Using 3D Game Engines to Build Digital Twins for Industrial Machine Simulation and Testing

Bachelor Thesis - FS 2025

Department: Computer Science
Field of Study: Software Engineering

Authors

Isaia Brassel
isaia.brassel@ost.ch
Matriculation: 22-173-264

Silvan Kisseleff
silvan.kisseleff@ost.ch
Matriculation: 22-176-135

Advisors

Prof. Dr. Christoph Gebhardt
christoph.gebhardt@ost.ch
Advisor

Thomas Zehnder
thomas.zehnder@avm.swiss
External Partner

Version: 1.0.0
Date: 2025-07-01

Abstract

Developing control software for industrial machines requires careful planning. However, it is still crucial to test the software before deploying it on a physical machine. To create meaningful tests, a digital twin that can simulate the behavior of the real machine is required. Yet, creating such digital twins can be both costly and complex – and often, they still fall short when it comes to offering realism. In light of these challenges, AVM Engineering AG approached us with the task of evaluating whether modern game engines are suitable for developing digital twins of their industrial systems. Our objective was to design a workflow for creating such digital twins and to demonstrate it using parts of a package distribution machine.Research into digital twins and 3D simulation highlighted the potential game engines provide with their advanced physical simulation capabilities. To determine the most suitable platform for implementing digital twins, we analyzed and compared the leading game engines – Unity and Unreal Engine 5. Based on performance, visual fidelity, extensibility, and licensing terms, we chose Unreal Engine 5. For reliable and standardized data exchange with industrial equipment, we used OPC UA, the de facto communication standard in industrial automation.To enable communication between Unreal Engine 5 and industrial controllers, we required a bridge between the OPC UA server and the game engine. For this, we integrated open62541, an open-source OPC UA implementation in C, into the game engine. To simplify the communication with the server we created custom Blueprint functions, the visual scripting language of Unreal Engine, for a simple drag-and-drop experience when building a digital twin. These building blocks enable read, write and subscribe operations from the game engine to the OPC UA server. In addition, we investigated the process of converting existing CAD models into assets suitable for use in Unreal Engine 5. This allows for highly realistic representations of industrial components within the digital twin.The integration of modern game engine technology with industrial automation tools enables the creation of highly customized digital twins. In this case, it allowed for a real-time simulation of the control logic developed for the package distribution machine, providing a safe and efficient environment to test and validate the software under realistic operating conditions—without the need for the physical system. To showcase the capabilities of our Blueprint functions and to make our approach to digital twin development more accessible, we designed and implemented custom models inside Unreal Engine 5.These models were specifically created to replicate key elements of the package distribution system—most notably, the conveyor belt mechanism. The components not only illustrate common use cases when it comes to OPC UA communication but also incorporate realistic physics simulations to closely mimic real-world behavior. User testing confirmed that our construction-based approach is intuitive and effective, significantly supporting the understanding and development of digital twins for complex systems.

Management Summary

Context

In the development of control software for industrial machines, accurate testing is critical before deploying to physical systems. This testing requires digital twins, virtual representations of machines that simulate real-world behavior. However, creating such twins has traditionally been expensive, complex, and often lacking in realism, limiting their usefulness in software validation workflows. Existing digital twin software are either manufacturer specific only, have high licensing costs or simply fall short in terms of realism. AVM Engineering AG has been affected by this issue and has been looking for a solution to this problem.

Solution

Research into digital twins highlighted the potential to significantly simplify the development of digital twins by using game engines. These engines offer a powerful and flexible environment for creating realistic simulations, making them ideal for building digital representations of industrial machines. Unreal Engine 5, in particular, stands out due to its advanced graphics, real-time rendering capabilities and sophisticated physics engine, making it a suitable choice for creating digital twins of industrial machines.

However, Unreal Engine lacks native support for communication with industrial machines. To address this, we implemented a solution based on the open62541 library, which enables communication to the machines through OPC UA, a widely adopted standard for industrial automation. We developed a set of reusable Blueprint functions that allow developers to easily connect Unreal Engine to an OPC UA server, enabling read, write and subscribe operations for multiple different data types with minimal effort.

To showcase the capabilities of our solution, we created several different example models of industrial machines, including a conveyor belt, a pneumatic cylinder, sensors and a slide. These components are controlled in real time using a demo OPC UA server built with Node-RED and a web-based dashboard interface.

Our solution provides a flexible foundation for developers to create and customize their own digital twins. It is designed to be easily extended and adapted to specific use cases. User testing confirmed that the system is intuitive, easy to use, and accessible even to those with limited experience in game development or industrial systems.

Future work

Looking ahead, the project can be extended in several impactful ways. Enhancements could include supporting OPC UA events and alarms for richer real-time interactions, improving the realism of the conveyor belt simulation, and enabling connections to multiple OPC UA servers for complex systems. Additional opportunities involve integrating machine vision capabilities and expanding platform support to include Linux. These developments would further increase the flexibility, realism, and accessibility of digital twin solutions built with Unreal Engine.

Table of Contents

1 Introduction 1
1.1 Target audience 1

2 Glossary 2

Product description 3

3 Context 4
3.1 Situation 4
3.2 Digital twin 5
3.3 Open Platform Communications Unified Architecture 5
3.4 Unreal Engine 6

4 Deliverables 7
4.1 Analysis of game engines 7
4.2 Integration of OPC UA client in the game engine 7
4.3 Blueprint functions 7
4.4 Models 7
4.4.1 Pneumatic cylinder 7
4.4.2 Conveyor belt 8
4.4.3 Slide 8
4.4.4 Combination 9

5 Related work 10
5.1 Commercial solutions 10
5.1.1 Mitsubishi Melsoft Gemini 10
5.1.2 Azure Digital Twins 10
5.1.3 Siemens SIMIT 11
5.1.4 ABB RobotStudio 11
5.1.5 Visual Components 11
5.2 Similar projects 12
5.2.1 Theses and papers focusing on OPC UA communication 12
5.2.2 Theses and papers focusing on 3D visualization in Unreal Engine 12

6 Requirements 14
6.1 Functional requirements 14
6.1.1 FR1: Connect to OPC UA server 14
6.1.2 FR2: Read boolean from OPC UA server 14
6.1.3 FR3: Write boolean to OPC UA server 15
6.1.4 FR4: Subscribe to boolean from OPC UA server 15
6.1.5 FR5: Pneumatic cylinder simulation 16
6.1.6 FR6: Conveyor belt simulation 16
6.1.7 FR7: Slide simulation 16
6.1.8 FR8: Spawn packages 16
6.1.9 FR9: Delete packages 16
6.2 Non-functional requirements 17
6.2.1 NFR1: Functionality on Windows and macOS 17
6.2.2 NFR2: The setup of our project requires no additional installments. 17
6.2.3 NFR3: The documentation is written in a detailed and explanatory way 17

6.2.4 NFR4: Simple and easy to understand graphics ... 18
6.2.5 NFR5: The project does not require a high-performance computer ... 18

Product realization ... 19

7 Analysis ... 20
7.1 Utility value analysis of game engines ... 20
7.1.1 Criteria ... 20
7.1.2 Utility value analysis for Unreal Engine ... 22
7.1.3 Utility value analysis for Unity ... 24
7.1.4 Summary ... 25
7.1.4.1 Unreal Engine ... 25
7.1.4.2 Unity ... 25
7.1.5 Conclusion ... 25
7.2 Analysis of OPC UA clients ... 26
7.2.1 open62541 ... 26
7.2.2 Unified Automation C++ SDK ... 26
7.2.3 Softing C++ SDK ... 26
7.2.4 Node-OPCUA ... 26
7.2.5 opcua-asyncio ... 26
7.2.6 Conclusion ... 27

8 Architecture ... 28
8.1 System context diagram ... 28
8.2 OPC UA server ... 29
8.2.1 OPC UA server address space ... 30
8.2.2 OPC UA server logic ... 31
8.2.3 Limitations ... 33
8.3 OPC UA client in Unreal Engine ... 34
8.4 Reading data from the OPC UA server ... 37
8.5 Writing data to the OPC UA server ... 39
8.6 Subscriptions ... 41

9 Project setup ... 45
9.1 Machine specifications ... 45
9.2 Installation ... 46
9.3 Version Control ... 47
9.4 Converting CAD files to Unreal Engine models ... 50

10 Models ... 55
10.1 Pneumatic cylinder ... 56
10.2 Package ... 59
10.3 Package spawner ... 61
10.4 Sensor ... 63
10.5 Conveyor belt ... 64
10.5.1 Conveyor belt movement ... 66
10.6 Slide ... 68

Summary ... 70

11 Summary ... 71

11.1 Results ... 71
11.2 Limitations ... 71
11.3 Future work ... 72

Appendix ... 73
11.4. Test runs ... 74
11.4.1. The whole project can be set up according to the README (10.4.2025) ... 74
11.4.2. The whole project can be set up according to the README (28.04.2025) ... 74
11.4.3. The whole project can be set up according to the README (03.06.2025) ... 74
11.4.4. The pneumatic cylinder can be extended and retracted. (03.06.2025) ... 75
11.4.5. The pneumatic cylinder can be extended and retracted. (03.06.2025) ... 75
11.4.6. The conveyor belt can be started and transports package. (03.06.2025) ... 76
11.4.7. The packages slide down the slide. (03.06.2025) ... 76
11.4.8. A Blueprint actor with a read operation to the OPC UA Server can be created. (03.06.2025) ... 77
11.4.9. A Blueprint actor with a write operation to the OPC UA Server can be created. (03.06.2025) ... 77
11.4.10. A Blueprint actor with a subscription to the OPC UA Server can be created. (03.06.2025) ... 78

Figures ... 79

Bibliography ... 83

1 Introduction

Verifying the behavior of control software for industrial machines is a critical yet challenging task. In most scenarios, it is neither practical nor cost-effective to set up physical machines solely for software validation. This creates a strong need for digital twins—virtual representations of machines or components that allow for safe and efficient testing in simulated environments.

While digital twins can be created in both 2D and 3D, 3D simulations offer distinct advantages. They enable the realistic modeling of physical behaviors such as gravity and friction which are essential for accurately replicating real-world machine dynamics. This allows engineers to detect logic errors, evaluate edge cases, and optimize system performance without physical risks or costs.

Many commercial digital twin solutions are available but often come with high licensing fees, limited customization, or poor compatibility with existing control software. To overcome these limitations, this thesis explores the use of a general-purpose 3D game engine with built-in physics simulation to develop a cost-effective, flexible digital twin.

To establish communication between the virtual model and industrial control software, we utilized OPC UA (Open Platform Communications Unified Architecture)—a widely adopted protocol in automation systems known for its security and interoperability. This enabled seamless integration with various types of programmable logic controllers (PLCs) and other control platforms.

The thesis outlines the full development process—from initial design and environment setup to implementing real-time physics and OPC UA communication—culminating in a functional prototype. The project was carried out in collaboration with AVM Engineering, whose expertise in automation helped ensure the model's industrial relevance and practical applicability.

By using open standards and accessible tools, this work demonstrates a scalable approach to building digital twins, empowering engineers to validate control systems in realistic 3D environments without the need for costly physical setups.

1.1 Target audience

This documentation is intended for developers and engineers interested in the creation of digital twins for industrial machines. The target audience is expected to possess a foundational understanding of programming principles, 3D game engines, and industrial communication protocols such as OPC UA. While the thesis provides brief introductions to these technologies to establish context, it does not aim to offer an exhaustive tutorial or theoretical background on each topic. Instead, the focus is on the practical application of these tools in the development of a functional digital twin prototype, showcasing the integration of physical simulation and real-time control communication.

In addition to serving as a technical guide, this documentation functions as the primary result-based report for our bachelor thesis. It is intended for review by our academic advisor, subject matter experts, and any potential stakeholders or competitors interested in the feasibility and implementation of our approach. By detailing our methodology, tools, and outcomes, we aim to contribute to ongoing discussions around cost-effective and adaptable solutions for validating industrial control systems in virtual environments.

2 Glossary

Blueprints Visual scripting system that allows developers to create game logic without writing code using a node-based interface.

Blueprint actor A type of Blueprint that represents an object in the game world, which can have properties, behaviors, and can be placed in the level.

Blueprint function An implementation of a node in Unreal Engine's Blueprint system, which can be used to encapsulate reusable logic or functionality. A Blueprint function can be used in the visual scripting system Blueprints.

Content Drawer The primary area for creating, importing, organizing, viewing and managing content assets in Unreal Engine.

Conveyor belt A mechanical system consisting of a continuous loop of material that transports objects or materials from one location to another, commonly used in manufacturing and logistics.

C++ A high-performance, general-purpose programming language known for its efficiency, object-oriented features, and extensive use in game development, system programming, and real-time applications.

Datasmith A plugin for Unreal Engine that facilitates the import of CAD data and other 3D assets into the engine, enabling seamless integration of complex models into Unreal projects.

Devcontainers A feature in Visual Studio Code that allows developers to create consistent and reproducible development environments using Docker.

Digital twin Virtual representation of a physical object, system or process to simulate, analyze and optimize performance.

Git A distributed version control system that helps track changes in code and collaborate with others efficiently.

Homebrew A package manager for macOS that simplifies the installation of software and tools via the command line.

Node-RED A flow-based development tool for visual programming, allowing users to wire together devices, APIs and online services using a browser-based flow editor. Used for building an OPC UA server in this project.

OPC UA Open Platform Communication Unified Architecture is a data exchange standard typically used in industrial machine communication.

open62541 open-source implementation of the OPC UA standard in C designed for industrial automation and IoT applications.

PLC Programmable Logic Controller is an industrial digital computer designed to automate and control machinery, processes, and production lines.

Pneumatic cylinder A cylinder that can block physical objects that can extend or retract using pneumatic forces.

Unreal Engine Powerful real-time 3D game engine developed by Epic Games, used for game development, virtual production and immersive experiences.

.NET A free, cross-platform, open-source developer platform for building many different types of applications, including web, mobile, desktop, gaming, and IoT.

Part I
Product description

3 Context

In this chapter we present the context of our bachelor thesis. Specifically, we introduce key technologies that are related or relevant to our thesis project and define the deliverables of our work.

3.1 Situation

AVM Engineering AG is a company that specializes in consulting and developing controlling software for industrial machines for various industrial partners. They are often faced with situations where they develop software without the ability to visually verify its correctness. Since it is not feasible to set up the industrial machines in their offices physically, they want to be able to have a digital twin of the machine. Having such a digital twin enables verification whenever and wherever required. AVM Engineering AG would like to be able to simulate the machine and verify the behavior of the controlling software visually. The goal is not to fully automate this process but rather have a good visual representation of it. The testing would be part of the integration and system testing, it is not part of the unit testing during active development. This would allow AVM Engineering AG to verify the software before deploying it onto a live machine.

Before this project AVM Engineering AG relied on 2D simulations using an SVG image with movable parts as seen in Figure 1. Their simulation software then moved these parts accordingly. This approach limited AVM Engineering AG in several different ways. Firstly the simulation was limited to 2D and therefore a splitting of the conveyor belt could not be modeled easily. In addition to splitting the conveyor belt, it would also be difficult to introduce a slide to the simulation. Not only because of the 2D limitation but also because the simulation software was not able to simulate physical properties like gravity or friction of the belt. An adaptation of the machine layout meant that the simulation software had to be adapted as well. This was a tedious process and required a lot of time and effort. Lastly the 2D simulation required a lot of manual reasoning to understand what is going on in the simulation because the machine in real life is not limited to two dimensions. The 2D view has limitations in visualization, making it difficult to accurately simulate elements like friction-based slides or sharp 90-degree curves.

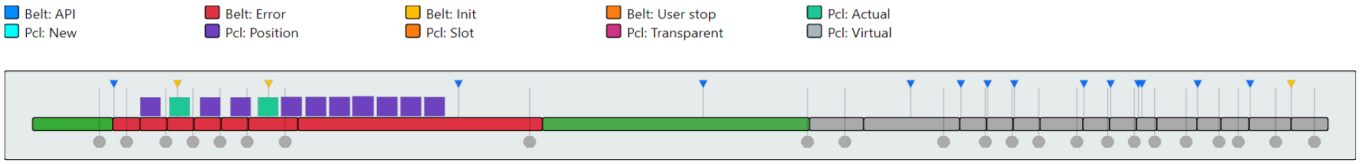

Figure 1: Existing 2D simulation which is utilizing a SVG with moveable parts.

Figure 2: Real life example of the package sorting machine constructed in the 2D SVG.

3.2 Digital twin

A digital twin is defined as a dynamic virtual copy of a physical asset, process, system or environment that looks like and behaves identically to its real-world counterpart [1]. The concept of digital twins was introduced in the 1960s by NASA. They wanted a physical replication of their spacecraft at ground level which would match the specifications of the active spacecraft that would be on a mission. This allowed them to quickly figure out what was causing problems in the spacecraft in outer orbit and how to resolve them. This implementation was still analog and physical rather than digital. The first digital twins of a physical object were observed in the 1970s and 1980s also developed by NASA [2]. They developed simulations of spacecraft equipment to help monitor and predict system behavior—an effort largely driven by the need to prevent incidents like the Apollo 13 mission failure in 1970. With the fourth industrial revolution, also known as Industry 4.0, it became much easier and affordable to create digital twins due to the digitalization and use of automation.

Digital twins reduce maintenance costs and give developers of controlling software a method of verification without risking physical damage to the machine [1]. A digital twin enables testing of machine designs and iteration on the design without the need to produce a physical prototype. Training for the specific machine can be done virtually, increasing safety and quality assurance. Enabling interaction in real time with the machine through the digital twin enhances production efficiency and minimizes maintenance disruptions by enabling predictive maintenance and process optimization.

Digital twins can be split up into three classes [3]. The first class is a simple digital model. It is a representation of an existing or planned physical object that does not use any form of automated data exchange between the physical object and the digital object. A change of state in the physical object or machine has no direct effect on the digital object and vice versa. The second class is a digital shadow that has a one-way data flow from the physical object or machine to the digital one. A change of state in the physical object or machine is reflected in the digital clone, but not the other way around. The third and final classification is a full digital twin, where data flows in both directions. A change of state in either the physical or digital object is reflected in both.

For this thesis, we use the term ‘digital twin’ to refer specifically to a digital model, meaning a representation of a physical object without automated data exchange between the physical and digital counterparts. This distinction is necessary because, although the concept of a digital twin is applicable, our work does not consider a physical counterpart of the machine.

3.3 Open Platform Communications Unified Architecture

Open Platform Communications Unified Architecture (OPC UA) is a standard used for the secure exchange of data in the industrial automation space and is used to control industrial machines. OPC UA [4] is the successor standard to the Open Platform Communications (OPC) standard. The previous standard is an interoperability standard for the secure exchange of data in the industrial automation space. It was first released in 1996 with the purpose of abstracting programmable logic controller (PLC)-specific protocols such as Modbus or Profibus into a standardized interface. The initial version was specific to the Windows platform. Due to this limitation, a second standard, OPC UA, was created to be a service-oriented, platform-independent, scalable and extensible standard. OPC UA has security built into the standard with encryption, authentication and auditing.

OPC UA introduces key features such as discovery of available servers, a hierarchical address space for structured data, on-demand read/write access, subscriptions for real-time updates, event notifications, and method execution [5]. OPC UA is hardware independent and can run on traditional PC hardware, cloud-based servers, PLCs, microcontrollers and more. The new standard can also run on various operating systems such as Windows, macOS, Android or any distribution of Linux and more.

In OPC UA, the address space is made up of nodes. An OPC UA node is like an entity representing some information [6]. Each node contains a unique identifier to distinguish between them, which also enables reading or writing

values to and from it [7]. In order to communicate with the server that contains the nodes, a client is required. A client is responsible for the communication with the server. It opens a connection via the Transmission Control Protocol (TCP) to the server. TCP is the primary protocol used to communicate with the server but OPC UA also supports Hypertext Transfer Protocol Secure (HTTPS), Message Queuing Telemetry Transport (MQTT), Advanced Message Queuing Protocol (AMQP), User Datagram Protocol (UDP) and WebSockets.

3.4 Unreal Engine

Unreal Engine[1] is one of the leading real-time 3D creation tools for photorealistic visuals and immersive experiences. It features dynamic physics, lifelike animation capabilities and is extensible to specific needs [8]. The engine is widely used across various industries, including game development, film production, animation, live events, automotive design, simulations, and architecture [9].

The community is very large and the engine is used by large corporations as well as by hobbyists. There are thousands of tutorials [10] online that teach everything from basic usage to very detailed nuances of the engine and the underlying pipeline.

The developer and maintainer of Unreal Engine is Epic Games [11]. The engine was first showcased in the 1998 first-person shooter video game “Unreal” and has been developed and released ever since then [12]. Unreal Engine is written in C++ and is highly adaptable and features a high level of portability supporting desktop, mobile, console and virtual reality applications. The first version of the engine that was downloadable for free was Unreal Engine 4 that was released in April 2014. The latest version, Unreal Engine 5, was released in April 2022 and the source code is available on Github.

Epic Games uses a royalty pricing model, charging five percent of all revenue that surpasses USD $1 million [11]. Games that are exclusively published on their own game distribution platform Epic Games Store are exempt from any royalty payments.

A project in Unreal Engine can be programmed in C++ or by using the visual scripting system called Blueprints. Blueprints is a node-based programming language that allows developers to create game logic without writing code. It is a powerful tool that enables rapid prototyping and iteration of game mechanics. Blueprints is designed to be user-friendly and accessible so that even developers with little to no programming experience can manage to create complex game logic. Unreal Engine provides a vast library of pre-built nodes that can be used directly for most common tasks. For more advanced tasks, developers can create custom Blueprint functions by implementing them in C++ and exposing them to the Blueprints system. This approach allows developers to combine the performance and flexibility of C++ with the ease of use and rapid development capabilities of Blueprints.

[1] In this thesis, we analyze both Unreal Engine and Unity. We ultimately selecting Unreal Engine. To provide context, we introduce Unreal Engine here at the beginning.

4 Deliverables

In the following, we list the deliverables of our project.

4.1 Analysis of game engines

We will analyze different game engines that are well known and widely used. In the analysis, we will take multiple factors into account and weigh them against each other. Based on the analysis, it will show why we decided to use Unreal Engine as the game engine for this project.

4.2 Integration of OPC UA client in the game engine

A major part of the project is to enable developers to create a digital twin of a machine. To achieve this, we need to integrate an OPC UA client into the game engine in a way that is easy to use and understand. The integration will be designed for reusability across different projects and machines.

4.3 Blueprint functions

We will leverage the Blueprints system of Unreal Engine to create building blocks, called Blueprint functions. Blueprint functions are reusable pieces of logic that can be used in the visual scripting system Blueprints to perform specific tasks or operations. They can be thought of as custom nodes that encapsulate a specific functionality, making it easier to use and understand within the visual scripting environment of Unreal Engine Blueprints system. These functions can be created in C++ and are then exposed to the Blueprints system with annotation macros. The main responsibility of the custom Blueprint functions that we will create is to enable communication with the OPC UA server without the need to write C++ code. We will abstract the complexity of the OPC UA client and provide a simple interface for interacting with the server. The building blocks will be reusable and can be extended for more advanced and specific use cases.

4.4 Models

We will develop some example models. These will be simple implementations of parts of a package transporting machine. These models serve as a demonstration of how to use the custom Blueprint functions and how to interact with the OPC UA server. The models will be simple enough to be understandable and easy to use, but complex enough to demonstrate the capabilities of the custom Blueprint functions we developed.

4.4.1 Pneumatic cylinder

The first model is a pneumatic cylinder. The pneumatic cylinder can be extended and retracted through a pressure valve. The valve is controlled by an OPC UA server provided by AVM Engineering AG.

When the valve is opened, the cylinder extends and when fully extended, a sensor notifies the server of the final position. When the valve is depressurized, the cylinder retracts and when fully retracted, a sensor notifies the server of the final position.

The following two images, represent a mockup of the pneumatic cylinder in two different states. Figure 3 shows the retracted state and Figure 4 shows the extended state. The images are not part of the simulation, but rather a mockup.

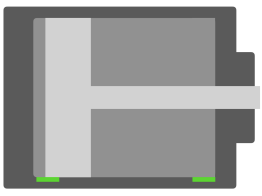

Figure 3: 2D mockup of a retracted pneumatic cylinder and its green sensor which will detect the position of the piston.

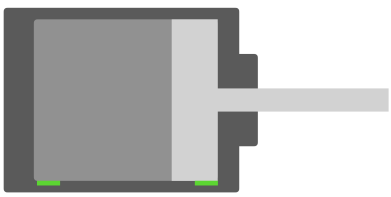

Figure 4: 2D mockup of an extended pneumatic cylinder and its green sensor which will detect the position of the piston.

4.4.2 Conveyor belt

The second model is a conveyor belt that can be turned on. On top of the conveyor belt, packages can be placed and they will move as soon as they are in contact with the conveyor belt. Collision will also be a part of the model, preventing packages from passing through one another. The conveyor belt has a speed that can be set through the OPC UA server, as well as a target position. The target position is incremented by the server based on the speed of the conveyor belt at every tick of the server. This target position is then used to determine the position of the packages on the conveyor belt[2].

A mockup of the conveyor belt is shown in Figure 5, where there is a conveyor belt with three packages on top of it. Additionally, two sensors that are used to detect the packages.

Figure 5: Mockup of a the conveyor belt with three packages and two sensors for detecting packages.

4.4.3 Slide

The third and final model is a slide for packages where gravity and friction must be simulated. Collision should also occur to prevent packages from sliding through one another.

A mockup of the slide is shown in Figure 6, where there is a slide with two packages on it. The slide has a slope and the packages will slide down the slide due to gravity. The packages will also collide with each other and will not be able to go through one another. The slide can be combined with a conveyor belt that feeds the packages into the slide, as seen in Figure 6.

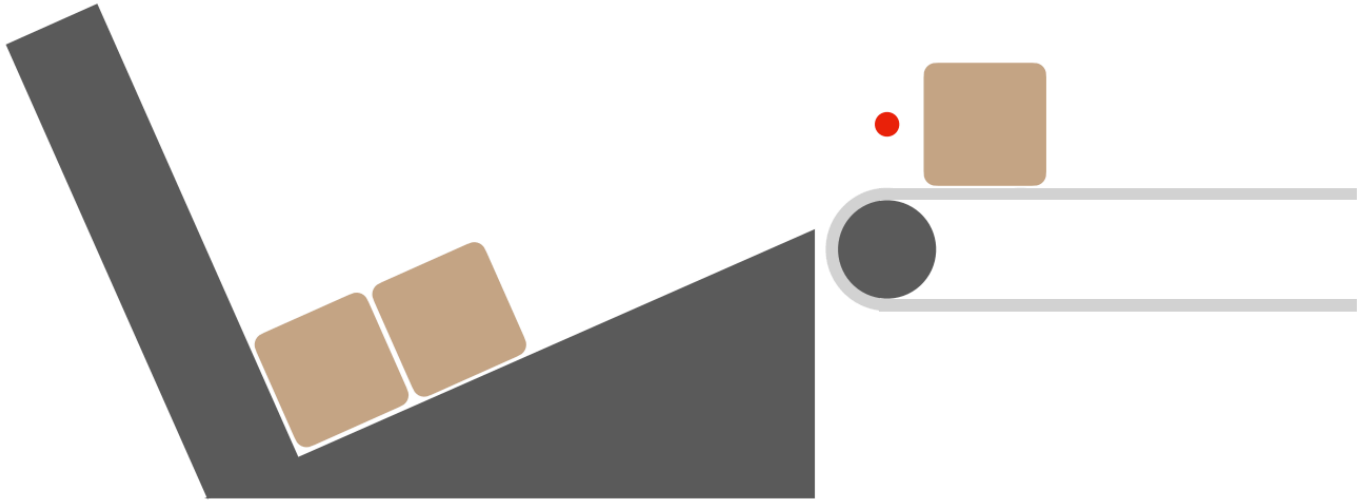

Figure 6: Mockup of a slide at the end of the conveyor belt. It will hold the packages as they fall of the belt and has a sensor for counting the amount of packages that slid down.

[2]We do not use the speed because a real conveyor belt would also use the target position of the servomotor instead of a fixed speed.

4.4.4 Combination

In the end, it is possible to combine these models into a single configuration where a pneumatic cylinder could prevent packages from traveling along the conveyor belt and then at the end of the belt, the packages could slide down the slide, as seen in Figure 7.

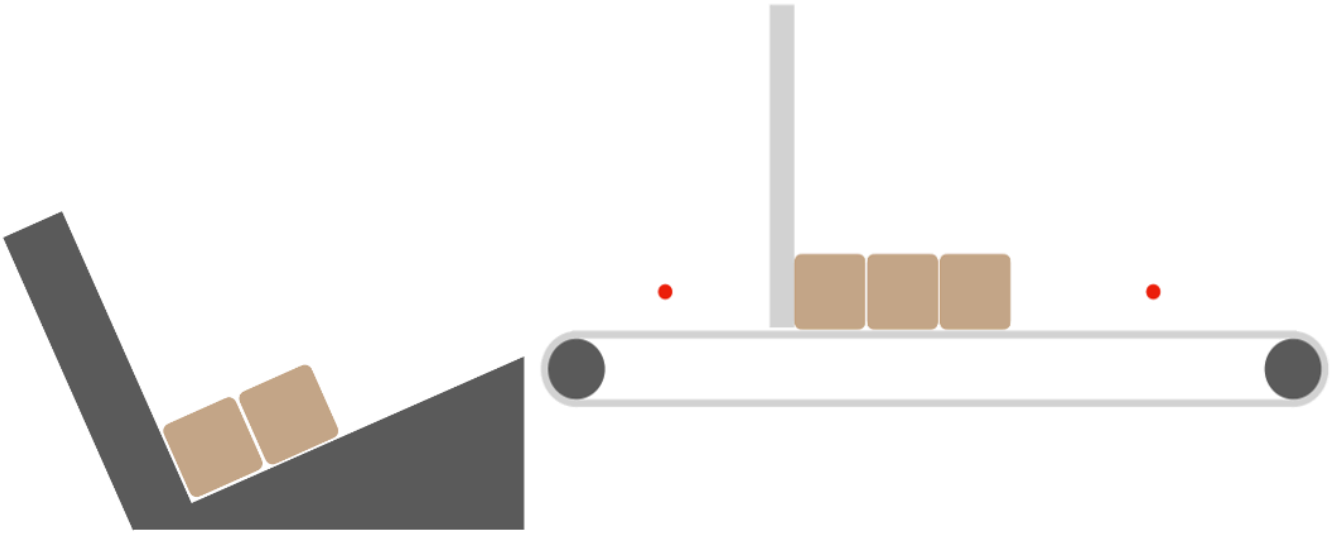

Figure 7: Mockup of the combination of all the models showing a complete machine lineup. The wall that prevents the packages from moving is meant to represent a pneumatic cylinder. However, since it is difficult to depict in a 2D model, we used a wall instead for this mockup.

5 Related work

Digital twins are well established in the industrial context and many different tools already exist today. Most of the sophisticated products are developed by major brands like Mitsubishi, ABB or Siemens. The drawback of these software product is, that they charge expensive licensing fees and that their customization usually is limited.

5.1 Commercial solutions

To have an overview over the products of the major brands we created a short summary for each one of them during our research.

5.1.1 Mitsubishi Melsoft Gemini

Gemini is a well-established software that allows layout planning of entire factories, production, operation flow simulation, digital twin creation for verification and virtual commissioning [13]. The software allows importing custom parts into the simulation and writing custom scripts that control the operations of the machine. Mitsubishi does not publicly announce the price of the software, likely because it is a premium product where the price varies from customer to customer.

Advantages

- **Seamless integration with the Mitsubishi ecosystem:** It integrates well with the Mistubishi factory automation components, reducing the time and complexity of connecting and configuring real-world hardware.
- **Support for custom screnarios and visualization:** The platform allows for the creation of custom simulation scenarios, including visualization of production flows, equipment behavior, and process logic, making it useful for training, testing and optimization.

Disadvantages

- **Limited flexibility:** Custom use cases involving third-party hardware or non-Mitsubishi platform can be challenging due to limited native support or more complex integration paths.
- **Customization requires technical expertise**: While it supports custom usage, implementing deeply customized features or complex logic simulations often requires advanced scripting or in-depth knowledge of this specific tool.

5.1.2 Azure Digital Twins

Azure is a platform-as-a-service (PaaS) and provides a digital twin solution that is very capable and feature-rich [14]. It allows for the modeling of factory space and machine compositions, enabling the software to simulate complex situations. Azure Digital Twins cannot directly communicate through OPC UA, which would hinder and slow down a communication between Azure Digital Twins and an OPC UA server. It uses a pay-what-you-need pricing model instead of larger upfront licensing fees.

Advantages

- **Real time monitoring:** Azure Digital Twins serves as a real time monitoring application that visualizes the current state of the factory.
- **Pricing:** As mentioned above, they use a pay-what-you-need pricing model. This can turn out cheaper than other competitors due to its configurable nature, where customers only pay what they really use in their in the project and not whole packages.

Disadvantages

- **Service Limits and Throttling:** Azure Digital Twins is limited due to its online nature. Adjustments, bad connectivity or a bad throughput can have a negative impact on the performance. Although this is not important for this project, reviewing this shows, that an offline solution will suffer from such connection problems.
- **Data Security**: Since it is an online service via the Azure cloud, security leaks can always happen. It can never be as safe as an offline solution since the data could possibly always be accessed via the network traffic. But this

is a topic that Azure takes very seriously and they have a lot of security measures in place to protect the data. However, it is still a risk that needs to be considered.

5.1.3 Siemens SIMIT

SIMIT from Siemens is also a feature-rich solution that natively supports OPC UA [15]. It focuses on simulating and testing of industrial machines, but without visualizing the machine and factories in a 3D space. The licensing is also not publicly available, but is suspected to be similar to Mitsubishi Melsoft Gemini.

Advantages

- **Training environment:** The platform provides a realistic training environment when the simulations comes to data. It will return exact measurements of the simulated machines, but it is not possible to view it in 3D. The goal is to let unfamiliar workers practice on the simulation, before they can work on the real-world machines.
- **Comprehensive testing:** SIMIT also allows users to do real-world simulations for testing purposes. These are precise and contain a lot of data about the machine. This can be done in advance of a real-world machine.

Disadvantages

- **Integration challenges:** similar to Mitsubishi Melsoft Gemini, Siemens SIMIT works perfectly fine with Siemens machines, but problems can occur when using third-party systems.
- **Learning curve:** Since Siemens SIMIT is also a standalone product, one would have to learn a lot of new functions. It is necessary to understand how the software works to avoid real-life casualties.

5.1.4 ABB RobotStudio

ABB RobotStudio is a simulation and offline programming software for industrial robots and machines [16]. It allows users to create, simulate and optimize robot programs in a virtual environment. The software is designed to work with ABB robots and controllers, making it a powerful tool for users in the robotics industry.

Advantages

- **Strong VR support**: RobotStudio has strong support for virtual reality (VR) and augmented reality (AR) applications. This allows users to visualize and interact with their robot programs in a more immersive way, allowing also allowing for better training and understanding of the machine.
- **Offline programming**: RobotStudio allows users to program robots offline, which can save time and reduce the risk of errors during programming. This is especially useful for complex applications where real-time programming may not be feasible.
- **Integration with ABB robots**: RobotStudio is designed to work seamlessly with ABB robots and controllers, making it a powerful tool for users in the robotics industry. This allows for easy integration and programming of ABB robots.

Disadvantages

- **Limited to ABB robots**: RobotStudio is primarily designed for use with ABB robots and controllers, which may limit its applicability for users working with other brands or types of robots. This can be a significant drawback for users who require a more flexible solution.
- **Cost**: RobotStudio is a commercial software, and the cost of licensing can be a barrier for some users. This may limit its accessibility for smaller companies or individuals who are looking for a more affordable solution.

5.1.5 Visual Components

Visual Components is a 3D simulation software that allows users to create and simulate complex manufacturing processes [17]. It is designed for use in the manufacturing and industrial sectors, providing tools for modeling, simulation, and analysis of production systems.

Advantages

- **Entire factory simulation**: Visual Components allows users to create and simulate entire factories, including machines, robots, and other components. This enables users to visualize and optimize their production processes in a virtual environment.
- **Protocol support**: The software supports a wide range of protocols, including OPC UA, which allows for easy integration with other systems and devices without any vendor lock-in. This makes it a powerful tool for users looking to connect their simulation with real-world data.

Disadvantages

- **Cost**: Visual Components is a commercial software, and the cost of licensing can be a barrier for some users. This may limit its accessibility for smaller companies or individuals who are looking for a more affordable solution.
- **Learning curve**: The software can be complex and may require a significant amount of time to learn and master. This can be a barrier for users who are new to simulation software or who have limited experience in the field.

5.2 Similar projects

Since most of the papers and thesis focus either on the OPC UA communication or the 3D visualization we write two summaries for either type.

5.2.1 Theses and papers focusing on OPC UA communication

A number of theses center their contributions on the use of OPC UA for connecting digital twins with real-world industrial systems. These projects typically focus on data communication, control integration, and system monitoring, often prioritizing backend architecture over immersive visualization.

For instance, the digital twin of a warehouse project integrates OPC UA within Unreal Engine 4.26 specifically for TwinCAT 3, noting incompatibilities with Unreal Engine 5 [18]. Similarly, the OPC UA client/gateway-based architecture for SCADA systems develops a hybrid framework emphasizing safety and fatigue monitoring rather than 3D immersion [19].

Other works explore industrial data exchange through web services, like the real-time two-way data transfer framework, which employs OPC UA as a data backbone but emphasizes a web-based API for interaction [20]. The thesis Industry 4.0 - Digital twins and OPC UA also applies OPC UA to connect a robotic production cell with its digital counterpart, but uses Visual Components instead of a real-time 3D game engine [21]. Likewise, the immersive digital twin and web-based digital twin theses implement OPC UA in Unity to monitor Industry 4.0 processes, with a focus on communication protocols over immersive fidelity [22], [23].

In contrast to these works, our approach uniquely integrates OPC UA directly into Unreal Engine 5, achieving seamless, secure, and real-time data access. We address the compatibility issues others encountered and provide not just data connectivity, but also full 3D immersion, bridging industrial machine data with high-performance visualization in UE5.

5.2.2 Theses and papers focusing on 3D visualization in Unreal Engine

Several projects investigate the visualization capabilities of Unreal Engine 5 (UE5) or Unity in the context of digital twin development, emphasizing immersive 3D environments, high-fidelity rendering, and simulation tools.

The wind turbine simulation study showcases UE5's ability to simulate physical conditions like wind speed and turbine mechanics in a virtual setting. Although visually advanced, this project doesn't involve live machine integration via industrial protocols like OPC UA [24]. Similarly, the master thesis on digital twins leverages UE5 and a RESTful API for real-time synchronization but omits OPC UA entirely [25].

Another notable example is the Unreal Engine 5 based digital twin platform, which proposes a modular, user-friendly architecture for building scalable digital twins. However, it focuses on UI/UX design and development efficiency rather than backend communication with control systems [26]. The paper Which architecture should be

implemented… discusses scalable data architectures in mixed reality scenarios using IoT data in UE5, yet does not apply OPC UA for real-time machine interfacing [27].

Additional work like Combining digital twins of robotic systems… integrates high-end simulation tools (e.g., VEROSIM [28]) with UE5 to enhance user interaction and physical accuracy, but lacks industrial communication protocols [29].

Our project distinguishes itself by combining immersive 3D visualization in Unreal Engine 5 with live, industrial-grade data streaming via OPC UA. We enable real-time, bidirectional communication with physical systems, resulting in a digital twin that not only looks realistic but also operates in sync with actual machines—a level of integration not achieved by the visualization-centric projects listed above.

6 Requirements

In the following subchapters, we describe the requirements for the building blocks and the simulation. The requirements are divided into functional and non-functional requirements. The functional requirements describe the behavior of the simulation and the building blocks. The non-functional requirements describe the quality of the simulation and the building blocks.

6.1 Functional requirements

In this chapter we describe the functional requirements for the simulation and the Blueprint functions that we will create. Certain requirements are kept brief, while the most important requirements are described with a fully dressed requirement template. The template for the use cases is based on Craig Larman's template [30] for brief and fully dressed use cases.

6.1.1 FR1: Connect to OPC UA server

Main success scenario

- The Unreal Engine establishes a single connection to the OPC UA server. The connection is accessible from all components of the simulation and can be shared in order to reduce the number of connections.

Alternate scenario

- If the connection cannot be established, the simulation should clearly display that the connection failed, and the user should be able to try to connect again.

6.1.2 FR2: Read boolean from OPC UA server

Scope Blueprint functions
Level User-goal
Primary actor Unreal Engine application
Stakeholders and interests

- **Developers**: The developers want to read a boolean value from the OPC UA server. The developers want to be notified, if the read was not successful.
- **OPC UA server:** The server prefers that the client does not overload the server with too many requests.

Preconditions

- Connection to the OPC UA server is established.
- OPC UA node that contains a boolean value exists on the server.

Main success scenario

1. Blueprint function checks if a connection is established.
2. Blueprint function requests the boolean value from the server.
3. Blueprint function receives the response.
4. Blueprint function parses the response to a boolean.
5. Blueprint function passes the boolean back to the callee.

Alternate scenario

- If the connection is not established, the Unreal Engine should display an error message.
- If the OPC UA node does not exist, the Unreal Engine should display an error message.
- If the response is not a boolean, the Unreal Engine should display an error message.

Extensions The read request could also be applied to other data types like integers, floats, strings, etc.

6.1.3 FR3: Write boolean to OPC UA server

Scope Blueprint functions
Level User-goal
Primary actor Unreal Engine application
Stakeholders and interests

- **Developers**: The developers want to write a boolean value to the OPC UA server. The developers want to be notified if the write was not successful.
- **OPC UA server:** The server prefers that the client does not overload the server with too many requests.

Preconditions

- Connection to the OPC UA server is established.
- OPC UA node with boolean value exists on the server.

Main success scenario

1. Blueprint function checks if a connection is established.
2. Blueprint function sends the write request to the server.
3. Blueprint function receives the response.
4. Blueprint function parses the response to a boolean.
5. Blueprint function passes the boolean back to the callee.

Alternate scenario

- If the connection is not established, the Unreal Engine should display an error message.
- If the OPC UA node does not exist, the Unreal Engine should display an error message.
- If the response is not a boolean, the Unreal Engine should display an error message.

Extensions The write request could also be applied to other data types like integers, floats, strings, etc.

6.1.4 FR4: Subscribe to boolean from OPC UA server

Scope Blueprint functions
Level User-goal
Primary actor Unreal Engine application
Stakeholders and interests

- **Developers**: The developers want to subscribe to a boolean value from the OPC UA server. The developers want to be notified if the subscription was not successful.
- **OPC UA server:** The server prefers that the client does not overload the server with too many requests.

Preconditions

- Connection to the OPC UA server is established.
- OPC UA node with boolean value exists on the server.

Main success scenario

1. Blueprint function checks if a connection is established.
2. Blueprint function sends a subscribe request to the server.
3. Blueprint function is notified as soon as the boolean changes.
4. Blueprint function triggers the callback function with the new boolean value.

Alternate scenario

- If the connection is not established, the Unreal Engine should display an error message.
- If the OPC UA node does not exist, the Unreal Engine should display an error message.

Extensions The subscription could also be applied to other data types like integers, floats, strings, etc.

6.1.5 FR5: Pneumatic cylinder simulation

Main success scenario The simulation of a pneumatic cylinder is displayed in the Unreal Engine. The cylinder can be extended and retracted by a boolean value from the OPC UA server. The pneumatic cylinder is animated, and the user can see the cylinder moving in real time. The cylinder blocks the path of anything that tries to pass it. The cylinder notifies the server when it is fully extended or retracted.

6.1.6 FR6: Conveyor belt simulation

Main success scenario The simulation of a conveyor belt is displayed in the Unreal Engine. The conveyor belt can be turned on, or off. Additionally, when turned on, the desired speed can be set. The conveyor belt is animated and the user can see the belt moving in real time. The belt moves anything that is on top of it.

6.1.7 FR7: Slide simulation

Main success scenario The simulation of a slide is displayed in the Unreal Engine. The slide makes use of friction and gravity in real-time. Packages that slide down the slide will increase a counter above it, to show how many packages passed the slide.

6.1.8 FR8: Spawn packages

Main success scenario In the OPC UA server, one can create packages with specific dimensions and a certain weight. These packages are then spawned in the Unreal Engine. The packages are displayed in the simulation and can interact with the rest of the simulation. The packages get a unique ID which is used to identify them.

6.1.9 FR9: Delete packages

Main success scenario In the OPC UA server, one can delete packages by specifying the desired package through the unique package ID. If a package with that ID exists, it is deleted from the simulation. If no package with that ID exists, nothing happens.

6.2 Non-functional requirements

The following non-functional requirements are listed and categorized according to ISO/IEC 25010 [31].

6.2.1 NFR1: Functionality on Windows and macOS

Category Compatibility -> Interoperability

Preconditions Unreal Engine and C++ are installed and set up correctly.

Description The Unreal Engine project, used to design the digital twins, must run on both Windows and macOS.[3]

Acceptance criteria Users can start the project in Unreal Engine on Windows and macOS, and can work on it or adapt it further.

Verification process

- Clone the Git repository.
- Setup according to the README.
- Open the project in Unreal Engine.
- Check if the application can complete all processes successfully.

Verification period After every construction sprint, the project will be checked for compatibility on the two different operating systems.

6.2.2 NFR2: The setup of our project requires no additional installments.

Category Flexibility -> Installability

Preconditions Unreal Engine and C++ are installed and set up correctly.

Description The project, must be easy to set up. The user should be able to run it without additional installments on Windows and with only a single installment on macOS.

Acceptance criteria A user should be able to clone the Git repository and open the project in Unreal Engine on Windows with no additional installments and on macOS with a single additional brew installment.

Verification process

- Clone the app from the gitlab repo.
- (macOS: install open62541 via brew.)
- Open the project in Unreal Engine.
- Setup according to the README.
- Run the demo digital twin.

Verification period After every construction sprint.

6.2.3 NFR3: The documentation is written in a detailed and explanatory way

Category Product Quality -> Quality use

Preconditions -

Description The documentation helps developers that are unfamiliar with the project, to develop their own feature.

Acceptance Criteria A user familiar with Unreal Engine, but new to the project, can create their own digital twin relying solely on the documentation. He should not need to look up for help online, (excluding unrelated Unreal Engine knowledge).

Verification process

- Have someone who is familiar with Unreal Engine start up the app.
- Verify they can independently create a digital twin using only the documentation.

Verification period After the “M4 Conveyor belt model is finished” milestone.

[3] Linux is not supported because our customer has no Linux devices and thus was not the focus of this thesis.

6.2.4 NFR4: Simple and easy to understand graphics

Category Interaction Capability -> User Error Protection
Preconditions -
Description Digital twin elements should be visually distinguishable.
Acceptance criteria A user familiar with Unreal Engine can identify static objects, fixed movable objects, freely movable objects, and contact lasers by observation.
Verification process

- Have someone who is familiar with Unreal Engine start up the app.
- Verify they can correctly identify elements based on appearance.

Verification period After the "M4 Conveyor belt model is finished" milestone.

6.2.5 NFR5: The project does not require a high-performance computer

Category Performance Efficiency -> Resource Utilization
Preconditions The graphic settings are set according to the README.
Description The project should run smoothly on a standard office computer without the need for a good GPU.
Acceptance criteria The Unreal Engine project should run on a basic office computer. We will test this on the office computers of AVM Engineering AG.
Verification process

- Have someone who is familiar with Unreal Engine from AVM Engineering AG start up the app on their office computer.
- The whole project should run on at least 30+ FPS.

Verification period After the "M3 Cylinder model is finished" milestone.

Part II
Product realization

7 Analysis

In this chapter, we analyze the game engines and the OPC UA clients to determine the most suitable options for our thesis project. For this thesis, we considered two game engines: Unity and Unreal Engine 5. These two engines were selected because they are the most widely used, well-documented, and actively maintained game engines available today. Both offer advanced physics capabilities, robust 3D rendering, and large, active communities, which are essential for a project of this scope. Additionally, both engines provide free licenses for educational or non-commercial use, making them accessible for academic research. Other engines were excluded due to limited feature sets, smaller communities, lack of long-term support, or restrictive licensing that does not align with the open and extensible nature required for our thesis.

7.1 Utility value analysis of game engines

To determine the best game engine for our project, we will perform a utility value analysis. First, we will assign an importance weight to each criterion, ranging from zero to one, where zero means it is not important, and one means it is essential. Next, we will research each engine's performance on these criteria and rate how well each engine meets them, using a scale of one to ten (where one means it does not support the criterion, and ten means it fully supports it). To calculate the final score, we will multiply the importance weight by the fulfillment rating for each engine. The engine with the highest total score will be the best fit for our project.

7.1.1 Criteria

To evaluate the best solution, we need criteria for a good rating. These are the criteria we chose:

- **Assistance:** Are there people we know that have knowledge about this game engine? Since we both have little knowledge about either game engine, a person that can support and help us in the developing process can be really helpful.
- **License:** Since this will be an open source project with no commercial use, a license policy that suits this project is really important. We want to make this software free to use without any license. Additionally, we want to minimize potential costs for our external partner AVM Engineering AG.
- **Programming language:** Preference is given to a game engine that utilizes a programming language with which both team members are already familiar, or that is similar to languages they know, to facilitate a more efficient start to the construction phase. It is also important that the programming language is widely supported, with an extensive selection of libraries and examples available, to minimize the need for developing functionality from scratch.
- **Performance:** Since the main goal is to develop digital twins for testing industrial machine logic, high-end graphics performance is not essential. However, the simulation should still operate smoothly and reliably during use.
- **Compatibility:** Broad compatibility across operating systems is essential to ensure that the software can be developed and executed on both macOS and Windows platforms. This facilitates collaboration and increases the accessibility of the simulation environment.
- **3D Rendering Capabilities:** The visual complexity of the constructions required for this project is relatively low, and advanced visual effects such as sophisticated shadows or lighting are not essential. However, it remains important that the engine provides clear and accurate visualization of components and their interactions to support effective simulation and analysis.
- **Community support:** AVM Engineering AG plans to use this tool also in the future, so long term support is crucial. A good support of a capable community is also an important criteria. It helps the development process if libraries and code is kept updated.

- **Flexibility:** The goal of this thesis is to build a foundation for further development in the area of digital twins. Therefore, it is important that this foundation is flexible and can be adapted to all sorts of industrial machines.

7.1.2 Utility value analysis for Unreal Engine

Criteria	Weight	Score	Total
Assistance	0.15	6	0.9
License	0.25	9	2.25
Programming language	0.15	4	0.6
Performance	0.05	5	0.25
Compatibility	0.1	7	0.7
Graphics	0.05	9	0.45
Community support	0.2	9	1.8
Flexibility	0.05	6	0.3
Total	**1**	**55**	**7.25**

Table 1: Utility value analysis for Unreal Engine.

- **Assistance:** Our external Partner AVM Engineering AG has developers that use Unreal Engine in their free time for smaller projects. We could ask for help there, but because they also only have the basic knowledge and are not directly involved in the project, it would not be feasible to rely on the assistance from these developers. (6)
- **License:** Unreal Engine provides a free license for smaller companies or individuals [32]. They only require a license when the company or individual surpasses a gross product revenue of one million USD. For schools and educational projects there is no license fee. For larger companies it depends on the product if one either pays a royalty or a seat-based license fee. If the product requires the game engine at runtime like a game or VR simulation, one has to pay 5% of all gross revenue that is directly attributable to the product, using Unreal Engine. If the product used Unreal Engine in the process but not at runtime like a movie show, one has to pay a seat-based licensing fee. Unreal Engine only requires a royalty payment once a substantial amount of revenue is generated from the product, making it a more financially attractive option. (9)
- **Programming language:** Unreal Engine uses either C++ or a visual node-based programming called Blueprints. Silvan has basic knowledge of C++ but has not used the language in a variety of projects. Isaia has not worked with the language previously and is not very familiar with it. (4)
- **Performance:** Unreal Engine is optimized for high-end, AAA[4] games with advanced graphics, which means it can be resource-intensive. While it is possible to develop smaller projects and adjust settings to reduce hardware demands, the engine generally requires more powerful computers to run smoothly. With appropriate optimizations, it can still perform adequately on standard office machines. (5)
- **Compatibility:** Unreal Engine runs on Windows macOS, and Linux. The downside is, that it has really limited support for web [33]. We could not find any long-term support versions of Unreal Engine, but we do not think that this is necessary because one does not really update the game engines of a game or in this case a simulation. It is important that a reliable and up-to-date stable version is available for use in this project. (7)
- **Graphics:** Unreal Engine is famous for its high-quality graphics and realistic rendering. It is widely used for AAA games, films, architectural visualization, and simulations because of its advanced rendering capabilities. Unreal Engine uses multiple systems in their engine. Lumen is Unreal Engine's real-time global illumination system that provides realistic lighting, dynamic reflections, good performing ray tracing [34]. Furthermore, there is Nanite, Unreal's virtualized geometry system, enabling high-detail environments, billions of polygons on screen at once, no need for level of detail or manual optimizations [35]. Unreal Engine also offers physically-based rendering for

[4] **AAA** (pronounced "triple-A") games are large-scale, high-budget video games produced by major studios, known for their advanced graphics, complex gameplay, and high production values.

realistic materials [36]. Unreal Engine offers advanced graphical capabilities, but many of these features are not essential for the requirements of this project. (9)

- **Community support:** Unreal Engine benefits from a large and active community. Numerous tutorials, forums, and online resources are freely available, making it accessible even for beginners. The community is supported by Discord servers, subreddits, and comprehensive official documentation, providing extensive support for a wide range of topics and questions. (9)
- **Flexibility:** Unreal Engine offers significant power through C++ and the Blueprint system, but its complexity can make large projects harder to manage. The engine is optimized for high-end visuals by default, which is more than required for our use case. Unreal Engine 5 lacks native web support, which could limit future deployment options if AVM Engineering AG wishes to run simulations in a browser. Its strict code conventions and extensive built-in features can make deep customization more challenging. However, Unreal Engine's robust built-in physics system is a strong advantage, as it enables adaptable simulations without requiring custom physics implementations. (6)

7.1.3 Utility value analysis for Unity

Criteria	Weight	Score	Total
Assistance	0.15	8	1.2
License	0.25	4	1
Programming language	0.15	7	1.05
Performance	0.05	8	0.4
Compatibility	0.1	10	1
Graphics	0.05	7	0.35
Community support	0.2	10	2
Flexibility	0.05	8	0.4
Total	**1**	**62**	**7.4**

Table 2: Utility value analysis for Unity.

- **Assistance:** Our advisor Dr. Christoph Gebhardt has worked with Unity on multiple occasions and could help us during the development process directly. If the project would be used later on by AVM Engineering AG, they would have to build up the experience for the game engine first, reducing the score a bit. (8)
- **License:** Unity comes with a free license for small businesses and individuals that make less than 200,000 USD in revenue or funding in the prior 12 months [37]. For businesses that make more than 200,000 USD are required to purchase a Unity Pro license, business with more than 1,000,000 USD, are required to purchase a Unity Industry license. Our school project would fall under the free plan and our external partner AVM Engineering AG would have to purchase a Unity Industry license. (4)
- **Programming language:** Unity uses C# as its main programming language, which both Isaia and Silvan have prior experience with, though neither is an expert. (7)
- **Performance:** Unity is generally lightweight but can become demanding depending on the project. It performs well on a wide range of hardware, from low-end laptops to high-end gaming PCs. (8)
- **Compatibility:** Unity is compatible with almost every environment. Windows, macOS, Linux, Mobile, Web, consoles, and more. (10)
- **3D Rendering Capabilities:** Unity's graphics are not as advanced as those of Unreal Engine, but they are more than sufficient for our needs. Unity offers a high-definition render pipeline with realistic lighting, reflections, high-quality shadows, physically based rendering, and support for high-quality assets and large-scale environments [38], [39]. It also provides a universal render pipeline for mobile and cross-platform development. While less complex than Unreal Engine, Unity's lightweight design and straightforward architecture offer greater flexibility in rendering. (7)
- **Community support:** Unity boasts one of the largest and most diverse developer communities, spanning mobile, web, and PC platforms. This diversity brings a wide range of perspectives and solutions to various challenges. A significant advantage is the Unity Asset Store, where users can access and share libraries and models, often for free [40]. Unity also supports external and custom user-created libraries, which can help reduce development effort by providing reusable components that may already fit our project needs. (10)
- **Flexibility:** Unity is quite a flexible game engine. As mentioned above it uses C# and Bolt (visual scripting). A big plus for Unity is the compatibility across multiple platforms. Unity supports Web, Mobile, and AR/VR. Unity also supports 2D graphics. It does not force a specific workflow on developers, which makes it flexible for different programming styles. (8)

7.1.4 Summary

To summarize this analysis we created a list with the most important pros and cons of either game engine.

7.1.4.1 Unreal Engine

Pros

- Free license for non-commercial use
- High-quality and visually appealing graphics
- Extensive built-in tools, making it a highly versatile and powerful engine
- Large community with excellent support

Cons

- Relies on C++ as the primary language, which is complex and has a steeper learning curve
- Limited native support for web platforms
- Heavy and resource-intensive game engine

7.1.4.2 Unity

Pros

- Utilizes C# as its programming language, which is straightforward and user-friendly
- Lightweight game engine with essential and effective built-in features
- Has a large community and an asset store, offering user-created libraries and resources
- Offers native support for web deployment

Cons

- Requires an annual license fee of USD 4,000.
- Graphics quality is not as advanced as Unreal Engine

7.1.5 Conclusion

After presenting the analysis results to AVM Engineering AG, the license cost associated with Unity was deemed a significant drawback. Considering this and the relatively close utility values of both game engines, we have decided to proceed with **Unreal Engine** for this thesis.

7.2 Analysis of OPC UA clients

In order to connect to an OPC UA server, we will need a client that handles the request, session and state management. There are multiple different implementations of the OPC UA standard in different languages.

7.2.1 open62541

open62541 [41] is an open-source implementation of the OPC UA standard in C, using the C99 standard. It is licensed under the Mozilla Public License Version 2.0 [42], and thus can be used for free even in commercial projects. It runs on different platforms like macOS, Linux, Windows, or even embedded systems. It has a small memory footprint and supports subscriptions, read/write operations, method calls, alarms and conditions, events, encryption, and authentication. Because the client is written in C and Unreal Engine uses C++, it would be possible to directly use the library in the C++ source code without the need to implement a separate layer for Unreal Engine to talk to the OPC UA server.

7.2.2 Unified Automation C++ SDK

Unified Automation is a software company specializing in OPC UA technology, providing toolkits, SDKs, and development frameworks for industrial communication. Unified Automation distributes a C++ library that is capable of creating an OPC UA client in order to talk to an OPC UA server. It requires a license for commercial projects and is closed sourced [43]. The target platform is limited to Windows and Linux. The client supports read/writes, subscriptions, method calls, and alarms. Since it is a C++ library it could be directly used in an Unreal Engine project without the need to implement a separate layer for Unreal Engine to talk to the OPC UA server. We consider it a less suitable alternative to open62541, as it is closed-source, requires a commercial license, and lacks support for macOS. These limitations make it unsuitable for use in an academic thesis intended for open publication.

7.2.3 Softing C++ SDK

Softing provides a C++ SDK for OPC UA client development, similar to Unified Automation [44]. While it offers comparable features, it also requires a license for usage, with pricing details not publicly disclosed. The supported platforms include Windows, Linux, and VxWorks. Given the licensing requirements, lack of macOS support, and the additional cost, this SDK is less suitable for our project. Consequently, we opted not to use this library as it does not align with our project's goals and constraints.

7.2.4 Node-OPCUA

Node-OPCUA is an open-source client/server implementation of the OPC UA standard in JavaScript, using Node.js [45]. The library is feature rich and supports all required operations and subscriptions. The library is licensed under MIT [46], and thus can be freely used in commercial projects. Because Unreal Engine uses C++ we would need to create a separate layer through which we could communicate with the OPC UA server from the engine. This would be a possible alternative if our preferred solution open61541 would not work. There is also a plugin for Unreal Engine that enables support for JavaScript in Unreal Engine. The feature is experimental and not widely used, for this reason we would prefer to not use it for our bachelor thesis. We want to keep the architecture as lean as possible.

7.2.5 opcua-asyncio

opcua-asyncio is an open-source client/server implementation of the OPC UA standard in Python [47]. The library is feature rich and supports all required operations and subscriptions. The library is licensed under LGPL Version 3 [48]. Because Unreal Engine uses C++, we would need to create a separate layer, through which we could communicate with the OPC UA server from the engine. Similar to the approach with Node-OPCUA an extra layer would mean extra effort, a bigger application, more languages, and overall a less organized project. If using an additional layer becomes unavoidable, we would still prefer Node-OPCUA due to its open-source nature.

7.2.6 Conclusion

We prefer open-source and free use over closed-source and licensing. Additionally, we aim to keep the system complexity as low as possible to reduce maintenance requirements, while ensuring that the solution remains easily extensible. For this reason, we decided to use open62541 instead of the other libraries because it is open-source, free to use in commercial projects, already written in C so it can be compiled and used in the C++ project of the game engine without the need for an extra layer. Additionally, the library has all the functionality we require to create a digital twin and is actively maintained and developed. For these reasons, open62541 is the best option in the context of our project.

8 Architecture

In this chapter, we summarize decisions, design patterns and explain the architecture of the software components and how they interact with each other.

8.1 System context diagram

The system context diagram shows the overall architecture of the simulation. The simulation is divided into two main components: the OPC UA server and the Unreal Engine simulation as seen in Figure 8. The OPC UA server is responsible for providing the data to the simulation and the Unreal Engine simulation is responsible for rendering the state of the machines. The tester interacts with the OPC UA server through a web-based dashboard that is provided by the server. The tester can control the simulation by changing the state of machines. The Unreal Engine simulation listens to the changes in the OPC UA server and updates the state of the visualized machines accordingly.

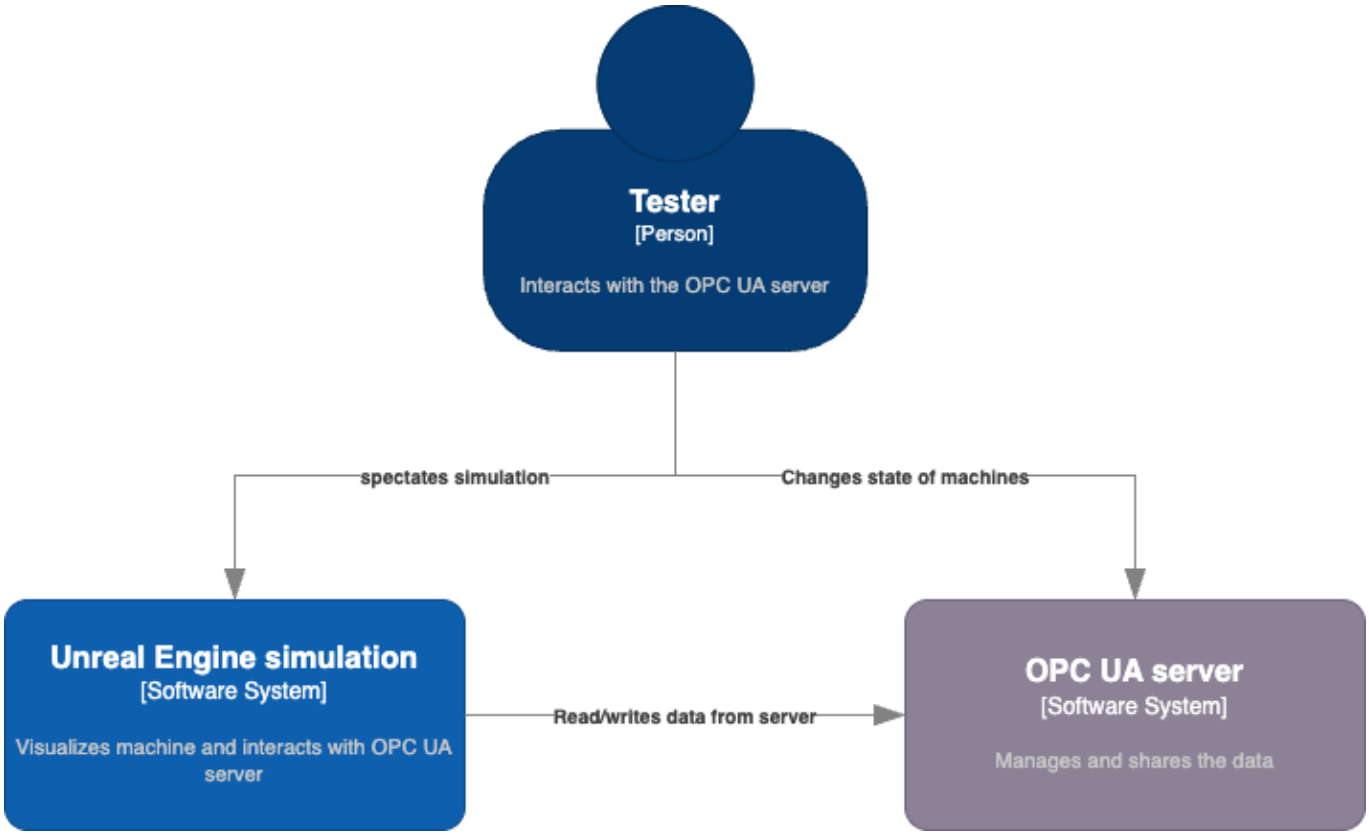

Figure 8: System context diagram for the whole setup and interaction between the Unreal Engine the OPC UA server and a test user.

8.2 OPC UA server

The OPC UA server is implemented using Node-RED. Node-RED is a graphical programming tool that uses a flow-based programming approach where nodes are used to create application logic. It is based on JavaScript and is designed for the Internet of Things (IoT) and data flow applications. Node-RED provides a web-based interface as seen in Figure 9 for creating flows. We use `node-red-contrib-opcua` in order to define the OPC UA server address space. In order to interact with the control software we use `@flowfuse/node-red-dashboard` to create a simple dashboard that allows us to create simple buttons and sliders to control the simulation as seen in Figure 10. The dashboard is accessible from any web browser and allows us to control the simulation in real time. While AVM Engineering AG developed the majority of the server logic and dashboard, we implemented the logic for creating and deleting packages on the server due to time constraints.

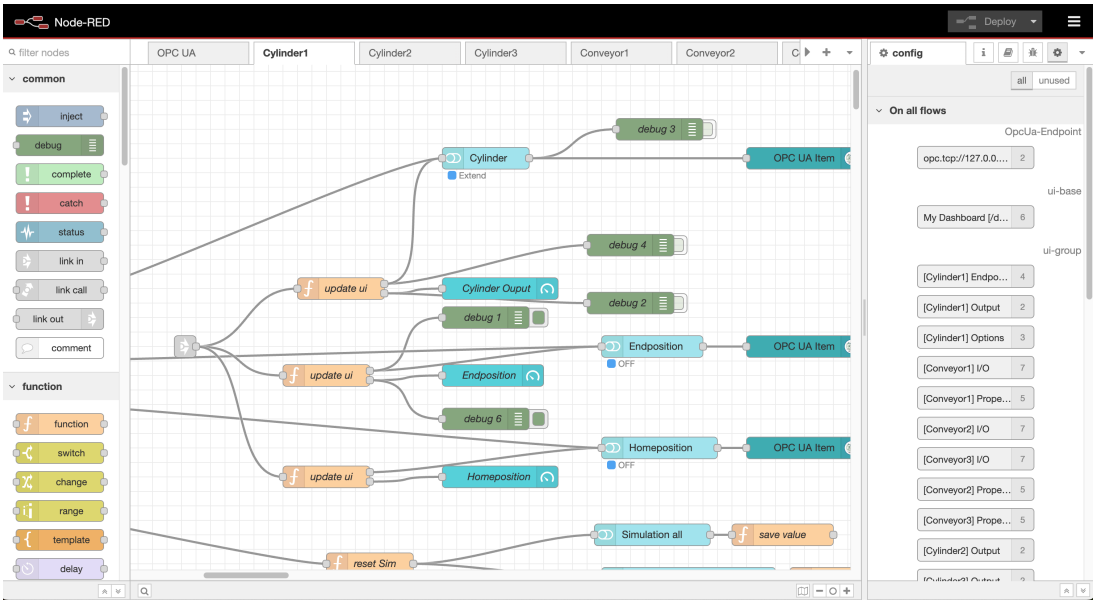

Figure 9: Node-RED editor configuration view for the Cylinder1 of the OPC UA server.

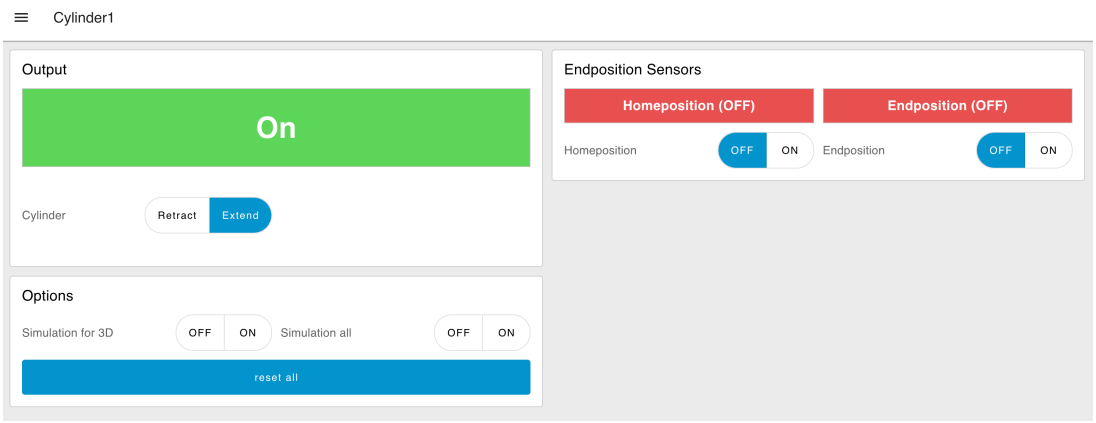

Figure 10: Node-RED editor dashboard view for the Cylinder1 of the OPC UA server.

The OPC UA server is a representation of a PLC used to control the simulation. The server enables the simulation to read and write values from the OPC UA nodes and to subscribe to value changes. AVM Engineering AG chose Node-RED to quickly deliver functionality with a simpler server setup, avoiding the complexity and cost of a custom PLC server. However, this choice limits the server's ability to handle rapid data changes, creating a bottleneck in our simulation. Despite this limitation, it does not significantly impact our thesis, as the focus is on implementing and integrating the OPC UA client with Unreal Engine.

8.2.1 OPC UA server address space

The OPC UA server address space, as explained in Section 3.3, is defined in the Node-RED editor. The address space is a hierarchical structure that defines the nodes and their data types. We created a simple address space where each machine is represented by a folder where multiple nodes reside. The nodes have unique node ids that are used to identify them. The nodes have a data type and a default value that is used to initialize them. In order to add a new node to the address space, one can simply open the Node-RED editor, and open the `Config` tab where the node `Compact-Server-Refresh` is located. Inside this node one can add the new OPC UA node as seen in Figure 11 an in Listing 1.

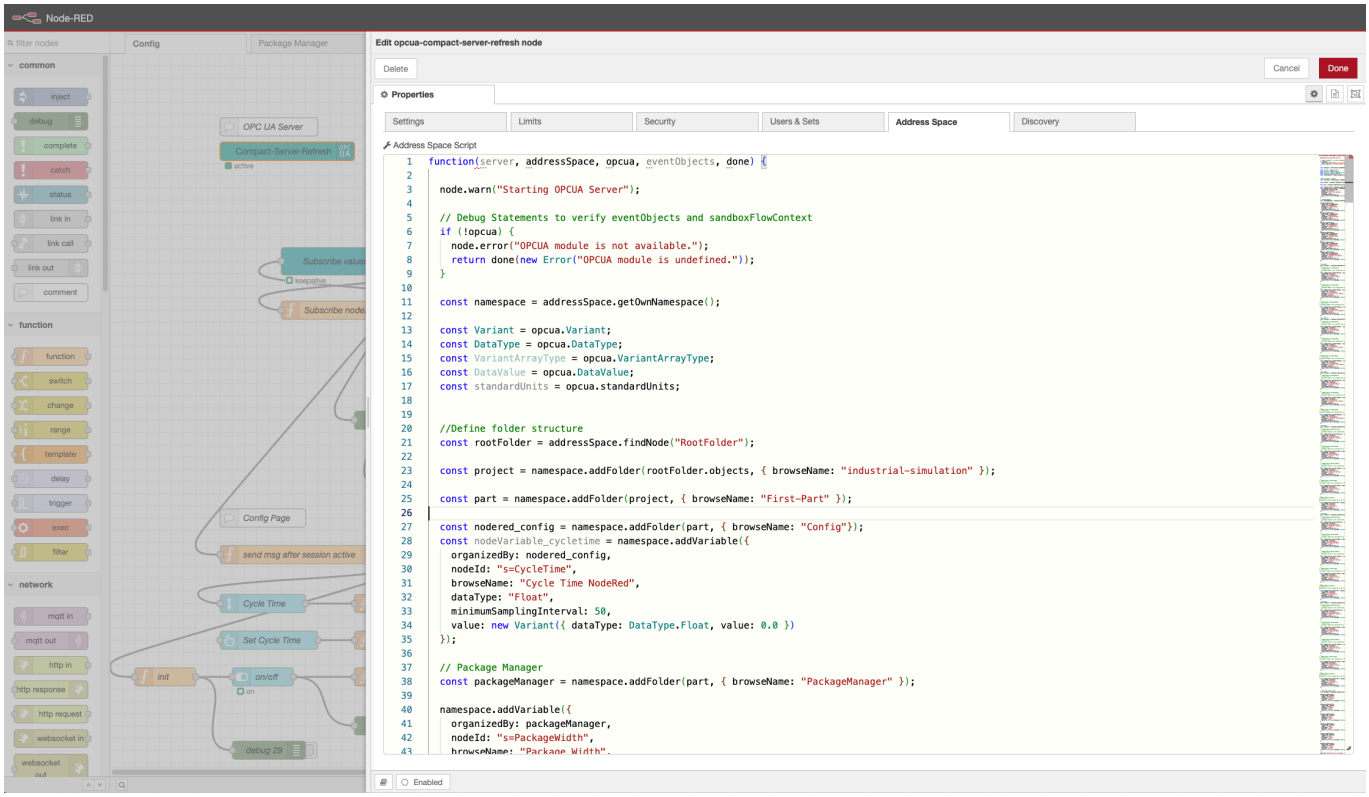

Figure 11: Address space editor where the OPC UA nodes are defined and can be added

```
const packageManager = namespace.addFolder(part, { browseName: "PackageManager" });

namespace.addVariable({
  organizedBy: packageManager,
  nodeId: "s=PackageWidth",
  browseName: "Package Width",
  dataType: "Float",
  minimalSamplingInterval: 50,
  value: new Variant({ dataType: DataType.Float, value: 1 })
});
namespace.addVariable({
  organizedBy: packageManager,
  nodeId: "s=PackageHeight",
  browseName: "Package Height",
  dataType: "Float",
  minimalSamplingInterval: 50,
  value: new Variant({ dataType: DataType.Float, value: 1 })
});
```

Listing 1: Snippet of the address space configuration of the package manager where the variables `PackageWidth` and `PackageHeight` are defined

8.2.2 OPC UA server logic

To define custom business logic for the server we created custom Node-RED flows that are used to process the data. The flows are created using the Node-RED editor and can be easily modified. The flows are responsible for creating the dashboard and the logic behind the buttons and sliders. For example as seen in Figure 12 the package manager has four input fields where the dimensions of the package and its weight can be defined. The values are then sent to the OPC UA server using the `OPC UA Item` node that is passed back to the OPC UA server. Additionally it has two buttons, one that when pressed counts up the package counter which is a simple `Int32` value to which the simulation subscribes to. This allows the simulation to be notified whenever a package should be initiated. The other button is used to delete a package. In order to delete a package a `PackageID` which is simply the counter value of the package is required. When the button is pressed the ID set in the input field is sent to the OPC UA server and the simulation is notified that the package with the given ID should be deleted.

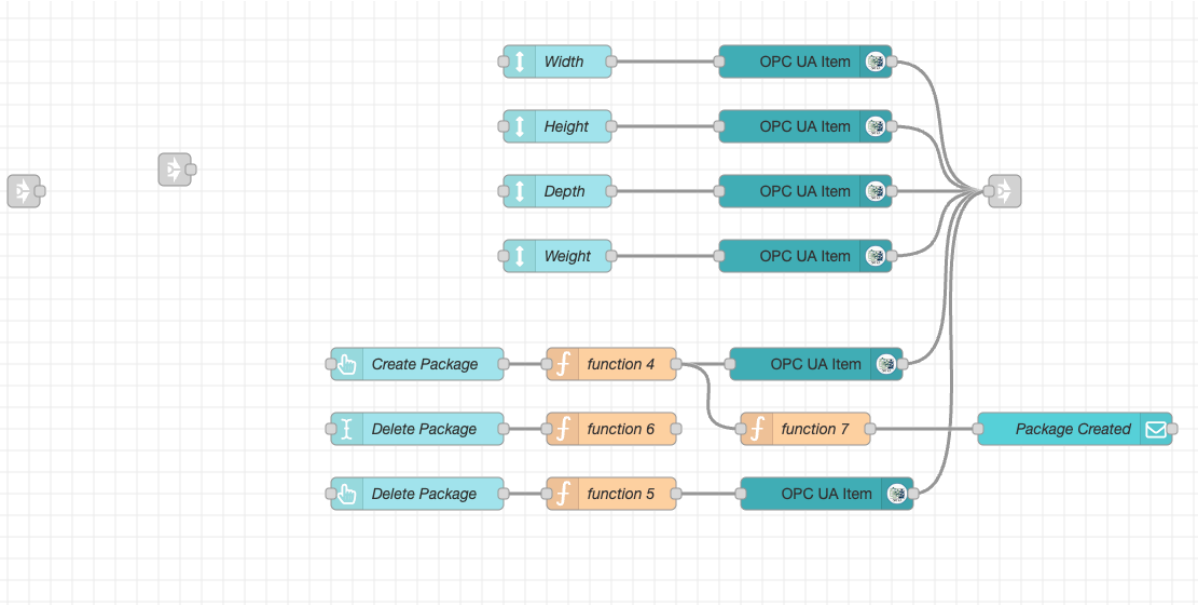

Figure 12: Flow configuration for the package manager where multiple input fields are defined and two buttons that increase the respective counter on the OPC UA server.

The input nodes that were configured in Figure 12, also generate a fully functional web dashboard where the values can be changed as seen in Figure 13.

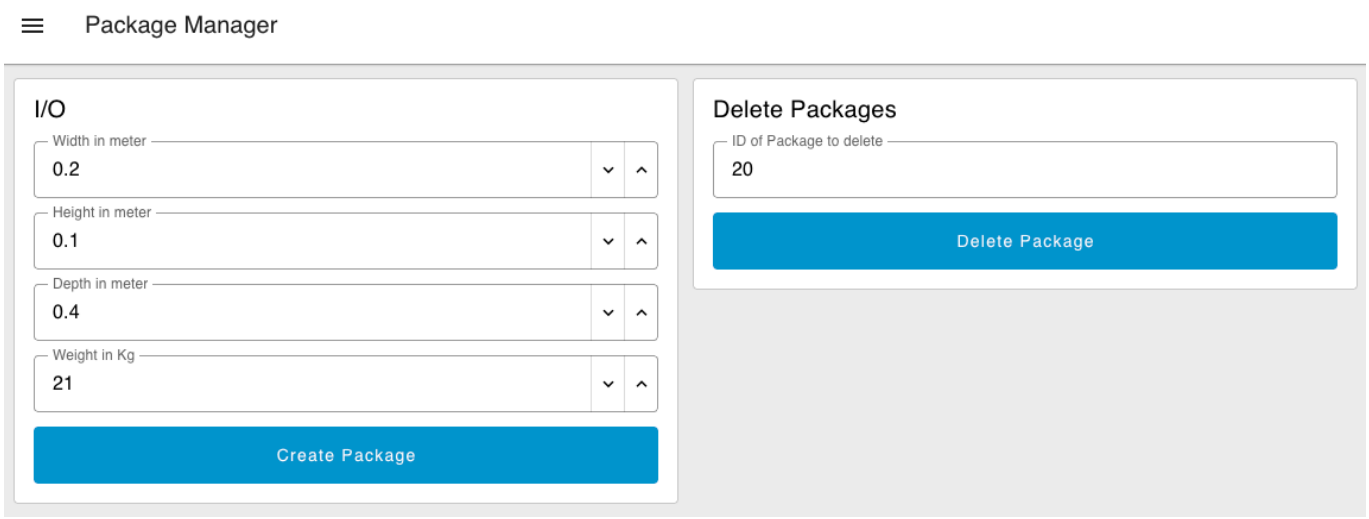

Figure 13: Dashboard generated by the nodes for the package manager where packages can be created and deleted.

The logic behind the other flows is similar in concept but differ in the details. AVM Engineering AG implemented the logic and dashboard for the remaining parts of the package sorting system.

8.2.3 Limitations

The OPC UA server is not comparable to a real PLC server. It is a simple server that is used for demonstration purposes only and was chosen because of its simplicity and ease of use. Many of the limitations of the project are due to the limitations of the server and not due to the simulation or the building blocks. The server is not capable of handling rapid data changes and is not designed for real-time data-intensive applications. Instead of being able to read data changes rapidly we had to implement a cycling timer on the server where the interval of the server, in which it accepts changes, can be adjusted. In tests we figured out that the minimum interval, where the server can still perform, is 200 ms which is not ideal for a real-time application, but is sufficient for our use case and demonstration of the building blocks. Additionally, the server is limited in the number of subscriptions and connections that it can handle. Due to this, when opening to many subscriptions, an error message is printed to the screen and to the log as seen in Figure 14. To increase the number of subscriptions, we had to manually adjust the source files inside the Node.JS packages, by increasing the hard coded amount of subscriptions per connection as documented in the `README` file inside the folder `/server/`.

Figure 14: Subscription error message because the server is limited in the number of subscriptions it can handle

The server interface utilizes Node-RED, which is based on JavaScript. However, JavaScript does not support certain data types like `Int64`. As a result, the server cannot handle operations such as reading or writing `Int64` values or performing calculations on them. Despite this limitation, the simulation itself can still read and write `Int64` values, as the necessary building blocks for these operations are implemented and tested. If the server must process `Int64` values, a workaround is to split them into two `Int32` values. Alternatively, `Int32` values can be used directly for server-side logic. This limitation only affects the server's ability to process `Int64` values and does not impact the simulation's functionality.

8.3 OPC UA client in Unreal Engine

Many components inside the Unreal Engine simulation require communication with the OPC UA server. In order to keep connections minimal, we opted for a shared client between the components. We store the client in the game instance [49]. In Unreal Engine, a game instance is a global persistent class that exists throughout the game's entire runtime, allowing you to store and manage data that needs to persist between levels or maps. The game instance is created at the beginning when starting the simulation and will not be destroyed until the simulation is shutdown. This allows us to create a single connection at the beginning and keep it open until the end. The game instance is accessible from within the components and, thus, the client is accessible from all components.

The OPC UA client is implemented using the open62541 library. The library is a C implementation of the OPC UA standard. To integrate the C library into Unreal Engine, we precompiled the library on both Windows and macOS. The binaries together with the header files are imported into the Unreal Engine build pipeline as runtime dependencies. On Windows the library is linked dynamically and on macOS it is linked statically because at the time of writing this thesis, the dynamic linking for macOS was not working. Due to this limitation it is required to install open62541 on macOS with Homebrew in order to use the project. The header files differ for Windows and macOS. Due to this we adjusted the build script to include the correct header files depending on the platform of the machine that was running Unreal Engine as seen in Listing 2.

```
string Open62541Path = Path.Combine(ModuleDirectory, "../ThirdParty/open62541");
if (Target.Platform == UnrealTargetPlatform.Mac)
{
  PublicIncludePaths.Add(Path.Combine(Open62541Path, "macOS/include"));
  PrivateIncludePaths.Add(Path.Combine(Open62541Path, "macOS/include"));
      string Open62541Lib = Path.Combine(Open62541Path, "lib/libopen62541.1.4.10.dylib");
      PublicAdditionalLibraries.Add(Open62541Lib);
      RuntimeDependencies.Add(Open62541Lib);
}
else if (Target.Platform == UnrealTargetPlatform.Win64)
{
  PublicIncludePaths.Add(Path.Combine(Open62541Path, "windows/include"));
  PrivateIncludePaths.Add(Path.Combine(Open62541Path, "windows/include"));
      string Open62541Lib = Path.Combine(Open62541Path, "lib/open62541.lib");
      PublicAdditionalLibraries.Add(Open62541Lib);
      RuntimeDependencies.Add(Open62541Lib);
}
```

Listing 2: Build script that includes the correct header files and links the correct library depending on the platform.

Additionally for Windows we had to include the `winsock2.h` and `ws2tcpip.h` header files in order to use the socket library as seen in Listing 3. These includes are required in files where read or write operations are executed.

```
#ifdef _WIN32
  #include <winsock2.h>
  #include <ws2tcpip.h>
#endif
```

Listing 3: Windows specific includes for the socket library.

The behavior and configuration of the client is defined in the `Source/industrialSimulation/Private/Industrial_Sim_GameInstance.cpp` file as seen in Listing 4. To adjust the behavior of the client, one can change the settings there manually.

```
void UIndustrial_Sim_GameInstance::Init()
{
  Super::Init();
  UAClient = UA_Client_new();
  UA_ClientConfig_setDefault(UA_Client_getConfig(UAClient));

  UE_LOG(LogTemp, Log, TEXT("Attempting to connect to OPC UA server at URL: %s"), *ServerURL);

  // adjust to the correct server url
  UA_StatusCode status = UA_Client_connect(UAClient, TCHAR_TO_ANSI(*ServerURL));
  if (status != UA_STATUSCODE_GOOD)
  {
      UE_LOG(LogTemp, Error, TEXT("OPC UA Connection Failed with error code: 0x%X"), status);
      UA_Client_delete(UAClient);
      UAClient = nullptr;

      GetWorld()->GetTimerManager().SetTimerForNextTick(this,
&UIndustrial_Sim_GameInstance::TryShowErrorWidget);
  }
  else
  {
      UE_LOG(LogTemp, Log, TEXT("OPC UA Connection Successful!"));
  }
}
```

Listing 4: Configuration of the OPC UA client server connection which can be adjusted to the needs. The server URL can be changed in the GameInstance.

To configure the server connection, we created a custom Blueprint wrapper around the game instance where the variables can easily be adjusted without the need of recompiling the entire source code. The wrapper is editable in the Unreal Engine editor by double clicking on the “GameInstance” Blueprint under `/All/Content/SimBlank/Blueprints/`, as seen in the following two figures.

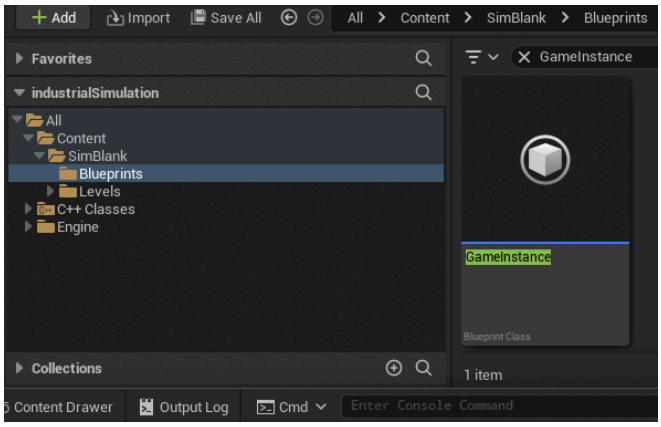

Figure 15: Content Drawer inside the Unreal Engine editor. This is where the GameInstance is located.

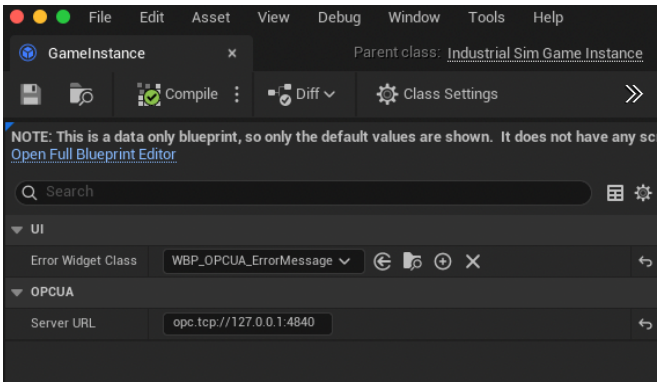

Figure 16: OPC UA client connection settings to configure the error message visualizer and the server connection url.

)

The game instance is accessible through the world context. In order for a custom blueprint to access the client, the blueprint declaration requires the world context as a meta information as seen in Listing 5.

```
UFUNCTION(BlueprintCallable, Category = "OPC UA", meta = (WorldContext = "WorldContextObject"))
static FOptionalBool ReadBoolean(UObject* WorldContextObject, const FString& NodeId);
```

Listing 5: Declaration of a function that is accessible as a Blueprint in the Unreal Engine with the `WorldContext` as a meta parameter.

Inside the function definition we can then access the game instance and the client as seen in Listing 6.

```
UIndustrial_Sim_GameInstance* GameInstance = Cast<UIndustrial_Sim_GameInstance>(
  WorldContextObject->GetWorld()->GetGameInstance()
);
UA_Client* UAClient = GameInstance->GetUAClient();
```

Listing 6: Code snippet that accesses the game instance from the `WorldContext` in order to use the global instance of the OPC UA client.

8.4 Reading data from the OPC UA server

We created custom Blueprint functions that simplify the communication to the OPC UA server for the developers. The Blueprint functions are accessible from the Unreal Engine editor and can be used with ease as seen in Figure 17.

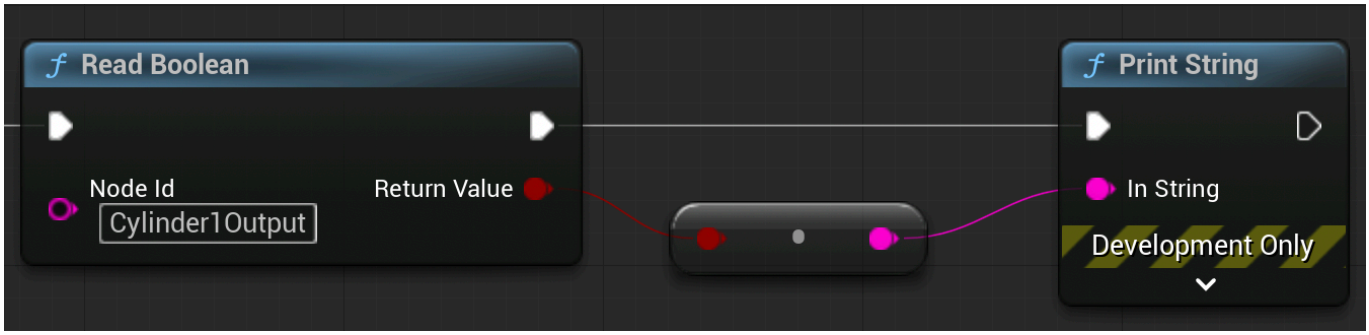

Figure 17: Example Blueprint configuration that reads a boolean value from the OPC UA server and prints the value to the screen.

Whenever a developer requires to read a value from the OPC UA server, they can simply add the dedicated Blueprint function to their existing logic and connect the given input and output pins. The Blueprint function requires the OPC UA node ID of the value to be read. The world context is automatically passed to the function and is required in order to access the game instance and the OPC UA client. As seen in Figure 18, the read operation retrieves the global client instance and then tries to read the value from the server. The read operation is blocking and will wait until the value is received before triggering the following nodes. The returned status code is checked if the operation was successful or not. If the operation was not successful, the function will print an error message to the log and to the screen so that the developer is informed about the error. The status code can be checked manually in the status code documentation. If the returned value is not of the data type of the dedicated node, e.g. the "Read Boolean" function expects the data type to be a boolean, a message is printed to the log and to the screen. If everything went well, the function will return the value making it available for processing in subsequent Blueprint functions.

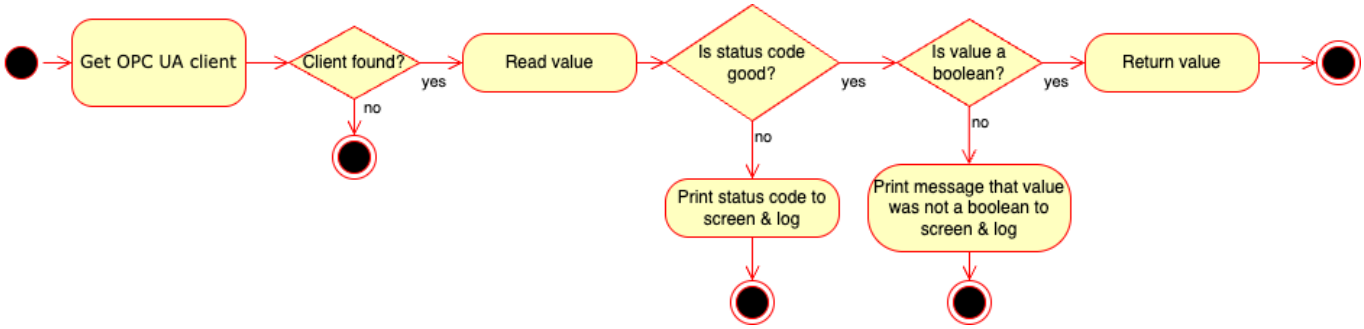

Figure 18: Flow chart diagram that displays how the read request of the boolean value from the OPC UA server is handled. This also applies to other data types.

We implemented custom Blueprint functions for the most common OPC UA data types that should cover most of the use cases. The following table shows the implemented Blueprint functions and the corresponding OPC UA data types and the Unreal Engine data types.

Blueprint function	**OPC UA data type**	**Unreal Engine data type**
Read Boolean	Boolean	bool
Read Double	Double	double
Read Float	Float	float
Read Int32	Int32	int32
Read Int64	Int64	int64
Read String	String	FString

Table 3: A list of all the Blueprint functions that can read a specific data type value from the OPC UA server

8.5 Writing data to the OPC UA server

Besides reading data from an OPC UA server it is also possible to write data to an OPC UA server. Identically to the reading functions the Blueprint functions for the writing operations are accessible from the Unreal Engine editor and can be used in the same way. An example for this would be shown in Figure 19.

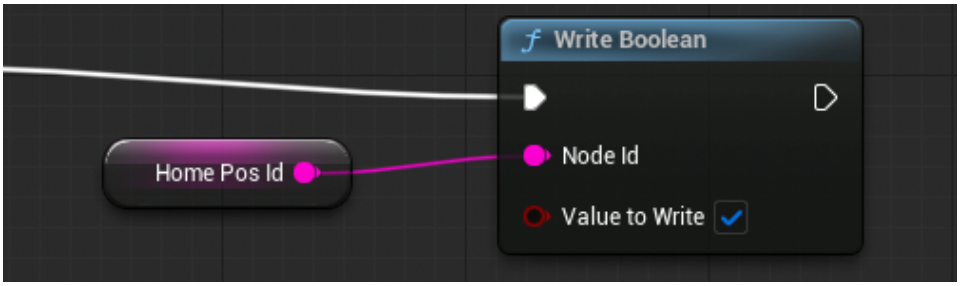

Figure 19: Example Blueprint configuration that writes a boolean value to the OPC UA server. Home Pos Id is a variable that represents the OPC UA node ID.

The logic behind a write Blueprint works very similar to the logic behind the read operation. A Blueprint function can be added to the existing logic and the input pins can be connected. The operation does not return a value like the read operation. The Blueprint requires the OPC UA node ID to which the operation should write the value to and the value itself. As seen in Figure 20, the write operation retrieves the global client instance and then tries to write the value to the server. The status code is checked if the operation was successful or not. If an error occurred, the Blueprint will print the error message to the log and to the screen. If the status code is as expected, the Blueprint function is finished and the next Blueprint function gets called.

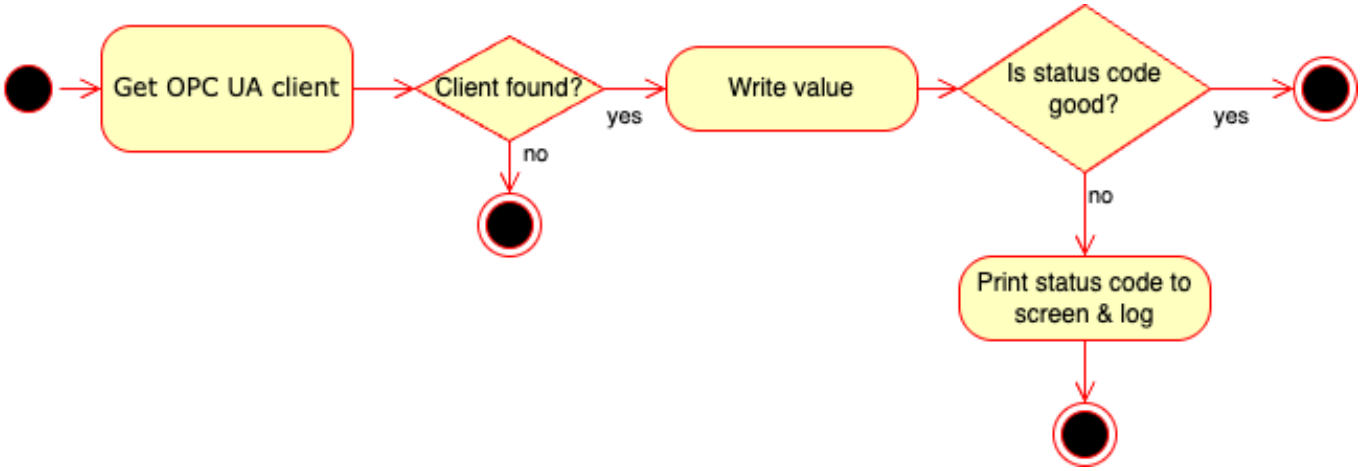

Figure 20: Flow chart diagram that displays how the write request of the value to the OPC UA server is handled.

We implemented custom Blueprint functions for the most common OPC UA data types that should cover most of the use cases. The following table shows the implemented Blueprint functions and the corresponding OPC UA data types and the Unreal Engine data types.

Blueprint function	OPC UA data type	Unreal Engine data type
Write Boolean	Boolean	bool
Write Double	Double	double
Write Float	Float	float
Write Int32	Int32	int32
Write Int64	Int64	int64
Write String	String	FString

Table 4: A list of all the Blueprint functions that can write a specific data type value to the OPC UA server

We are aware that there are many more data types available in the OPC UA standard. We decided to implement only the most common ones in order to keep the project simple, understandable and maintainable. If a developer requires a different data type, they can easily implement it themselves using the existing Blueprint functions as a reference. The implementation of the Blueprint functions is done in C++ and is accessible in the `Source/industrialSimulation/Private/OPC_UA.cpp` file. The declaration can be found in the `Source/industrialSimulation/Public/OPC_UA.h` file. For certain data types like the `Write String` node, we had to convert the data type to the Unreal Engine specific `FString` data type. It should be noted that the same could apply to other data types as well.

If one would decide to extend the existing supported data types with additional ones, it is important to note that after extending the files, the project has to be recompiled in order for the changes to take effect. Existing usages of the Blueprint functions could be invalidated due to the recompilation. We highly suggest to restart Unreal Engine after serious changes to the source code to avoid any caching issues with Unreal Engine.

8.6 Subscriptions

Some components in the simulation need to act when a certain value changes in real-time. One could simply poll the OPC UA server for changes, but this is inefficient and can lead to performance issues. Instead, we use OPC UA subscriptions [50] to get notified when a value changes. The OPC UA subscriptions differs from a traditional server-to-client communication like WebSockets because the server does not know the client. The client has to request what OPC UA node should be monitored, but the client has to periodically check if any changes happened to the OPC UA node. In open62541, this is done by calling `UA_Client_run_iterate()` periodically.

In our simulation, we setup a global timer that calls this function every 0.099 seconds with a very short blocking timeout of one millisecond in order to keep the simulation smooth and performant as seen in Figure 21. This results in a polling interval of 0.1 seconds. The timer is setup in the game mode as soon as the simulation is started and is destroyed when the simulation is stopped. This ensures that the simulation is always checking for changes but components that are interested in the subscriptions do not have to setup a timer to call the function themselves. The `UA_Client_run_iterate()` requires the active OPC UA client instance. For this reason we access the game instance where the OPC UA client is stored and give the OPC UA client as a parameter to the blueprint function. If the interval or the blocking time should be changed, open `BP_SimGameMode` and adjust the values as needed inside the Unreal Engine editor.

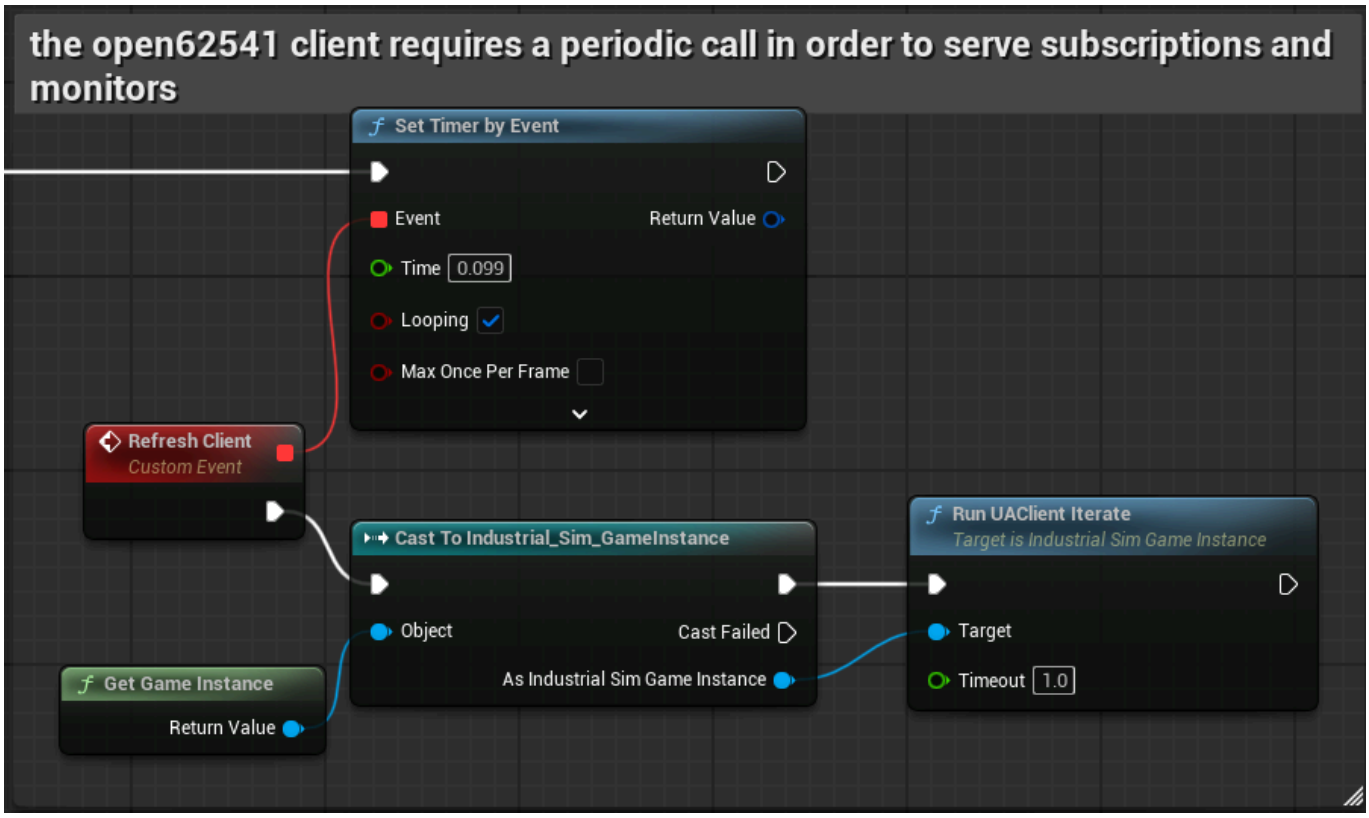

Figure 21: Blueprint function configuration that sets up a timer to call the `UA_Client_run_iterate()` function periodically in order to check for changes in the active subscriptions.

To subscribe to a certain OPC UA node, a Blueprint actor is required to have a `OPCUASubscriber` as a local variable as seen in Figure 22. The `OPCUASubscriber` is an instance that stores all the information about the subscription and is responsible to trigger the correct callback whenever the value changes.

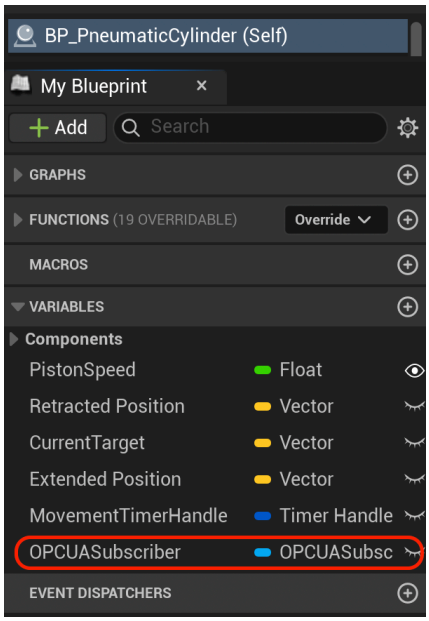

Figure 22: The custom created variable of type OPCUASubscriber that holds the information about the subscription and the callback function.

The variable requires an instance which has to be initiated with the `Create OPCUASubscriber` Blueprint function. Then the instance can subscribe to a node with the Blueprint function `Subscribe to a Boolean Node` using the OPC UA node ID and specify the callback that should be triggered whenever the value changes as seen in Figure 23. The custom event that can be called has the updated value as a parameter which can be processed further according to the requirements. Additionally, it also has an additional parameter called `Source Timestamp`. This parameter is a timestamp that represents the time when the value was requested for change by another OPC UA client, not when the server received and applied the change. In our example, this would be the time when a value is changed in the web-based dashboard. This timestamp is important for real-time systems like the conveyor belts that have to do some calculations considering the timespan between two datapoints. In Figure 23, it triggers a further custom event depending on the value of the new boolean.

A more advanced usage of the subscriptions is seen in the conveyor belt as seen in Section 10.5.

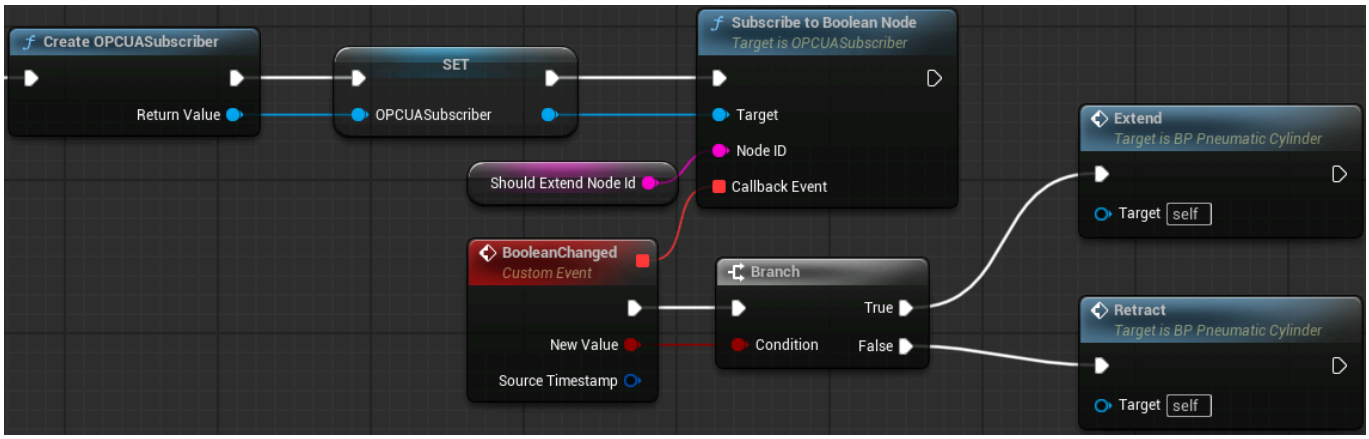

Figure 23: Blueprint function configuration that setups a subscription to a boolean OPC UA node with a callback function that either retracts or extends the pneumatic cylinder.

The `Subscribe to Boolean` Blueprint function is responsible for creating the subscription and setting up the monitored item. If the setup fails, a visual error message is displayed on the screen and printed to the log informing the developer of the error.

In Figure 24, we show the sequence diagram of the subscription process with the example of a pneumatic cylinder where the cylinder is extended and then retracted. The OPC UA client is created in the game instance and the subscription is initiated in the Blueprint actor. The subscriber periodically checks for new notifications from the monitored item due to the periodic call inside the game mode as documented at the beginning of this sub-chapter. The desired position is manually set in the dashboard or is automatically set via the PLC software. If a notification is received, the subscriber calls the callback function with the new value. The callback function then processes the new value and triggers the custom events that turn the correct sensors on or off depending on the value of the boolean.

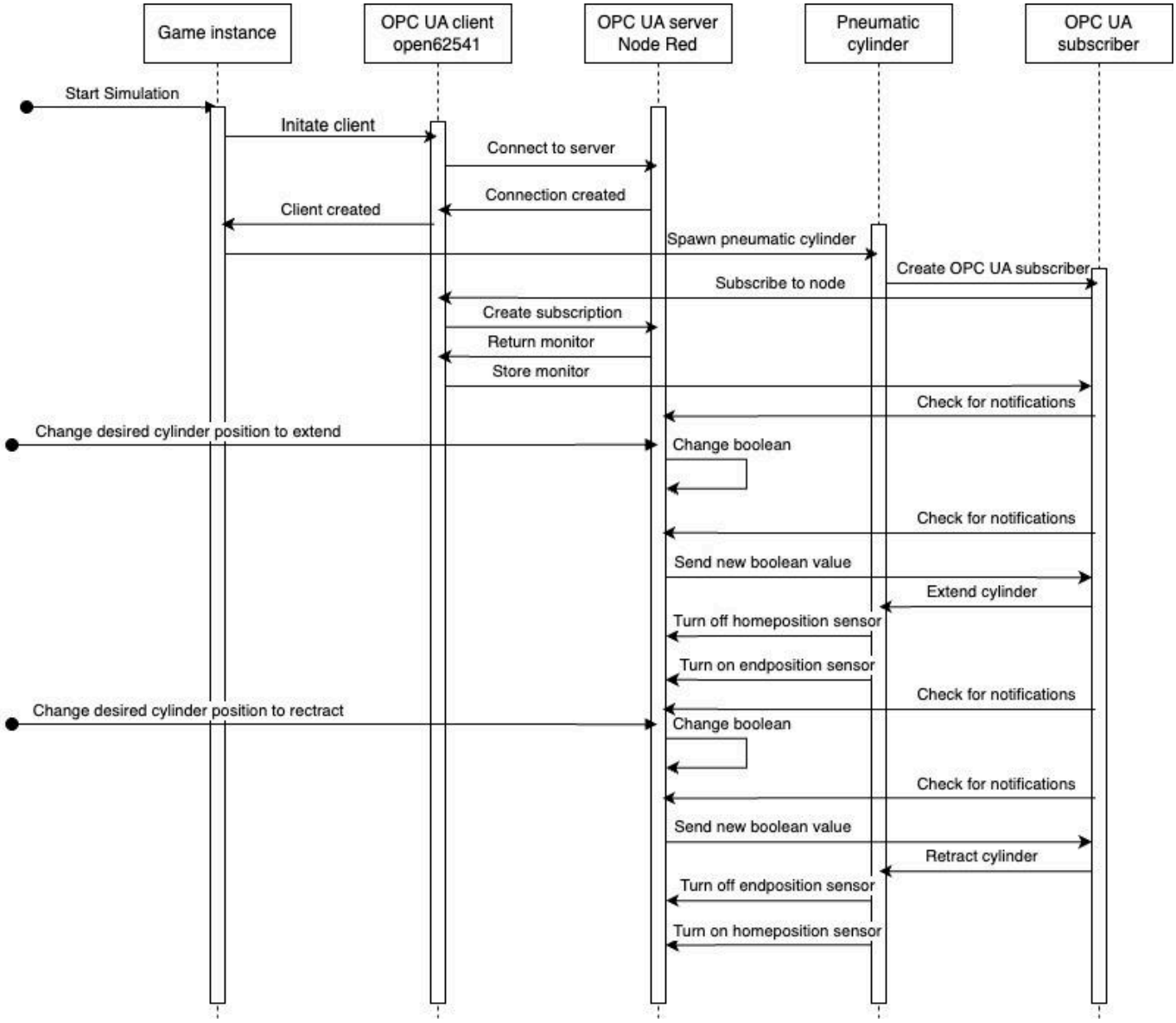

Figure 24: Sequence diagram that shows the entire interactions between the simulation and the OPC UA server. Shows the process of the pneumatic cylinder that creates a subscription to a boolean node and the process of receiving a notification from the server that triggers the respective callback function.

As with the read operations, we provided custom Blueprint functions for the most common OPC UA data types, enabling subscriptions to data changes for typical use cases. Table 5 shows the implemented Blueprint functions and the corresponding OPC UA data types and the Unreal Engine data types.

Blueprint function	OPC UA data type	Unreal Engine data type
Subscribe to Boolean Node	Boolean	bool
Subscribe to Double Node	Double	double
Subscribe to Float Node	Float	float
Subscribe to Int32 Node	Int32	int32
Subscribe to Int64 Node	Int64	int64
Subscribe to String Node	String	FString

Table 5: A list of all the Blueprint functions that can subscribe a specific data type value from the OPC UA server

We are aware that there are many more data types available in the OPC UA standard. We decided to implement only the most common ones in order to keep the project simple, understandable and maintainable. If a developer requires a different data type, they can easily implement it themselves using the existing Blueprint functions as a reference. The implementation of the Blueprint functions is done in C++ and is accessible in the `Source/industrialSimulation/Private/OPCUA_Subscriber.cpp` file. The declaration can be found in the `Source/industrialSimulation/Public/OPCUA_Subscriber.h` file. For certain data types like the `Subscribe to String Node`, we had to convert the data type to the Unreal Engine specific `FString` data type. It should be noted that the same could apply to other data types as well.

If one would decide to extend the existing supported data types with additional ones, it is important to note that after extending the files, the project has to be recompiled in order for the changes to take effect. Existing usages of the Blueprint functions could be invalidated due to the recompilation. We highly suggest to restart Unreal Engine after serious changes to the source code to avoid any caching issues with Unreal Engine.

In order to bind the callback function to the subscription, we use Unreal Engine dynamic delegates as seen in Listing 7. Currently the new value and the source timestamp are passed to the callback function. If other parameters are required, they can be added to the delegate function declaration in `Source/industrialSimulation/Public/OPCUA_Subscriber.h`. Note that the delegate function name has to match the number of logical parameters. So if you add a third parameter, the delegate function name has to change from `DECLARE_DYNAMIC_DELEGATE_TwoParams` to `DECLARE_DYNAMIC_DELEGATE_ThreeParams`. Currently Unreal Engine supports up to eight parameters in a dynamic delegate.

```
DECLARE_DYNAMIC_DELEGATE_TwoParams(FOnOPCBooleanChanged, bool, NewValue, const FDateTime&,
SourceTimestamp);
DECLARE_DYNAMIC_DELEGATE_TwoParams(FOnOPCFloatChanged, float, NewValue, const FDateTime&,
SourceTimestamp);
DECLARE_DYNAMIC_DELEGATE_TwoParams(FOnOPCDoubleChanged, double, NewValue, const FDateTime&,
SourceTimestamp);
DECLARE_DYNAMIC_DELEGATE_TwoParams(FOnOPCStringChanged, const FString&, NewValue, const
FDateTime&, SourceTimestamp);
DECLARE_DYNAMIC_DELEGATE_TwoParams(FOnOPCInt32Changed, int32, NewValue, const FDateTime&,
SourceTimestamp);
DECLARE_DYNAMIC_DELEGATE_TwoParams(FOnOPCInt64Changed, int64, NewValue, const FDateTime&,
SourceTimestamp);
```

Listing 7: Declaration of the dynamic delegates that are used to bind the callback functions to the subscription. Only the new value and the source timestamp are passed to the callback function. If other parameters are required, they can be added to the delegate function declaration.

9 Project setup

In this chapter, we guide how to install and setup the Unreal Engine in order to use it for a project.

9.1 Machine specifications

Unreal Engine is a powerful game engine that requires a decent machine to run. We opted to use low graphics settings in order reduce the minimal specifications required to run the project. We tested the project on the following machines and got fluent performance with a minimum of 45 frames per second (FPS) in the editor:

- MacBook Pro 14″ (2021) with Apple M1 Pro, 16 GB RAM
- MacBook Pro 16″ (2021) with Apple M1 Pro, 16 GB RAM
- Custom tower PC with AMD Ryzen 9 3900X 3.8 GHz, 32 GB RAM, NVIDIA RTX 2070 Super
- Dell Precision 7780 with Intel i9-13950HX 2.2 GHz, 64 GB RAM, NVIDIA RTX 3500 Ada
- Dell Precision Workstation 7760 with Intel Xeon W-11955M 2.6 GHz, 64 GB RAM, NVIDIA RTX A3000
- HP Zbook Fury 16 G11 with Intel i9-14900HX, 128 GB RAM

The project was mainly developed and tested on the two MacBook Pro machines. The other machines were tested at the end of the project to ensure the project runs on the machines AVM Engineering AG uses.

Recommended specifications:

- OS: Windows 11+ or macOS 14.5+ (no support for older versions or other OS)
- 16+ GB RAM
- M1 Pro or better
- optionally a dedicated GPU with 4+ GB VRAM

If the project is not running smoothly on the machine, the graphics settings can be changed in the editor. To do so, open the “Settings” menu in the top right corner of the editor and select “Engine Scalability Settings”. This will open a menu where one can change the graphics settings as seen in Figure 25. We recommend to set the settings to “Low” or “Medium” for a fluent experience.

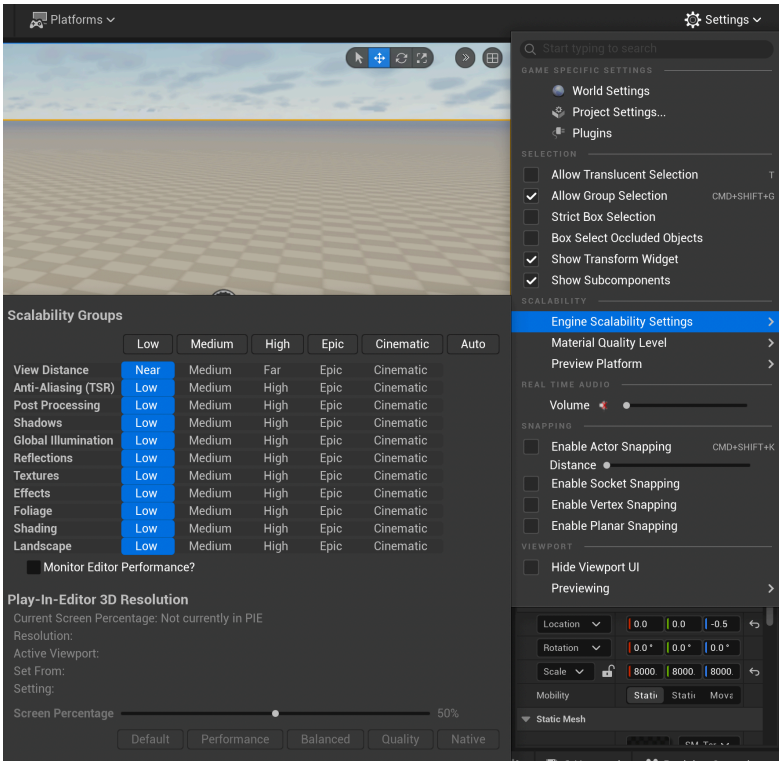

Figure 25: Engine Scalability Settings in the Unreal Engine editor with the recommended settings for maximum performance.

We used a Blueprint function that automatically sets the quality to “Low” in order to ensure a fluent experience when the project is started as seen in Figure 26. If this is not desired the node can be removed from the game mode Blueprint called `BP_SimGameMode`. This blueprint is located in the folder `Content/SimBlank/Blueprints`.

Figure 26: Blueprint functions to set the engine scalability to low by default.

9.2 Installation

Unreal Engine is free to download [51]. Follow the link and download the Epic Games launcher in order to install the Unreal Engine to the computer. One is required to create and log in to an Epic Games account to be able to use the engine. Once logged in, navigate to the Unreal Engine tab and click on the “Library” tab in order to select and install the Unreal Engine with the version **5.5.4**. After downloading the correct version one can start the engine as seen in Figure 27.

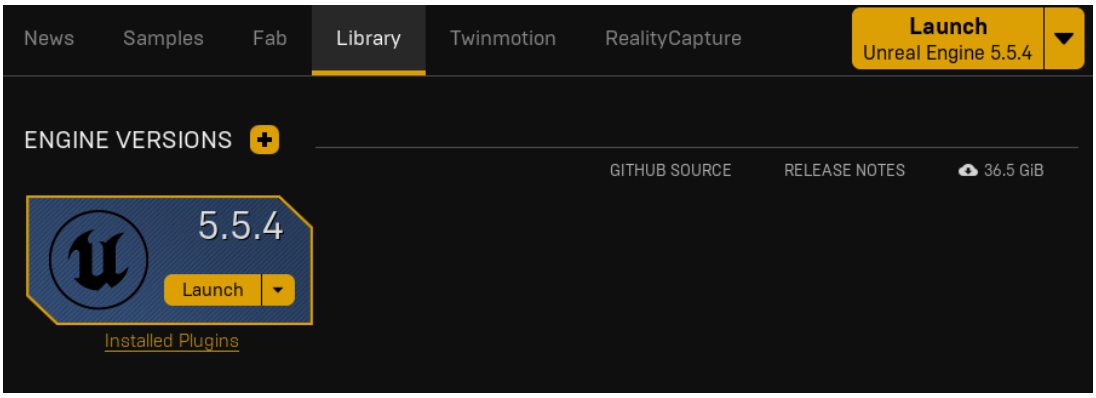

Figure 27: Starting screen in Epic Games client for Unreal Engine.

Once installed the project can be started by opening `/unreal/industrialSimulation.uproject` with Unreal Engine. When loaded, one can see the Unreal Engine editor. In the following legend, we explain the editor interface as seen in Figure 28.

1. Tab bar and menu bar ⟶ used to select general tools and configure the editor windows.
2. Toolbar ⟶ displays a group of commands providing quick access to commonly used tools and operations.
3. Viewport ⟶ displays the current loaded level and enabling to perform modifications to the world.
4. Outlier ⟶ displays all of the actors placed in to the level in a hierarchical tree.
5. Details ⟶ displays the details of the selected actor from the outlier menu.
6. Content Drawer ⟶ primary area for creating, importing, organizing viewing and managing content assets.
7. Bottom Toolbar ⟶ contains access to the output log, a shortcut to compile C++ code and access to the revision control feature.

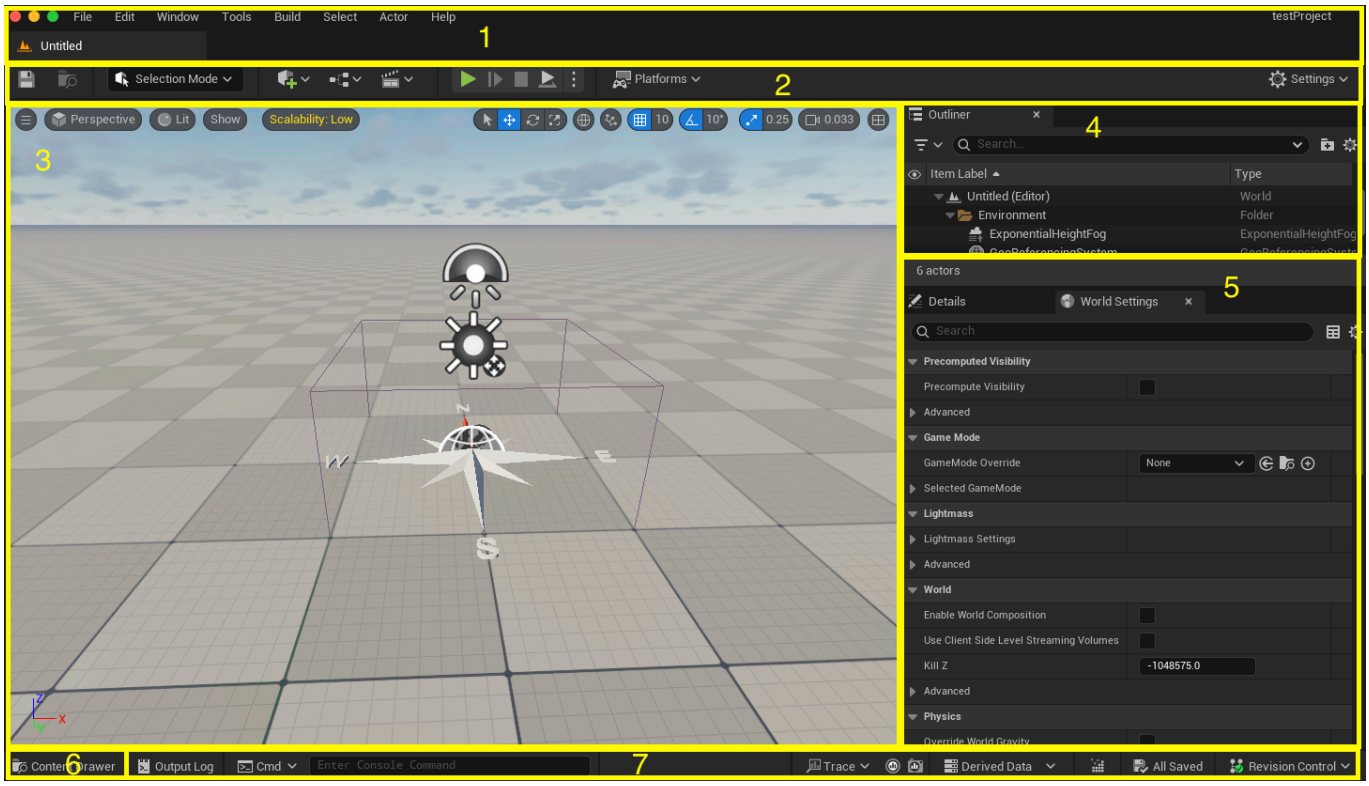

Figure 28: Unreal Engine editor interface for modeling a 3D level in the Selector Mode.

We strongly recommend completing a tutorial before starting this project to familiarize yourself with the editor, shortcuts, and basic architecture. Check out this tutorial to get started.

In order to use the project, set up the rest of the project as described in the README file in the root folder of the project.

9.3 Version Control

Unlike a traditional source code projects, a game engine projects contains many different assets in binary format. Git, the most used version control software, cannot understand the binary changes and thus cannot propose any data merges. When working on the project with multiple people, it is important to pull the changes from the Git server before starting to work on the project. Because the assets are in binary form, Git is only able to keep one of two different versions of the same asset resulting in possible lost changes when not handled with care. To avoid this, communicate with each other on which assets work is being done and pull early and often.

To track changes Unreal Engine has an integrated source control feature.

To enable version control one can simply press on “Revision Control” in the bottom left corner where one can setup the Git configs. Note that it is easier if the Git project is setup the command line tool and then attach Unreal Engine to the existing Git project config rather than starting a project from the UI.

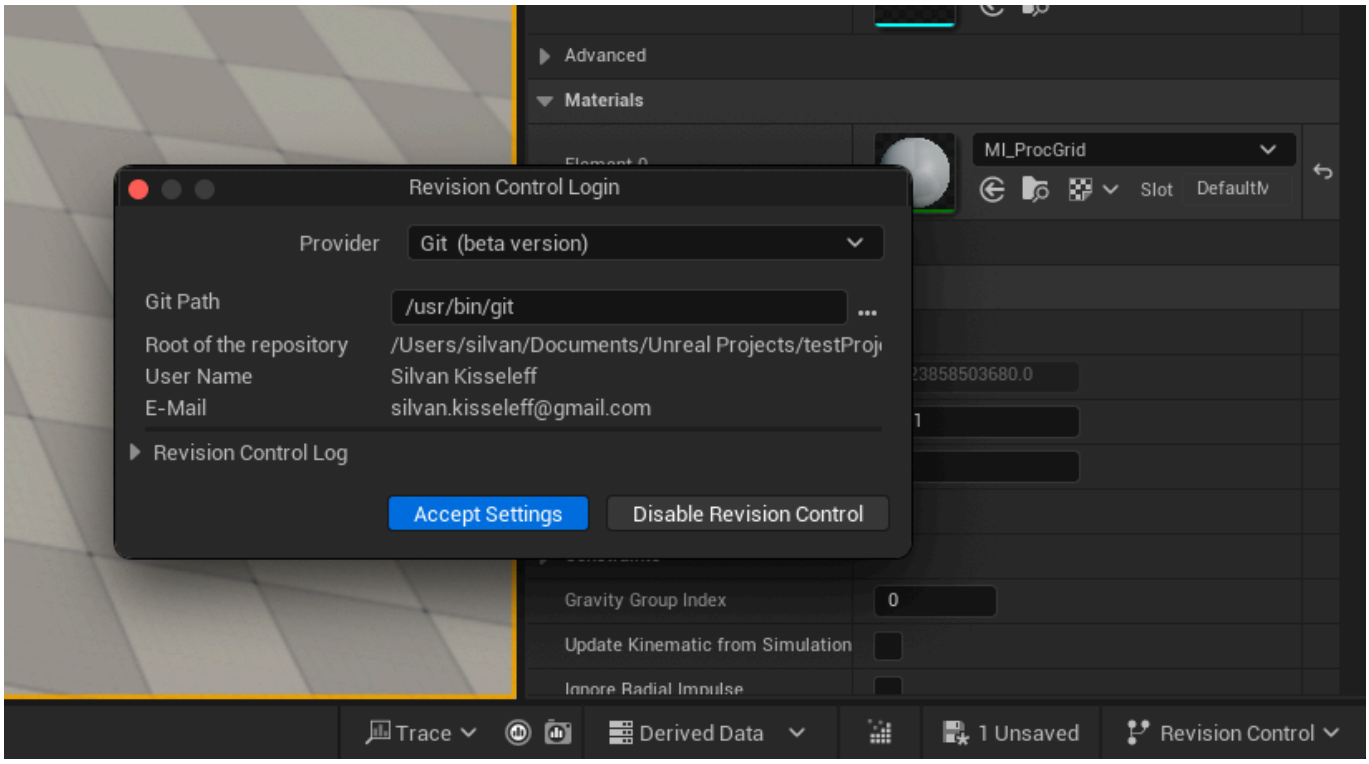

Figure 29: Settings window for the Unreal Engine Revision Control setup.

Now whenever one has changes, the version control system not only requires to user to save them but also check them in for the source control. To do so click again on the "Revision Control" tab and then click "Submit Content". This will open a dialog allowing one to specify the commit message for the changes.

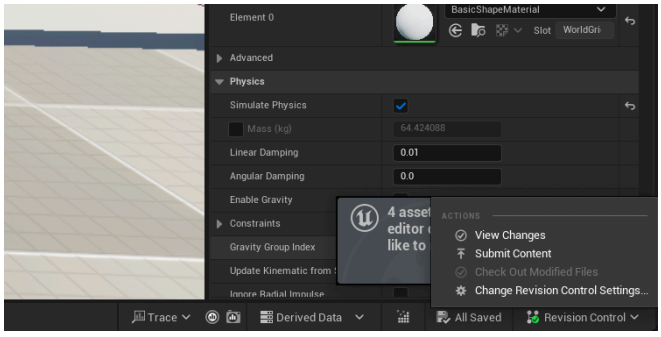

Figure 30: Submit content in Unreal Engine to the revision control via user interface.

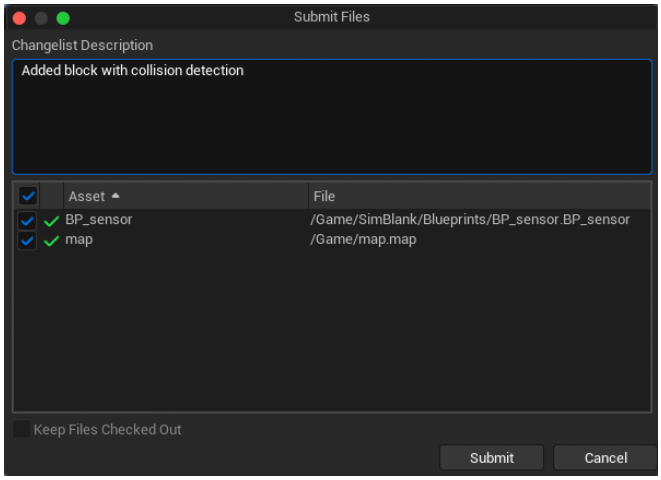

Figure 31: Select files and write a message for the commit in the Unreal Engine user interface.

Alternative to using the "Revision Control" feature, one could also manually track changes using the command line tool. To do so, open the command line tool and navigate to the project folder. Once there, one can track the changes using the following command:

```
# show changed files
git status
# add a specific changed file
git add <PATH_TO_FILE>
```

```
# add a meaningful commit message
git commit -m "<COMMIT_MESSAGE>"
```

Because the changes are mostly in binary form, it is important to make meaningful commit messages that state what and why things changed.

The changes are now tracked but need to be pushed to the Git server, this cannot be done through the UI so one will still be required to open the command line tool in order to push the commits.

```
# push changes to server
git push
```

9.4 Converting CAD files to Unreal Engine models

Rather than modeling the objects in Unreal Engine, one can leverage CAD models that are already available for real-world industrial machines. These CAD models are highly detailed and can be used in Unreal Engine with some conversion steps. There are multiple ways to convert CAD files to Unreal Engine models. Here, we will explain the two most common ways to do so.

There is a **limitation** with both approaches because the importers used, as of Unreal Engine 5.5.4, only work for the Microsoft Windows platform [52].

Datasmith Plugin in CAD software

For most CAD software, there is a plugin available that allows to export the CAD model into the Datasmith format that is developed by Epic Games specifically for Unreal Engine [53]. The Datasmith file format is a binary file that contains all the information about the model and can be used in Unreal Engine. This Datasmith file can be imported into Unreal Engine by simply dragging and dropping it into the content drawer. This will automatically create all the materials and textures that are used in the model [54].

Direct import of CAD files

Unreal Engine also supports the direct import of CAD files in certain file formats with the use of the "Datasmith CAD Importer"-plugin as seen in Figure 32. With this plugin, one can import CAD files directly into Unreal Engine without the need for a plugin in the CAD software. The plugin supports the most common CAD file formats [53].

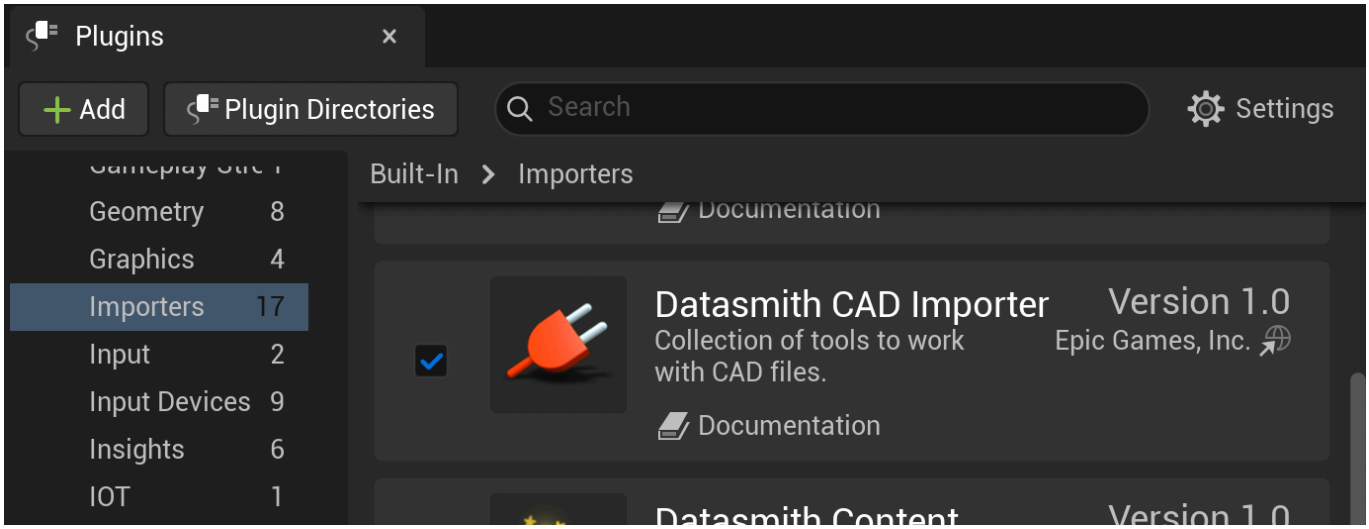

Figure 32: Datasmith CAD Importer plugin that can be enabled in the plugin manager of Unreal Engine. The plugin is responsible for converting CAD files to Unreal Engine models.

As a demonstration, we used a CAD model that we could freely download from GrabCAD [55]. The model is a conveyor belt and is contained in a single ".STEP" file. This file format stands for "Standard for the Exchange of Product Data" and is also known as ISO 10303 [56]. It is a common file format for CAD models because it contains the complete information about the model not just the geometry. The file can be imported into Unreal Engine by either simply dragging and dropping it into the content drawer or by using the import button under the "Add Object to the Project" button as seen in Figure 33.

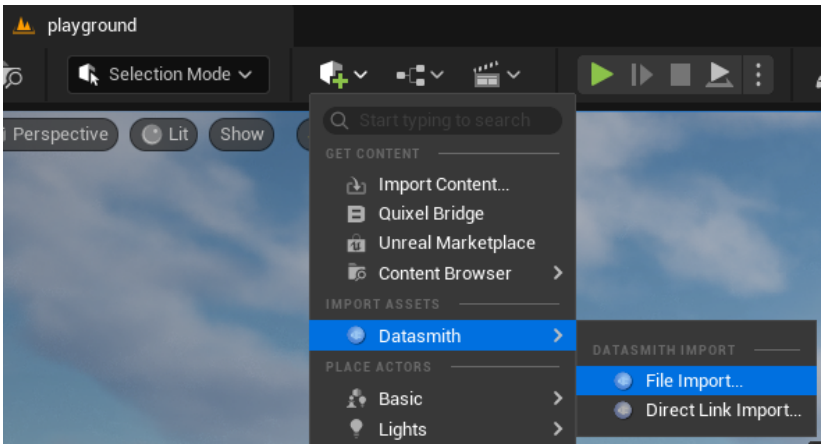

Figure 33: Import button to convert a CAD file to a Unreal Engine model.

After selecting the file and the location where the model should be imported, a dialog will open where one can select the import settings as seen in Figure 34. The settings are set to default values and should be sufficient for most cases.

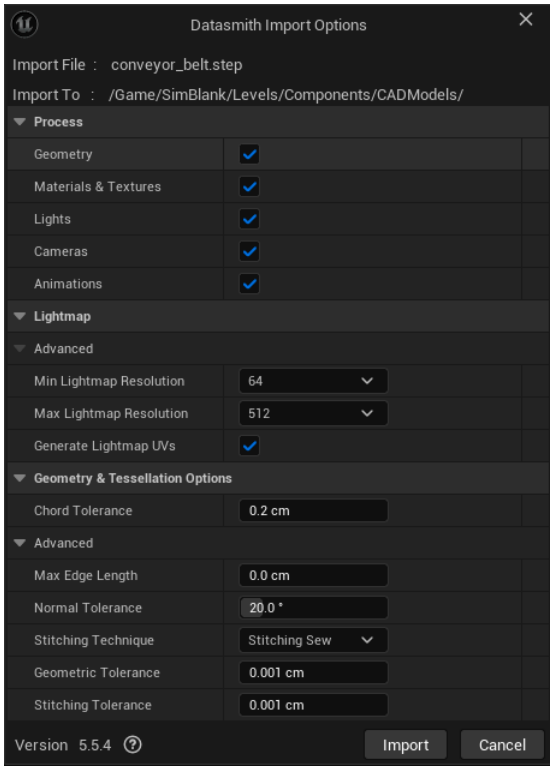

Figure 34: Import settings for the CAD file. The settings are set to default values and should be sufficient for most cases.

The import process will start and can take a while depending on the size of the model. The import process will create a new folder in the selected location with the name of the CAD file. This folder will contain all the assets that are used in the model. The folder as seen in Figure 35 will contain a “Datasmith Scene” which is a file that contains the information about the constellation of the model and the different parts that were just imported.

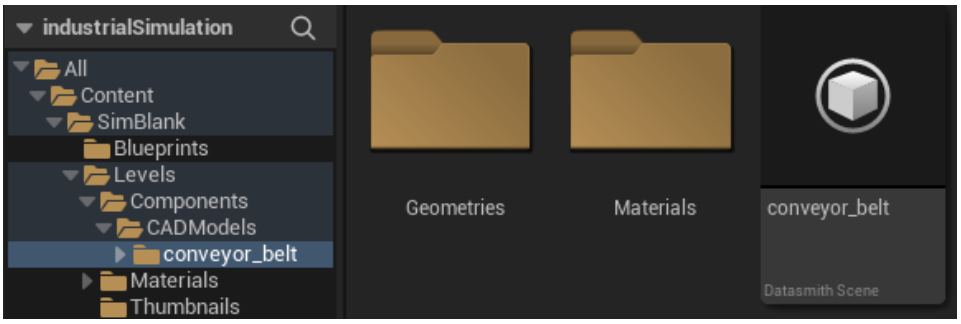

Figure 35: Imported CAD model in the content drawer. The folder contains all the assets that are used in the model and is represented by the Datasmith Scene file.

The Datasmith Scene file can be placed in the world by simply dragging and dropping it into the viewport. This will spawn the model in the world as seen in Figure 36.

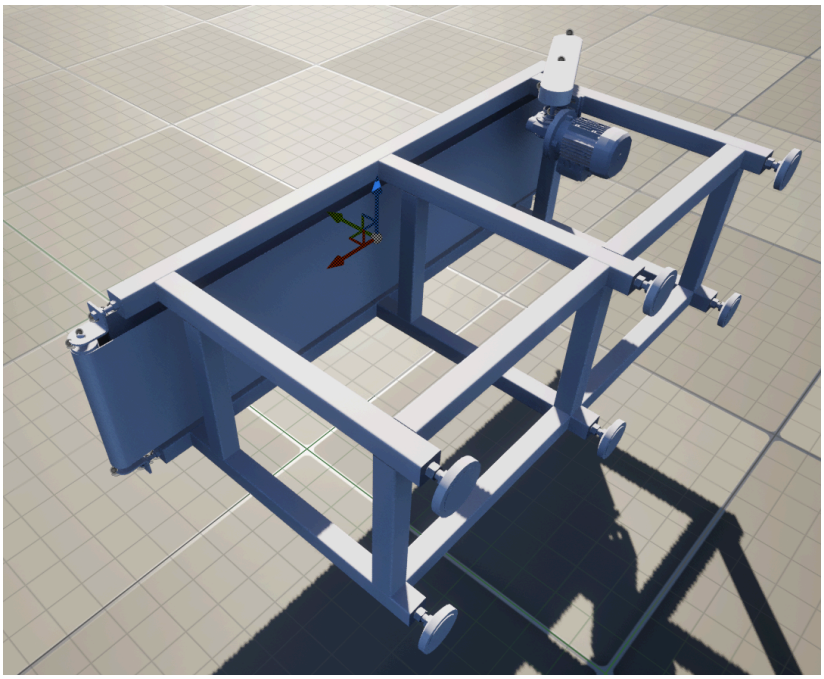

Figure 36: CAD model in the scene. The model here is by default on its side but can be rotated and moved like any other object in Unreal Engine.

The Datasmith Scene file has to be converted in to a Blueprint Actor in order to apply logic to the model. This can be done by selecting all the child components of the Datasmith Scene object in the Outlier tab on the right side and clicking on “Convert Selection to Blueprint Class” as seen in Figure 37. A dialog will open where one can select the name and location of the Blueprint class and the creation method. Select “Harvest Components” in order to create a new Blueprint class that contains the components but does not copy and logic. Note that depending on the CAD model hierarchy the conversion process will automatically create different scene root components for the model. This is not a problem but can be confusing. One can simply select the different meshes and move them to the same root component and delete the other root components to clean up the hierarchy. This is not necessary but makes it easier to work with the model.

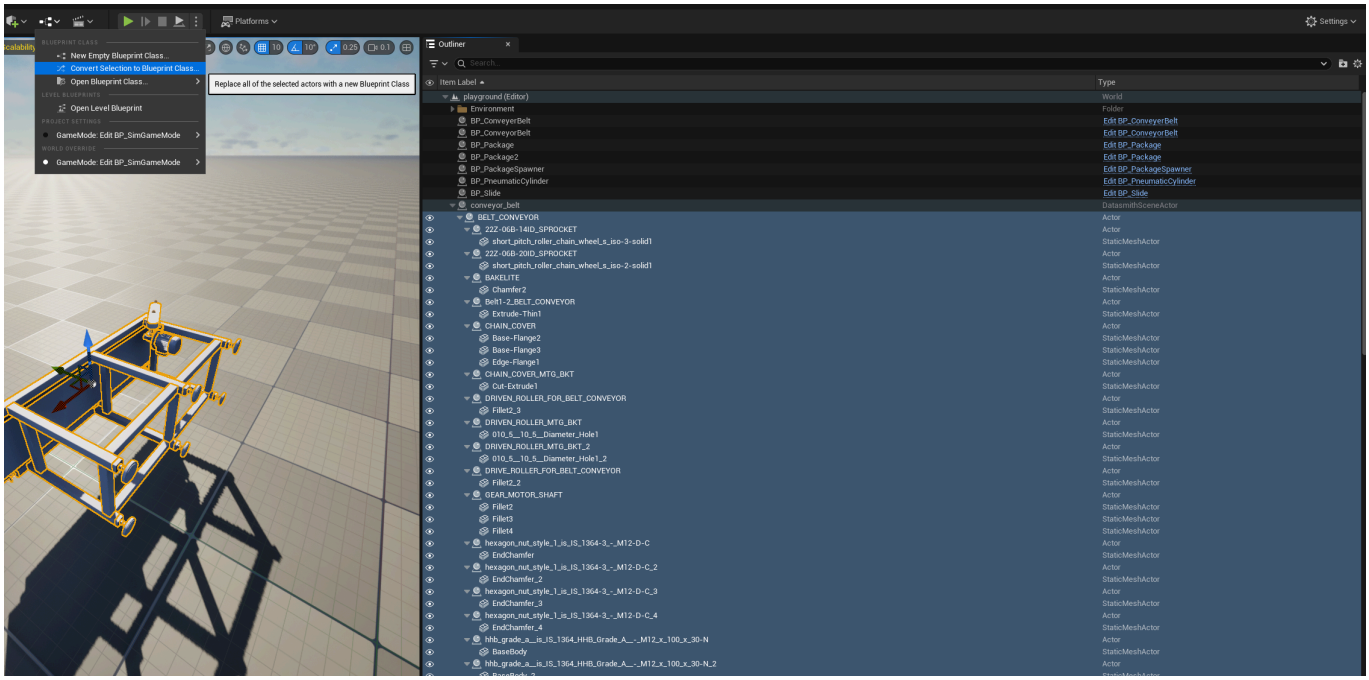

Figure 37: Convert Datasmith Scene to Blueprint Class by selecting all the child components and clicking on "Convert Selection to Blueprint Class".

With the resulting Blueprint class one can now add logic to the model as seen in Figure 38 and use it in the project as any other Blueprint class like the example models we created for demonstration purposes. The Blueprint class can be placed in the level by simply dragging and dropping it into the viewport.

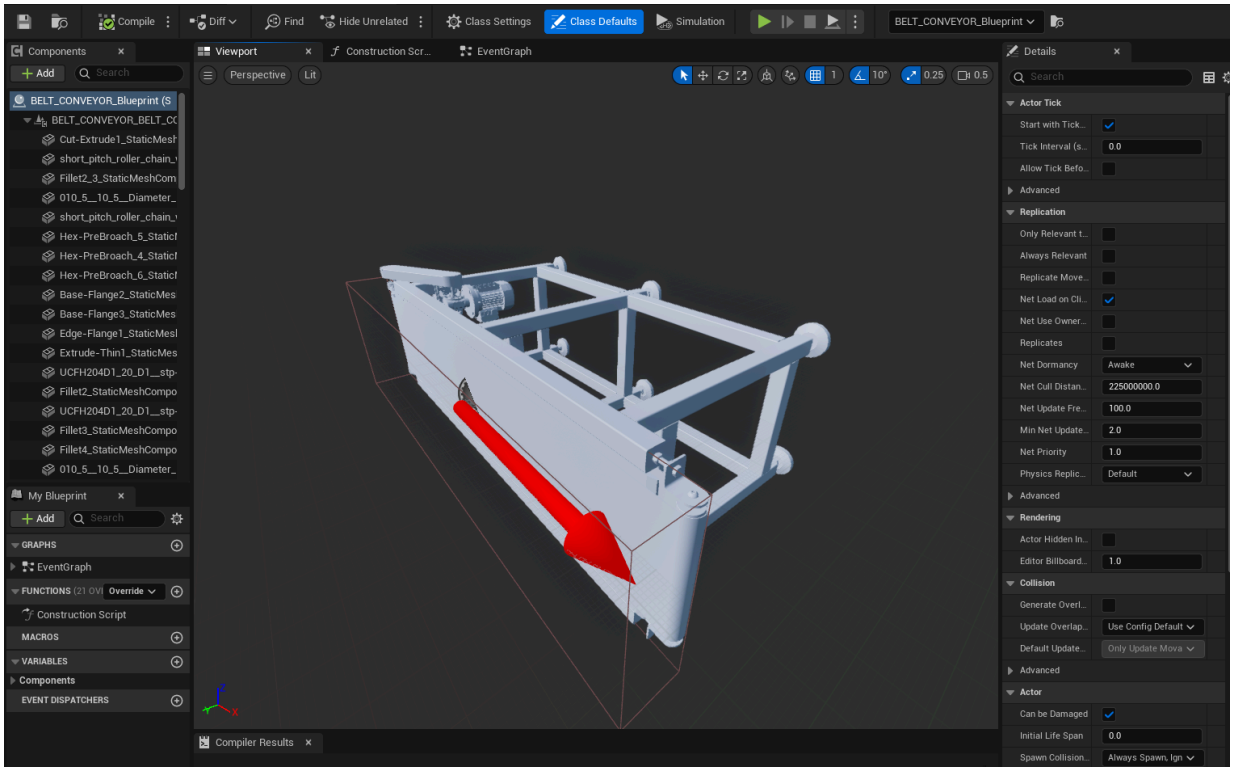

Figure 38: Converted CAD model to Blueprint Class. The class contains all the components of the model and logic can be added to it.

For demonstration purposes, we added a simple collision box to the model and a direction arrow to show the direction of the conveyor belt. The collision box is used to detect when a package is placed on the conveyor belt and the direction arrow is used to calculate the vector that is applied to the package. The logic is very simple and can be seen in Figure 39.

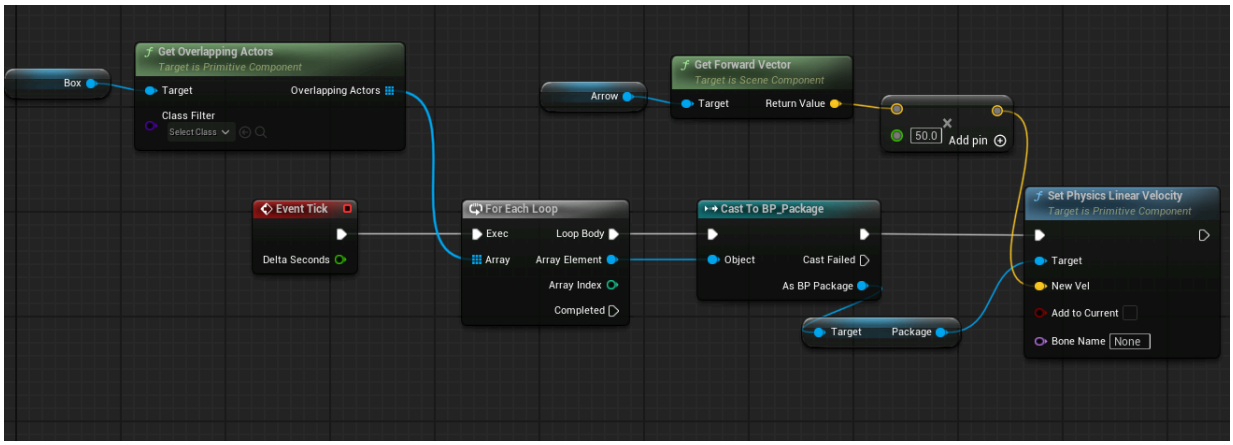

Figure 39: Logic of the converted CAD model. It detects when a package is inside the collision box and applies a force to the package in the direction of the conveyor belt.

In some cases there will be no collision mesh on the converted model. This will have the result that objects can pass through the mesh instead of being blocked like a physical barrier. If this is the case, select and edit the mesh in question. In the mesh editor, select “Auto Convex Collision” from the “Collision” dropdown menu as seen in Figure 40. A new panel will appear on the right hand side where the settings for the collision mesh can be defined and with the button “Apply” a collision mesh will be generated. Save the changes and objects should now be blocked by the new collision mesh.

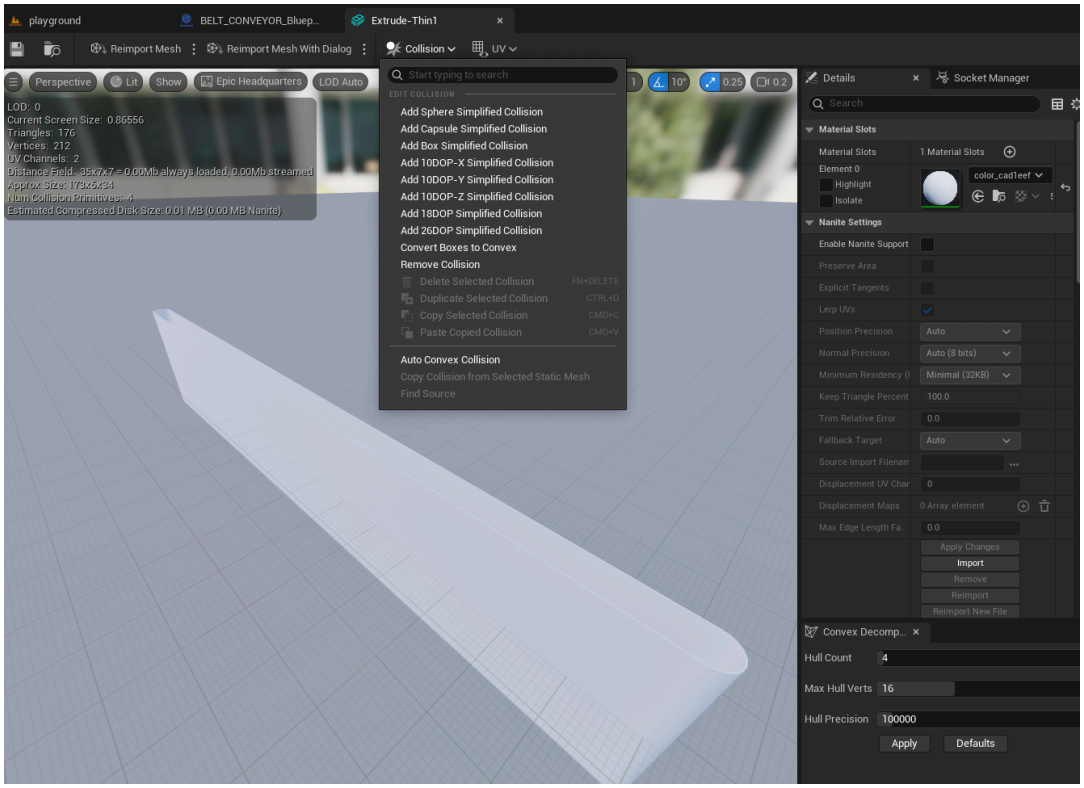

Figure 40: Mesh from the converted CAD model where there is no collision mesh. Menu open that generates a collision mesh automatically.

10 Models

In this chapter, we are going to describe the implemented models in the project. It will demonstrate how to use them and explain how they are implemented. The models are implemented in Unreal Engine 5.5.4 and are located in the folder `unreal/industrialSimulation/Content/SimBlank/Blueprints`.

In the Unreal Editor the models can be found in the content drawer as seen in Figure 41. These models can be dragged into the world to be used in the simulation.

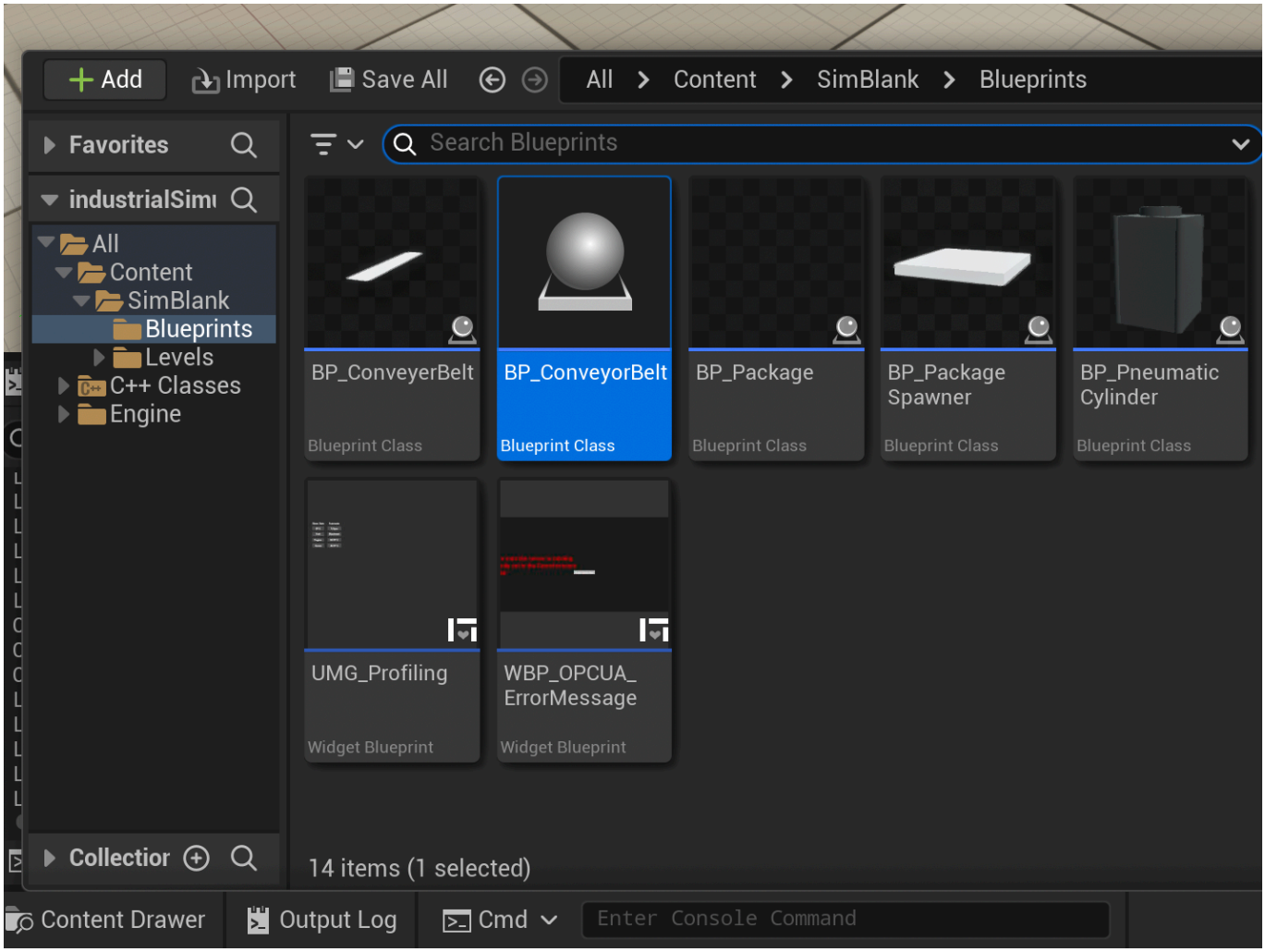

Figure 41: All the Blueprint models accessible in the content drawer.

10.1 Pneumatic cylinder

The pneumatic cylinder has a piston inside a casing that can be extended and retracted. To add the model to the world drag the respective model from the content drawer into the active world as seen in Figure 42.

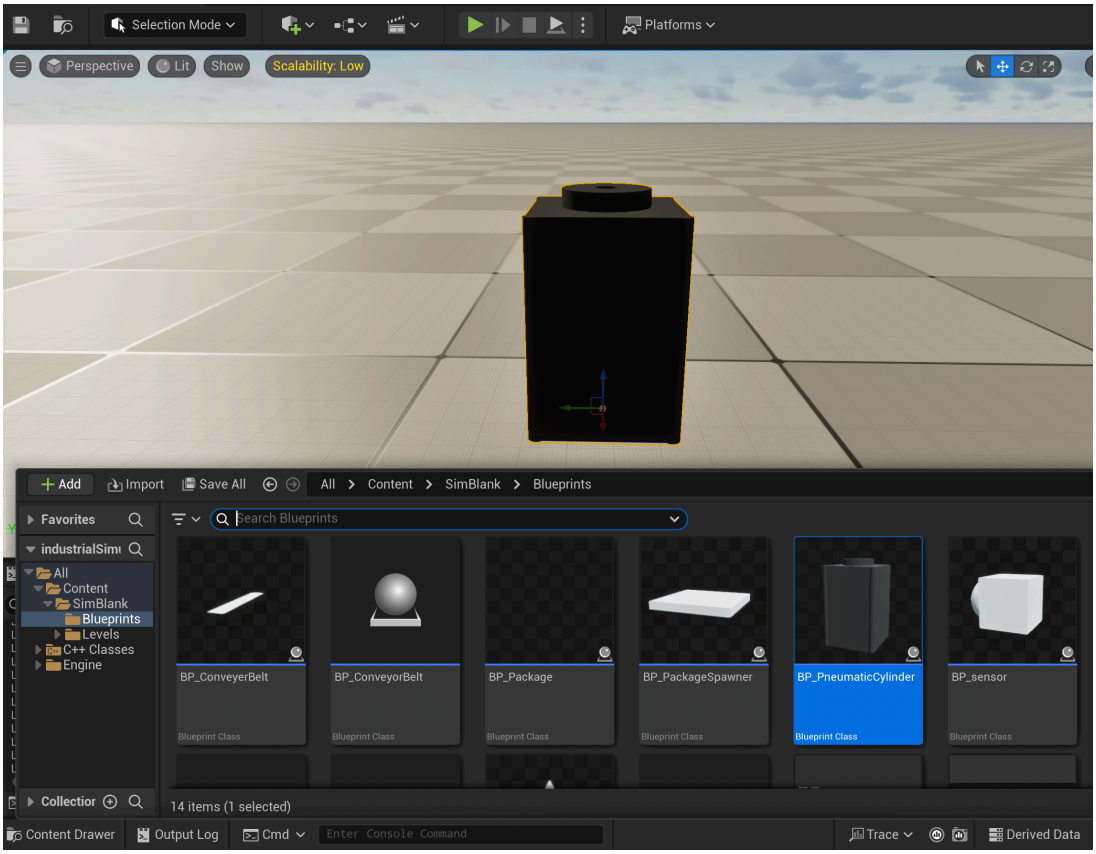

Figure 42: Placing the Blueprint model of a pneumatic cylinder in the world via drag and drop from the content drawer

The piston is by default in the upright position, but can be rotated and moved to the desired position. For this either press “E” or click on the two rotating arrows in the top left corner of the viewport to enter the rotation mode. Alternatively, the rotation can be set in the details panel on the right side of the screen as seen in Figure 43

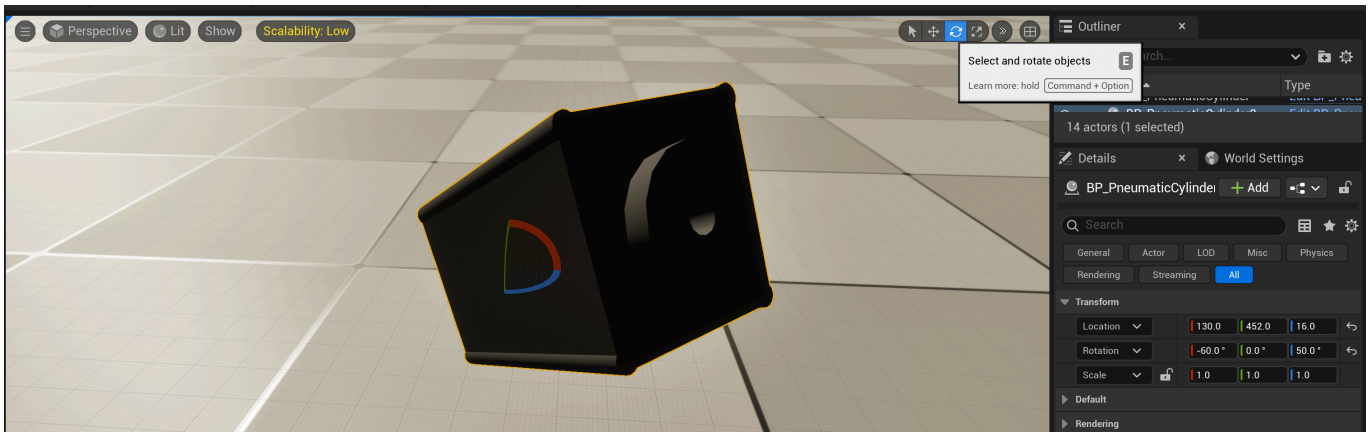

Figure 43: Rotating the pneumatic cylinder in the rotating mode, which can be selected in the top right corner

The controls of the pneumatic cylinder requires three different OPC UA nodes. A boolean node indicating if the pneumatic cylinder is extended or not, where true indicates that it should be extended and false indicates that it should be retracted. Next it requires two boolean nodes indicating if the pneumatic cylinder is extended or retracted. These boolean nodes are used to communicate to the OPC UA server the actual state of the pneumatic cylinder, while the first node is used to control the pneumatic cylinder. In order to set these node IDs, select the pneumatic cylinder and inside the details panel scroll down to the section "OPC UA" as seen in Figure 44. The `Is Retracted Node Id` is the node ID of the boolean node indicating if the pneumatic cylinder is in the retracted position or not. `Is Extended Node Id` is the node ID of the boolean node indicating if the pneumatic cylinder is in the extended position or not. `Should Extend Node Id` is the node ID of the boolean node indicating if the pneumatic cylinder should be extended or not. Next to the OPC UA settings there is also a section called "Settings" where the speed of the pneumatic cylinder can be set. The speed is defined as cm/centiseconds. We use centiseconds so that the animation is smoother and more precise but still not too computationally demanding. Setting the speed to 0.1 means the piston will move at a rate of ten centimeters per second. Lastly the `Extract Distance` is the distance the piston will extend. This value can be adjusted but it should be noted that the piston itself has a certain length, if the value is set to a value larger than the length of the piston it will visually break out of the casing. The `Extract Distance` is set in centimeters and the default value is 50 centimeters.

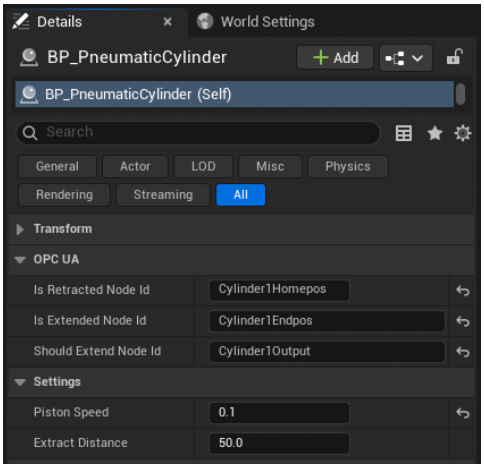

Figure 44: All the possible settings for the pneumatic cylinder. These are the variables that are used behind the scenes in the Blueprint functions.

Whenever a pneumatic cylinder is dragged into the world, these variables have to be set in order for it to work. The specific OPC UA node ID has to exist in the OPC UA server. If missing or if the OPC UA node ID cannot be found, an error message will appear on screen and in the log as seen in Figure 45. The error code can manually be checked in the status code documentation.

Figure 45: This Error message that, we created, is displayed on the screen when the node IDs are not set or not found on the OPC UA server or if the type of the node is not correct.

To view the internal logic of the pneumatic cylinder, double click on the model in the content drawer. This will open the Blueprint editor as seen in Figure 46. The logic is divided into three different sections. The first section is the “Setup” section where the model is set up. It calculates the extended and retracted position with the use of vector math. Additionally, it also sets up the subscription to the OPC UA server and defines the callback whenever the value changes. The next section is the actual callback for when the value for the “Should Extend Node Id” changes. When it should extend it sets the target position to the extended position and turns off the retracted state. When it should retract, it sets the target position to the retracted position and turns off the extended state. Lastly a timer is setup with a static interval that moves the actual piston. The logic for that is located in the third section where the direction, current position and target position are considered to figure out in which direction for how many units the piston should be moved. After moving the piston in the right direction it checks if it already reached the target position. If true it turns off the timer and waits until it is turned on again. This logic allows us to move the piston in a smooth way, and stay performant because we do not need to check every frame if the piston should be moved. Furthermore, it also allows us to change direction even during the movement. Due to the speed variable, we can control how fast the piston should move.

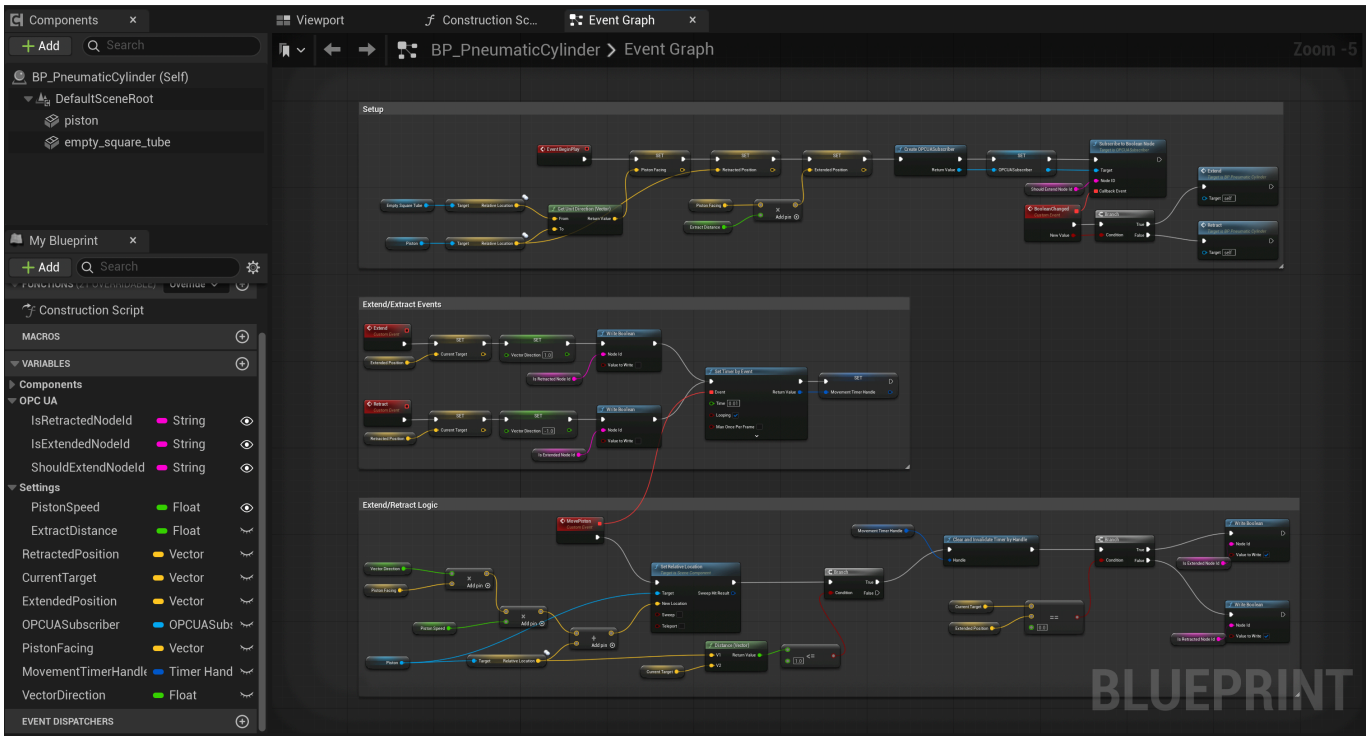

Figure 46: The whole Blueprint logic of the pneumatic cylinder, separated in setup where the subscription and other internal variables are set, the callbacks for when the value changes and the actual movement of the piston.

10.2 Package

The package is a simple box that consists of a width, height, depth and weight. These variables can be set when dragging the package to the world in the details panel as seen in Figure 47.

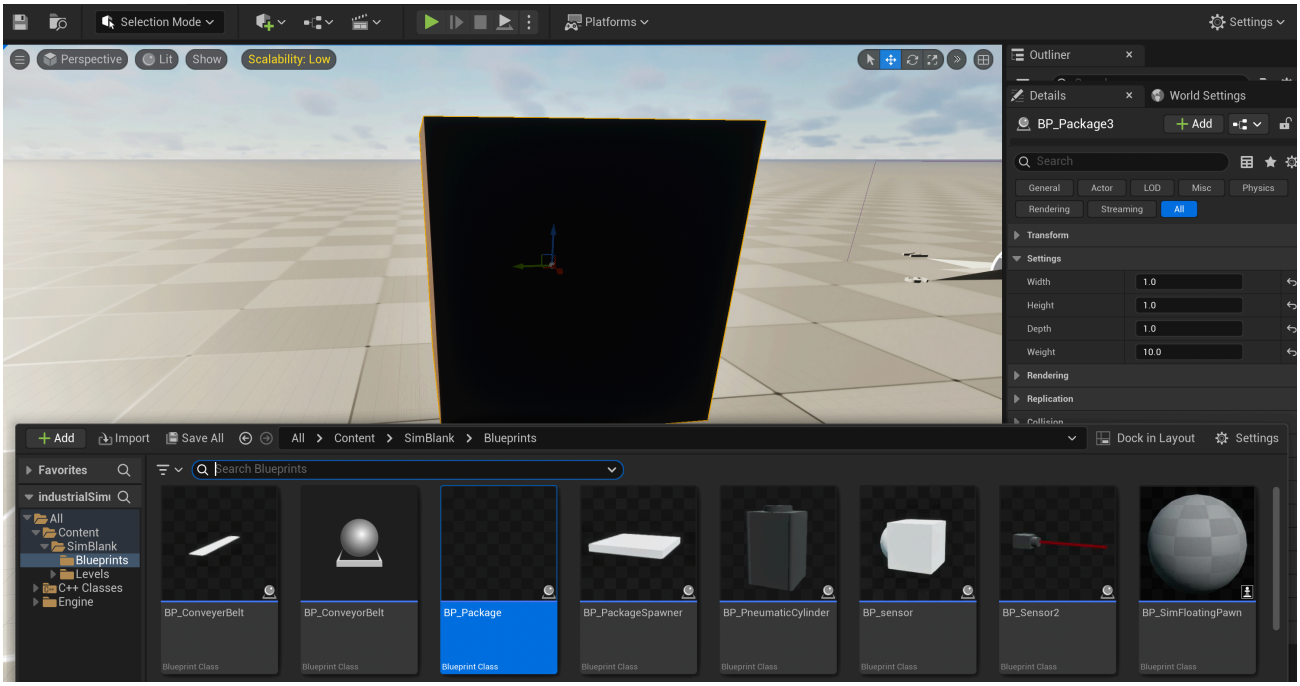

Figure 47: Placing the Blueprint package in the world via drag and drop from the content drawer.

The package has a construction script where the width, height, depth and mass are set as seen in Figure 48. The dimensions are in meters and the mass in kilogram.

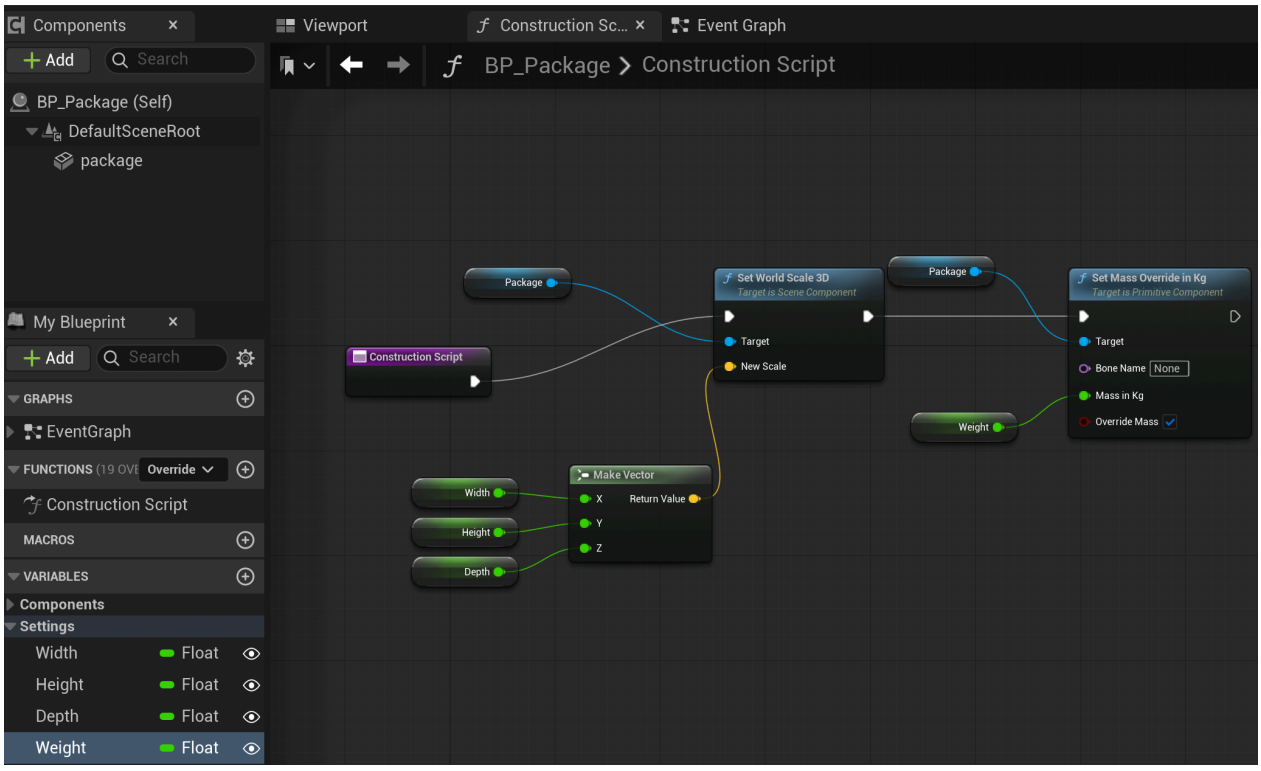

Figure 48: Construction script of the package where the width, height, depth and mass are set to the values of the variables.

The package is a physical object that has physics enabled. This means that the object will only be moved with physical forces and allows properties like friction and gravity to have an impact on the object. Additionally, to define other physical properties a physical material is used as seen in Figure 49.

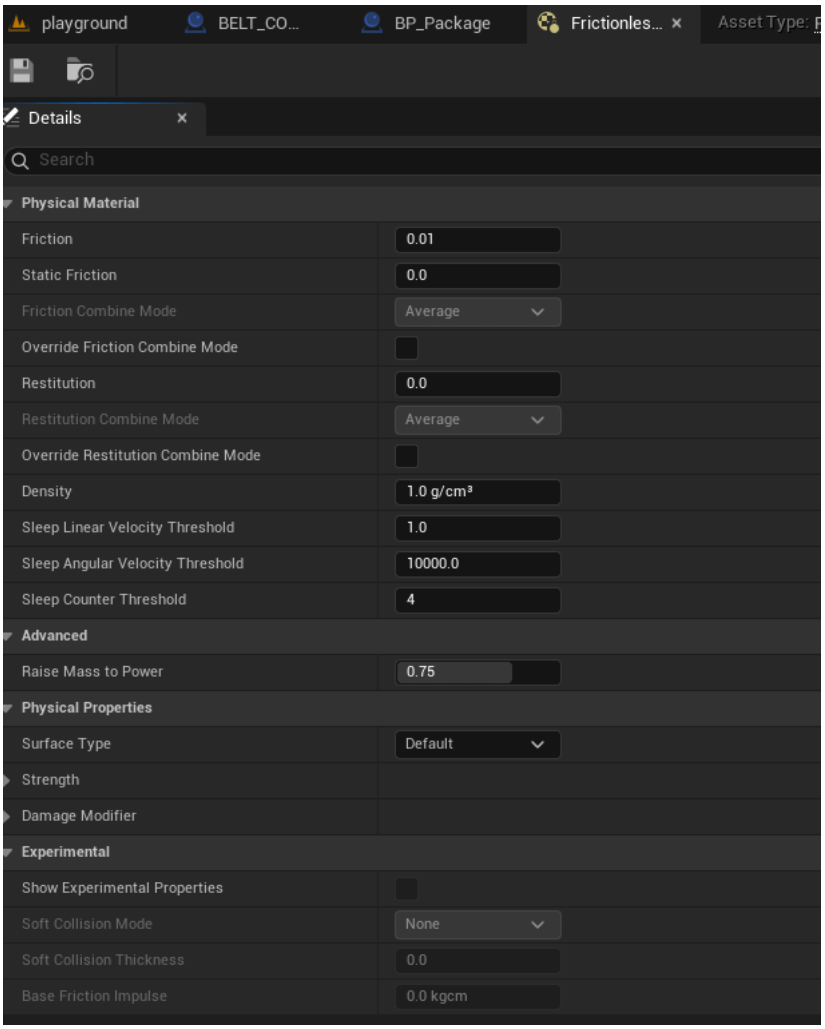

Figure 49: Physical material properties of the package. It has a low friction in order to move smoothly on the conveyor belt.

The same material is used for the conveyor belt surface as seen in Section 10.5. This combination creates a low-friction environment where objects move smoothly without being excessively slowed down. The friction is not set to zero to avoid an unrealistic “ice-like” effect, where objects would continue sliding indefinitely. After experimenting with various values, we found that the chosen settings allow objects to move at the intended speed while maintaining realistic behavior.

10.3 Package spawner

In order to allow dynamically spawned packages during run-time, a package spawner is implemented. This model is a simple box that can be placed in the world as seen in Figure 50.

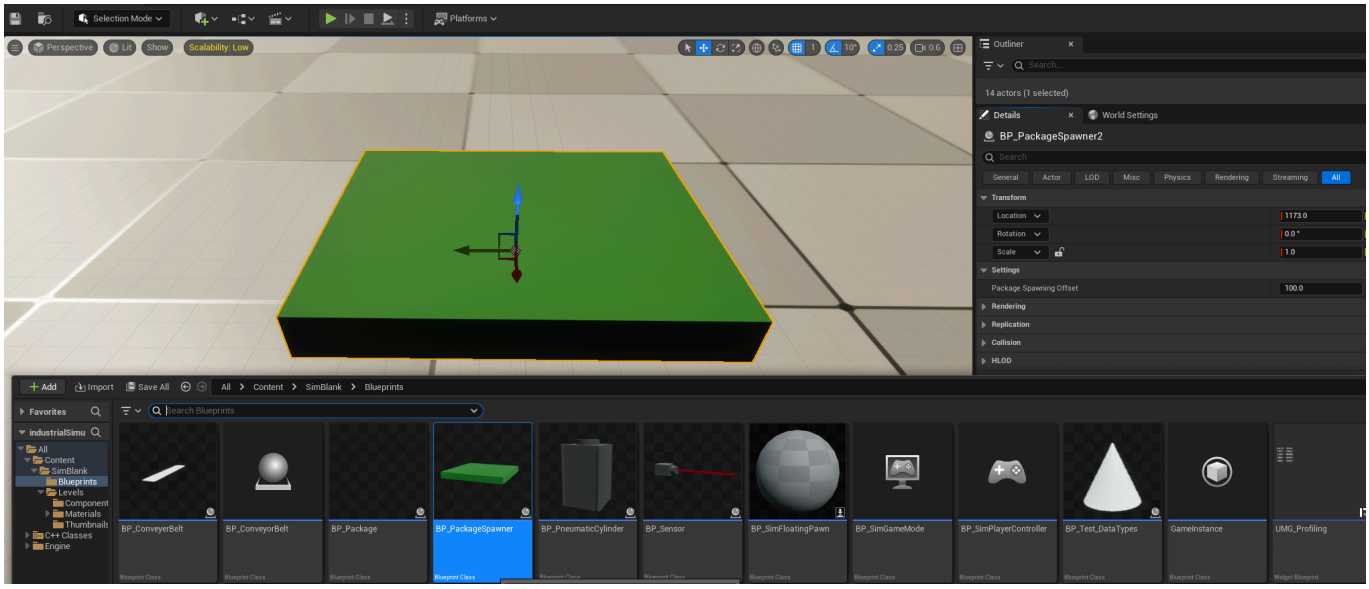

Figure 50: Placing the Blueprint package spawner in the world via drag and drop from the content drawer.

The package spawner has a setting in the details panel where the Z offset can be set as seen in Figure 51. The Z offset is the vertical offset of the package spawner where the package will be spawned. The Z offset is in centimeters and the default value is 100 meaning that the package will be spawned one meter above the package spawner.

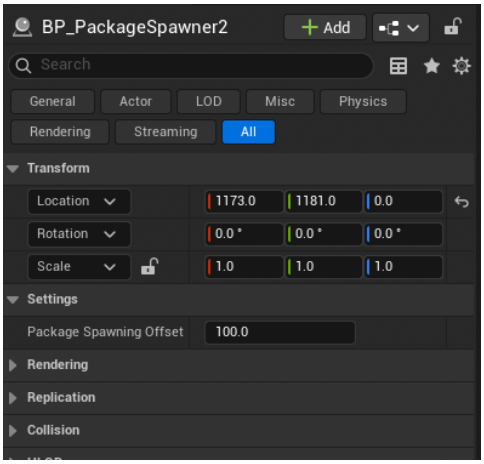

Figure 51: Settings for the package spawner where the Z offset of where the packages will be spawned can be set.

The package spawner listens to the OPC UA server for a counter node of type `Int32`. Whenever the counter node is increased, a new package will be spawned at the Z offset. The spawner reads the width, height, depth and mass of the package from the OPC UA server and sets the values of the package accordingly. The package spawner creates the package in the world and saves a reference to the package together with the package ID, which in this case is the value of the counter node. The reference and the package ID is saved in a map, where the key is the package ID and the value is the reference to the package. This allows us to easily manage the packages and enables the simulation to delete them later on. In addition to the counter node, the package spawner also listens to a separate `Int32` node that indicates which package should be deleted. Whenever the value of the delete node is changed, the package spawner will check if the package ID exists in the map. If it does, it will remove the package from the

world and delete the reference in the local map. This constellation allows us to spawn packages and delete them dynamically during the simulation without being limited to the order of when the packages were created.

The logic can be viewed in detail in the Blueprint editor as seen in Figure 52.

We did not expose the OPC UA node IDs like in the other object because there should only be one instance of the package spawner in our use case.

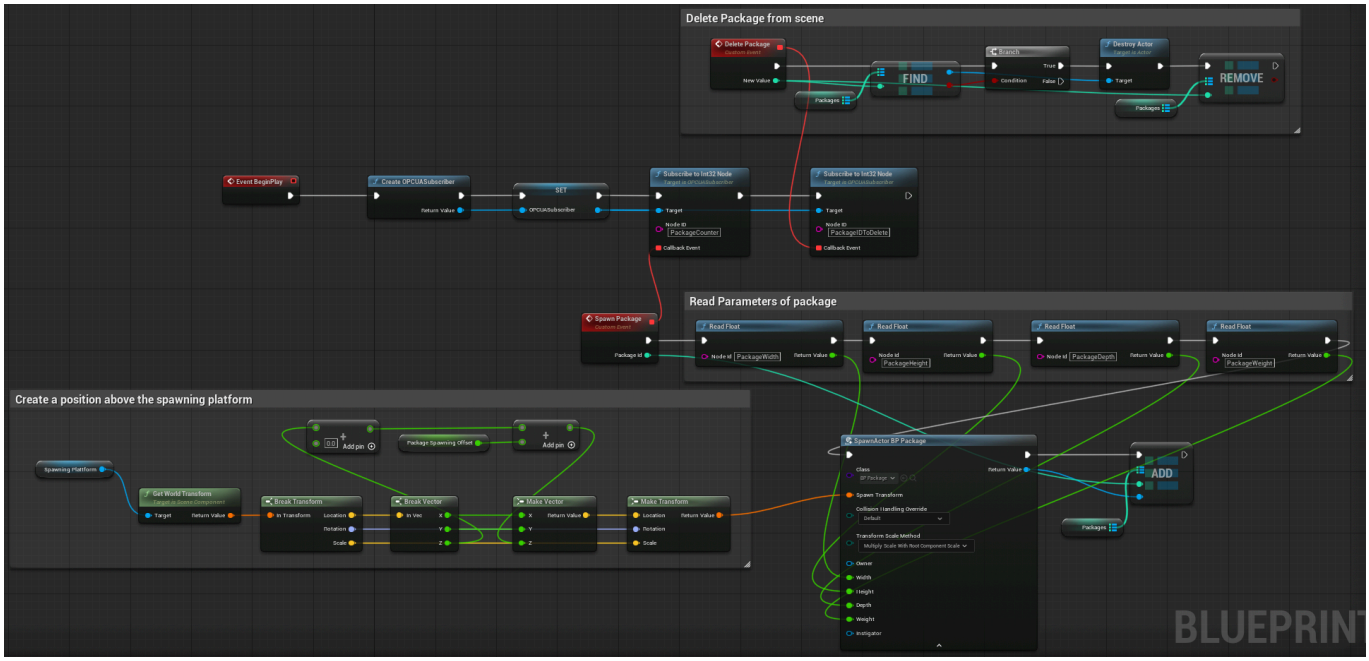

Figure 52: Blueprint logic of the package spawner that spawns and deletes packages dynamically during runtime.

10.4 Sensor

The sensor is a box with a collision box that can be placed in the world as seen in Figure 53. In the “OPC UA” settings of the details panel, the OPC UA node ID for the boolean node indicating whether the sensor is triggered is specified. The sensor will trigger whenever an object with collision turned on enters the beam in front of the sensor. When the object leaves the beam the sensor will be triggered again turning off the signal on the OPC UA server as seen in Figure 54.

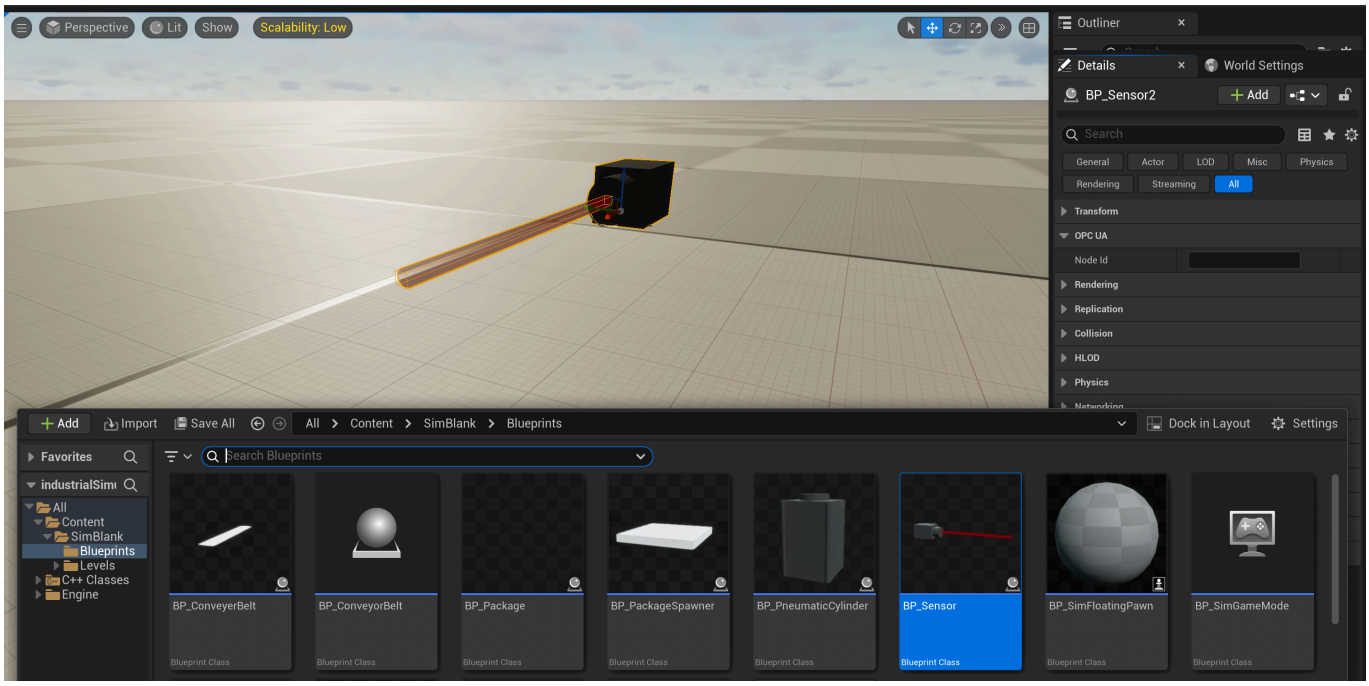

Figure 53: Placing the Blueprint sensor in the world via drag and drop from the content drawer.

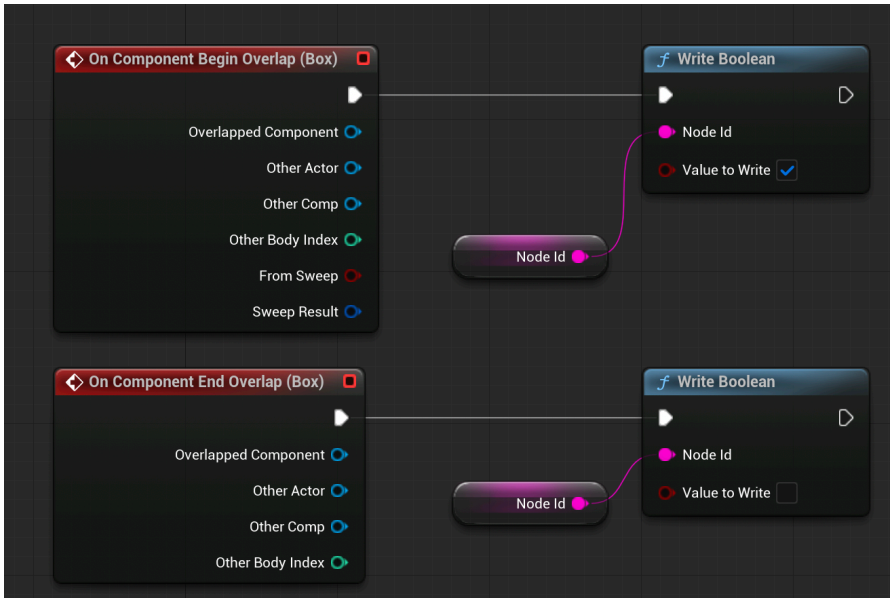

Figure 54: Blueprint logic of the sensor that writes true to a boolean node in the OPC UA server when an object enters the beam and false when it leaves the beam. The node ID can be set in the details panel of the sensor.

10.5 Conveyor belt

The conveyor belt is a simple box that moves packages along a path. To make it look more like a conveyor belt we added a texture that moves with the exact speed of the packages. The conveyor belt can be placed in the world as seen in Figure 55.

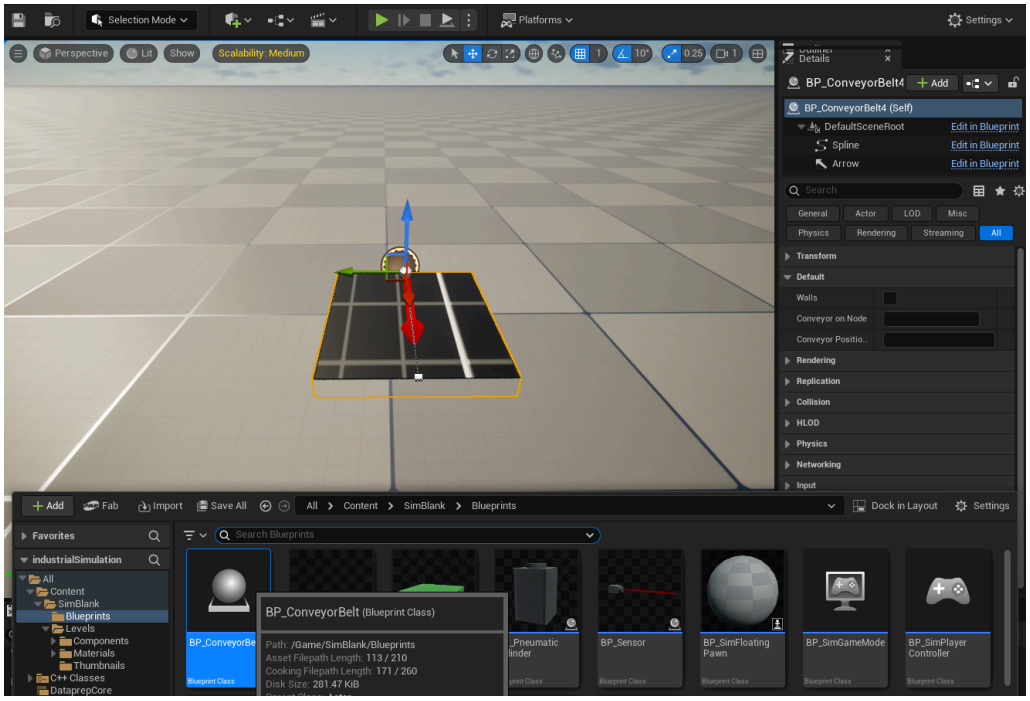

Figure 55: Conveyor belt model being dragged into the world from the content drawer. The settings for the conveyor belt can be set in the details panel on the right side of the screen.

The conveyor belt can be formed by using splines. A spline in Unreal Engine is a curved path defined by control points, which can be used to create smooth and flexible shapes or movement paths for objects just like conveyor belts. The spline points can be selected and adjusted as seen in Figure 56. The spline points can be moved around to create the desired shape of the conveyor belt. To add a new spline point, you can right click on the spline and select “Duplicate Spline Point” or press and hold “ALT” and drag out from an existing spline point from either one of the axis arrows. The conveyor belt will automatically adjust to the new spline points and create linear segments between them.

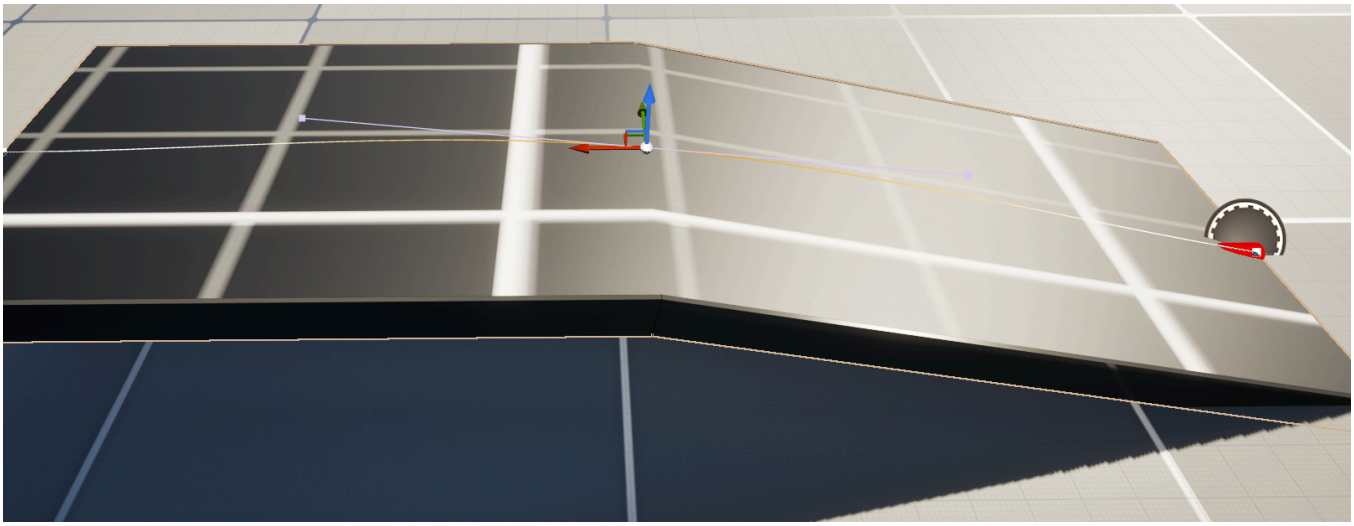

Figure 56: Conveyor belt splines being raised forming a linear path for packages to move along.

The conveyor belt includes an option to toggle walls on or off. The walls are used to prevent packages from falling off the conveyor belt. If a mix between walls and no walls is required, drag multiple conveyor belts into the world and set the walls on the desired conveyor belts as seen in Figure 57.

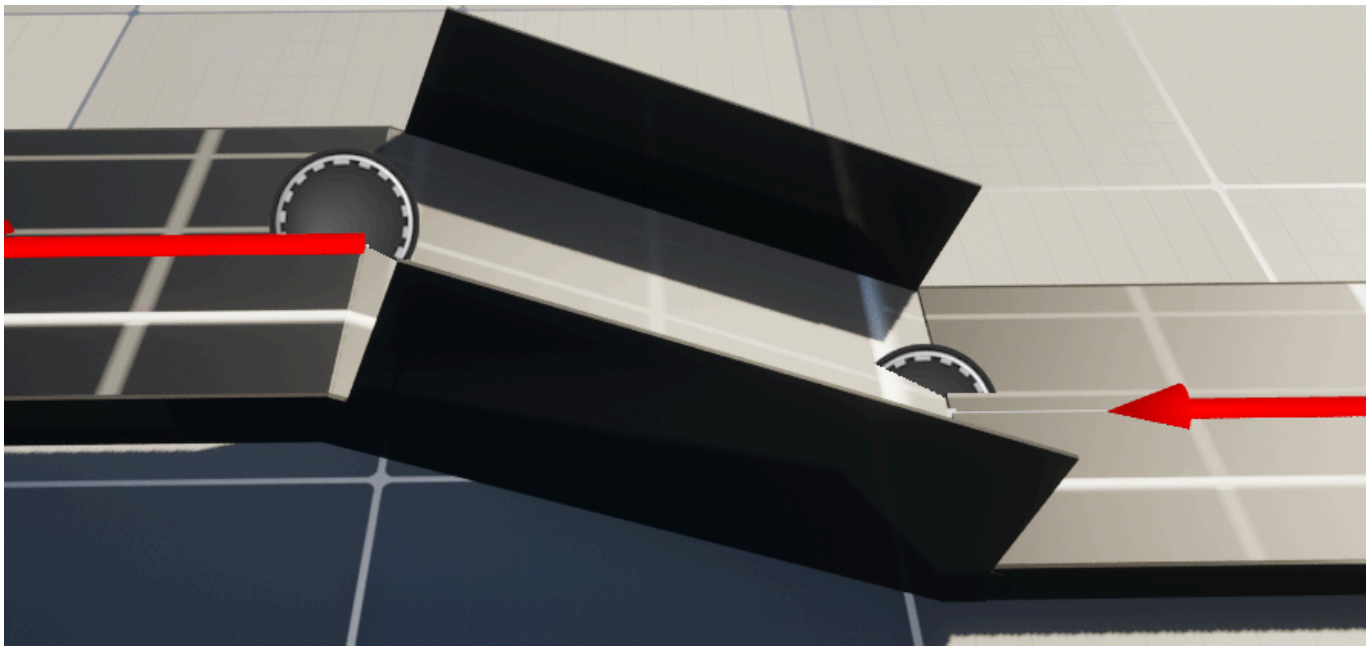

Figure 57: Multiple conveyor belts with different wall settings forming one long conveyor belt. The first and last conveyor belt have no walls, while the middle conveyor belt has walls.

Additionally, the conveyor belt requires an OPC UA node of type `Boolean` that indicates if the conveyor belt is turned on or not. To control the speed of the conveyor belt, an OPC UA node of type `Float` is required. This float indicates the target position of the conveyor belt in centimeters. The conveyor belt will move towards this target position in a smooth way. These settings can be set in the details panel as seen in Figure 58.

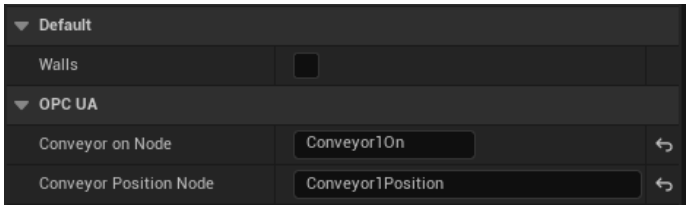

Figure 58: Settings of the conveyor belt. Walls toggles between having walls or not, the `Conveyor on Node` is the OPC UA node ID of the boolean node indicating if the conveyor belt is turned on or not and `Conveyor Position Node` is the OPC UA node ID of the float node indicating the target position of the conveyor belt in centimeters.

Whenever a conveyor belt is dragged into the world, these variables have to be set in order for it to work. The specific OPC UA node ID has to exist in the OPC UA server. If missing or if the OPC UA node IDs cannot be found, an error message will appear on screen and in the log as seen in Figure 45.

To view the internal logic of the conveyor belt, double click on the model in the content drawer. This will open the Blueprint editor as see in Figure 59.

In the construction script the logic is set up to calculate the spline points and the direction of the conveyor belt. It adds the meshes of the conveyor belt and the walls if they are enabled.

10.5.1 Conveyor belt movement

In the event graph, the logic is divided into multiple different sections. The first section is the "Setup" section where the model is set up. It sets up the subscriptions to the OPC UA server for the specified OPC UA nodes and defines the callback whenever the value changes. Furthermore, it handles the change of the target position. The target position refers to on what position around the conveyor belt the starting point would be. Whenever the target position changes, the system compares the new target position and timestamp with the previous position and the timestamp of the last change to calculate the conveyor belt's velocity. The velocity is calculated as the difference between the new target position and the previous position divided by the difference between the new timestamp and the previous timestamp. This allows us to calculate the velocity of the conveyor belt in a smooth way, without having to check every frame if the target position has changed. When the conveyor belt is running, it will try to move all the packages on top of it every frame.

The second section is the "Set speed of conveyor material" section. This has nothing to do with the movement of the packages and just exists to move the material (texture), with the exact same speed as the packages.

The third section is the "Run every tick" section. This section gets executed every tick and checks if the conveyor belt is on. If it is, the next section "Move items on conveyor belt" get started.

The section "Move items on conveyor belt" is the fourth section. This contains the whole logic to move the packages along the conveyor belt. We had to ensure, that the packages always follow the conveyor belt, so we could not just set an initial direction when the packages first touch the conveyor belt. We made the packages follow the conveyor belt's spline, which runs along its center. To achieve this, we calculate the vector from each package to the nearest point ahead of it on the spline. Here the movement velocity, calculated in the setup section, gets applied on the packages. The movement velocity is set every frame in order to minimize the velocity loss that occurs due to friction and other forces acting on the packages.

The final section, "Stop packages on conveyor belt," handles the event triggered when the conveyor belt is turned off. When the conveyor belt is turned off, it set the velocity of all packages on top of the conveyor belt to zero, so that they do not move anymore.

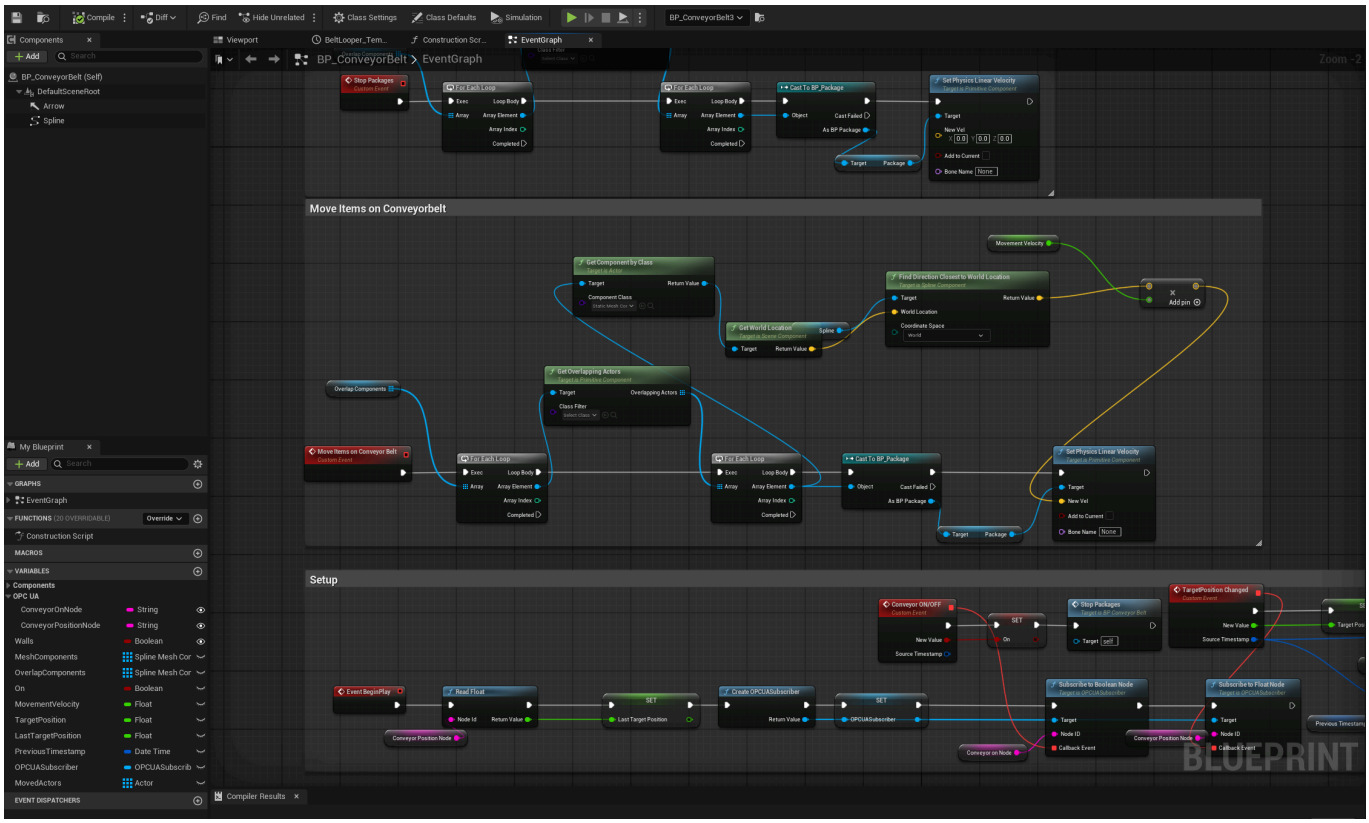

Figure 59: The whole Blueprint logic of the conveyor belt, separated in setup where the subscriptions to the OPC UA server are set up, the callbacks for when the value changes and the actual movement of the conveyor belt.

10.6 Slide

The slide is a simple box with walls around it and a slope where the packages can slide down. The slide additionally has a sensor in front of it that detects when a package is about to slide down. Above the slide there is a floating text that displays the counter of the packages that passed the sensor. To add the model to the world drag the respective model from the content drawer into the active world as seen in Figure 60.

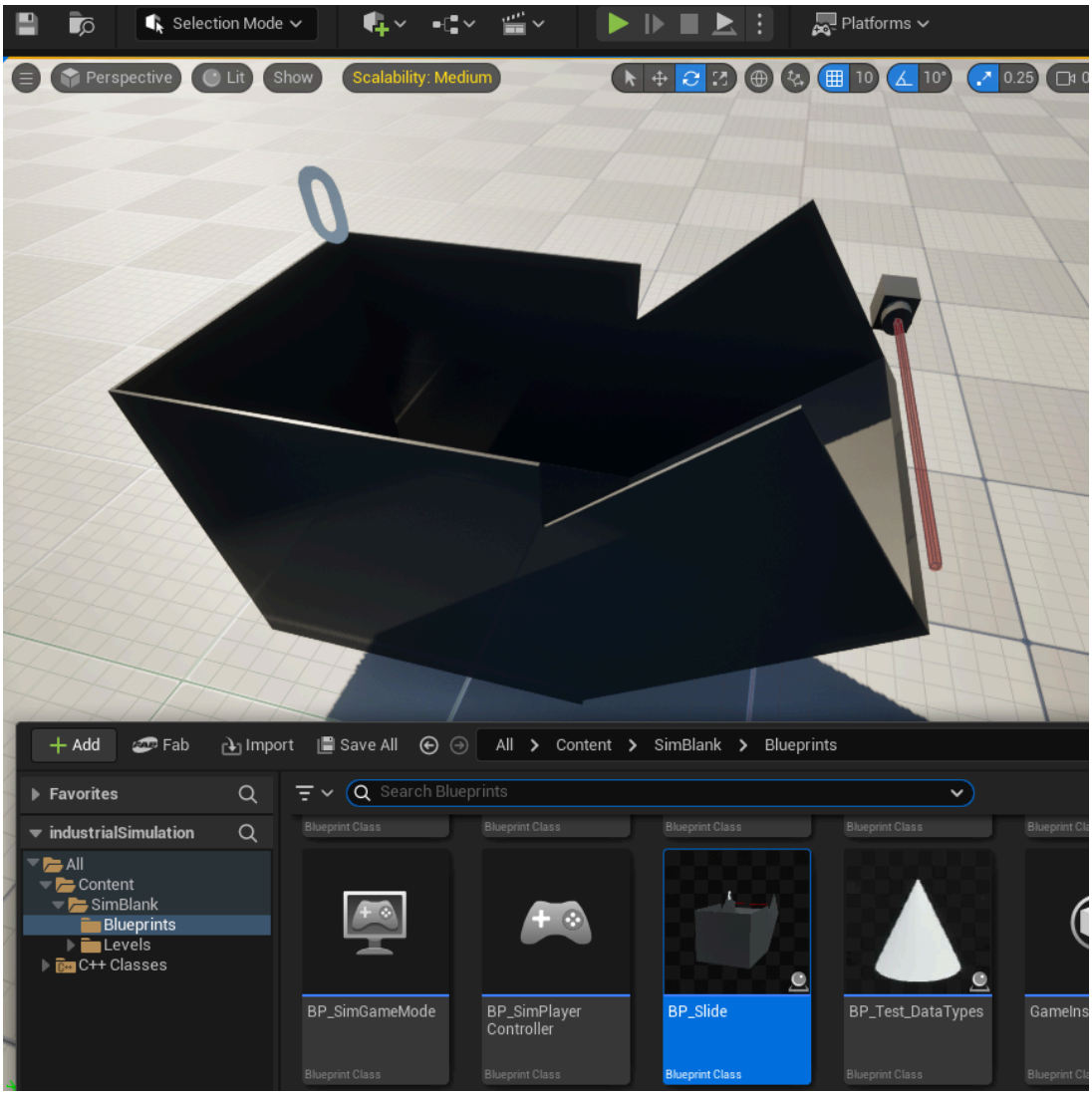

Figure 60: Placing the Blueprint slide in the world via drag and drop from the content drawer.

Due to the simple nature of the slide there is no need to set any variables or settings in the details panel. In order for the packages to slide down smoothly and avoid being stuck on the slope, the floor has a physical material where friction is set to zero. This allows the packages to slide down and not being slowed down by friction. This is true for both the slope and the floor at the end of the slides where the packages can be collected.

The logic for the slide is very minimal as well. There is only a collision detection at the sensor in front of the slide. Whenever something enters the sensor it counts up an internal counter. The value of the counter is displayed in the floating text above the slide. The counter is reset whenever the simulation is restarted. The logic can be viewed in the Blueprint editor as illustrated in Figure 61.

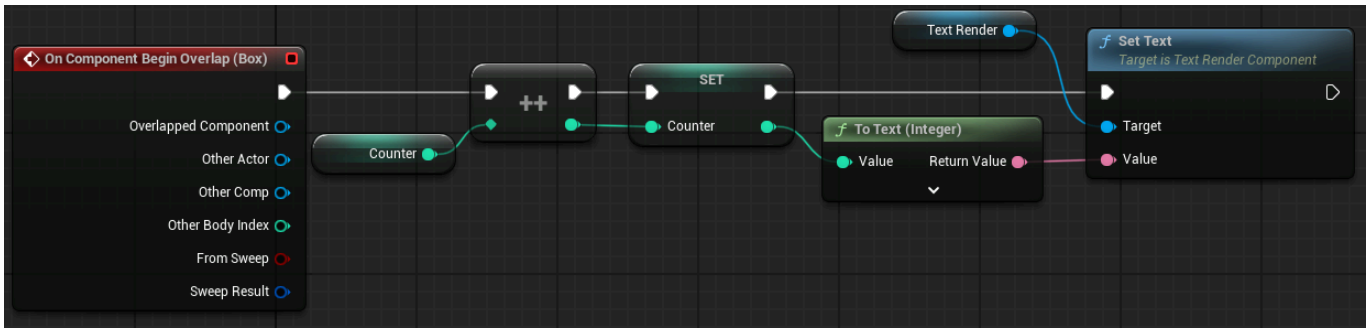

Figure 61: Blueprint logic of the slide that counts up the packages that passed the sensor and displays the counter in a floating text above the slide.

Part III
Summary

11 Summary

This chapter summarizes the project and gives an overview of the results. It also provides a brief outlook on future work and alternative approaches.

11.1 Results

The project successfully demonstrated that game engines like Unreal Engine can be used to develop a digital twin. We successfully integrated an OPC UA client into Unreal Engine with open62541 and utilized it to communicate with an OPC UA server. To simplify the development process, we created reusable building blocks by abstracting the communication to the server into simple Blueprint nodes that enable a developer, with minimal knowledge of the underlying programming language C++, to create complex logic to interact with the server. We enabled the developers to read data from the server, write data to the server and subscribe to data changes enabling them to build real time systems. We demonstrated the use of these building blocks with various different models. We utilized the powerful physical simulation capabilities of Unreal Engine 5 to simulate the behavior of physical objects. We explained the process of transforming a CAD model into a format that can be used in Unreal Engine and demonstrated the process with a simple example where we converted a simple CAD model into an Unreal Engine actor and added some basic logic to it. We also showed how to use version control with Git and demonstrated how to work in a team with multiple developers on the same project.

With our project as a base it should be possible to create a digital twin of a real-world system that can be used for simulation, testing, analysis and optimization. The project also serves as a foundation for future work in this area, as it provides a solid base for further development and experimentation with digital twins in Unreal Engine.

11.2 Limitations

Although we could leverage the powerful physical simulation capabilities of Unreal Engine, we still had to make some compromises in the accuracy of the simulation. For instance, the conveyor belt simulation is not as accurate as it could be. Instead of a moving surface with high friction that transports the objects on top of it, we had to move the objects themselves by applying force to them to simulate the movement of the conveyor belt. This was due to the fact that Unreal Engine does not support moving surfaces with high friction out of the box. If more time could have been spent on the project, we could have implemented a custom solution to achieve this.

Additionally, the OPC UA server we used for the project was not a real-world server, but rather a mock server that we created for testing purposes. By using Node-RED, we were able to create a simple OPC UA server that could be used to test the communication with the Unreal Engine client. However, this server is limited in its capabilities and in its throughput. This meant that we could only update data at a certain rate before the server would struggle to keep up with the requests. Due to this limitation, we had to limit the granularity of the data that updates for example the conveyor belt target position. Also it limited the amount of subscriptions per connection that we could have. This setting was not directly configurable in the Node-RED OPC UA server, so we had to work around it by manually adjusting the generated source files of the server. If not adjusted the client would be limited to a maximum of ten subscriptions which would not be sufficient for a more complex system. These bottlenecks do not affect the Unreal Engine client itself and our implementation of the Blueprint functions that interact with the server. If a real-world OPC UA server is used, these limitations would not be present and the client would be able to handle more complex systems with more data and more subscription.

Unreal Engine is a powerful but also very complex tool. It has a steep learning curve and requires a lot of time to get used to it. The project was a good opportunity to learn how to use the engine and explore its capabilities. However, for a realistic digital twin project, it would be highly recommended to have a some prior knowledge of the engine and its features especially the physics engine and the Blueprint system. Then it would be possible to create a more complex and realistic digital twin with more precision.

We acknowledge that the models that we created as a demonstration of the Blueprint functions are not very detailed in the way they look. This is due to the fact that we did not have CAD models available for the objects and had to create them from scratch without any prior knowledge of the modeling tools in Unreal Engine. The models are sufficient to demonstrate the functionality of the Blueprint functions and the communication with the OPC UA server, but they do not represent a realistic digital twin of a real-world system. We explored the workflow of how to convert a CAD model into a format that can be used in Unreal Engine and demonstrated the process with a simple example.

The OPC UA client that we used in Unreal Engine, open62541, is a powerful library that provides a lot of functionality for OPC UA communication. However, we encountered a lot of issues when we tried to compile the library for Unreal Engine. This was because the build for macOS was partially broken and the build for Windows was not compatible with the pregenerated binary for macOS. Due to this we had to include some OS-specific code in the project to handle the differences between the two platforms and require the user to install the library manually on macOS. This is not ideal but until the library is fixed, this is the only way to make it work on both platforms.

11.3 Future work

The project serves as a foundation for future work in the area of digital twins in Unreal Engine. There are many possibilities to extend the project and create more complex and realistic digital twins. Some ideas for future work include:

Support OPC UA events and alarms: Changes in a system can be communicated via events and alarms. These notifications are asynchronous and can be used to trigger actions. The delivery of these events also utilizes the subscription and monitor mechanism of OPC UA like the subscriptions for data changes. New Blueprint functions could be created to handle these events and alarms which would trigger actions in the Unreal Engine client. This would allow for more complex interactions with the server and would also enable to build a digital twin that represents a real-world system with real-time data and events in order to visualize the system and its current state.

More realistic conveyor belt simulation: As mentioned in the shortcoming section, the conveyor belt simulation is not as accurate as it could be. A more realistic simulation could be implemented where the surface of the conveyor belt actually moves and transports the objects on top of it due to high friction. This would require more extensive knowledge of the Unreal Engine physics engine.

Support multiple OPC UA servers simultaneously: The current implementation uses a single connection to a single OPC UA server. If a more complex system that consists of multiple subsystems is to be represented, it would be necessary to support multiple OPC UA servers simultaneously. This would also require an adaptation of the current Blueprint functions to enable them to specify which server the communication should be done with.

Integrate machine vision capabilities: Many industrial systems rely on machine vision technologies. A basic implementation could involve a barcode scanner that reads and processes barcodes on objects. More advanced functionality might include using a camera to interpret handwritten labels, such as address labels, on items.

Support Linux devices: Currently, the project supports only Windows and macOS platforms. To make the project more inclusive and accessible, Linux support should be added. This would involve compiling the open62541 library for Linux and updating the build pipeline to accommodate Linux-specific requirements.

Part IV
Appendix

11.4. Test runs

Here we display all the testing that was done during the project. For privacy reasons we did not state the names of the testers but instead assigned them numbers.

11.4.1. The whole project can be set up according to the README (10.4.2025)

Description In this test the instructions for the set up of the project are tested. The user should be able to set up the project with just the README.

Preconditions The newest version of the repository is cloned.

Tester Test_User_1

Test steps

Step	Description	Result
1	Setup the whole project by only following the instructions in the README of the whole project and the README of the server.	Failed

Table 6: Manual user test on setting up the whole project.

Result The tester was not able to start our project in Unreal Engine with just the help of our README. For the whole setup to work, the .NET developer pack and the .NET sdk is needed. This had to be found out during the debugging process.

Tester feedback The description is well written besides the fact, that the .NET developer pack was missing.

11.4.2. The whole project can be set up according to the README (28.04.2025)

Description In this test the instructions for the set up of the project are tested. The user should be able to set up the project with just the README.

Preconditions The newest version of the repository is cloned.

Tester Test_User_2

Test steps

Step	Description	Result
1	Setup the whole project by only following the instructions in the README of the whole project and the README of the server.	Failed

Table 7: Manual user test on setting up the whole project.

Result The tester successfully set up and ran the entire project in Unreal Engine. However, our README lacked sufficient setup instructions. Specifically, it did not mention the requirement for Visual Studio, which led to a crash. The tester had to consult the crash report to identify this missing dependency.

Tester feedback Everything worked as described. The tester would have liked a link to Visual Studio, so he does not have to look into the crash report.

11.4.3. The whole project can be set up according to the README (03.06.2025)

Description In this test the instructions for the set up of the project are tested. The user should be able to set up the project with just the README.

Preconditions The newest version of the repository is cloned.

Tester Test_User_3

Test steps

Step	Description	Result
1	Setup the whole project by only following the instructions in the README of the whole project and the README of the server.	Successful

Table 8: Manual user test on setting up the whole project.

Result The tester successfully set up and ran the entire project in Unreal Engine.

Tester feedback Everything worked as described. The tester would have appreciated a hint about also installing the C++ plugins in Visual Studio.

11.4.4. The pneumatic cylinder can be extended and retracted. (03.06.2025)

Description In this test the functionality and correctness of the pneumatic cylinder is tested. It can extend and retract based on a switch of the OPC UA server.

Preconditions The project is running in Unreal Engine, the OPC UA server is working and the dashboard can be opened on the OPC UA server.

Tester Test_User_1

Test steps

Step	Description	Result
1	Drag the pneumatic cylinder Blueprint from the content drawer into the world.	Successful
2	Configure the `Should extend Node ID`, the `Is Retracted Node ID` and the `Is Extended Node ID` to match a pneumatic cylinder on the server.	Successful
3	Run the application and tab out to the dashboard window in the browser.	Successful
4	Switch the pneumatic cylinder to `ON` so it gets extended.	Successful
5	Switch the pneumatic cylinder to `OFF` so it gets retracted.	Successful

Table 9: Manual user test for the pneumatic cylinder

Result The pneumatic cylinder in the simulation extends when the switch is on `ON`. As soon as the cylinder is extracting, the `Home Position` in the dashboard gets switched to `OFF` and the `Output` is switched to `ON`. When the cylinder is fully extracted the `End Position` in the dashboard gets switched to `ON` and the `Output` gets turned to `OFF`.

Tester feedback Everything worked as described. The tester was not able to identify the OPC UA node IDs, that were required to be set in the simulation on first sight.

11.4.5. The pneumatic cylinder can be extended and retracted. (03.06.2025)

Description In this test the functionality and correctness of the pneumatic cylinder is tested. It can extend and retract based on a switch of the OPC UA server.

Preconditions The project is running in Unreal Engine, the OPC UA server is working and the dashboard can be opened on the OPC UA server.

Tester Test_User_3

Test steps

Step	Description	Result
1	Drag the pneumatic cylinder Blueprint from the content drawer into the world.	Successful
2	Configure the `Should extend Node ID`, the `Is Retracted Node ID` and the `Is Extended Node ID` to match a pneumatic cylinder on the server.	Successful
3	Run the application and tab out to the dashboard window in the browser.	Successful
4	Switch the pneumatic cylinder to `ON` so it gets extended.	Successful
5	Switch the pneumatic cylinder to `OFF` so it gets retracted.	Successful

Table 10: Manual user test for the pneumatic cylinder

Result The pneumatic cylinder in the simulation extends when the switch is on `ON`. As soon as the cylinder is extracting, the `Home Position` in the dashboard gets switched to `OFF` and the `Output` is switched to `ON`. When the cylinder is fully extracted the `End Position` in the dashboard gets switched to `ON` and the `Output` gets turned to `OFF`.

Tester feedback Everything worked as described. The tester was familiar with Unreal Engine and had no troubles to find the necessary fields.

11.4.6. The conveyor belt can be started and transports package. (03.06.2025)

Description In this test the functionality and correctness of the conveyor belt is tested. It can be started via the Node-RED dashboard and transports packages. The conveyor belt can also be reversed.

Preconditions The project is running in Unreal Engine, the OPC UA server is working and the dashboard can be opened on the OPC UA server.

Test steps

Tester Test_User_1

Step	Description	Result
1	Drag the conveyor belt Blueprint from the content drawer into the world.	Successful
2	Configure the `Conveyor on Node`, `Cycle Time Node`, `Conveyor Position Node` with the nodes of the `Conveyor1` on the Dashboard.	Successful
3	Drag the package spawner Blueprint from the content drawer into the world underneath the start of the conveyor belt so the packages later can fall onto the belt.	Successful
4	Run the application and tab out to the dashboard window in the browser.	Successful
5	Create a package with the help of the package manager view in the dashboard. Use the values `50` for `width`, `height` and `depth`. Use `10` for kilograms.	Successful
6	Set `Conveyor Speed` to `50` and start the conveyor belt switching `Conveyor` to `ON`.	Successful

Table 11: Manual user test for the conveyor belt.

Result The conveyor belt is running. It transports the package in the direction the conveyor belt faces.

Tester feedback Everything worked as intended, and the steps were described precisely.

11.4.7. The packages slide down the slide. (03.06.2025)

Description In this test the ability of the package to slide down the slide is tested. The slide is not controlled via the Node-RED dashboard and is something that works on its own. For package spawn the server is still needed.

Preconditions The project is running in Unreal Engine, the OPC UA server is working and the dashboard can be opened on the OPC UA server.

Test steps

Tester Test_User_1

Step	Description	Result
1	Drag the slide Blueprint from the content drawer into the world.	Successful
2	Drag the package spawner Blueprint form the content drawer into the world in a way the packages can fall right onto the slide and slide down.	Successful
3	Run the application and tab out to the dashboard window in the browser.	Successful
4	Create a package with the help of the package manager view in the dashboard. Use the values `50` for `width`, `height` and `depth`. Use `10` for kilograms.	Successful
5	CSet `Conveyor Speed` to `50` and start the conveyor belt switching `Conveyor` to `ON`.	Successful

Table 12: Manual user test for the package slide

Result The package slide down the slide and increase its counter by one.
Tester feedback Everything worked as intended, and the steps were described precisely.

11.4.8. A Blueprint actor with a read operation to the OPC UA Server can be created. (03.06.2025)

Description In this test the functionality and accessability of the creation of a new Blueprint actor together with an OPC UA read operation is tested. The user is able to access a boolean value from an existing node on the server side of the Node-RED server.
Preconditions The project is running in Unreal Engine, the OPC UA server is working and the dashboard can be opened on the OPC UA server.
Test steps
Tester Test_User_1

Step	Description	Result
1	Create a new Blueprint actor and name it `BP_TestActorRead`.	Successful
2	Add the mesh `TestMesh` to the Blueprint actor.	Successful
3	In the Event Graph create the logic to read a boolean from a node of the Node-RED server. The logic should get triggered with the event `Event BeginPlay`.	Successful
4	Read the boolean value and print it out using the `Print String` Blueprint function, when the level is started.	Successful

Table 13: Manual user test on creating a read operation with Blueprint

Result The level can be started and the `BP_TestActorRead` makes a read call to the Node-RED server. It then prints out the value it reads, this should be either `True` or `False`, depending on the node the user chose to read.
Tester feedback Everything worked as intended, and the steps were described precisely.

11.4.9. A Blueprint actor with a write operation to the OPC UA Server can be created. (03.06.2025)

Description In this test the functionality and accessability of the creation of a new Blueprint actor together with an OPC UA write operation is tested. The user is able to write a boolean value to an existing node on the Node-RED server.
Preconditions The project is running in Unreal Engine, the OPC UA server is working and the dashboard can be opened on the OPC UA server.
Test steps
Tester Test_User_1

Step	Description	Result
1	Create a new Blueprint actor and name it `BP_TestActorWrite`.	Successful
2	Add the mesh `TestMesh` to the Blueprint actor.	Successful
3	In the Event Graph create the logic to write a boolean value to a node of the Node-RED server. The logic should get triggered with the event `Event BeginPlay`.	Successful
4	Check if the boolean value changes on dashboard of the Node-RED server.	Successful

Table 14: Manual user test on creating a write operation with Blueprint

Result The level can be started and the `BP_TestActorWrite` makes a write call to the Node-RED server. The value on the dashboard of the Node-RED server should change from `False` to `True` or from `True` to `False` depending on the node the user selected.

Tester feedback Everything worked as intended, and the steps were described precisely.

11.4.10. A Blueprint actor with a subscription to the OPC UA Server can be created. (03.06.2025)

Description In this test the functionality and accessability of the creation of a new Blueprint actor together with an OPC UA subscription is tested. The user is able to subscribe to a boolean value of an existing node on the Node-RED server.

Preconditions The project is running in Unreal Engine, the OPC UA server is working and the dashboard can be opened on the OPC UA server.

Test steps

Tester Test_User_1

Step	Description	Result
1	Create a new Blueprint actor and name it `BP_TestActorSub`.	Successful
2	Add the mesh `TestMesh` to the Blueprint actor.	Successful
3	In the Event Graph create the logic to subscribe to a boolean value of a node of the Node-RED server. The logic should get triggered with the event `Event BeginPlay`.	Successful
4	Every time the subscribed value changes, the logic should print the value using the `Print String` Blueprint function.	Successful

Table 15: Manual user test on creating a subscription operation with Blueprint

Result The level can be started and the `BP_TestActorWrite` makes a write call to the Node-RED server. The value on the dashboard of the Node-RED server should change from `False` to `True` or from `True` to `False` depending on the node the user selected.

Tester feedback Everything worked as intended, and the steps were described precisely.

Figures

Figure 1 Existing 2D simulation which is utilizing a SVG with moveable parts. 4

Figure 2 Real life example of the package sorting machine constructed in the 2D SVG. 4

Figure 3 2D mockup of a retracted pneumatic cylinder and its green sensor which will detect the position of the piston. 7

Figure 4 2D mockup of an extended pneumatic cylinder and its green sensor which will detect the position of the piston. 7

Figure 5 Mockup of a the conveyor belt with three packages and two sensors for detecting packages. 8

Figure 6 Mockup of a slide at the end of the conveyor belt. It will hold the packages as they fall of the belt and has a sensor for counting the amount of packages that slid down. 8

Figure 7 Mockup of the combination of all the models showing a complete machine lineup. The wall that prevents the packages from moving is meant to represent a pneumatic cylinder. However, since it is difficult to depict in a 2D model, we used a wall instead for this mockup. 9

Figure 8 System context diagram for the whole setup and interaction between the Unreal Engine the OPC UA server and a test user. 28

Figure 9 Node-RED editor configuration view for the Cylinder1 of the OPC UA server. 29

Figure 10 Node-RED editor dashboard view for the Cylinder1 of the OPC UA server. 29

Figure 11 Address space editor where the OPC UA nodes are defined and can be added 30

Figure 12 Flow configuration for the package manager where multiple input fields are defined and two buttons that increase the respective counter on the OPC UA server. 32

Figure 13 Dashboard generated by the nodes for the package manager where packages can be created and deleted. 32

Figure 14 Subscription error message because the server is limited in the number of subscriptions it can handle 33

Figure 15 Content Drawer inside the Unreal Engine editor. This is where the GameInstance is located. 35

Figure 16 OPC UA client connection settings to configure the error message visualizer and the server connection url. 35

Figure 17 Example Blueprint configuration that reads a boolean value from the OPC UA server and prints the value to the screen. 37

Figure 18 Flow chart diagram that displays how the read request of the boolean value from the OPC UA server is handled. This also applies to other data types. 37

Figure 19 Example Blueprint configuration that writes a boolean value to the OPC UA server. Home Pos Id is a variable that represents the OPC UA node ID. 39

Figure 20 Flow chart diagram that displays how the write request of the value to the OPC UA server is handled. 39

Figure 21 Blueprint function configuration that sets up a timer to call the `UA_Client_run_iterate()` function periodically in order to check for changes in the active subscriptions. 41

Figure 22 The custom created variable of type OPCUASubscriber that holds the information about the subscription and the callback function. 42

Figure 23 Blueprint function configuration that setups a subscription to a boolean OPC UA node with a callback function that either retracts or extends the pneumatic cylinder. 42

Figure 24 Sequence diagram that shows the entire interactions between the simulation and the OPC UA server. Shows the process of the pneumatic cylinder that creates a subscription to a boolean node and the process of receiving a notification from the server that triggers the respective callback function. 43

Figure 25 Engine Scalability Settings in the Unreal Engine editor with the recommended settings for maximum performance. 45

Figure 26 Blueprint functions to set the engine scalability to low by default. 46

Figure 27 Starting screen in Epic Games client for Unreal Engine. 46

Figure 28 Unreal Engine editor interface for modeling a 3D level in the Selector Mode. 47

Figure 29 Settings window for the Unreal Engine Revision Control setup. 48

Figure 30 Submit content in Unreal Engine to the revision control via user interface. 48

Figure 31 Select files and write a message for the commit in the Unreal Engine user interface. 48

Figure 32 Datasmith CAD Importer plugin that can be enabled in the plugin manager of Unreal Engine. The plugin is responsible for converting CAD files to Unreal Engine models. 50

Figure 33 Import button to convert a CAD file to a Unreal Engine model. 51

Figure 34 Import settings for the CAD file. The settings are set to default values and should be sufficient for most cases. 51

Figure 35 Imported CAD model in the content drawer. The folder contains all the assets that are used in the model and is represented by the Datasmith Scene file. 52

Figure 36 CAD model in the scene. The model here is by default on its side but can be rotated and moved like any other object in Unreal Engine. 52

Figure 37 Convert Datasmith Scene to Blueprint Class by selecting all the child components and clicking on “Convert Selection to Blueprint Class”. 53

Figure 38 Converted CAD model to Blueprint Class. The class contains all the components of the model and logic can be added to it. 53

Figure 39 Logic of the converted CAD model. It detects when a package is inside the collision box and applies a force to the package in the direction of the conveyor belt. 54

Figure 40 Mesh from the converted CAD model where there is no collision mesh. Menu open that generates a collision mesh automatically. 54

Figure 41 All the Blueprint models accessible in the content drawer. 55

Figure 42 Placing the Blueprint model of a pneumatic cylinder in the world via drag and drop from the content drawer 56

Figure 43 Rotating the pneumatic cylinder in the rotating mode, which can be selected in the top right corner 56

Figure 44 All the possible settings for the pneumatic cylinder. These are the variables that are used behind the scenes in the Blueprint functions. 57

Figure 45 This Error message that, we created, is displayed on the screen when the node IDs are not set or not found on the OPC UA server or if the type of the node is not correct. 57

Figure 46 The whole Blueprint logic of the pneumatic cylinder, separated in setup where the subscription and other internal variables are set, the callbacks for when the value changes and the actual movement of the piston. 58

Figure 47 Placing the Blueprint package in the world via drag and drop from the content drawer. . . 59

Figure 48 Construction script of the package where the width, height, depth and mass are set to the values of the variables. 59

Figure 49 Physical material properties of the package. It has a low friction in order to move smoothly on the conveyor belt. 60

Figure 50 Placing the Blueprint package spawner in the world via drag and drop from the content drawer. 61

Figure 51 Settings for the package spawner where the Z offset of where the packages will be spawned can be set. 61

Figure 52 Blueprint logic of the package spawner that spawns and deletes packages dynamically during runtime. 62

Figure 53 Placing the Blueprint sensor in the world via drag and drop from the content drawer. 63

Figure 54 Blueprint logic of the sensor that writes true to a boolean node in the OPC UA server when an object enters the beam and false when it leaves the beam. The node ID can be set in the details panel of the sensor. 63

Figure 55 Conveyor belt model being dragged into the world from the content drawer. The settings for the conveyor belt can be set in the details panel on the right side of the screen. 64

Figure 56 Conveyor belt splines being raised forming a linear path for packages to move along. 64

Figure 57 Multiple conveyor belts with different wall settings forming one long conveyor belt. The first and last conveyor belt have no walls, while the middle conveyor belt has walls. 65

Figure 58 Settings of the conveyor belt. Walls toggles between having walls or not, the `Conveyor on Node` is the OPC UA node ID of the boolean node indicating if the conveyor belt is turned on or not and `Conveyor Position Node` is the OPC UA node ID of the float node indicating the target position of the conveyor belt in centimeters. 65

Figure 59 The whole Blueprint logic of the conveyor belt, separated in setup where the subscriptions to the OPC UA server are set up, the callbacks for when the value changes and the actual movement of the conveyor belt. 67

Figure 60 Placing the Blueprint slide in the world via drag and drop from the content drawer. 68

Figure 61 Blueprint logic of the slide that counts up the packages that passed the sensor and displays the counter in a floating text above the slide. 69

Bibliography

[1] "What are Digital Twins?." Accessed: Mar. 11, 2025. [Online]. Available: https://unity.com/topics/digital-twin-definition

[2] "The Past: History of Digital Twins." Accessed: May 29, 2025. [Online]. Available: https://foundtech.me/history-of-digital-twins-in-industry-past-and-future/?lang=en

[3] W. Kritzinger, M. Karner, G. Traar, J. Henjes, and W. Sihn, "Digital Twin in manufacturing: A categorical literature review and classification," *IFAC-PapersOnLine*, vol. 51, no. 11, pp. 1016–1022, 2018, doi: https://doi.org/10.1016/j.ifacol.2018.08.474.

[4] "What is OPC?." Accessed: Mar. 11, 2025. [Online]. Available: https://opcfoundation.org/about/what-is-opc/

[5] "Unified Architecture - Landingpage." Accessed: Mar. 11, 2025. [Online]. Available: https://opcfoundation.org/about/opc-technologies/opc-ua/

[6] "What is a OPC-UA node?." Accessed: May 29, 2025. [Online]. Available: https://stackoverflow.com/questions/55050908/what-is-a-node-in-opc-ua#:~:text=the%20node%20concept%20of%20OPC,what%20they%20are%20used%20for.

[7] "OPC UA specification - 2 Terms, definitions and abbreviated terms." Accessed: Mar. 11, 2025. [Online]. Available: https://reference.opcfoundation.org/Core/Part1/v105/docs/2

[8] Engine Unreal, "Unreal Engine," *Retrieved from Unreal Engine: https://www.unrealengine.com/en-US/unreal-engine-5*, 2018, Accessed: Mar. 11, 2025. [Online]. Available: https://d1.awsstatic.com/awsmp/solutions/mksol-files/media-entertainment/AWSMP-MandE-Game-Tech-Datasheet-Epic-Games.pdf

[9] "Unreal Engine." Accessed: Mar. 12, 2025. [Online]. Available: https://www.unrealengine.com/en-US

[10] "Learn Unreal Engine." Accessed: Mar. 12, 2025. [Online]. Available: https://dev.epicgames.com/community/unreal-engine/learning?sort_by=views_count

[11] "About Epic Games." Accessed: Mar. 12, 2025. [Online]. Available: https://www.epicgames.com/site/en-US/about?lang=en-US

[12] "Unreal Engine." Accessed: Mar. 12, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Unreal_Engine

[13] "Mitsubishi Melsoft Gemini 3D." Accessed: Mar. 06, 2025. [Online]. Available: https://emea.mitsubishielectric.com/fa/products/software-overview/melsoft-gemini

[14] "Azure Digital Twin." Accessed: Mar. 06, 2025. [Online]. Available: https://azure.microsoft.com/en-us/products/digital-twins

[15] "Siemens SIMIT." Accessed: Mar. 06, 2025. [Online]. Available: https://www.siemens.com/global/en/products/automation/industry-software/simit.html

[16] "ABB RobotStudio." Accessed: Apr. 01, 2025. [Online]. Available: https://new.abb.com/products/robotics/de/software-digitale-loesungen/robotstudio

[17] "Visual Components." Accessed: Apr. 01, 2025. [Online]. Available: https://www.visualcomponents.com/

[18] K. Rongiers, "Digital Twin of the Vertical Warehouse Application." Accessed: Mar. 11, 2025. [Online]. Available: https://www.theseus.fi/handle/10024/861994

[19] "An OPC UA client gateway-based architecture for SCADA systems with automatic mental fatigue detection application." Accessed: Jun. 04, 2025. [Online]. Available: https://summit.sfu.ca/item/36074

[20] "Real-time two-way data transfer with a digital twin via web interface." Accessed: Jun. 04, 2025. [Online]. Available: https://aaltodoc.aalto.fi/items/08de97f7-5af6-435a-b189-c21c8254fd94

[21] "Industry 4.0 - Digital Twins and OPC UA." Accessed: Jun. 04, 2025. [Online]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2561319

[22] "An Immersive Digital Twin Applied to a Manufacturing Execution System for the Monitoring and Control of Industry 4.0 Processes." Accessed: Jun. 04, 2025. [Online]. Available: https://www.mdpi.com/2076-3417/14/10/4125

[23] "An Immersive Digital Twin Applied to a Manufacturing Execution System for the Monitoring and Control of Industry 4.0 Processes." Accessed: Jun. 04, 2025. [Online]. Available: https://www.mdpi.com/2076-3417/14/10/4125

[24] J. V. Sørensen, Z. Ma, and B. N. Jørgensen, "Potentials of game engines for wind power digital twin development: an investigation of the Unreal Engine," *Energy Informatics*, vol. 5, no. 4, p. 39, doi: 10.1186/s42162-022-00227-2.

[25] R. Piha, "Unreal Twins," 2024. Accessed: Apr. 01, 2025. [Online]. Available: https://diglib.uibk.ac.at/download/pdf/10338423.pdf

[26] "Design of a UE5-based digital twin platform." Accessed: Jun. 04, 2025. [Online]. Available: https://arxiv.org/abs/2407.03107

[27] "Which architecture should be implemented to manage data from the real world, in an Unreal Engine 5 simulator and in the context of mixed reality?." Accessed: Jun. 04, 2025. [Online]. Available: https://arxiv.org/abs/2305.09244

[28] "Verosim solutions." Accessed: May 31, 2025. [Online]. Available: https://www.verosim-solutions.com/

[29] "Combining Digital Twins of Robotic Systems with State-of-the-Art Game Engines for Enhanced Visualization and User Interfaces." Accessed: Jun. 04, 2025. [Online]. Available: https://largestructureproduction.sdu.dk/thesis/combining-digital-twins-of-robotic-systems-with-state-of-the-art-game-engines-for-enhanced-visualization-and-user-interfaces/

[30] Craig Larman, *UML and Patterns - Use Cases*. pp. 63–72. Accessed: Mar. 18, 2025. [Online]. Available: https://www.craiglarman.com/wiki/downloads/applying_uml/larman-ch6-applying-evolutionary-use-cases.pdf

[31] "ISO/IEC 25010." Accessed: Mar. 24, 2025. [Online]. Available: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

[32] "Unreal Engine 5 - Licensing options." Accessed: May 31, 2025. [Online]. Available: https://www.unrealengine.com/en-US/license

[33] "Unreal Engine 5 - WebGPU support." Accessed: May 31, 2025. [Online]. Available: https://forums.unrealengine.com/t/anyone-interested-in-webgpu-support-for-unreal-engine-5/1910658

[34] "Unreal Engine 5 - Lumen." Accessed: May 31, 2025. [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/lumen-global-illumination-and-reflections-in-unreal-engine

[35] "Unreal Engine 5 - Nanite." Accessed: May 31, 2025. [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/nanite-virtualized-geometry-in-unreal-engine

[36] "Unreal Engine 5 - Physics." Accessed: May 31, 2025. [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/physics-in-unreal-engine

[37] "Unity - Licenses." Accessed: May 31, 2025. [Online]. Available: https://unity.com/products

[38] "Unity - Features." Accessed: May 31, 2025. [Online]. Available: https://docs.unity3d.com/6000.1/Documentation/Manual/FeatureSets.html

[39] "Unity 3D - Physics." Accessed: May 31, 2025. [Online]. Available: https://docs.unity3d.com/6000.1/Documentation/Manual/PhysicsOverview.html

[40] "Unity Asset Store." Accessed: May 31, 2025. [Online]. Available: https://assetstore.unity.com/

[41] "open62541." Accessed: Mar. 12, 2025. [Online]. Available: https://www.open62541.org/

[42] "Mozilla Public License Version 2.0." Accessed: Mar. 12, 2025. [Online]. Available: https://www.mozilla.org/en-US/MPL/2.0/

[43] "C++ Based OPC UA Client & Server SDK (Bundle)." Accessed: Mar. 12, 2025. [Online]. Available: https://www.unified-automation.com/products/server-sdk/c-ua-server-sdk.html

[44] "OPC UA Clients & Server Integration in C++ Applications based on Windows." Accessed: Mar. 12, 2025. [Online]. Available: https://industrial.softing.com/products/opc-ua-and-opc-classic-sdks/opc-ua-c-sdks-for-windows.html

[45] "Node-OPCUA." Accessed: Mar. 12, 2025. [Online]. Available: https://node-opcua.github.io/

[46] "The MIT License - Open Source Initiative." Accessed: Mar. 12, 2025. [Online]. Available: https://opensource.org/license/mit

[47] "opcua-asyncio." Accessed: Mar. 12, 2025. [Online]. Available: https://github.com/FreeOpcUa/opcua-asyncio

[48] "GNU Lesser General Public License." Accessed: Mar. 12, 2025. [Online]. Available: https://www.gnu.org/licenses/lgpl-3.0.html

[49] "UGameInstance." Accessed: Mar. 17, 2025. [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/API/Runtime/Engine/Engine/UGameInstance

[50] "UA Part 4 Services - 5.13.1 Subscription model." Accessed: Mar. 18, 2025. [Online]. Available: https://reference.opcfoundation.org/Core/Part4/v104/docs/5.13.1

[51] "Download Unreal Engine." Accessed: Mar. 12, 2025. [Online]. Available: https://www.unrealengine.com/en-US/download

[52] "Datasmith Limitations." Accessed: May 14, 2025. [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/datasmith-supported-software-and-file-types?application_version=5.4#exportpluginsformacos

[53] "Datasmith File Formats." Accessed: May 14, 2025. [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/datasmith-supported-software-and-file-types

[54] "Datasmith." Accessed: May 14, 2025. [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/datasmith-plugins-for-unreal-engine

[55] "GrabCAD." Accessed: May 14, 2025. [Online]. Available: https://grabcad.com/library/belt-conveyor-148

[56] "STEP File Format." Accessed: May 14, 2025. [Online]. Available: https://www.adobe.com/creativecloud/file-types/image/vector/step-file.html