id,
3      "tags": {Tag_name:Length},
4      "fields": {Field_name:Value}
5  }
```

The digital twin should always store the historical data. That is, any new data that goes into the system should be inserted instead of getting updated. Therefore, the client implementation only has select and insert methods for use. On the other hand, update and delete methods exist, but they are not implemented.

The select method is implemented to retrieve only the latest value of all the tags existing in a measurement. As explained, the digital twin never changes existing data. Therefore, it is done to optimize the selection process. However, any historical data can be retrieved by providing a timestamp to influx as influx adds a timestamp to data if none have been provided during insertion.

Insert method requires a tuple of list of dictionaries to work. Although it looks unnecessarily complicated at first, both NVDB and influx communicate with JSON, and there is no need to convert the data type at all. Therefore, simply parsing the nvdb or other api data to tags and fields would be enough to insert them into the database. The first item of the tuple contains all the tags that will be inserted in JSON or dictionary, and the second item will contain the corresponding fields. Therefore, no additional parsing is required.

```
1  def insert(self, id, characteristics):
2
3      tags = characteristics[0]
4      fields = characteristics[1]
5      json_body = []
6      for i in range(len(tags)):
7          json_body.append(
8              {
9                  "measurement": id,
10                 "tags": tags[i],
11                 "fields": fields[i]
12             }
13         )
14
15     isOk = self.client.write_points(json_body)
16     if isOk:
17         return True
18     return False
```

5.3.3 OPC UA Client

Similar to the previous clients, the OPC client is created during the initialization of a twin and destroyed with the twin. Also, it has its configuration to pass the address of the OPC UA Server for it to connect. This client is used to connect to the previously discussed OPC UA emulation server in section 5.2, and its primary role is to parse the data from the emulation server and return it to the digital twin for it to store in the influx database.

It contains a pull method to fetch data from the server it connects. The method receives an object id as an argument. This id is used in the emulation server to create namespaces.

The method retrieves the namespace array from the server and compares it with the argument. When the correct namespace is found, it pulls all the nodes and converts node name, value pairs into tags and fields to make it easier for the twin to insert into influx.

Algorithm 3 OPC UA Client Pull

Require: Client from opc_ua, OPC UA Client
1: Fetch, $Namespace_Array \leftarrow Client$
2: **for each** $Namespace \in Namespace_Array$ **do**
3:     **if** id Namespace **then**
4:         **for each** $Node \in Namespace$ **do**
5:             $Tag \leftarrow Node.name$
6:             $Field \leftarrow Node.value$
7:         **end for**
8:         Return All [Tag, Field]
9:

5.3.4 NVDB Client

NVDB component is used to fetch characteristic data from the NVDB API for the framework. Unlike the other clients, it does not have its class and only contains function definitions. The primary purpose of this implementation is to fetch the core data of the tunnel. The major characteristic information for the tunnel itself and its tunnel objects are retrieved by API

calls to NVDB, and all the ids are stored and used throughout the lifetime of any digital twin instantiation. The first API call contains characteristics and the children ids of the main tunnel. Secondary calls are made to the children with the received ids. The framework treats these ids as unique items. That is, the object ids in the initial pull from the NVDB will have their own measurements in the database, and the OPC server will be queried to fetch data with their ids.

The digital twin is used to link the data from any source to the procedural models. Therefore, the nvdb client implementation is made to link the API database to the digital twin. Moreover, periodic calls are made to the API throughout the lifetime of the digital twin. The period of these calls can be configured from the configuration file specific for similar intervals.

The main flow of the component can be summarized in three steps:

1. Fetch tunnel with its id
2. Manage the tunnel characteristic data
3. Repeat first two steps for all the children objects
4. Repeat first three steps every **T** time

The tunnel characteristic information is parsed accordingly to the influx client requirements. The data is fetched and separated into tags and fields. For instance, if a tunnel length is to be parsed, the tag becomes the length, and the field becomes its actual length value. Later, the tag and fields are inserted into their corresponding measurement.

```
1  def parse_characteristics_data(data):
2      fields = []
3      tags = []
4      for field in data:
5          key = field['id']
6          value = field['navn']
7          tags.append({key: value})
8          fields.append({"value": ...
               value+":"+str(field['verdi'])})
9      return tags, fields
```

5.3.5 Digital Twin

The last part of this section is dedicated to the instances of digital twins. In other words, this subsection will explain the entire flow and discusses the steps of the digital twin lifetime, using the previously explained components.

A digital twin is created by starting the digital twin framework with a tunnel id. Then it reads a configuration file to determine some critical paths for third party applications, such as procedural generation program or Unity, or the virtual twin. Important paths are:

- Procedural Generation Script Path
- Procedural Generation Output Path
- Python Path
- Unity Path
- Unity Project Path
- Virtual Twin Path

When a digital twin is initialized, it uses the tunnel id to create the following components respectively.

- Broker Client
- Influx Client
- NVDB Client
- OPC UA Client

Later, it launches its awake method, subscribing to the broker server through the broker client, and starts the NVDB initialization as described in section 5.3.4. After the API calls are done, it starts two timer threads for the OPC UA client to pull the sensor data and for the nvdb client to make another

request, respectively. Therefore, these timers will ensure that a regular data flow feeds the digital twin throughout its lifetime.

Throughout the lifetime, whenever a data change is discovered by the twin, the influx client is passed the fresh data, and the broker client is notified.

If a twin is ever destroyed, destroy method is called to stop the internal components respectively. That includes stopping the broker, database and OPC clients and the timer threads.

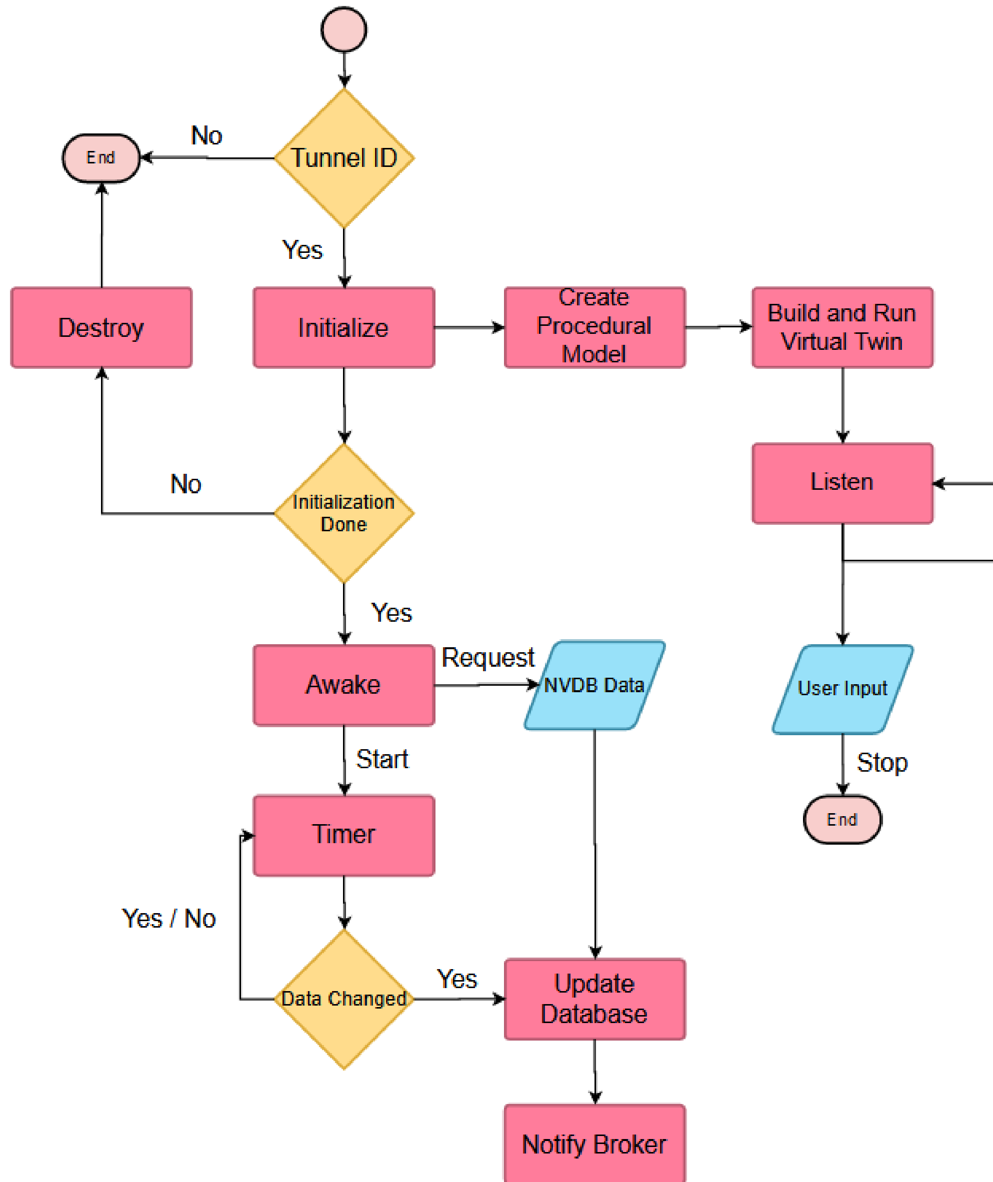


Figure 5.1: Digital Twin Execution Flow

Simultaneously with the component initialization, the digital twin framework makes a call for a procedural model generation. The procedural model

generator handles the generation process and places the model for Unity to discover, and the model is built into the virtual twin application. The build is called using Unity bash command by the digital twin framework.

```
1 os.system(UnityPath + " -quit -batchmode -projectPath " + ...
      UnityProjectPath + " -executeMethod Build.PerformBuild")
```

The reason for a build is to enable the procedural model to be used in the virtual twin is that unity does not support real-time model loading. Therefore, the model is built inside the application to make it usable in the virtual twin. However, as a digital twin does not planned to be restarted often, this is not a costly process.

After the build is done, the virtual twin is launched, and the main framework thread is blocked. If a user stops the twin framework, the destroy process starts to kill these processes in a controlled way.

5.4 Virtual Twin

The virtual twin component is one of the core applications of the entire distributed digital twin framework project. Previously, this project was started to demonstrate the procedural models that were generated as the output of another thesis. Later, it was upgraded to make the tunnels and tunnel object interactable and make the tunnels more realistic.

The scope of this work, however, is to create real-time representations of tunnels. That is, the interactable procedural models from before had to be enhanced one more time to connect them to a real-time data flow.

The virtual twin implementations include minor and major modifications to the existing software as well as the addition of internal components. The components that fell into the scope of this thesis and therefore modified or implemented are visualized below.

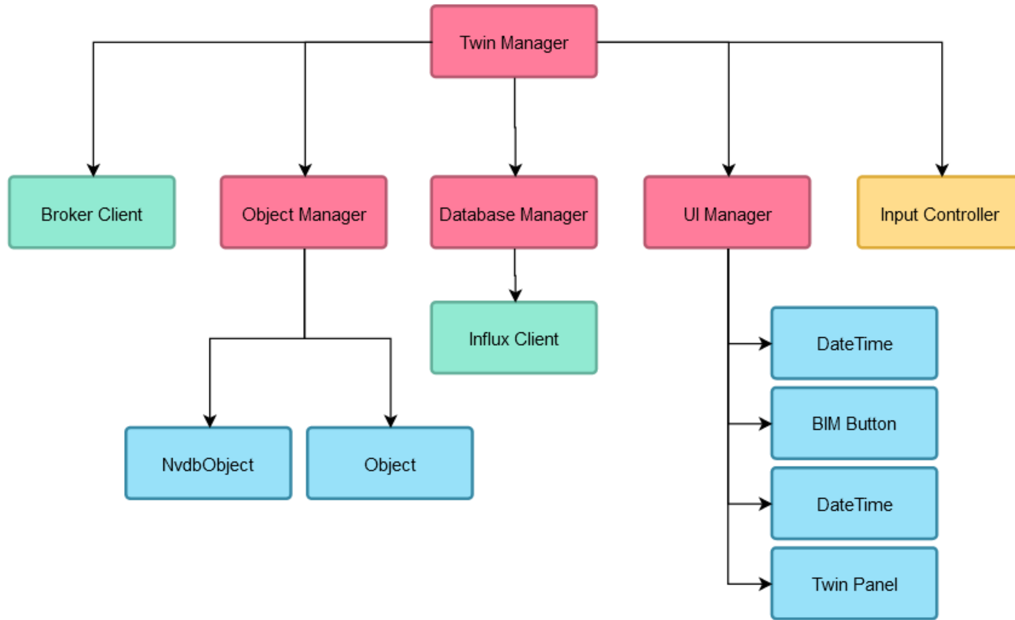


Figure 5.2: Virtual Twin Component Hierarchy

The following sub sections will discuss the implementation details, respectively.

5.4.1 Object Management

The object manager is possibly the most important component of the virtual twin. Its main role in the whole implementation is to discover, initialize and instantiate objects. The virtual twin project depends on the previously generated procedural model. The model is created by the procedural model generator using Maya2020 and placed into the virtual twin by the digital twin framework. Although the implementation of the procedural model generator is beyond the scope of this thesis, the generated model should be explained briefly to explain this and the following components better.

The procedural tunnel model is composed of multiple 3D objects. Those

are walls, roads, parking shoulders and tunnel objects. Although the model includes graphics for walls and roads, it only creates colored boxes for the objects. This issue was addressed and resolved in the previous work using the object manager class while making the same objects interactable. The object manager discovered the tunnel objects from their names, initialized their positions and rotations and instantiated new graphics.

In this thesis' scope, the object manager is again used to add more functionality to the tunnel and its objects. The NvdbObject class is used to abstract the new functionality, and the class is attached to the relevant objects procedurally. During the discovery, the object manager checks for the objects inside the tunnel graphic and attaches a script to all of them.

```
1 private void InitializeObject(Transform tunnel, Transform obj)
2 {
3     ...
4     obj.gameObject.AddComponent<NvdbObject>();
5     ...
6 }
```

Additionally, the main tunnel object itself is an essential component for the virtual twin because the digital twin can obtain information regarding the tunnel as well. In order to be able to show this information in the virtual twin, the object manager class not only attaches the NvdbObject class but also registers it to the Twin Manager class as the main tunnel.

The object manager handles the initialization of each tunnel object individually. However, the entire virtual twin system also needs to be managed to eliminate the internal synchronization issues. In other words, no object should be looking for data that do not exist yet, and a class should be responsible for that. The Twin manager class is built to address this issue. It is a singleton class, and by its definition and implementation, it only has one instance per a virtual twin program. It is configured to start as the first instance, and static links are provided for it to store all the relevant manager and controller instances.

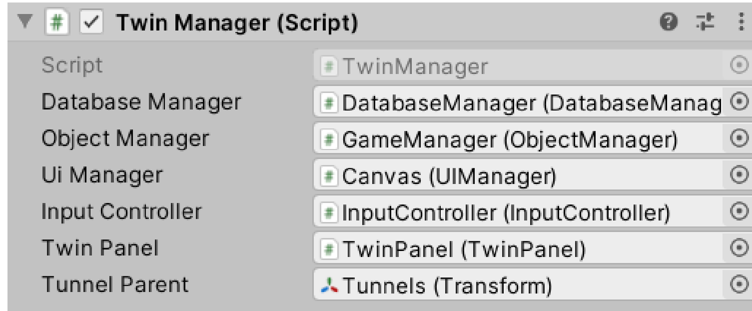


Figure 5.3: Twin Manager Links

Additionally, as the procedural tunnel model does not exist before the digital twin framework, the TunnelLoader script is used to create and place it under its predefined Tunnel Parent object, which can be seen in Figure 5.3. After the object manager discovers and registers the tunnel, the twin manager instance store it as well. Similarly, a BIM model can be placed under virtual twin resources by the digital twin or manually to start an additional sequence to load a BIM model to the scene. Only in this case, a BIMLoader instance will create the BIM model, place it in its predefined position and register it to the twin manager.

The virtual twin relies on a procedural model to be built beforehand to launch. Basically, the graphics are needed to visualize all the collected data. Therefore, if the model can not be loaded, the application will quit. Moreover, the tunnel id that the digital twin framework is tightly connected is passed to the virtual twin in the name of the main tunnel object. That is, if the tunnel can not be loaded, it is simply not possible to link the twins together.

5.4.2 Broker Client

The broker client allows the virtual twin to communicate with the broker server, as it is explained before. Besides the programming language related differences, there exist some minor implementation choices.

The first one is channel discovery. The framework creates its broker client on

launch by passing it the tunnel id. However, in the virtual twin's case, the broker client initialization is triggered by the main tunnel object discovery. In other words, the twin manager fires an event when the object manager registers the tunnel object to it.

```
1  // Twin Manager
2  public void SetTunnel(NvdbObject tunnel)
3  {
4      _tunnel = tunnel;
5      TunnelInitialized?.Invoke(tunnel.name);
6  }
7
8  //Broker Client
9  private void OnTunnelInitialized(string channel)
10 {
11     _channel = channel;
12     SetupClient();
13     SendSubscribeMessage(_channel);
14 }
```

Later, the client is initialized by the SetupClient method by reading the server address from config, and the subscription message is sent with the channel name. This completes the initialization of the virtual twin's broker communication.

The most critical functionality is triggered when a notification is received. The client detects the TCP message via a callback, and if the received message is a notification, it fires the NotificationReceived event. This event is subscribed by all the NvdbObjects in the virtual twin and will cause them to pull from the influx database.

5.4.3 NDVB Object

This section will briefly explain the aforementioned NvdbObject class. This class is attached on run time, on "Start" of ObjectManager instance. It contains an id member and a worker object. Respectively, the id is that particular nvdb object's id in the NVDB and is used in the virtual twin to link that object to the influx server. The latter is an instance of the Object class, which manages the interactions of that object. For example,

a door has two classes attached to it. The first one is the NvdbObject class, which uses the id to send pull requests to the database to retrieve data. The second one is the Door class, which inherits from the Object class to actually open or close the door according to the data that the NvdbObject requests. Thus, when the NvdbObject receives the NotificationReceived event, it requests the current data and commands the Object to do the work. The process so far is presented with the below sequence diagram.

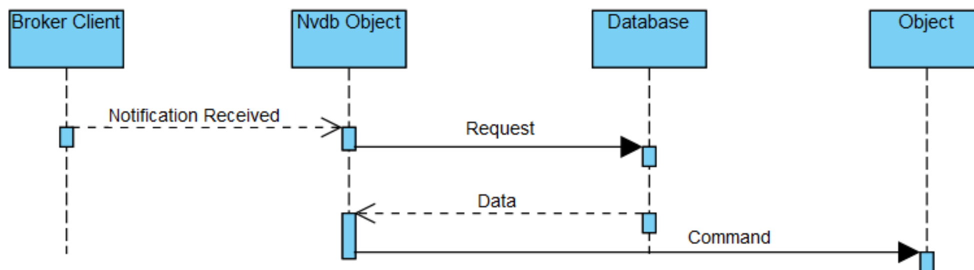


Figure 5.4: Nvdb Object Command Sequence

A vital implementation detail must be noted that due to the asynchronous nature of the database call, the regular synchronous calls might not yield expected results. For instance, the UI might not be registered with the correct items if the call to UI is made in an asynchronous body. Therefore, The NvdbObject only stores the information that is received from the database in the call body. The process of the aforementioned data is done in the Update method, which is called every frame.

5.4.4 Database Manager

In the previous section, the database communications are abstracted. On the other hand, this chapter is dedicated to explaining the implementation of the database manager class.

Similar to the broker client, the database manager reads from its config when its singleton instance is created to determine the server address. Later, when the twin manager discovers the main tunnel object, the database manager

receives the event to create the database connection. The tunnel id will be used to determine which database to be used.

An **async** SelectData method is used to retrieve the requested data asynchronously, not to block the virtual twin. When it receives the response from the database, it will return to the Nvdb Object instance that requested the data initially.

```
1 public async Task<List<string>> SelectData(string ...
      measurement, long time)
2 {
3     ...
4 }
```

In order to work, the method requires a measurement, which is the object id and a time value. The implemented history feature uses the time value. Basically, only the most recent entity for all tags at the given time will be received by the database. The built-in timestamp value of influx is used to achieve this if the framework had not stored a timestamp for the data.

Lastly, unlike the scripting languages, like Python, in this case, C# requires a data transfer object (DTO) to map JSON format. That is, a class is needed to store the response from the database. Usually, a class for each object would be a desirable implementation choice to keep the implementation simple and easy to understand. Having an individual definition for all classes will ease the developers' understanding. However, having a procedural system dictates no hard coding or, preferably, nothing static at all, but everything to be generic or dynamic. Therefore, the data transfer object can not have any tag names or tag ids as they tend to be different for all the objects. For example, a sign may have a sign type tag, but a door might lack it. Moreover, having a DTO for all objects would require code generation.

The solution to this was to implement a generic class that only stores the timestamp and a field-value pair. As all the objects have those fields, the generic class works. The TunnelInfo class is used with Influx macros to enable this.

```
1 public class TunnelInfo
2 {
3     [InfluxTimestamp]
4     public DateTime Timestamp { get; set; }
5
6     [InfluxField("value")]
7     public string Value { get; set; }
8 }
```

Both the timestamp and the value is accessible by the Nvdb Object once it is returned to it. However, as SelectData is an async method, it must be noted that the method it uses must also be an async method. Moreover, the call to it should await the response, and no synchronous calls in the same body will be blocking. Therefore, the NvdbObject instances manage the received data in their Update, as explained above.

5.4.5 Procedural Model Generator

The procedural model code is responsible for the creation of the procedural 3D models. The process starts with a given tunnel id and ends with a model generated by Maya2020. Although this program was not implemented during the development of the virtual twin or by the author, a crucial change has been made to make the virtual twin work as intended. Hence, there is a need to mention this as a subsection to the virtual twin.

In the previous implementation, the tunnel objects are generated as placeholder boxes and left only with their group, set, and object names. For example, a wall or a road was named a "polysurface" followed by an integer to address their order, or an emergency station was named "T1_N_dstasjon_N_A_Set1_Obj1". Although it was named after logical conventions, it lacked the object id, which is the most critical logic for the virtual twin. Therefore, a modification was done to the code to embed the object ids into the graphics. In order to achieve this, API call to the NVDB in the procedural generation code was extended to all the tunnel objects to retrieve their ids, and the ids are placed at the end of the names. For instance, the name "T1_N_dstasjon _N_A_Set1_Obj1_ObjectID271551714" can be given an example emergency station name with the current version. This object id is parsed by the NvdbObject and used to send requests to the

database. The walls or the road parts, however, lack the characteristics information in the NVDB. Thus, their names are not updated.

5.4.6 Interfaces

The virtual twin is essentially a component to visualize all the collected data by the digital twin. Therefore, it can be argued that the main feature is the interfaces, screens or basically any functionality to present data. The implementation to present data will be explained in this section.

Twin Panel

The twin panel is the main UI display method. It is used to present the data of a particular NvdbObject. It is typically hidden from the user until the object of interest is interacted with by the user. The interaction is a user click. The user click event will be caught by the clicked object's 3D collider, which is procedurally attached to the NvdbObject by the object manager. The event then will be forwarded to the NvdbObject instance of the clicked object.

The NvdbObject contains a reference to the twin panel. This reference is set on Awake of the Nvdb Object by retrieving it from the twin manager. It is used to populate the panel with data and to show it to the user.

```
1  private void Awake()
2  {
3      ...
4      _twinPanel = TwinManager.sInstance.GetTwinPanel();
5      ...
6  }
7
8  public void ShowObjectInfo()
9  {
10     _twinPanel.Populate(FieldNames, FieldValues);
11     _twinPanel.SetDisplayedObjectId(_id);
12     _twinPanel.gameObject.SetActive(true);
13 }
```

The population is done hierarchically and dynamically. The twin panel is essentially a window with scaling rules. The rules prevent it from overflowing when there is too much data to show. It hosts a vertical layout group to restrict each line space. The other important element is the InfoRow. An InfoRow is the object that stores exactly one-row information. That is, it only stores and visualizes a key and a value pair. It has a horizontal layout group to set rules against overflowing and to arrange spacing. When the NvdbObject starts populating the panel, it starts pooling InfoRows. In other words, it tries to use already existing components to populate itself. If there are not enough row objects, the twin panel will create more. Each time a row is created or reused, it will be populated with one key-value pair and will be listed in the panel as active. Thus, the objects will be created dynamically and populated hierarchically.

Additionally, the information about the tunnel itself can be seen by clicking on the information button on the top left of the interface. The button click event will be received and forwarded by the UI manager to the NvdbObject instance attached to the main tunnel object itself. This call will be made through the twin manager statically.

History

The date time selection tool at the top left of the user interface allows the user to select a date to be used as the timestamp in the database requests. After it is set to a specific date, the NvdbObjects will start using that value in their requests. That will restrict the returned data to a particular timestamp. Thus, no data after that time will be received.

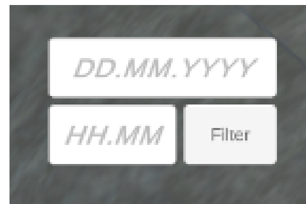


Figure 5.5: History Tool

The implementation is similar to the other event-driven logic throughout

the virtual twin. When a user sets the date and optionally the time input boxes in a valid format, which is shown in Figure 5.5, and pushes the filter button, the UI manager fires the TimeChanged event.

```
1 public void OnFilterButtonClicked()
2 {
3 ...
4 string tmp = _dateField.text + " " + _timeField.text;
5 DateTime time = DateTime.ParseExact(tmp, "dd.MM.yyyy ...
      HH.mm", ...
      System.Globalization.CultureInfo.InvariantCulture);
6                 long timestamp = ...
                      Util.ConvertToTimeStamp(time);
7                 TimeChanged?.Invoke(timestamp);
8 ...
9 }
```

This event is caught by the NvdbObjects, and the time value passed with the event is stored. The event will also start the same sequence with a notification received event to refresh the NvdbObjects' data.

```
1 public void OnTimeChanged(long time)
2 {
3     _time = time;
4     OnNotificationReceived();
5 }
```

An invalid input will set the timestamp to the current time, and an empty time field will mean 23.59 to filter the latest data at that date.

BIM View

The UI hosts a button with a Tunnel Logo to enable switching between models if a BIM model was discovered by the BIM loader. That is if the digital twin or the user placed a model before building the virtual twin.

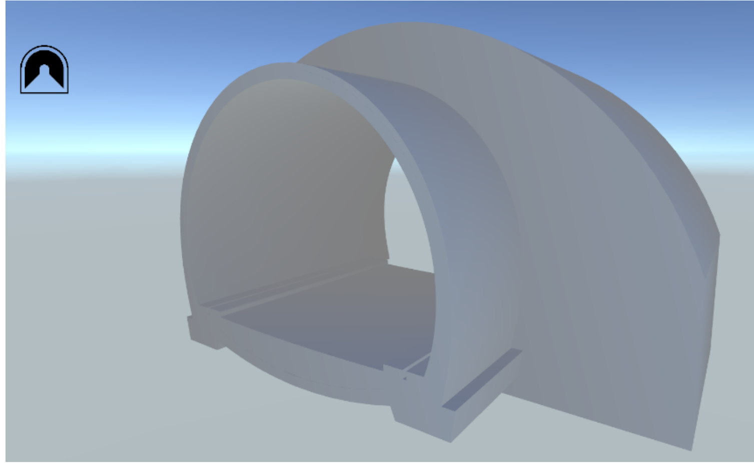


Figure 5.6: BIM Model in Virtual Twin

The BIMLoader places a BIM model under a previously given location, which is the BIM Manager object under object hierarchy, and the twin manager registers to itself. Therefore, whenever a user clicks the button to switch models, an event will be fired by the UI Manager, and it will trigger a model switching sequence in the twin manager. The manager is responsible for setting the camera accordingly and showing or hiding the models. Unlike the procedural model, however, the BIM model lacks interactivity as it is loaded as an FBX model to satisfy Unity's constraints. Future implementations can, of course, enable a more interactive model.

Chapter 6

Evaluation

This chapter is dedicated to evaluating the current state of the project. The proposed architecture and the implementation will be justified by presenting the test results and showing the ability to enable the use cases mentioned earlier.

The digital framework's responsibility is to produce digital twins for Norwegian road tunnels, and as the work is to collect data from multiple data sources, the test results will demonstrate the twin's ability to:

- React to changes in data.
- Speed to update the virtual twin.
- Ability to visualize data.

An example scenario will be presented to show that the implementation is capable of working as in the use case. Moreover, the proposed architecture allows multiple twins to be connected to one digital twin. Hence, the example views from the virtual twin can be observed in any one of the connected virtual twins.

Lastly, the known issues with the virtual twin will be listed and explained.

6.1 Experimental Setup

The digital twin framework needs certain requirements to be satisfied before executing a digital twin. Those requirements will be discussed in the following section. Later, the tests and their execution flows will be explained, and the results will be presented.

6.1.1 Requirements

The applications in the framework are programmed and tested in Windows OS. The digital twin framework and the supporting programs are written in Python 3, but the virtual twin is coded in Unity-C#. The digital twin framework builds the virtual twin during its execution, and Unity supports many platforms from Linux to macOS. Hence, making the virtual twin work in those environments with a minor code change should be possible. However, due to the lack of resources, that is, a Linux machine, for instance, this is left as a future improvement. It must be noted that the Unity engine is still needed to go through with the build. Therefore, a macOS or Windows machine is still needed, but after the build, the virtual twin executable can be used in the built platform.

Moreover, the python code does not depend on any platform, but the framework uses the influx database, and in the current state of the project, it uses influx's Windows build. However, influx also supports other platforms, and by replacing the database, it should be possible to use it on other platforms.

The procedural model generation also depends on Maya 2020, which supports Windows, macOS and Linux.

The software, necessary tools and their platform dependencies with the used version are shown below.

Software	Program	Version	Platform
DT Framework	InfluxDB	1.8.4	Windows/Linux/MacOS
DT Framework	Unity	2019.4.5f1	Windows/MacOS
Virtual Twin	InfluxDB	1.8.4	Windows/Linux/MacOS
Model Gen.	Maya2020	-	Windows/Linux/MacOS

During the development and evaluation, the same machine has been used, and it has the following characteristics. They are presented as the host computer has a significant impact on the time calculations.

CPU	Intel Core i7 8750H @ 2.20GHz
Memory	16GB
Operating System	Windows Home 2004

6.1.2 Test Execution

There are three different time-related tests conducted during the testing phase of the project. The system was tested to analyze the reaction of the digital twin and the virtual twin. The tests check the time for the digital twin to receive the OPC emulation data, the time for the notification of data change to reach the virtual twin and the time for the digital twin to update itself.

In order to test the reaction and the update speed of the digital twin, the Broker server and the clients are utilized. In the regular execution of the system, the broker clients send notifications to the broker server whenever there is a change in the database. New lines are added to the config files to configure the broker clients and the server to communicate with a test message, which carries a timestamp value. This timestamp is the exact time that the digital twin or the virtual twin receives the changes in data, which are saved into separate files, which are configured as well. Later a difference checker program checked these two files to compare the timestamp values. The system was left running for an hour with the maximum emulation settings, two files containing the timestamps were produced, and the first 500 entries were taken. The difference checker then compared the two files by subtracting one line from another to find differences. In order not to complicate the test code, all the tests are conducted individually, and the results are shown for five hundred entries, where an entry is a change by the emulation server.

Digital Twin Reaction Time

The reaction time is calculated by producing two timestamp values in the digital twin. The first one is created before the pull from the OPC emulation server, and the other one is created after comparing the received values with the database and noticing a change. As the digital twin pulls the data from the server regularly and as that time is configurable, there is not much of a need to calculate that part. Instead, the configured pull time, that is, ten milliseconds, is added to the results.

Number of Entries	Fastest	Slowest	Mean	Median
500	3123ms	5266ms	4081.28ms	4035ms

The OPC emulation server updates the server every second in its maximum emulation settings. Moreover, the digital twin pulls the data every millisecond. However, the digital twin pulls the latest data from the database and compares the fresh data with the database to check if there is a change or not. Although these test results show that the system works relatively slow as it averages 4.035 seconds per pull, it must be noted that this is actually a very exhaustive test. In the current state, the server emulates all the signs and lights every second, increments the heartbeat values, and emulates the status of the nodes. However, the real bottleneck is indeed the database, as it is queried every millisecond and is overwhelmed.

Still, the result shows that there is a need for improvement in digital twins' reaction ability.

Notification Time

The notification time is calculated by running the entire project for an hour in the local machine. The digital twin produced timestamp values right after it updates the database and the virtual twin produced another timestamp just after it received a notification from the server. The results are as follows.

Number of Entries	Fastest	Slowest	Mean	Median
500	7ms	145ms	59.035ms	54.0ms

This difference refers to the time the digital twin sends the notification for the change in the database and just after the virtual twin receives that notification. In other words, it is the time needed for the virtual twin to realize that there is a change and to initiate the process to retrieve the data.

One of the most important abilities of a digital twin is that to be able to update itself as close to real-time as possible. The test shows that the mean of this update is 54 milliseconds, which is a satisfying result.

Virtual Twin Update Time

This time interval is calculated by subtracting the time after the virtual twin receives a notification from the time the virtual twin updates all the objects from the database.

Number of Entries	Fastest	Slowest	Mean	Median
500	746.01ms	845.87ms	790.61ms	789.17ms

The results show that once the virtual twin receives the notification for it to fetch data, it actually updates itself quite fast. It must be noted that these values are for the virtual twin to update all the objects that are linked with the digital twin. Therefore, a lot faster results can be achieved by querying only one object every time. In fact, the virtual twin sends a select query each time the user selects an object. Thus, receiving the data much faster.

6.2 Visualization

The visual aspect of the project is can not easily be tested by scripts, but it must be shown instead. This subsection will present multiple images from a particular tunnel with a small scenario to demonstrate the virtual twin's ability to visualize data.

The particular scenario is an emergency scenario, where a car fire is created from the virtual twin's user interface. This scenario is chosen as it is pretty likely to show the virtual twin's interaction capabilities as well as its ability to reflect the incoming data.

The scenario can be briefly explained as a car starting to burn during regular traffic in a tunnel. Later the operator responds by changing the signs in front of the tunnel to a stop or a no entry sign, and once the fire is dealt with, it will be switched back to its initial value. The lights will be turned on to emphasize the ability to react to the changes in the lighting objects.

Figure 6.1: Tunnel with Traffic

The image above was taken right after the launch of the virtual twin application. It shows the final UI of the project with a generated tunnel model. The simulation panel, which is located at the top right of the image, was developed before to control the car fire simulation scenarios. A generated car can be seen, the lights are off, and the sign shows a speed limit of 60.

After some time and after multiple cars have been generated, a fire scenario was started from the simulation dropdown.

Figure 6.2: Car Fire

This simulation was designed to stop the car traffic right after the fire starts. In other words, the car generation stops, and the already generated cars stop moving while the selected car starts burning. The fire can be extinguished by using the extinguishers in the tunnel.

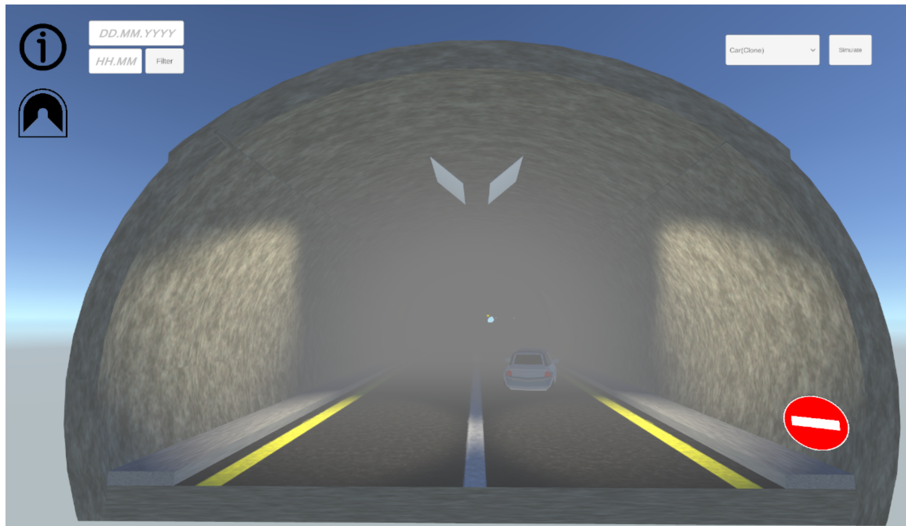


Figure 6.3: During Tunnel Emergency

Meanwhile, in a real-life scenario, it is expected that the people who are authorizing the signs to change the signs to control the traffic. Therefore, this demo scenario manually imitates such a case. It is seen from the image that the sign now changed to a no entry sign. Moreover, the light objects can reflect the real-time changes of the sensor servers. In order to demonstrate that ability, the lights in the tunnel were switched on before taking the previous image. All the changes in any linked object in a tunnel can be observed from the information panels by simply clicking on the particular object. More examples with different tunnels can be found in the appendix.

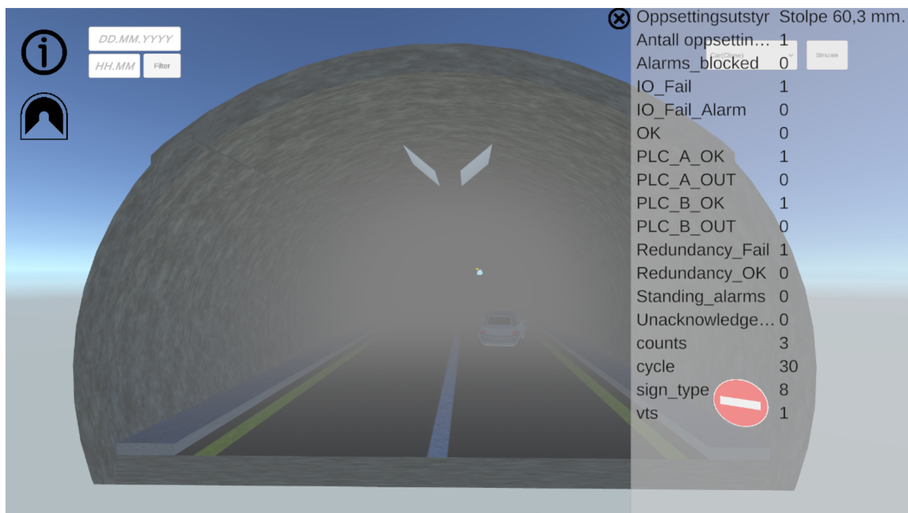


Figure 6.4: Information Panel of Sign

The example simulation shows that the virtual twin can indeed link the data from the nvdb to 3D objects and can further enhance them with interactions. Moreover, the virtual twin's ability to reflect on the changes is demonstrated. The current version of the virtual twin can place emergency stations, doors, technical cabinets, road signs and lights with their corresponding interactions and their links to the digital twin. Although not all the interactions are coded for each object, the reason for this is the lack of access to the actual data. Instead, it provides the dynamic creation of objects with their corresponding behavior scripts with empty bodies. These bodies can be further coded with respect to the real data later. Moreover, the digital twin does not forward changes in the virtual twin to the digital twin. Both of these improvements will be discussed later in this thesis.

6.3 Known Issues

During the implementation and the testing phases of the project, several issues were found and fixed. However, there are some that are left because of time constraints. These minor issues will be addressed in this section.

6.3.1 Object Interaction Issue

Unity uses a physics object named collider to handle the user interactions. Normally, these colliders are created manually while creating the objects in the scene. However, as the models are created procedurally and because of the differences in coordinate systems of Maya and Unity, these colliders might not be created in their correct positions. Although the manual tests show no issues with the colliders after the fixes, the number of tunnels and tunnel objects make it hard to test them all.

The problem with the colliders is that if they are placed in the wrong positions, the user might not be able to interact with the respective object because it is the collider, which is essentially interacted by the user, not the texture.

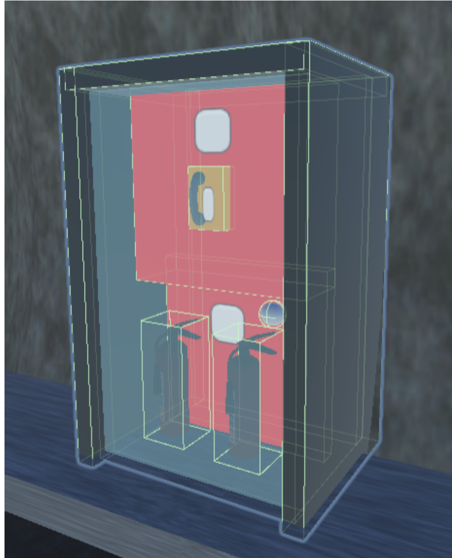


Figure 6.5: Manually Attached Object Collider

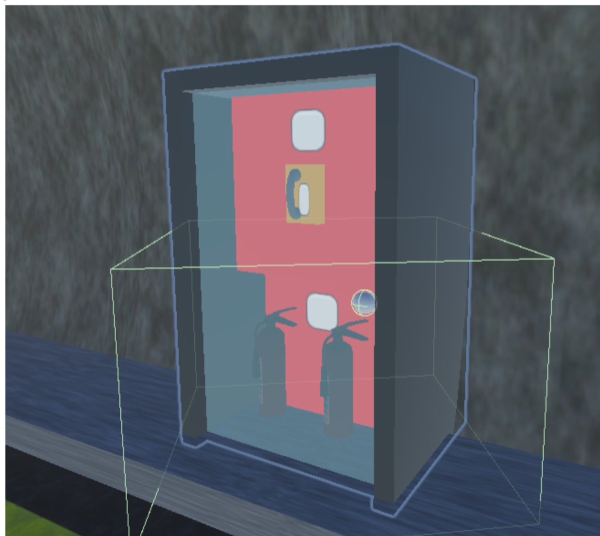


Figure 6.6: Generated Collider

In the example, the user would not be able to interact with the object below if they try to click the top part of the emergency station.

This issue was addressed by attaching colliders manually and saving the objects as Unity prefabs and then replacing them with the placeholders. Moreover, the generated colliders were used to call the information panel to show the digital twin information. That is, the different colliders pass user interaction to different scripts. However, this creates an inconvenience for the user and should be replaced.

6.3.2 Information Panel Wrapping Issue

The information panel generates the information rows for all new entries. For instance, if the object has twenty different tags in the database, the panel will generate twenty rows. However, the issue where the object has more elements than the actual screen size has not been addressed thoroughly. The panel scales itself and its child objects, which are the rows, according to the screen size. Hence, the larger the screen gets, the larger the panel becomes. Moreover, as the number of row objects increases, theirs sizes shrink. However, it is clear that the objects can not shrink smaller than a limit for them to stay readable. Therefore, there exists a hard limit to stop them from shrinking after some point, which basically prevents displaying more items after that limit. Hence, a solution that involves pages should be implemented. However, due to time constraints, these issues could not be addressed.

6.3.3 Emulation Server VTS Information

The OPC data includes a VTS field to indicate the objects' ownership information. This value can be between 1 to 6. These values have meaning in the digital twin and will be converted to the name of the respective VTS authority. However, an issue where the emulation server changes the VTS information together with the **counts** value has been seen several times. In other words, the emulation server increments the VTS field every **Cycle** times as it does with the counts field. This causes the value to go out of scope and not be converted to a meaningful value.

6.3.4 Broker Server Exception

There is a known issue with the broker server, where it recovers from an exception when an application that was connected to it crashes. Although it is observed that the server never crashes, which means it continues its work normally, the exception prevents a potential client from using the same client port. Hence, this issue should be fixed because of resource management and optimization concerns.

Chapter 7

Conclusion

The digital twin framework was built to procedurally create digital twins for the existing road tunnels in Norway. The aim was to use available information sources like NVDB to emulate the non-available but essential sources and to link them with the procedurally created 3D models to have working and presentable digital twin instances.

This thesis proposes a distributed architecture, where the instantiated digital twin and a virtual twin communicate with a middle broker program over a network. The design and the implementation were slowed down by the procedural nature of the work, as it caused every decision and the work done to be generic. However, the resulting software is believed to be modular enough for any future additions, removals or updates.

It is shown that a digital twin framework can be created from the existing and emulated data and can be used to simulate basic to advance scenarios. The experiments show that the data computing speed of the framework is fair. Moreover, the visuals are believed to be enough to explain the current state of the project and the potential the project has. However, both the software and the visuals need more work for this framework to be the state-of-the-art framework to generate digital twins for road tunnels.

The initial objectives of the project were mostly completed. Literature for the digital twins was studied, and a design to produce digital twins

for the Norwegian road tunnels was created and implemented. The sensor communications were included in the presented system, and due to the absence of actual data, the sensor data was emulated. All the experiment results, visuals and prototype tunnels were left as proof of work.

7.1 Future Work

7.1.1 Features

BIM is an essential technology that can be helpful to push the standards of this digital twin higher. Thus, its information must be added to the software. The current state of the digital twin allows the twin to fetch data from OPC UA servers and the nvdb. Also, multiple **fbx** models can be added to increase the visual capabilities of the virtual twin. However, the essential information of a 4D BIM could not be added because of the lack of time. However, a parser project for XML-like data was thought to handle the task. The **ifc** format creates an internally very well-linked file and can be read and parsed by a program. The initial thoughts and ideas are presented in this subsection.

- The tunnel objects in the 4D BIM must have the same IDs as the NVDB to be able to be linked to the existing software.
- The objects may or may not have an existing timeline in their name. The case where some objects might not have time-domain information must be handled.
- The objects' data must be saved with their existing timestamp fields. Otherwise, the database will act like it is a current entry.
- The fields and values of the objects must be saved according to the existing rules of the database to prevent additional rework in the virtual twin.

7.1.2 Improvements

Platform

The current state of the project is only tested in Windows OS, but the implementation is coded in generic ways to allow the builds for the other platforms. Provided that the framework has access to machines that have Maya2020 and Unity, the virtual twin can be built for any operating system, including the mobile ones.

Functionality

Currently, the virtual twin supports a handful of visualization of the data it receives. Some of these visualizations are for the light objects and the signs. However, it can be improved by adding data that can be shown by the 3D objects to the OPC server. The current version of the software is limited to emulation data. Moreover, all the data it manages are created by assumptions and deductions from online documents. Therefore, with the actual data, the functionality of the twin can be enhanced further. The functionality might include but is not limited to the behavior and interactions of the emergency stations, road signs, emergency lights, and doors.

Additionally, the current state of the procedural model generation limits the digital twin as it alters the coordinates of the tunnel objects by warping them after the object placement. This is an obstacle on the way to place new objects, according to the nvdb sources. Hence, making the objects appearing after the digital twin initialization harder. Although it is still possible to place new objects accordingly, it requires more effort than average as the coordinates for the old and new objects would be in different systems. Furthermore, Unity does not support dynamic model loading. That is, after creating new objects, the virtual twin or an asset bundle, including the tunnels, needs to be rebuilt and restarted. This does not necessarily mean that the digital twin should also be rebooted.

Another similar functionality would be to allow the movement of the tunnel objects according to the nvdb data. The current version of the twin only

supports the management of the characteristic data as the procedural model generation handles the geometrical data. However, in case of a location change, the twin must adapt and change the positions and the rotations of such objects. This, however, would be affected by the constraints and limitations regarding the coordinate systems, which are stated above.

In the case of 3D BIM, the virtual twin supports **fbx** models to be loaded in the project. However, they are required to be placed in the virtual twin resources. Having a source for the BIM, the digital twin can download and place the models inside the virtual twin, similar to the functionality for the procedural models.

Textures and Images

The current version of the digital twin uses free texture files from the internet. Although their qualities are enough to demonstrate the digital twin's capabilities, they look pretty different than the actual tunnels in Norway. Therefore, replacing the "placeholder" textures and images with properly designed ones can change much in the visual capabilities of the software.

Bibliography

[1] F. Tao, H. Zhang, A. Liu, and A. Y. C. Nee, “Digital Twin in Industry: State-of-the-Art,” IEEE Transactions on Industrial Informatics, vol. 15, no. 4, pp. 2405–2415, Apr. 2019, doi: 10.1109/TII.2018.2873186.

[2] K. E. Harper, C. Ganz, and S. Malakuti, “Digital Twin Architecture and Standards,” Nov. 2019.

[3] B. Galloway and G. P. Hancke, “Introduction to Industrial Control Networks,” IEEE Commun. Surv. Tutorials, vol. 15, no. 2, pp. 860–880, 2013, doi: 10.1109/SURV.2012.071812.00124.

[4] “News,” Jun. 16, 2011. https://web.archive.org/web/20110616214614/http://www.opcfoundation.org/Default.aspx/02_news/02_news_display.asp?id=845MID=News (accessed Mar. 09, 2021).

[5] “What is OPC?,” OPC Foundation. https://opcfoundation.org/about/what-is-opc/ (accessed Mar. 09, 2021).

[6] “open62541/open62541,” GitHub. https://github.com/open62541/open62541 (accessed Mar. 09, 2021).

[7] “Introduction — open62541 1.2.0-rc2-44-ge5eba7bd documentation.” https://open62541.org/doc/current/ (accessed Mar. 09, 2021).

[8] “Unified Architecture,” OPC Foundation. https://opcfoundation.org/about/opc-technologies/opc-ua/ (accessed Mar. 09, 2021).

[9] L. Roepert, M. Dahlmanns, I. B. Fink, J. Pennekamp, and M. Henze, “Assessing the Security of OPC UA Deployments,” p. 2.

[10] S.-H. Leitner and W. Mahnke, “OPC UA – Service-oriented Architecture for Industrial Applications,” p. 6.

[11] “Statens Vegvesen southern region – Nordre Vestfold - Tratec Norcon.” https://www.tratec.no/en/references/statens-vegvesen-nordre-vestfold (accessed Mar. 09, 2021).

[12] Drury, Bill (2009). "Control Techniques Drives and Controls Handbook" (2nd edition) - Knovel . Institution of Engineering and Technology. pp. 508–.

[13] “About Modbus Organization.” https://modbus.org/about_us.php (accessed Mar. 13, 2021).

[14] “Modbus FAQ.” https://modbus.org/faq.php (accessed Mar. 13, 2021).

[15] Modbus Messaging on TCP/IP Implementation Guide V1.0b (PDF), Modbus Organization, Inc., October 24, 2006, (accessed Mar. 13, 2021).

[16] “jamod - About.” http://jamod.sourceforge.net/ (accessed Mar. 13, 2021).

[17] A. Stanford-Clark and H. L. Truong, “MQTT For Sensor Networks (MQTT-SN) Protocol Specification,” p. 28, 1999.

[18] M. Singh, M. A. Rajan, V. L. Shivraj, and P. Balamuralidhar, “Secure MQTT for Internet of Things (IoT),” in 2015 Fifth International Conference on Communication Systems and Network Technologies, Apr. 2015, pp. 746–751, doi: 10.1109/CSNT.2015.16.

[19] “What is MQTT? Why use MQTT?,” IBM Developer. https://developer.ibm.com/technologies/messaging/articles/iot-mqtt-why-good-for-iot/ (accessed Mar. 13, 2021).

[20] “Quality of service and connection management.” https://www.ibm.com/support/knowledgecenter/SSMKHH_10.0.0/com.ibm.etools.mft.doc/bc62020_.htm (accessed Mar. 13, 2021).

[21] Y. Zheng, S. Yang, and H. Cheng, “An application framework of digital twin and its case study,” J Ambient Intell Human Comput, vol. 10, no. 3, pp. 1141–1153, Mar. 2019, doi: 10.1007/s12652-018-0911-3.

[22] “Digital Twin in manufacturing: A categorical literature review and classification,” IFAC-PapersOnLine, vol. 51, no. 11, pp. 1016–1022, Jan. 2018, doi: 10.1016/j.ifacol.2018.08.474.

[23] F. Tao et al., "Digital twin-driven product design framework," International Journal of Production Research, vol. 57, no. 12, pp. 3935–3953, Jun. 2019, doi: 10.1080/00207543.2018.1443229.

[24] "The National Road Database," Statens vegvesen. https://www.vegvesen.no/en/professional/roads/national-road-database (accessed Mar. 15, 2021).

[25] "NVDB Datakatalog." http://labs.vegdata.no/nvdb-datakatalog/ (accessed May 17, 2021).

[26] "File formats for BIM." https://www.designingbuildings.co.uk/wiki/File_formats_for_BIM (accessed May 29, 2021).

[27] "What is BIM?," NBS. https://www.thenbs.com/knowledge/what-is-building-information-modelling-bim (accessed May 29, 2021).

[28] A. Koutamanis, "Dimensionality in BIM: Why BIM cannot have more than four dimensions?," Automation in Construction, vol. 114, p. 103153, Jun. 2020, doi: 10.1016/j.autcon.2020.103153.

[29] pcholakis, "4D , 5D , 6D BIM," LEAN Construction Project Delivery Methods - Job Order Contracting, IPD, 5D BIM, Mar. 05, 2010. https://buildinginformationmanagement.wordpress.com/2010/03/05/4d-5d-6d-bim/ (accessed May 29, 2021).

[30] "Building Information Modelling," Pinsent Masons. https://www.pinsentmasons.com/out-law/guides/building-information-modelling (accessed May 29, 2021).

[31] "What is DDS?" https://www.dds-foundation.org/what-is-dds-3/ (accessed May 29, 2021).

[32] Boschert S., Rosen R. (2016) Digital Twin—The Simulation Aspect. In: Hehenberger P., Bradley D. (eds) Mechatronic Futures. Springer, Cham. https://doi-org.ezproxy.uis.no/10.1007/978-3-319-32156-1_5

[33] S. Jain, N. Fong Choong, K. Maung Aye, and M. Luo, "Virtual factory: an integrated approach to manufacturing systems modeling," International Journal of Operations Production Management, vol. 21, no. 5/6, pp. 594–608, Jan. 2001, doi: 10.1108/01443570110390354.

[34] E. Bottani, C. Assunta, T. Murino, and S. Vespoli, “From the Cyber-Physical System to the Digital Twin: the process development for behaviour modelling of a Cyber Guided Vehicle in M2M logic,” Sep. 2017.

[35] M. Shen, L. Wang, and T. Deng, Digital Twin: What It Is, Why Do It, Related Challenges, and Research Opportunities for Operations Research. 2021.

[36] “Project Koningstunnel,” Infranea. https://infranea.com/en/project/koningstunnel/ (accessed Jun. 08, 2021).

[37] “Tunnelware.” https://www.tunnelware.io/about (accessed Jun. 08, 2021).

[38] “TUNNEL DIGITALIZATION CENTER.” https://www. tunneldigitalizationcenter.com/En/Home (accessed Jun. 08, 2021).

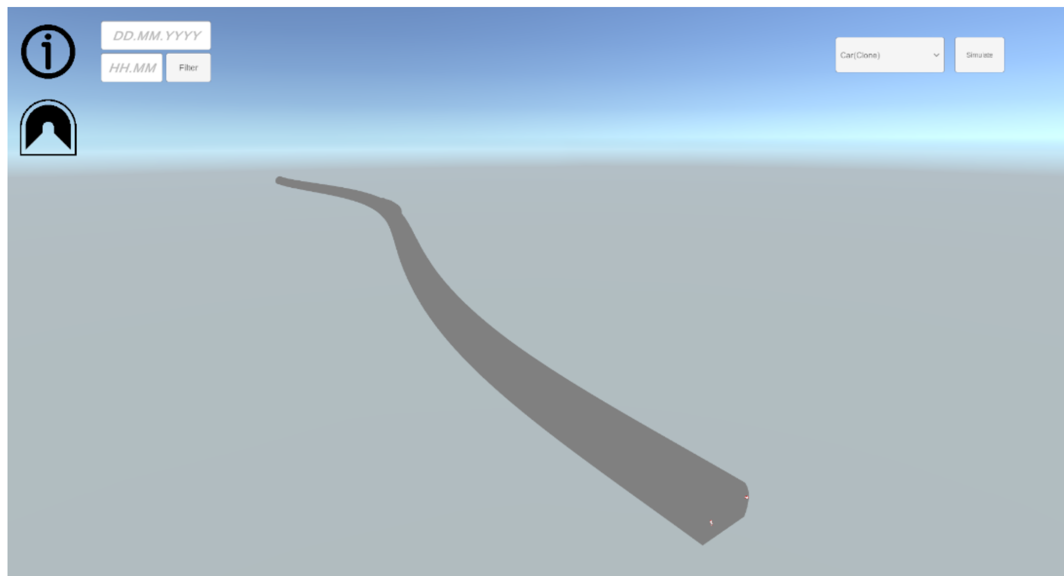
[39] “Standard OPC-grensesnitt for Trafikkstyresystem i Vegtrafikksentralen i Oslo (VTS) Versjon 3.2d - PDF Free Download.” https://docplayer.me/25671416-Standard-opc-grensesnitt-for-trafikkstyresystem-i-vegtrafikksentralen-i-oslo-vts-versjon-3-2d.html (accessed Jun. 12, 2021).

[40] Stava, G., Knutsen, H. and Henriksen, S., 2020. Procedural 3D Modeling of RoadTunnels: a Norwegian Use-case.

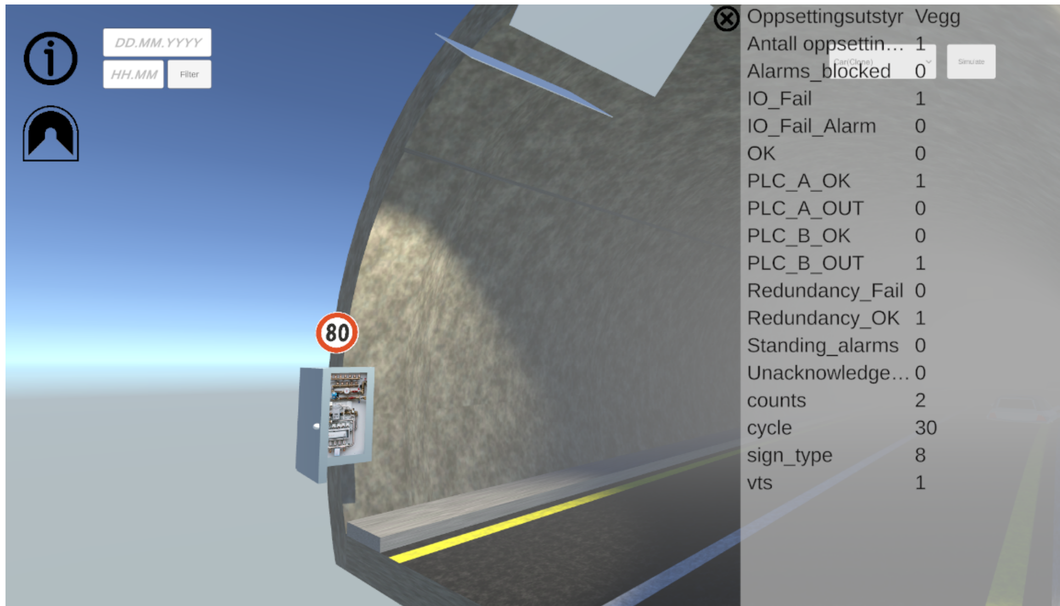
Appendix A

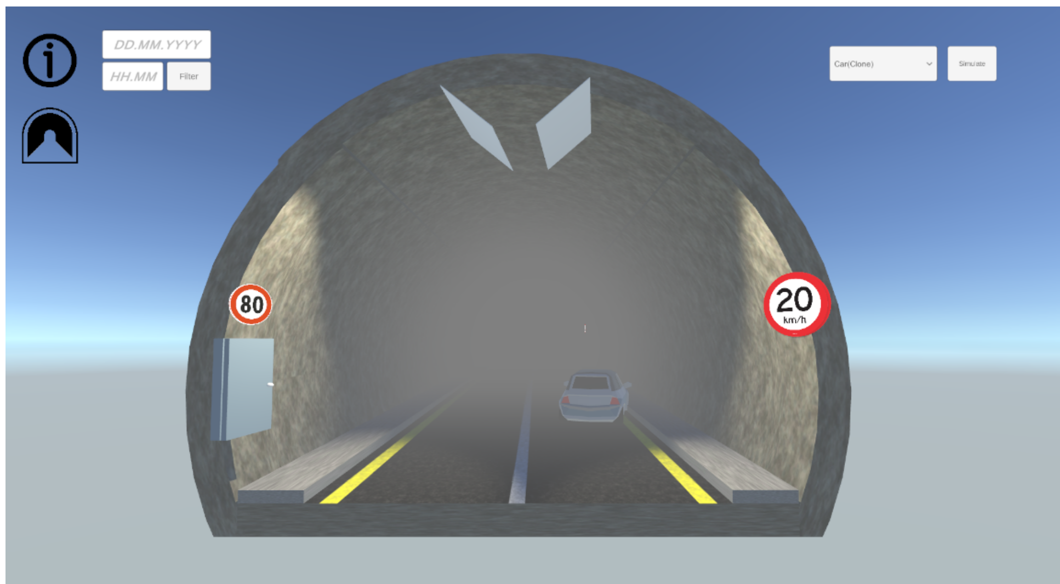
Appendices

A.1 Images From Tunnels



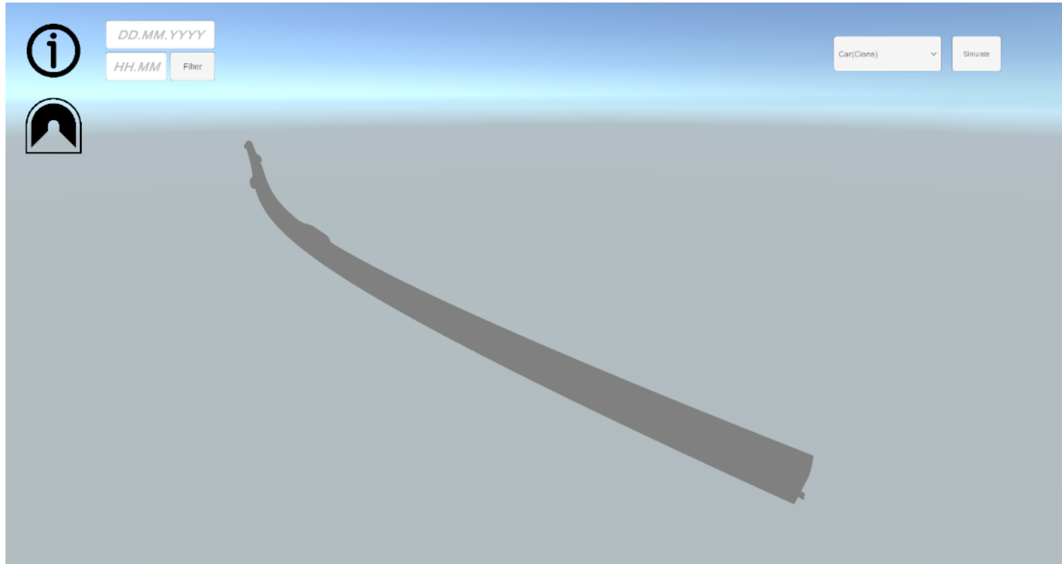
A.1 Images From Tunnels

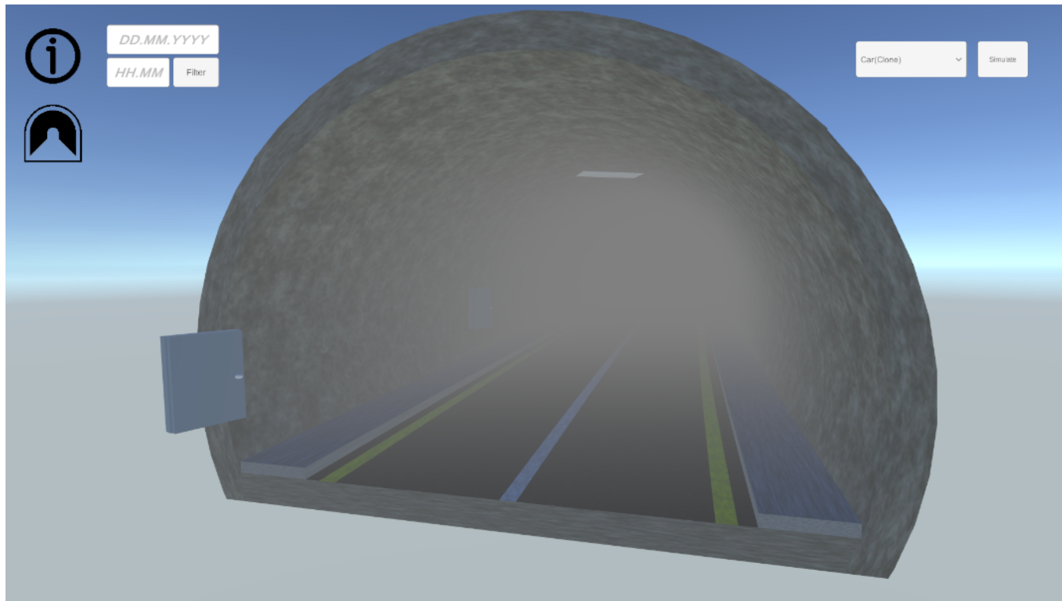




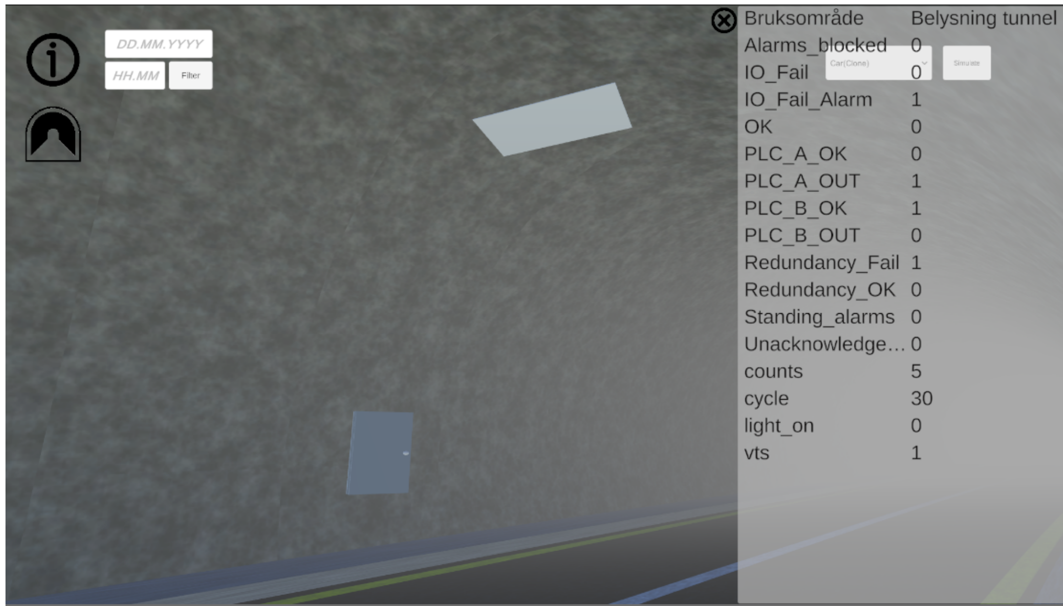
A.1 Images From Tunnels

A.1 Images From Tunnels





A.1 Images From Tunnels



A.2 Setup and Run

The source code is in its GitHub repository, which can be found in "https://github.com/TunnelSafety/DigiTUN". Also, the procedural generation code is located in "https://github.com/TunnelSafety/3D-tunnel/tree/Digitun", note that its in Digitun branch.

A.2.1 Minimum System Requirements

CPU	Intel Core i7 8750H @ 2.20GHz
Memory	16GB
Operating System	Windows Home 2004

It must be noted that the settings above might not be minimum. However, it is the only set of settings that this program was run.

A.2.2 Required Programs

Program	Version	Platform
InfluxDB	1.8.4	Windows/Linux/MacOS
Unity	2019.4.5f1	Windows/MacOS
InfluxDB	1.8.4	Windows/Linux/MacOS
Maya2020	-	Windows/Linux/MacOS

A.2.3 Required Libraries

Language	Library
Python	opcua
Python	nvdb
Python	json
Python	requests
Python	influxdb
Unity	Vibrant.InfluxDB
Unity	Newtonsoft.Json
Unity	DG.Tweening

Note that the mentioned libraries are required on top of the standard libraries. Hence, there might be a need to get additional libraries if the system does not include some standard libraries.

A.2.4 Configuration

The python libraries can be downloaded and integrated using pip or can be managed manually. Other than that, with the libraries, everything is ready to execute.

For the Unity part, it is suggested to get the required libraries using NuGet. It can be found as an add-in to the virtual studio, which is also integrated into Unity. For instance, the vibrant package can be found in "https://www.nuget.org/packages/Vibrant.InfluxDB.Client/" to retrieve it by NuGet.

The configuration for Unity was made correctly, and it should come readily when the code is checked out from the Github repository. However, if any errors occur due to configuration, the following steps must be done.

Unity requires some system and player settings to be configured. Check that the build settings are correct from File->Build Settings. The platform must be set to PC, MAC Linux Standalone. Although Windows Universal Platform can also be selected, it is not tested thoroughly. Architecture must

be chosen as x86_64. Lastly, make sure that on the top of the window, Scenes/Tunnel is selected as the only scene.

From the build settings menu, navigate to Player Settings->Player. Under the settings for PC, MAC Linux Standalone platform, scroll down to **configuration**. The settings under should be chosen as **Scripting Backend** : Mono and **Api Compatibility Level** : .NET 4.x. Note that higher .NET versions should be okay as well but were not tested.

All the code development was done in Unity 2019.4.5f1 and Python 3. However, note that the procedural 3D model generation was coded with Python 2. Thus, Python 2 is also required to run it.

A.2.5 Execution

The normal execution for the digital twin framework will be explained in this section.

The framework works with multiple standalone applications running together, and the order of execution is essential. Both the OPC emulation server and the broker server must be running before the digital twin framework. Therefore, the logical execution would start by running the emulation server and the broker server first. Later, the digital twin framework should be executed.

OPC emulation server generates a config file for a given tunnel. This id should be typed in code in main.py.

Correct execution for the framework is Python main.py <Tunnel_id>. This will start the run sequence. Note that the paths must be correct for the framework to find and execute the following.

- Procedural 3D Model Generator
- Maya2020
- Unity

A.2 Setup and Run

The framework will run an external code to generate and place a 3D model, connect to sources, build, and run the virtual twin. If, at some point, this execution fails, all these steps can be manually completed by running the individual applications.