Universität Osnabrück
Institut für Informatik
AG Technische Informatik
Prof. Dr.-Ing. Mario Porrmann

Master's Thesis

Mixed Reality Environment for Robot-in-the-Loop-Testing using Digital Twins

Jonas Kittelmann

Betreuer: Prof. Dr.-Ing. Mario Porrmann
M.Sc. Philipp Gehricke

Osnabrück, 23 December 2025 M 00

Abstract

Hier sollte in einem Abstract kurz der Inhalt der Arbeit erläutert werden.
Zuerst auf deutsch. [**NOTFINAL**] Then an abstract of the thesis in English should follow.

Contents

1 Introduction

Robotic and Autonomous Systems (RAS) combine multiple disciplines, including control engineering and robotics, mechanical engineering, electronics, and software engineering. While traditional testing relies on domain-agnostic software metrics like code coverage and failure counts, the multidisciplinary nature of RAS requires an extension of these methods to verify the system's autonomous operation, continuous movement, and ability to handle uncertainty. For researchers and engineers specializing in software testing, adapting existing techniques for RAS presents a significant challenge. Therefore, there is extensive literature dedicated to proposing and evaluating different testing techniques and processes, showing how essential it is to establish structured methods for ensuring the reliability of these complex systems. [AMV23]

This validation challenge highlights a fundamental tension between simulation-only testing and full-scale physical experimentation, which are the two established methods in robotics evaluation. Simulation is central to robotic development, offering a way to test control logic and experiment with system configurations before committing them to hardware [Hu05; Mic04]. Simulations provide a rapid prototyping environment that minimizes infrastructure costs and logistical difficulties [Mic04; DRC+17]. Furthermore, they utilize virtual time to execute computationally expensive algorithms significantly faster than real-time hardware, enabling efficient design exploration [Mic04]. A discontinuity often occurs during the transition to real hardware, where the models validated in simulation are discarded and the code is rewritten from scratch, introducing potential errors [Hu05]. Additionally, simulation environments rarely replicate the full complexity of the physical world, missing nuances such as sensor noise, surface friction, and the unique manufacturing variances found in individual motors [BM18]. Testing with real hardware allows for the highest fidelity and controls to be refined based on the presence of all the unpredictable physics and variability that simulations cannot replicate [Hu05; CCC+21]. However, this approach has its own drawbacks. Conducting physical experiments can be resource-intensive and time-consuming and requires significant funds, manpower, and infrastructure [Hu05; DRC+17; Mic04]. Moreover, virtual environments are essential for validating scenarios that are hazardous or physically inaccessible in the real world, such as bomb disposal

missions, space exploration, or pilot emergency training [BM18; Hu05]. Testing the entire system using only real components is also severely limited for complex, large cooperative systems because of issues relating to complexity and scale [Hu05]. As neither of these approaches seems satisfactory on its own, an intermediate solution is required that could connect simulation with real-world experimentation [Hu05].

To bridge this gap, hybrid methods like Robot-in-the-Loop (RitL) simulation have emerged, which allow physical robots to operate within virtual environments [Hu05]. This is often centered on a digital twin, which is a real-time virtual replica of the physical robot [AA23] and employs Augmented Reality (AR) as a Mixed Reality medium to facilitate user interaction [MV20]. The combination of these technologies provides a robust paradigm for the comprehensive testing and validation of robotic systems.

The Virtual Environment for mobile Robotic Applications (VERA) framework integrates digital twins, AR, and vehicle-in-the-loop testing into a single system [Geh24]. VERA provides a modular platform for creating, managing, and visualizing synchronized virtual environments, which are presented both in simulations and as real-world projections [Geh24]. This master's thesis builds directly upon the foundation laid by this original framework.

While the VERA framework provides a strong conceptual foundation, its original technical implementation contained critical limitations regarding physical realism, scalability, and user interaction [Geh24]. Its custom environment manager lacked a realistic physics engine, while its 2D visualizer, built with Pygame, showed performance degradation when handling a large number of dynamic objects, leading to delayed and incomplete visualizations [Geh24]. Furthermore, while an AR interface was implemented, the original thesis identified immersive Virtual Reality (VR) interaction as a key area for future work [Geh24].

This thesis overcomes those limitations by offering a novel, hardware-accelerated architecture capable of real-time physics simulation to replace the custom manager and visualizer used within VERA. To address these needs, a game engine is selected to replace the custom solutions, specifically chosen to fulfill the requirements for high-fidelity graphics, integrated physics, and native support for VR. All communication and data synchronization between the physical robot and the simulation environment is carried out with the Robot Operating System 2 (ROS 2) [MFG+22], a middleware framework used within state-of-the-art robotics.

The core goal is to create a real-time digital twin [AA23] of the Educational Modular Autonomous Robot Osnabrück (EMAROs) robot [GTP25], integrating its complete model, live sensor data, and physical properties such as mass, inertia, and collision

models. This digital twin serves as the foundation for Robot-in-the-Loop [Hu05] testing scenarios. Additionally, this project extends the original framework [Geh24] to include Virtual Reality (VR) [EM21] and desktop interfaces for scenario modification, robot control, and data visualization. This also includes reimplementing and enhancing VERA's AR floor projection feature by using the selected engine's rendering tools to improve visual quality and system capabilities.

[**NOT FINAL**]The remainder of this thesis is structured as follows: after reviewing the basic concepts and technologies, such as RitL, Digital Twins, Mixed Reality, ROS 2, and several simulation engines in Chapter 2, an in-depth review of the original framework of VERA is presented, along with its limitations. In Chapter 3, the functional and non-functional requirements for the new system are identified. The architecture and implementation of the new framework are explained in detail in Chapter 4, with particular reference to the integration of the game engine with ROS 2, the development of the digital twin, and the mixed reality interaction system. Then, a quantitative evaluation regarding the performance of the system is presented along with a discussion of the results in Chapter 5. Finally, Chapter 6 summarizes the thesis contributions and gives an outlook on possible future research.[**NOT FINAL**]

2 Background and Related Work

Einführung in das kapitel [**NOT FINAL**]

2.1 Robot-in-the-Loop-Testing

The validation and verification of modern Robotic and Autonomous Systems (RAS) is a significant challenge due to their complexity [AMV23]. This is because they integrate software, mechanical and electrical engineering all at once [AMV23]. A central problem is ensuring that the software and hardware work together seamlessly, especially when testing both simultaneously [AMV23]. The X-in-the-Loop (XitL) simulation paradigm addresses this challenge [AAR+19]. It provides a framework to integrate the flexibility of software simulation with the realism of physical experiments [AAR+19].

A foundational XitL method is Hardware-in-the-Loop (HitL) simulation, which is a technique for testing mechatronic systems [MTH22; AAR+19]. The main principle of HitL is creating a closed loop between the real hardware that is being tested and a simulation that represents the rest of the system or its operational environment [MTH22]. This setup effectively simulates the hardware to respond as if it were operating in a real system, which allows for testing across a wide range of virtual scenarios [AAR+19]. The main reason for using HitL is to shorten development cycles and prevent costly or dangerous failures by making exhaustive testing possible before the system is completely implemented [MTH22]. This is why HitL is essential in industries like automotive, aerospace and robotics, where real-world testing can be too expensive, dangerous, or even impossible [AAR+19].

Robot-in-the-Loop (RitL) [MTH22] simulation extends the HitL concept. Instead of just a component, the hardware under test is a complete robotic system, such as an uncrewed vehicle [MTH22]. RitL replaces components of a pure simulated setup with the actual robot, increasing the realism of testing [Hu05]. [**NOT FINAL**]As illustrated in Figure **??**[**NOT FINAL**], a typical RitL configuration has the robot's real actuators operating in the physical world, while its sensors interact with a simulated environment

instead [Hu05; MTH22]. This creates a hybrid setup where, for example, the robot might use virtual sensors to see objects in the simulation but use its real motors to move physically [Hu05]. To keep the physical and virtual worlds synchronized, the real robot often has a virtual counterpart in the simulation whose state is updated as the physical robot acts and moves [Hu05].

Figure 2.1: [**NOT FINAL**]The real robot's actions affect its virtual counterpart and the virtual environment provides sensor data back to the real robot.[**NOT FINAL**]

The main benefit of RitL is that it uses the dynamics of actual hardware for repeatable testing of high-level software, such as navigation algorithms [MTH22]. This method avoids the expense, complexity and risk of full physical testbeds while being a secure and useful substitute [MTH22]. RitL becomes therefore a tool for safely evaluating system performance in the lab [Hu05; MTH22]. This is especially important when working on projects that are hard to replicate, such as large-scale robot swarms or Mars rover missions, or when safety is at risk [Hu05; MTH22].

The RitL paradigm has been widely applied to validate complex autonomous systems, particularly in the automotive field using Vehicle-in-the-Loop (VitL) testing. For example, the Dynamic Vehicle-in-the-Loop (DynVitL) architecture integrates a real test vehicle with the high-fidelity CARLA simulator, which operates on the Unreal Engine [DSR+22]. As shown in Figure 2.2, this approach allows a vehicle on an empty track to be stimulated by sensor data from a virtual world [DSR+22]. This allows automated driving functions to be safely and reliably tested in situations that would be hazardous to physically replicate [DSR+22]. CARLA and the Unreal Engine are also used in a similar Vehicle-in-Virtual-Environment (VVE) method which establishes a closed loop in which the motion of the real vehicle is tracked and reflected in the virtual world, making it possible for its control systems to respond to simulated events [CCG+23].

Figure 2.2: An example of a VitL setup. A real car on a test track connected to a high-fidelity simulator like CARLA that generates virtual traffic and sensor data. [DSR+22]

These examples show a trend towards using game engine based simulators to evaluate autonomous vehicles. This method falls somewhere between physical testing, where safety and consistency can be problematic, and pure software simulation, which frequently lacks vehicle dynamics fidelity [CCG+23]. Some systems even stimulate the vehicle's actual sensors. For instance, the Radar Target Simulator (RTS) can feed artificial radar echoes from a virtual scene to the vehicle's actual radar sensor [DKK+21]. This allows for end-to-end validation of the entire perception and control pipeline [DKK+21].

The X-in-the-Loop approach is found in many different areas. In marine robotics, a VitL framework was developed to test the long-term autonomy of a robot swarm. In this system, an Autonomous Surface Vehicle (ASV) interacts with multiple simulated underwater sensor nodes to test cooperative behaviors without the logistical cost of deploying a full swarm [BVM20]. In the aerospace domain, the RFlySim platform uses

a Field Programmable Gate Array (FPGA)-based HitL system to create a high-fidelity simulation for testing Unmanned Aerial Vehicle (UAV) autopilot systems in a lab, which reduces the need for expensive and risky outdoor flights [DKQ+21].

These approaches continue to move toward deeper integration. The Scenario-in-the-Loop (SciL) paradigm creates a mixed reality that blurs the boundary between physical testing and virtual simulation [VTS21]. Unlike standard HitL methods, SciL allows the simulation to actively drive the physical environment [HLT+19]. A central software system uses virtual triggers to physically actuate real-world infrastructure, such as moving robotic dummies or changing traffic signals [HLT+19]. Simultaneously, the system manipulates the vehicle's perception using Augmented Reality (AR) to inject synthetic data directly into the vehicle's communication network alongside physical sensor readings [VTS21]. As a result, the vehicle cannot distinguish between physical objects and virtual ones [VTS21]. This indistinguishable perception compels the vehicle to execute real control maneuvers, such as emergency braking, in response to purely digital stimuli [VTS21].

2.2 Digital Twins

The Digital Twin is an important concept in many current X-in-the-Loop frameworks. The Digital Twin serves as the virtual counterpart to a physical system, acting as a virtual copy that allows real-time monitoring and simulation [AA23]. The idea is to create a digital information model of a physical system that stays linked to it for the duration of its lifecycle [GV17].

This concept was initially known as the Information Mirroring Model and consists of three core elements: the physical product, its corresponding virtual model and the data connection that connects them [GV17; AA23]. A diagram of this structure is shown in Figure 2.3. The virtual model extends beyond a representation of physical dimensions; it is often a detailed simulation that can model mechanical, electrical and software properties of a system [LWS+21]. Information moves in both directions, so sensor data can flow from the real world to update the virtual model [GV17; LWS+21]. In turn, the virtual model can also send commands back to control or optimize the physical system [GV17; LWS+21]. This continuous, bidirectional data exchange characterizes a Digital Twin [AA23].

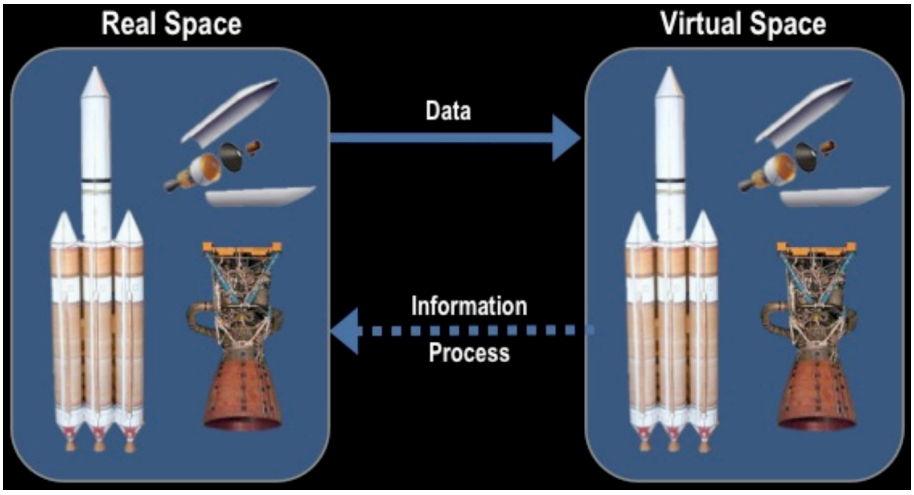


Figure 2.3: The foundational concept of a Digital Twin, illustrating the three core components: the physical entity, the virtual model and the bi-directional data connection that links them [GV17; LWS+21]. [Gri15]

Digital Twins in robotics are frequently created using simulation software and the middleware that connects the virtual simulation to the actual hardware is often the Robot Operating System (ROS) [MFG+22]. Stączek et al. used the Gazebo simulator with ROS to create a digital twin of a factory floor, which they used to test and optimize the navigation algorithms of a mobile robot in narrow corridors [SPD+21]. To validate a deep learning-based harvesting robot, R. Singh et al. adopted a similar strategy and created a digital twin of a greenhouse in Gazebo [BSH24]. Other simulators like CoppeliaSim and Webots are also common. Magrin et al. used CoppeliaSim and ROS to create a digital twin as a learning tool for mobile robot design [MCT21], while Marques et al. used Webots for an Automated Guided Vehicle (AGV), synchronizing it via an OPC-UA server [MRS24]. These examples show a common approach of using simulators to simulate robot kinematics and sensor feedback for closed-loop testing.

More recently, game engines have become a popular choice for creating digital twins, as they offer more realistic graphics and physics. Unity [Uni23], for instance, is used to build highly realistic virtual environments, as seen in Figure **??**. Pérez et al. utilized Unity to develop a digital twin of a robotic cell, which is a manufacturing station where robots, conveyors and safety systems work together to perform tasks [PRR+20]. In this framework, the digital twin acts as a live mirror of the physical cell [PRR+20]. The system uses the game engine to simulate physical laws and mechanical behaviors, ensuring that the virtual robots act exactly like the real ones [PRR+20]. To achieve this synchronization, the virtual model connects directly to the facility's control network to

read real-time data from the sensors [PRR+20]. This connectivity allows engineers to virtually validate the cell's design and logic before any equipment is installed and subsequently enables operators to train safely alongside the virtual robots [PRR+20].

Figure 2.4: [**NOT FINAL**]A comparison of robotic Digital Twin environments, showing a view from a traditional robotics simulator (left) versus a high-fidelity visualization from a modern game engine like Unity (right).[**NOT FINAL**]

Another important consideration is the technical capabilities of these engines. Yang et al. [YMZ20] simulated the physics of a UAV using Unity and its built-in NVIDIA PhysX engine. They also demonstrated the creation of virtual sensors, such as a Light Detection and Ranging (LiDAR) sensor, directly within the game engine using its raycasting API [YMZ20]. Research has also been done on the performance and reliability of these game engine-based frameworks. Kwon et al. developed a safety-critical Digital Twin integrated with Unity and ROS 2, which achieves minimal data transmission latency to enable the prediction of collisions in real-time [KYS+25].

2.3 Mixed Reality

Beyond the testing paradigm and the digital twin, the way a user interacts with the system is another part of a robotics framework. Virtual, Augmented and Mixed Reality have become promising technologies for improving the information exchange between humans and robots [WPC+22]. In robotics, Augmented Reality (AR) is an especially useful tool for enhancing Human-Robot Interaction (HRI) by integrating virtual objects into a real-world environment in real-time [MV20].

The relationship between these technologies is formally described by the foundational Reality-Virtuality Continuum concept, first introduced by Milgram and Kishino [MK94]. As illustrated in Figure 2.5, this continuum is a scale that is anchored by a purely real environment at one end and a completely virtual one at the other [MK94; SSW21; MV20].

Figure 2.5: The Reality-Virtuality Continuum, illustrating the spectrum from a real environment to a completely virtual one. [WPC+22]

A Virtual Reality (VR) environment is an endpoint of the continuum, where the user is totally immersed in and can interact with a fully synthetic world [MK94]. This approach is useful for HRI research, as it allows for testing interactions with virtual robots in scenarios where it might be unsafe or too expensive for physical hardware [WPC+22]. The general term Mixed Reality (MR) describes the entire spectrum between the two extremes, where real and virtual worlds are combined into a single display [MK94]. MR is made up of two main categories: AR and Augmented Virtuality (AV) [MK94; MV20]. AR is the process of adding virtual objects to a real environment [MK94]. This allows, for example, HRI researchers to place 3D data and intentions of a robot directly into the physical space of a user [WPC+22]. The opposite is Augmented Virtuality (AV), where a primarily virtual world is enhanced with elements from the real world, like live video feeds [MK94].

2.4 Robot Operating System 2

Robot Operating System 2 (ROS 2) is not an operating system but a middleware framework to simplify the creation of complex robotic systems [MFG+22]. It was redesigned from scratch to meet the challenges posed by modern robotics in domains ranging from logistics and agriculture to space missions and became the standard for both research and industry [MML+23; MFG+22]. At its core, ROS 2 is designed based on principles of distribution, abstraction, asynchrony and modularity that together allow the development of scalable and robust applications [MFG+22]. The architecture of ROS 2 is based on a distributed network of independent programs called Nodes [MFG+22]. A node is a single, self-contained executable that performs a specific task, such as controlling a motor, processing sensor data, or, in the case of this thesis, bridging communication between robots and the simulation [MFG+22]. The

nodes communicate through a set of defined patterns. The most common pattern is the publish-subscribe mechanism called Topics, illustrated in Figure 2.6 [MFG+22]. In this model, nodes can publish data as messages to a topic, while other nodes can subscribe to that topic to receive data asynchronously [MFG+22]. For tasks that require a direct request and a guaranteed response, ROS 2 offers Services, following a synchronous request-response pattern [MFG+22]. For long-running tasks that require continuous feedback and the ability to be preempted, ROS 2 provides a unique communication pattern called Actions [MFG+22]. An action consists of a goal, a feedback stream and a final result, making it optimal for managing tasks like navigation where the progress of a robot toward a goal has to be monitored over time [MMW+20a].

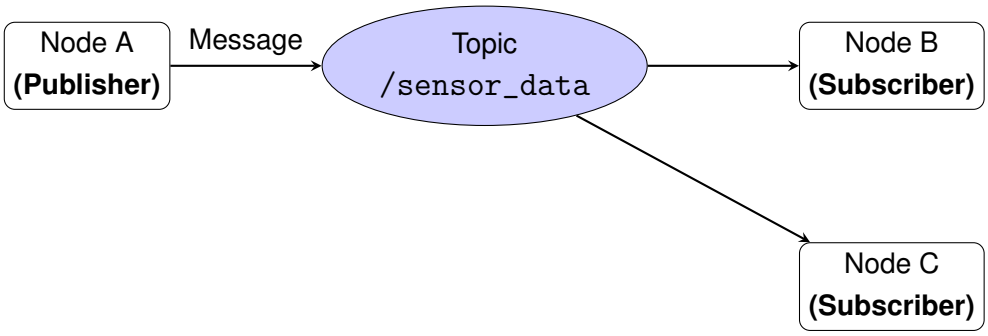


Figure 2.6: The ROS 2 publish-subscribe model. Node A publishes messages onto a central topic and any number of subscriber nodes (B and C) may receive that data without direct knowledge of the publisher.

An essential part of any mobile robot is coordinate frame management, which in ROS 2 is accomplished through its transform library, tf2 [Foo13]. The tf2 library standardizes the tracking of spatial relationships between the various parts of the robot and environment, placing them into a tree-like data structure, as illustrated in Figure 2.7 [Foo13]. This permits any node in the system to request at any time the position and orientation of any frame relative to another, a feature critical for transforming sensor data into a useful frame of reference [Foo13].

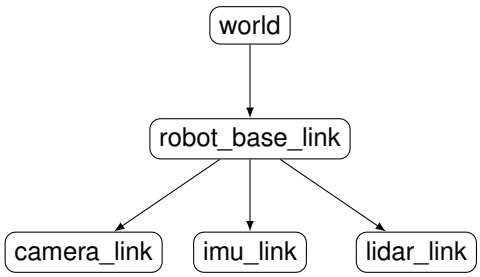


Figure 2.7: A simplified example of a ROS 2 transform tree (tf tree). The library maintains the hierarchical relationships so that a program can easily determine the transform from the `camera_link` to the `world` frame, for instance.

In the architecture of this thesis, ROS 2 provides the central data backbone, a term borrowed from digital engineering that defines an integrated communication layer for all relevant system knowledge [PKP+20]. This acts as the bridge that couples the physical EMAROs robot and its tracking system with the simulation environment. The digital twin of the EMAROs robot subscribes to ROS 2 topics to receive real-time data from the physical world and can synchronize its state through this communication layer [SKG+24]. For navigation of both the physical and virtual robots, this thesis utilizes Navigation2 (Nav2), the official, next-generation autonomous navigation framework for ROS 2 [MMW+20a]. Nav2 was designed from the ground up to orchestrate planning, control and recovery tasks using configurable Behavior Trees that are highly modular and can be changed at runtime to create custom navigation behaviors [MMW+20a]. Core to its operation, Nav2 separates the navigation task into two closely related, yet distinct components, namely a global planner, which seeks out an acceptable, long-range path through the environment and a local trajectory planner or controller, which generates velocity commands in order to follow that route while reacting to immediate obstacles [MML+23]. Using the Nav2 stack, a high-level goal, e.g. a coordinate in the environment, can be sent via a ROS 2 action to a robot and the system will take care of all complex path planning and collision avoidance autonomously.

2.5 The VERA Framework

This thesis directly builds on the Virtual Environment for mobile Robotic Applications framework, which was developed earlier by Gehricke [Geh24]. VERA was designed initially as a modular and scalable platform to bridge the validation gap between pure software simulation and real-world testing. It combines the concepts of Digital

Twins, AR and Vehicle-in-the-Loop testing to enable robots to interact with a virtual environment projected into the real world [Geh24].

The core idea of VERA is projecting a dynamic, interactive virtual world onto the physical floor where a real robot operates. The system synchronizes the virtual state with the actions of the physical robot in such a way that scenarios are possible where the robot can detect and react to virtual obstacles as if they were real. This system provides a flexible testbed for the development and evaluation of robotic applications without having to physically build complex environments [Geh24].

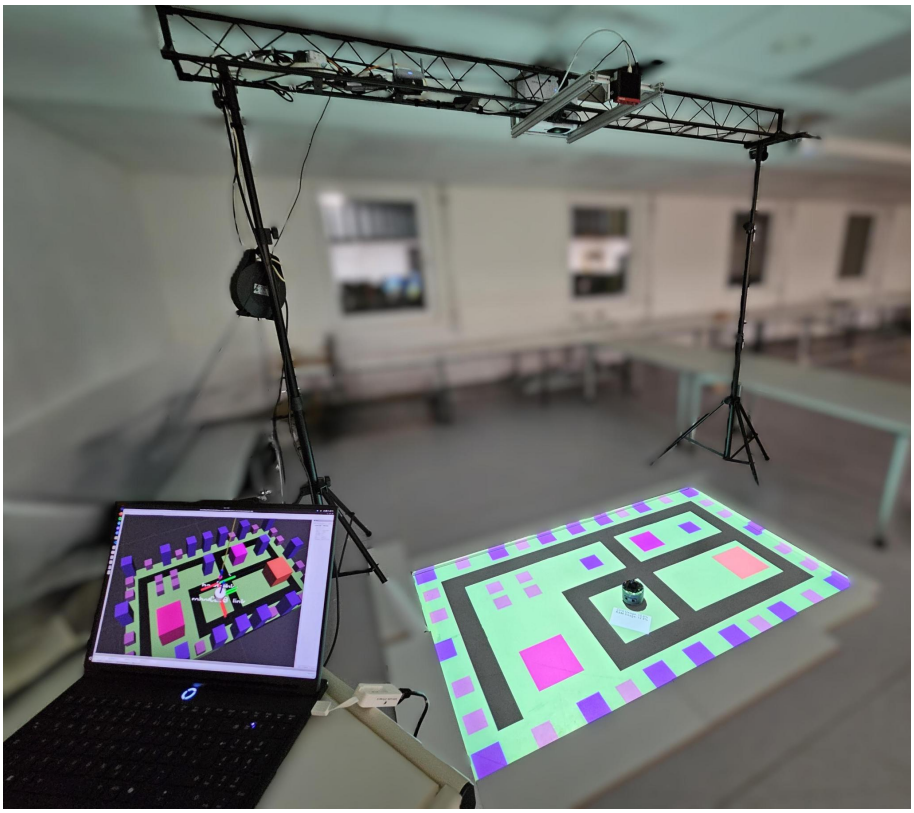


Figure 2.8: The physical setup of the VERA platform: A projector and tracking system are mounted on a frame above the test area. [Geh24]

2.5.1 EMAROs Test Platform

The capabilities of VERA are demonstrated by a mobile mini-robot called the Educational Modular Autonomous Robot Osnabrück (EMAROs), shown in Figure 2.9, which was developed at Osnabrück University specifically for research and education in robotics and artificial intelligence [GTP25]. The modular hardware architecture of

EMAROs is based on three main PCBs: a Host PCB containing high-level computing, a Power PCB managing energy supply and motor control and a Base PCB integrating ground sensors [Geh24]. The specific variant used in this work is equipped with a Raspberry Pi Compute Module 4 (CM4) running Ubuntu 24.04 and ROS 2 [Geh24].

Figure 2.9: The EMAROs robot is the robotic platform used in the testbed, equipped with a modular sensor suite and running ROS 2. [Geh24]

For obstacle avoidance, the robot is equipped with Time-of-Flight (ToF) distance sensors and an Inertial Measurement Unit (IMU) for orientation tracking [Geh24]. Moreover, there are two wide-angle cameras that can be configured for stereo vision applications or line-following tasks [Geh24]. Within VERA, EMAROs takes on the role of the robot, streaming real-time telemetry including odometry, battery status and sensor data to the simulation framework via ROS 2 to create a digital twin [Geh24].

2.5.2 Original VERA System Architecture

The software architecture of the original VERA framework was mainly a ROS 2 implementation and its structure included only three main components: the Virtual Environment Positioning System (VEPS), the Environment Manager and the Visualization System [Geh24].

Accurate localization is important to synchronize the physical robot with the projected virtual world and was provided by the VEPS in the original VERA framework [Geh24].

The original VEPS implementation relied on a depth-based approach where a ceiling-mounted Allied Vision Ruby 3D stereo camera captured point clouds of the test area [Geh24]. The point clouds were filtered by a Median Absolute Deviation (MAD) algorithm to calculate the centroid of the robot [Geh24]. However, for maximum robustness and precision for this thesis, the positioning system was updated to use an ArUco marker-based tracking system developed in parallel research by [Spr25], as shown in Figure **??**. Although the point-cloud method provided a markerless solution, the ArUco-based approach has higher reliability and lower noise [Spr25; Geh24]. By attaching a fiducial marker to the top of the EMAROs robot, the system is able to calculate the 6D pose (position and orientation) of the robot [Spr25]. This pose is published into the ROS 2 network, where it can instantly update the position of the digital twin [Spr25].

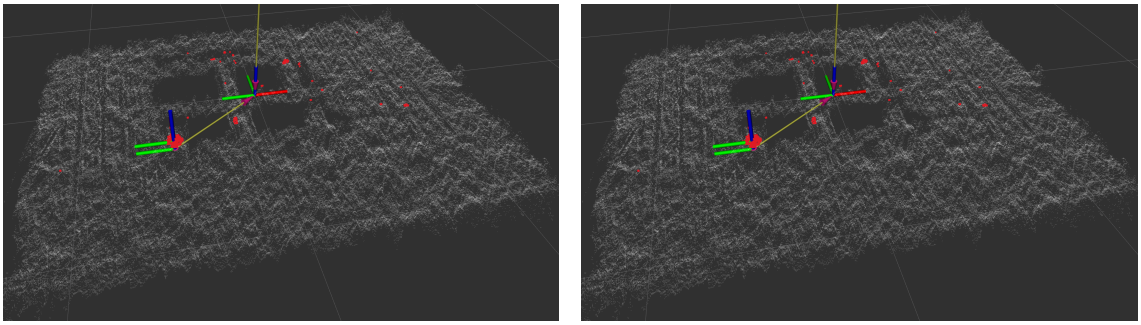


Figure 2.10: The original VERA used point clouds [Geh24], while the current iteration uses ArUco markers for pose estimation [Spr25].

[**ARUCO TRACKER BILD EINBAUEM**] The **Virtual Environment Manager** is the central brain of the simulation [Geh24]. Implemented as a C++ ROS 2 node, it parses environment definitions from standard Gazebo `world.sdf` files [Geh24]. It keeps track of the state of all virtual objects, including their positions and properties, such as whether an object is movable or static [Geh24]. One of the key roles of the manager is to manage the interactions of the robot with the virtual world [Geh24]. It operates a continuous loop, typically at 20 Hz, which checks the distance between the position of the robot given by the VEPS and the virtual objects [Geh24]. The manager triggers predefined actions if a collision is detected, such as removing a manipulatable object or stopping the robot [Geh24]. It relies on basic distance heuristics and not a physics engine, hence limiting interactions to simple ones [Geh24].

The **Visualization System** is responsible for rendering the virtual environment so it can be projected onto the physical floor [Geh24]. In the original VERA framework, this was achieved using a custom ROS 2 node and Pygame, a Python library designed for

2D game development [Geh24]. The visualizer subscribes to the object lists published by the Manager and the robot's path history [Geh24]. It provides a 2D, top-down, orthogonal view of the scene, drawing obstacles as simple geometric shapes [Geh24]. It gives Augmented Reality features, projecting dynamic information directly into the workspace [Geh24]. The driven path of the robot is displayed as a trail, while a system status panel with real-time CPU and battery usage is shown behind the robot, as illustrated in Figure 2.11 [Geh24].

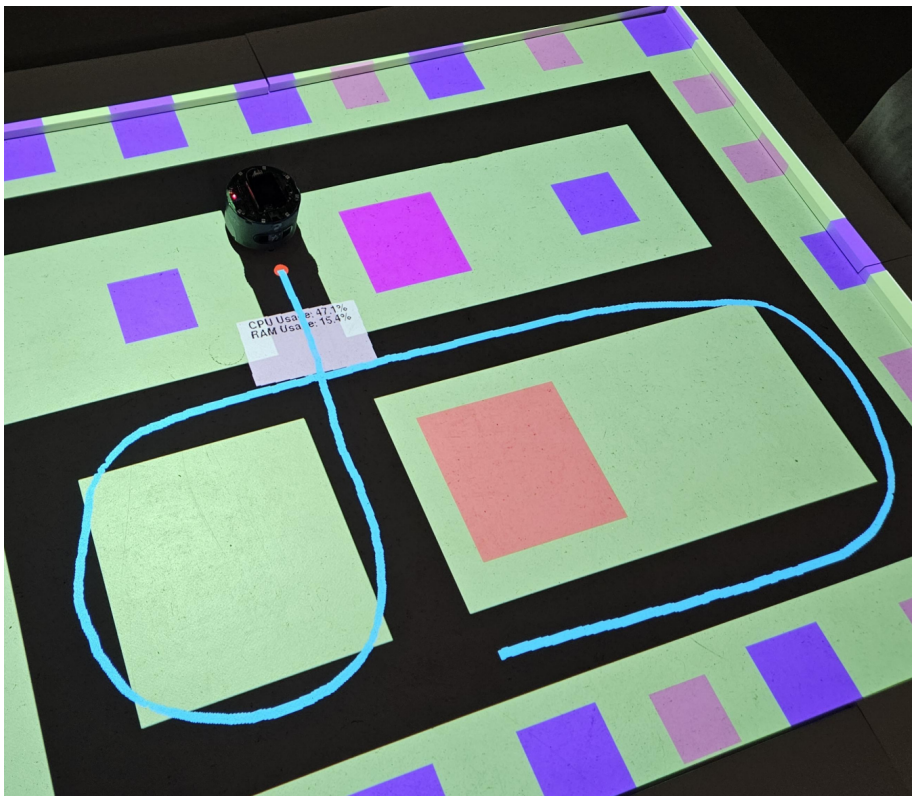


Figure 2.11: Original VERA visualization via Pygame, showing a projection of the robot's path and virtual obstacles onto the floor. [Geh24]

2.5.3 Identified Limitations

While VERA successfully demonstrated the concept of Robot-in-the-Loop testing with AR projections, its original software implementation contains several critical limitations, that this thesis sets out to overcome:

- **Lack of Realistic Physics:** The custom Environment Manager does not feature a physics engine [Geh24]. Collision detection is performed through simple radius

checks, meaning the robot cannot push objects or experience friction nor can it interact with complex geometries [Geh24]. This reduces the realism of the digital twin [Geh24].

- **Performance Scalability:** The Pygame-based visualizer utilizes the CPU for rendering and has performance-related issues for complex scenarios [Geh24]. Gehricke [Geh24] noted that in environments with a high number of dynamic objects (e.g., >800), the message queue became saturated, leading to considerable latency and incomplete visual updates.
- **Limited Interaction (No VR):** The system was designed strictly for 2D floor projections and Rviz [Geh24]. It does not support any kind of immersive VR interfaces that would enable a human operator to view the scene in 3D or to interact with the robot remotely [Geh24]. Therefore, the integration of VR was identified as a necessary future step for Human-Robot Interaction (HRI) [Geh24].

2.6 Simulation Engines for Robotics

The choice of a capable simulation engine is an important part in solving the limitations of the original VERA framework. The choice of platform affects the efficiency, accuracy and applicability of the digital twin [SKG+25]. Based on this, the review of simulation engines in this thesis is guided by four criteria which are required for a functional MR digital twin:

- **High Visual Fidelity:** Rather than focusing solely on graphical aesthetics, high visual fidelity is essential for generating accurate data for visual sensors (e.g., cameras) and ensuring user immersion in VR, addressing a significant limitation of robotics simulation platforms [DBF+25].
- **Physics Accuracy:** To accurately predict the kinematic and dynamic behavior of the EMAROs robot, the engine needs to have a strong physics backend [SKG+25].
- **Native VR/AR Support:** In order to extend VERA with the proposed Virtual Reality interfaces, the engine must have a framework for VR devices and provide capabilities for creating interactive environments without extensive middleware [CIR23].
- **Robust ROS 2 Integration:** ROS 2 forms the data backbone [PKP+20] of the system and should enable low-latency, high-throughput communication to synchronize the digital twin with the physical robot in real-time [SKG+25].

2.6.1 Robotics Simulators

Gazebo is the most widely used simulator in Industry 4.0 applications because of its deep integration with ROS, and it is open-source [SKG+25]. It delivers detailed physics simulations along with sensor data emulations, making it ideal for precise engineering applications [SKG+25]. However, while this simulator benefits from a large community and extensive documentation, there are noticeable drawbacks concerning visualization [DBF+25]. For example, the rendering quality of Gazebo is only moderate and not photorealistic to an extent that would enable immersive VR experiences [DBF+25]. Moreover, some works note that Gazebo has a steep learning curve for new users due to its complex configuration and less intuitive user interface compared to modern commercial tools [SKG+25; FCY25].

Webots is a well-established open-source platform, appreciated for cross-platform support and a large library of robot models [KYH+24]. Although it is quite effective in educational applications, the rendering capabilities of the software are somewhat dated compared to the current game engines, making it not fitting for high-fidelity MR graphics [KYH+24].

2.6.2 Physics-Based Simulators

A few other simulators were found in literature but considered less appropriate for the particular VR needs of this thesis. CoppeliaSim (previously known as V-REP) is widely applied in automation, yet it has less physics realism compared to the professional ones and its scaling is limited [FCY25]. MuJoCo does well with contact-rich tasks and Reinforcement Learning but works almost exclusively on the stability of the physics and not the visualization, so it lacks the inbuilt VR frameworks necessary for MR interaction [FCY25].

2.6.3 High Visual Fidelity and Game-Based Engines

Game engines have been widely adopted within the robotics community to overcome the visualization limitations of traditional robotics simulators.

NVIDIA Isaac Sim is the state-of-the-art in industrial simulation [DBF+25]. The platform is capable of photorealism through ray-tracing and has advanced GPU-accelerated physics through PhysX [DBF+25]. It excels in generating synthetic data for AI perception [KYH+24]. But Isaac Sim has a very high hardware barrier where high-end

NVIDIA RTX GPUs are specifically mandatory [DBF+25]. Being proprietary, it is more restricting for use in educational contexts compared to more open-source or widely accessible game engines [DBF+25].

Unreal Engine is renowned for photorealistic visuals and is capable of handling big, open-world environments [FCY25]. But it has a steep learning curve because of its dependency on C++ and therefore is not very well-suited for rapid prototyping compared to other engines like Unity [CIR23].

Unity Engine is establishing itself as the primary platform for Industry 5.0 applications, in particular those focusing on HRI and MR [FCY25]. It boasts a balance of high-fidelity graphics with a robust physics engine, PhysX [SKG+25]. Unity is also known for having one of the most mature MR ecosystems, along with an asset store, which lowers the barrier to entry for developers [SKG+25; CIR23]. Literature points out that the user-friendly interface of Unity and supportive community help with a softer learning curve compared to Gazebo or Unreal, particularly for users without deep C++ expertise [SKG+25; CIR23]. ROS 2 [MFG+22] is not natively supported by Unity. Support must be provided through third-party plugins. For the work described in this thesis, the ROS2 For Unity plugin from Robotec.AI has been used [Rob21]. This asset offers high-performance communication by loading the ROS 2 middleware layer (RCL layer) directly inside the Unity engine [Rob21]. Unlike bridged solutions, simulation entities now can act as native ROS 2 nodes in the context of Unity, thus respecting QoS and reaching lower latencies [Rob21].

2.7 Synthesis

The review of related work confirms that RitL testing [Hu05] combined with Digital Twin technology [AA23] sets up a paradigm for validating robotic systems. Current research highlights the need to move from basic kinematic simulations to high-visual-fidelity, physics-based environments that can more realistically simulate complex interactions and sensor data [SKG+25; DSR+22].

However, due to its reliance on custom software components, the original VERA framework has significant technical limitations [Geh24]. As explained in Section 2.5.3, the Pygame-based visualizer experiences performance bottlenecks in complex scenes and realistic object manipulation is impossible due to the lack of a dedicated physics engine [Geh24]. Moreover, the potential for researching Human-Robot Collaboration is limited by the lack of Virtual Reality support [Geh24].

To overcome these limitations, this thesis proposes integrating a simulation engine into the architecture of VERA using the ROS 2 [MFG+22]. The goal is to establish a robust, physics-enabled MR environment that bridges the gap between simulation and reality. This new development will be verified by creating demonstration scenarios where the EMAROs robot actively interacts with virtual elements and will serve as a proof of concept for improved system physical realism, scalability, and immersive interaction.

3 Requirements

This chapter establishes the functional and non-functional requirements for the Mixed Reality Environment. Intended as a validation tool for Robot-in-the-Loop (RitL) testing, the system must bridge the gap between virtual simulation and physical reality. The requirements follow a hierarchical structure, covering the simulation engine capabilities, system architecture, and digital twin interfaces. In addition, specific requirements regarding the autonomous agents acting within this environment to validate the system are designed.

3.1 Simulation Engine Capabilities

These requirements define the technical capabilities of the underlying simulation engine. The criteria provide a basis for the technology selection process in the implementation phase.

- **FR-01: Visual Fidelity:** The simulation engine shall be able to support rendering pipelines capable of producing image data representative of real-world lighting and material properties. This is required to provide realistic input for the computer vision algorithms of the robot.
- **FR-02: Physics Simulation:** Overcoming the limitations in the previous VERA framework, the engine shall provide a physics system that can resolve mass, friction, collision, and drag to simulate physical interactions such as a robot pushing obstacles.
- **FR-03: VR & AR Support:** The simulation engine shall support both VR and AR modes. It shall enable integration with VR and AR input and output devices, allowing users to interact with and visualize the environment seamlessly in either mode.

The system architecture is defined by limitations of the hardware and the need for standardized robotic communication.

- **FR-04: ROS 2 Integration:** The system shall use ROS 2 as the sole middleware for all communication. It shall act as a first-party in the network to minimize latency and shall make use of standard message definitions for all sensor streams, telemetry, and control commands to ensure interoperability with the physical robot.
- **FR-05: Time Synchronization:** The system shall act as the simulation time source and publish the clock signal. To prevent data drift in control algorithms, all simulated sensors and publishers shall synchronize their timestamps to this source.
- **FR-06: Scenario Management:** The system shall provide a mechanism to dynamically load and unload simulation environments via a network command during runtime. This is to enable test runs across different environments without restarting the application.

3.2 Digital Twin Interfaces

The system shall provide a digital representation of the robotic platform that mirrors its physical counterpart. This involves synchronization with real-world tracking, kinematic simulation, and the publishing of coordinate transforms.

- **FR-07: Pose Synchronization:** The system shall take real-time input from the external tracking system to synchronize the position and orientation of the digital twin with the physical robot.
- **FR-08: Tracking Failsafe:** The system shall provide a failsafe that halts the movement of the robot in case of an interruption of the tracking data stream beyond a defined threshold to avoid unwanted movements due to signal loss.
- **FR-09: Standalone Mode:** The system shall be able to simulate the behavior of the robot in the absence of physical hardware. It shall accept velocity commands and publish resulting odometry to test algorithms in a completely virtual environment.
- **FR-10: Coordinate Transforms:** The system shall publish dynamic coordinate transforms representing the motion of the robot.

3.3 Environment Capabilities

These requirements detail capabilities related to the virtual environment itself, such as generation of sensor data and mechanisms for dynamic interaction and visualization.

- **FR-11: Sensor Simulation:** The system shall provide simulated sensors able to generate data, including video feeds and ranging data. The sensors shall be capable of detecting both static environment structures and dynamic obstacles.
- **FR-12: Command Interface:** The system shall provide a generic network interface to receive commands and return status feedback to enable the robot to trigger simulation events.
- **FR-13: Dynamic Environment Interaction:** The system shall allow dynamic manipulation of the environment, including physical pushing and attaching of objects, modification of the state of objects, and changes in surface textures at runtime.
- **FR-14: Telemetry Display:** The system shall display real-time robot status data as floating UI elements attached to the digital twin within the 3D environment.

3.4 User Interface & Mixed Reality

These requirements outline how human operators visualize and interact with the system.

- **FR-15: AR Projection:** The system shall present a top-down, orthographic view of the scene. The projection parameters shall be configurable such that the virtual view can align 1:1 with the physical dimensions of the testbed floor.
- **FR-16: VR Interface:** The system shall provide a Virtual Reality mode in which the user can view the digital twin in 3D space. Information interfaces shall automatically orient themselves to stay readable from the user's perspective.
- **FR-17: Goal Setting:** The interface shall provide the user with the functionality to set navigation targets compatible with the Nav2 navigation stack by pointing at the virtual ground using the mouse or the VR controller.
- **FR-18: Visualization Control:** The user interface shall allow runtime modification of the visual environment by toggling the visibility or active state of various system components, such as the robot virtual body, lighting, or telemetry displays.

3.5 Robotic Application Requirements

These requirements are for the robotic applications developed to interact with the environment. These will help in demonstrating the capabilities of the system and testing the robot's response to dynamic scenarios.

- **FR-19: Sensor Dependence:** The robotic applications shall make decisions based only on the environmental data available via the simulated sensors. They shall not directly use any knowledge from the internal state of the simulation engine.
- **FR-20: Standardized Communication:** The robotic applications shall utilize standard robotic communication patterns to send action commands and monitor feedback topics to confirm task completion.

3.6 Non-Functional Requirements

The following constraints define the standards necessary to ensure reliability and extensibility.

- **NFR-01: Low Latency:** The system shall minimize the delay between the reception of a state update and the rendering of the corresponding visual frame. Latency shall remain lower than the simulation frame time to ensure immediate visual feedback.
- **NFR-02: Frame Rate:** The simulation shall maintain a constant target frame rate (at least 60 FPS) to avoid simulator sickness in VR [CKY20] and to ensure that AR projections remain visually stable and aligned with the physical environment.
- **NFR-03: Modularity:** The system architecture shall be modular, allowing individual components (e.g., sensor models, environment logic) to be added or modified without altering the core application.
- **NFR-04: Scalability:** The system shall support scenes with a high density of dynamic objects without violating frame rate stability requirements.

4 Implementation

This chapter describes the development of the Mixed Reality Environment, a platform designed to support RitL testing by combining simulation with AR projection and VR. The resulting system bridges the gap between purely digital simulations and real-world experiments by creating a common workspace where physical robots and virtual objects coexist and interact.

The central concept of this environment is the projection of a simulated world directly onto the laboratory floor. Rather than testing robots in completely physical setups or purely within software simulations, this system projects dynamic, physics-based elements into the real world. As can be seen from Figure **??**, the physical robot moves across the real floor while sensing and acting upon virtual objects. This projection has a dual purpose: it provides instant visual feedback to a human observer and enables the robot to receive sensor data that reflect the state of the virtual scene. This allows the robot to detect and respond to simulated entities as if they were physically present. [**ABBILDUNGFEHLT**]

Figure 4.1: The Mixed Reality Environment in operation. The physical robot interacts with a projected Smart Farming scenario, where the robot and the digital twin are synchronized via ROS 2.

To enable these applications, the system integrates several key functions into an architecture to realize the Mixed Reality Environment. First, it maintains a digital twin of the robot. This digital twin is synchronized with the movement of the physical hardware but can interact with the simulation's physics engine. This allows for complex actions, such as pushing virtual boxes or colliding with moving objects. This ensures that the simulation responds realistically to the robot's physical presence rather than serving as a static background.

In addition to maintaining a digital twin, the system also simulates sensors for the robot. It generates virtual LiDAR scans and camera images from the robot's perspective, allowing robotic applications to receive virtual sensor data. This enables thorough testing of navigation and vision algorithms within the mixed reality environment.

The architecture uses ROS 2 [MFG+22] as the primary communication backbone. By integrating ROS 2 nodes directly into the simulation engine, the virtual environment acts as a first-party participant in the robot's network. It manages time synchronization to prevent timing-related errors, handles the loading of different scenarios and provides two-way exchange of status data and control commands.

Besides sensor simulation, the environment also supports real-time diagnostic feedback. The system functions as a monitoring tool by projecting the robot's internal operating data back onto the physical workspace and the VR interface. Performance metrics, such as temperature, Random Access Memory (RAM) usage and location coordinates, are shown on information panels attached to the digital twin in the 3D environment. Moreover, the system shows the robot's live camera feed within the virtual scene. This allows operators to observe the robot's physical behavior alongside its internal status and visual perception in real-time.

The platform provides an adaptable and scalable testing solution by integrating physics, sensor simulation and autonomous control logic into a synchronized mixed reality environment. The only factor limiting the test environment's complexity is software, not its physical design. The technological decisions, system architecture and particular implementation of these elements will be covered in detail in the sections that follow.

4.1 Comparative Analysis

The simulation engines that were considered as candidates to implement the Mixed Reality Environment are: Gazebo [KH04], Isaac Sim [LMH+18], Unreal Engine [Epi19], and Unity [Uni23]. All were evaluated based on five critical criteria: visual fidelity, physics capabilities, learning curve, community support and native integration with VR, AR and ROS 2. Table 4.1 summarizes the results of this evaluation.

Table 4.1: Comparison of Simulation Engines for Mixed Reality Digital Twins [FCY25; KYH+24; SKG+25; CIR23].

Feature	Gazebo	Isaac Sim	Unreal Engine	Unity
Primary Use	Control & Navigation	AI & Photorealism	Photorealism	HRI & MR (VR/AR)
Visual Fidelity	Moderate	Very High	Very High	High
Physics Engine	ODE / Bullet	PhysX 5	Chaos / PhysX	PhysX
Learning Curve	Steep	Advanced	Steep (C++)	Moderate (C#)
Community	High (ROS)	Moderate	High (Gaming)	Very High (MR)
ROS 2 Integ.	Native	Bridge	Bridge	Plugin (Native)
Hardware	Low	Very High (RTX)	High	Moderate

4.1.1 Selection Rationale

Based on the comparative analysis, the Unity Engine was selected as the implementation platform for this thesis. This decision is driven by four key factors that align with the system requirements:

- **VR Framework** (FR-03): Unity has an integrated framework for VR applications [CIR23]. In contrast, simulators like Gazebo lack native VR support, which is critical for the proposed human-robot interaction interface.
- **Visual Fidelity & Physics** (FR-01, FR-02): Unity provides high-fidelity visualization alongside PhysX integration. This offers an optimal balance between performance and visual quality, avoiding the restrictive hardware requirements associated with NVIDIA Isaac Sim [FCY25].
- **Development Efficiency**: The use of C# scripting, combined with documentation and community support, makes Unity more accessible for rapid prototyping than the C++ environment of Unreal Engine [CIR23].
- **ROS 2 Integration** (FR-04): The availability of the `Ros2ForUnity` [Rob21] asset enables the simulation to function as a first-party participant in the ROS 2 network. This means it supports the low-latency communication requirement by avoiding external bridge applications.

4.2 System Architecture

The realization of the Mixed Reality Environment requires a distributed software architecture that connects the rendering capabilities of the Unity engine with real-time robotic control. The system is designed to satisfy the requirement for Middleware Integration (FR-04) by establishing a unified communication layer between the simulation host and the robotic applications using ROS 2.

4.2.1 Hardware and Software Topology

To ensure high performance and support for specific hardware peripherals, the system topology is consolidated into a single workstation acting as the central server, supported by distributed wireless clients. As shown in Figure 4.2, the architecture is divided into four distinct computational domains:

1. **Workstation (Windows 11 Host):** The primary host runs Windows 11 to support the Unity Engine and the ALVR (Air Light VR) Server [ALV25] natively. This layer manages the physics engine and renders the digital twin and its virtual environment. It physically interfaces with the visual peripherals: a ceiling-mounted Projector via HDMI for floor projection and an Overhead Tracking Camera via USB to provide localization data.
2. **Control Environment (WSL 2):** The computationally intensive robotic software runs on Ubuntu 24.04 within the Windows Subsystem for Linux (WSL) on the same workstation. This isolation allows the *Nav2 Stack*, *Robotic Applications Logic* and the *ArUco Tracking System* to run in a native Linux environment while communicating with the Windows host via a low-latency Localhost connection.
3. **Physical Robot (EMAROs):** The mobile robot[GTP25] operates as an independent node running Ubuntu 24.04 and ROS 2 Jazzy Jalisco. The onboard software is launched by the `emaros_launch` package, which manages hardware drivers for motor controllers and sensors. It connects to the system wirelessly via Wi-Fi.
4. **VR Interface (Meta Quest Pro):** The VR headset functions as a standalone client running Meta Horizon OS. It executes the ALVR Client [ALV25] application, which is responsible for decoding the incoming video stream and handling spatial input.

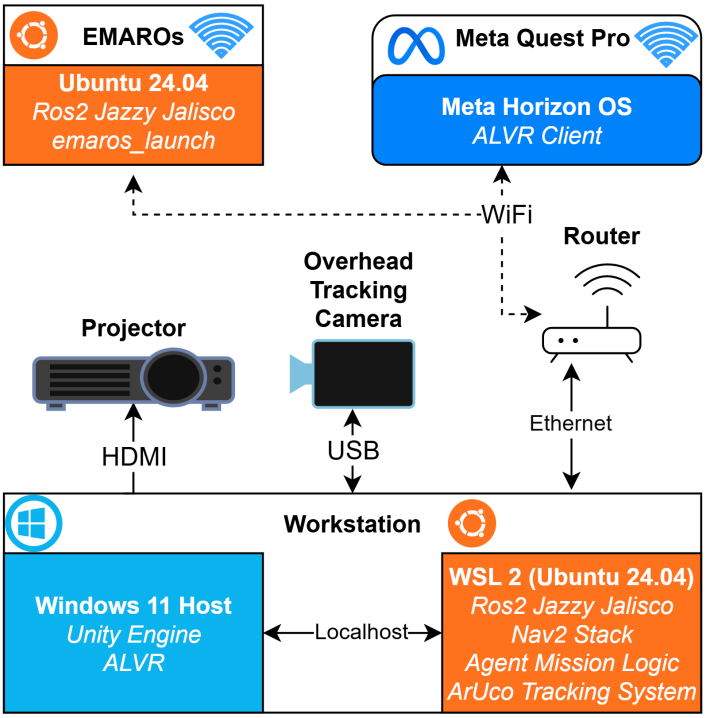


Figure 4.2: The Hardware and Software Topology of the Mixed Reality Environment. [**ABBILDUNG Andern**]

The VR integration (FR-16) relies on ALVR, an open-source remote rendering solution that splits the computational workload. The ALVR Server, running on the workstation, captures the rendered frames from Unity and encodes them into a high-bitrate video stream. This stream is transmitted via Wi-Fi to the ALVR Client on the Meta Quest Pro. The headset decodes and displays this video feed while simultaneously capturing head-tracking data and controller inputs, which are transmitted back to the server. [ALV25]

The network infrastructure is centered around a dedicated Router. The workstation is connected via Ethernet, while the EMAROs robot and Meta Quest Pro connect via Wi-Fi. This topology ensures that latency-sensitive traffic, such as the ALVR video stream and ROS 2 control commands, faces minimal interference.

4.2.2 Software Component Architecture

The software architecture follows a modular publisher-subscriber pattern. To integrate the Unity game engine with the robotic network, the system utilizes the `ROS2ForUnity` [Rob24] library (FR-04).

To facilitate modularity (NFR-03), the implementation relies on Unity's component-based architecture. In this framework, entities within the simulation, such as the robot or the environment, are represented as `GameObjects`. These objects function as containers that hold various functional modules known as *components*. By inheriting from Unity's base class `MonoBehaviour`, C# scripts gain access to the engine's event lifecycle. This allows them to be attached to `GameObjects` and execute logic in response to specific engine events, such as frame updates or physics steps.

The core of the network integration is the `ROS2UnityComponent` [Rob24] class. This component acts as a wrapper around the standard ROS 2 middleware. Inheriting from `MonoBehaviour`, it must be attached to a `GameObject` in the scene to function. Upon startup, this component initializes the ROS 2 context, ensures that a valid connection is established and handles the lifecycle of the ROS 2 nodes running within the simulation. It allows other scripts to access the ROS 2 network by referencing this central component to create publishers, subscribers and service clients.

The functionality of the digital twin is driven by such scripts that interact with the engine's internal state machine. To guarantee that the physical simulation remains deterministic while visualization remains smooth, the architecture uses specific phases of the Unity execution loop [Uni25]:

- **Initialization (`Start`):** This method is called once when the script is first enabled, before any frames are updated. In this architecture, it is used to retrieve the reference to the `ROS2UnityComponent` [Rob24], initialize the specific ROS 2 nodes and set up the necessary publishers and subscribers.
- **Physics Loop (`FixedUpdate`):** This method executes at a constant, user-defined time step, independent of the visual frame rate. All physics-based calculations, such as updating the robot's position based on received velocity commands or handling collision detection, occur in this loop. This ensures that the robot's physical behavior is consistent regardless of graphical performance.
- **Rendering Loop (`Update`):** This method runs once every frame. It is utilized for logic that is not physics-related, like updating UI elements, rendering diagnostic lines, or capturing camera frames for visualization. This separation ensures that high-frequency visual updates do not interfere with the stability of the physics engine.

Figure 4.3 provides an overview of how these Unity scripts interact with the external ROS 2 nodes via topics. Detailed descriptions of these components follow in Sections 4.3. and 4.4.

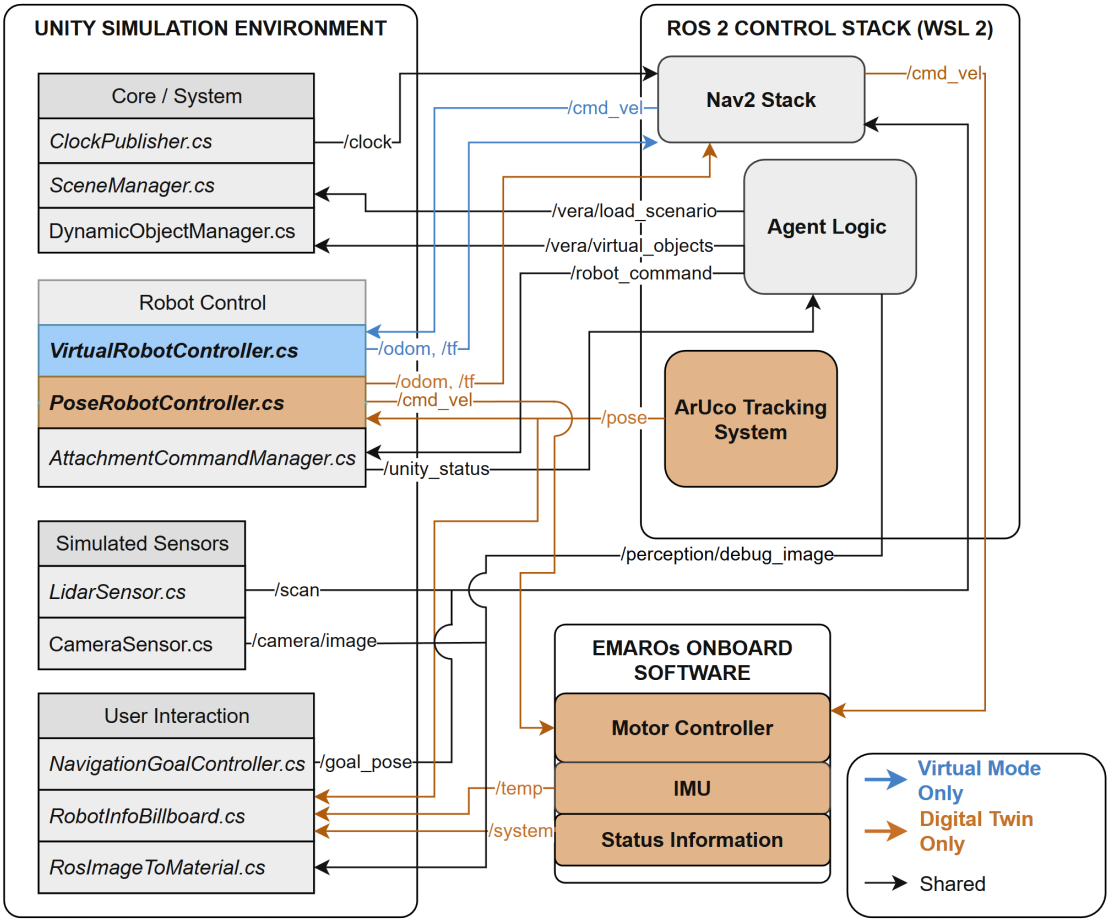


Figure 4.3: Software Component Architecture. Unity scripts act as ROS 2 nodes, publishing sensor data and subscribing to control commands using ROS 2 topics.

4.2.3 Time Synchronization

A critical challenge in Robot-in-the-Loop testing is how to synchronize time between the simulation and the robot's software. If the simulation runs slower or faster than real-time, the navigation algorithms of the robot may calculate velocities that are incorrect. To fulfill the requirement for Time Synchronization (FR-05), the system decouples the ROS network from the computer's system time.

The `ClockPublisher.cs` script in Unity acts as the master clock. It calculates the time since the start of the simulation and publishes it to the `/clock` topic. The ROS 2 nodes running on the workstation, such as the navigation stack, are configured

with `use_sim_time = true`, forcing them to synchronize their logic with the Unity engine.

To ensure consistency, a `RosTimeHelper.cs` utility is used by all ROS 2 publishers in the Unity scripts. This script ensures that data leaving Unity is stamped with the exact simulation time of the frame it was created in to prevent data drift.

4.3 Robot Representation and Control

The robot in the Mixed Reality Environment runs in one of two modes: either as a completely simulated virtual robot or as a synchronized digital twin of the physical EMAROs [GTP25] platform. To ensure consistency, both of these modes use the same visual and geometric model. This model is a simple cylinder that approximates the physical size of the robot for collision detection while keeping computational costs low. The behavior of this cylinder depends on the active control script, allowing the system to switch between physics-based simulation and position-based synchronization based on the requirements of the test. To select the desired mode, the user can enable in the Unity Editor either the model of the digital twin or the purely virtual robot.

4.3.1 The Purely Virtual Robot

In the standalone simulation mode, the robot is controlled by the `VirtualRobotController.cs` (FR-09). This component allows for the validation of high-level control logic in a safe environment before it is deployed on real hardware. Unlike simulations that teleport a robot to a target point, this implementation uses the Unity physics engine PhysX to simulate realistic movement.

The controller acts as a ROS 2 node that accepts velocity commands (`cmd_vel`) and publishes odometry data. To handle data exchange between the asynchronous ROS 2 thread and the Unity main thread, incoming commands are buffered in a thread-safe `ConcurrentQueue`. During the physics step of Unity (`FixedUpdate`), the controller retrieves the latest command and applies it to the `Rigidbody` of the robot.

The script does not directly change the position of the robot. Instead, it modifies the velocity of the physics body. By applying limits to acceleration, the system simulates mass and inertia for the robot. The robot can push lightweight virtual objects, but heavy obstacles or walls will stop its movement. This provides realistic feedback to the navigation system.

Simultaneously, the controller provides ground truth data. It tracks the movement of the robot in Unity, converts the coordinates to the ROS 2 standard [MFG+22] and publishes a timestamped odometry and tf messages (FR-10). This closes the control loop for the external navigation logic.

4.3.2 The Digital Twin Implementation

When operating in RitL mode, the `PoseRobotController.cs` takes control. In this configuration, the virtual robot stops acting as a dynamic entity and instead becomes a digital twin of the real EMAROs [GTP25] robot (FR-07).

The controller listens to a pose topic given by the external ArUco tracking [Spr25] to move the digital twin. The incoming pose is provided in pixel coordinates from the tracking camera. It must be mapped onto the defined virtual area. On every new incoming pose, the script converts the pixel coordinates into Unity world space. To ensure alignment, an offset parameter enables the operator to perform a calibration of the digital twin so that its center is aligned with the physical mounting position of the tracking marker. The script republishes the synchronized pose as odometry and tf messages (FR-10).

Unlike the purely virtual robot, the digital twin overrides the physics engine. It is set to a kinematic state, meaning it is immune to virtual forces. If the physical robot moves, the digital twin moves with it, ignoring virtual barriers. This allows the physical robot to push virtual objects with absolute force. This way, the state of the virtual world always reflects physical reality.

4.3.3 Safety and Failsafe Mechanisms

A major challenge in mixed reality testing is signal latency or loss. If the physical robot leaves the tracking area or if the ArUco marker is blocked, position updates will stop. Without a failsafe, the digital twin would freeze while the physical robot continues to move blindly, causing a mismatch between reality and simulation.

To prevent this, the `PoseRobotController.cs` implements a safety watchdog (FR-08). The system continuously checks how long it has been since receiving the last pose update. If this delay exceeds a given safety limit, the system assumes tracking has been lost.

When this occurs, the controller marks the tracking state as invalid. Although the digital twin will stop moving in the simulation, the physical Robot might continue

moving. For this reason, the system is designed to send a stop command over the `cmd_vel` topic to the motors of the physical robot. This ensures the robot does not perform actions without simulation guidance. Once the tracking system re-acquires the robot, the watchdog resets and the digital twin resumes synchronization immediately.

4.4 Environmental Simulation Capabilities

To serve as an effective testbed for robotic applications, the virtual environment provides a dynamic and interactive simulation context. It incorporates the capability to manipulate objects physically, generate realistic sensor data and modify the state of the environment persistently. This section describes the implementation of these features, fulfilling the requirements for Sensor Simulation (FR-11), Dynamic Environment Interaction (FR-13) and the Command Interface (FR-12).

4.4.1 Physics-Based Interaction

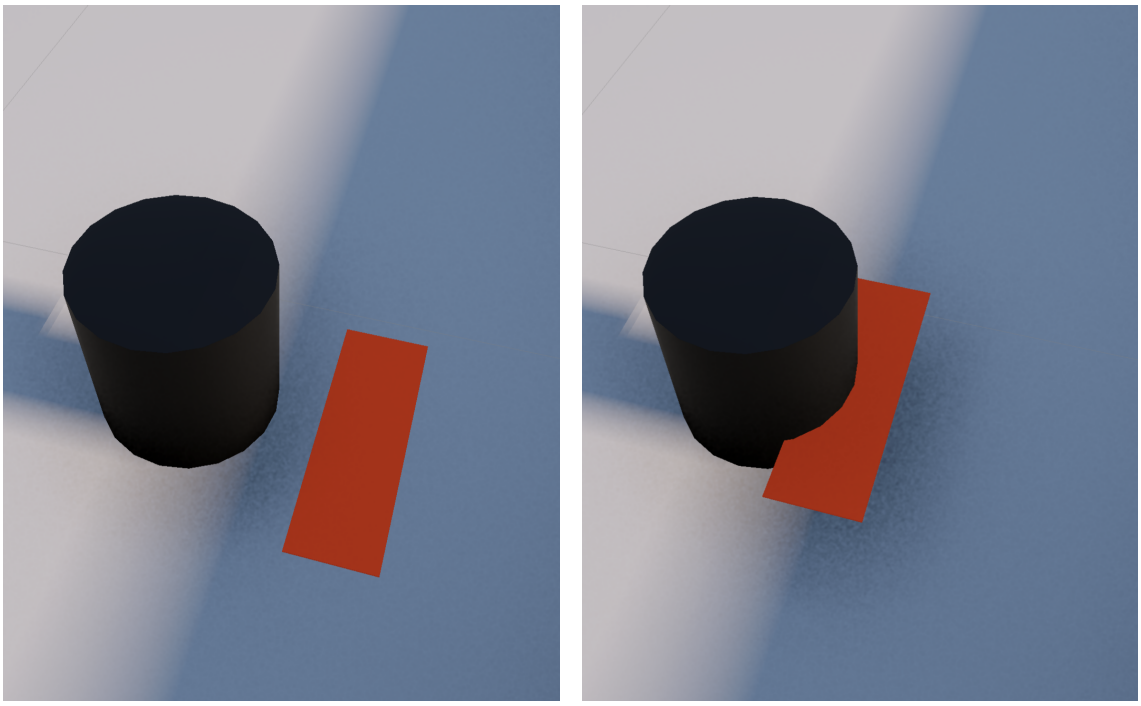
A key feature of the system is the robot's ability to physically connect with environmental objects. To achieve a rigid and predictable connection between the robot and manipulated items, the architecture uses a kinematic attachment method managed by the `AttachmentCommandManager.cs` and the `RobotAttachmentController.cs`.

ROS 2 command and control (FR-12) is provided via the `/robot_command` topic. Commands are sent as simple string messages (e.g., `"attach,plow"` or `"detach"`). It offers a single, consistent interface for the high-level robotic applications to request simulation state changes. The Command Manager runs on the Unity main thread and processes incoming entries from a thread-safe queue. On receiving an attach request, it searches for valid targets within a configurable radius using `Physics.OverlapSphere`.

When a valid target is identified, the system attaches it to the robot, as illustrated in Figure 4.4. In Unity, the object is attached as a child of the robot's transform hierarchy at a specific mount point. Crucially, the system modifies the physics state of the attached object (FR-13). Upon attachment, the script sets the object's `Rigidbody.isKinematic` property to true. This removes the object from the physics engine's dynamic calculations. This state change grants the robot's transform absolute authority over the object's position, eliminating any relative motion, jitter, or drift between the robot and the load during transport.

At the same time, the controller disables collisions between the robot and the tool using `Physics.IgnoreCollision`. Without this step, snapping the object to the

robot would cause their geometries to overlap. The physics engine would attempt to separate them forcibly, potentially causing the equipment to be ejected unpredictably.



(a) Detached State (b) Attached State

Figure 4.4: The robot attachment logic. (a) The robot approaches the target object. (b) The object is kinematically coupled to the robot chassis.

4.4.2 Surface Modification

To fulfill the requirement for dynamic environment interaction regarding surface textures (FR-13), the system implements a modification mechanism.

The modification logic relies on physics raycasting to identify the coordinates of the surface at the point of contact. When a valid hit is detected on the target surface, the `TrackPainter.cs`, which is attached to the robot model, calculates the pixel coordinates on the texture map. It then modifies the pixel buffer directly using `SetPixels32`, changing the color to a configured color and thickness. These changes are committed to the GPU using the `Apply` method. In complex scenarios, multiple robotic applications (e.g., the robot, a VR controller, or a mouse interface) may attempt to modify the terrain simultaneously. To manage this, the `SharedPaintTextureRegistry.cs` functions as a singleton manager. It maintains a dictionary of active textures and guarantees

that all painters operate on a single, shared instance of the `Texture2D`. This prevents race conditions where one application or user overwrites the work of another and guarantees data consistency across the simulation.

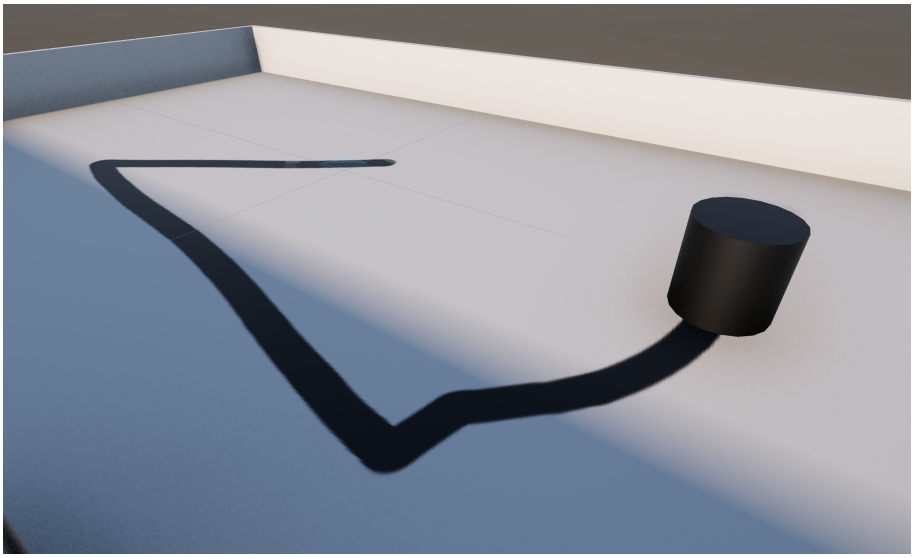


Figure 4.5: Demonstration of dynamic surface modification. The robot uses the `TrackPainter.cs` component to modify the floor texture in real-time based on its trajectory, creating a persistent black trail.

4.4.3 Sensor Simulation

To validate the perception algorithms of the robotic applications, the system provides simulated sensors capable of generating video feed and ranging data (FR-11).

The `LidarSensor.cs` component simulates a 2D planar laser scanner, which can be attached to the virtual model of the robot. This implementation utilizes the Unity physics engine's raycasting API to query the scene geometry. In every simulation step, the component executes a loop corresponding to the configured number of beams. For each beam, the system calculates a direction vector based on the scan angle.

For each beam a `Physics.Raycast` is cast along the computed direction. If the ray intersects a collider on the configured collision layer, its hit distance is recorded. Otherwise, the beam is reported as an infinite range. The collected distances are packaged into a standard `sensor_msgs/LaserScan` message. Immediately after the raycast loop finishes, the message is stamped with the current simulation time from the `ClockPublisher.cs` and published. This exact timestamping preserves

temporal alignment with the navigation stack and prevents the ROS 2 transform (TF) system from rejecting the scan due to extrapolation errors caused by unsynchronized timestamps.

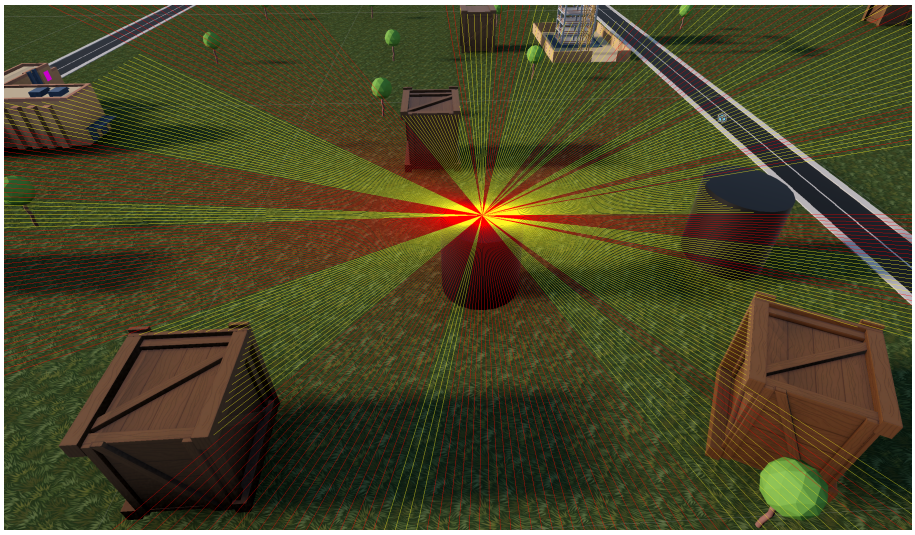


Figure 4.6: Visualization of the LiDAR simulation in the editor. Red debug lines indicate raycasts that did not hit an obstacle within range, while yellow debug lines indicate valid hits registered by the physics engine.

Visual perception is handled by the `CameraSensor.cs` component. This script attaches a dedicated Unity Camera to the robot model, which is configured to render the scene into a `RenderTexture` rather than the user's screen. This allows the simulation to generate visual data at a resolution independent of the application window.

The data extraction pipeline transfers the raw pixel data from the Graphics Processing Unit (GPU) memory to the Central Processing Unit (CPU) memory. Once on the CPU, the data is encoded into the appropriate format (e.g., raw RGB8 or JPEG) and published as a `sensor_msgs/Image` or `CompressedImage`. To initialize the sensor, the `RobotCameraManager` instantiates the camera prefab at runtime and attaches it to the defined mount point in the robot hierarchy.

4.4.4 Scenario and Object Management

To support the requirement for **Scenario Management** (FR-06), the architecture allows the system to switch between different test environments without restarting the entire application.

This is handled by the `SceneManager.cs` script. When it receives a command via the `/vera/load_scenario` topic, it performs a clean reset sequence:

1. **Shutdown:** The active ROS 2 connections are terminated. This ensures that all Unity nodes are properly removed from the network, preventing any lingering nodes from publishing outdated or invalid data.
2. **Load:** Unity loads the new environment scene asynchronously.
3. **Restart:** A fresh `ROS2UnityComponent` [Rob24] initializes, creating a new connection for the loaded scenario.

This fresh start approach ensures that Unity components specific to one scenario do not interfere with others.

Beyond static geometry, the environment must support the lifecycle management of transient entities, such as the delivery boxes in the Logistics scenario. This is handled by the `DynamicObjectManager.cs`, which acts as a bridge between the ROS 2 data stream and the Unity instantiation engine (FR-13).

The manager subscribes to the `/vera/virtual_objects` topic, processing a custom message structure that defines an object's ID, type and pose. Internally, the system maintains a dictionary mapping string identifiers to Unity Prefabs. When an `ADD` or `MODIFY` command is received, the system checks if the object already exists. If it is new, the corresponding prefab is instantiated into the scene. If it exists, its transform is updated. This component handles the coordinate conversion between the two systems, mapping the right-handed ROS 2 position and orientation to the left-handed Unity world space. This allows external scripts or test runners to populate the scene with obstacles dynamically.

4.5 Mixed Reality and User Interaction

A primary objective of the framework is to enhance the transparency of the robotic system by providing intuitive, real-time feedback to human operators. This is achieved through an MR interface that visualizes internal robot states and allows for direct environmental manipulation. This section details the implementation of augmented telemetry, sensor projection and interactive control systems designed for both desktop and VR contexts.

4.5.1 Environment Projection

To fulfill the requirement for a 1:1 Augmented Reality projection (FR-15), the system utilizes a dedicated Unity camera component configured to `orthographic` [Uni25]. Unlike perspective cameras, the orthographic view preserves parallel lines and consistent object scales regardless of their distance from the camera [Uni25]. This property is essential for projecting a map that aligns physically with the flat laboratory floor.

The camera is positioned at a fixed height looking downward. The alignment with the physical floor is achieved by calculating the exact `orthographicSize` parameter, which determines the vertical half-extent of the camera's viewing volume in world units [Uni25]. By mapping the pixel resolution of the projector to the metric dimensions of the projection area, as illustrated in Figure 4.7, the system ensures that one unit in the simulation corresponds exactly to one meter on the physical floor.

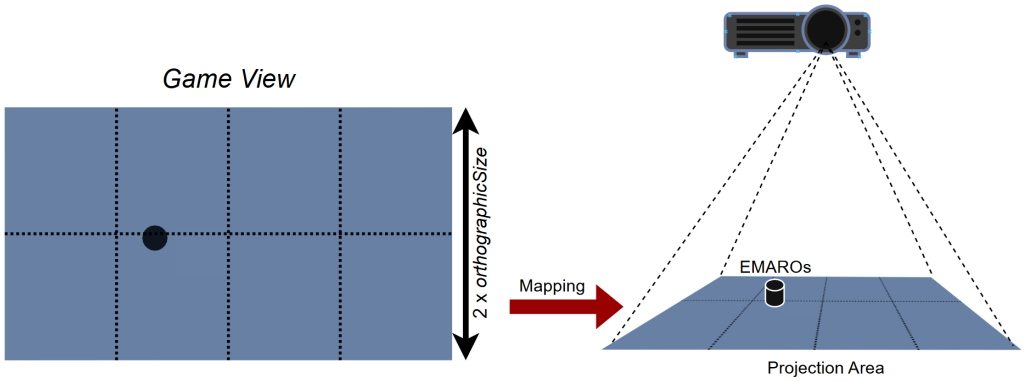


Figure 4.7: Schematic of the Environment Projection logic. The Unity Orthographic Camera (left) is calibrated such that its `orthographicSize` corresponds exactly to half the physical height of the projection area on the laboratory floor (right), ensuring a 1:1 metric scale.

To display this view on the physical floor, the ceiling-mounted projector acts as a secondary monitor for the workstation. During operation, the *Game View* window within the Unity Editor is undocked, moved to the projector's screen space and maximized to fill the projection area.

4.5.2 Augmented Telemetry and Diagnosis

To fulfill the Telemetry Display requirement (FR-14), the system implements the `RobotInfoBillboard.cs` component. This script serves as a central data aggre-

gator, establishing ROS 2 subscriptions based on a configurable list of topic definitions. It processes diverse message types and utilizes regular expressions to extract specific metrics, such as CPU and RAM usage, from text payloads.

For visualization in standard desktop views and AR projections, the system employs Unity's immediate mode GUI (`OnGUI`). The script calculates the screen-space coordinates of the robot's anchor using `Camera.WorldToScreenPoint` and applies rotational matrices to align the text. This ensures that the interface remains legible and strictly aligned with the screen plane, creating a floating overlay locked to the robot's position, as illustrated in Figure 4.8.

In contrast, screen-space rendering is unsuitable for Virtual Reality due to depth perception issues. To address this, the architecture supports a World Space Canvas for VR users. In this configuration, textual information is rendered onto a 3D plane floating physically above the digital twin. To ensure readability from any perspective, the canvas implements a constraint that continuously orients the UI to face the VR headset, allowing operators to inspect the robot's status naturally within the virtual environment.

4.5.3 Sensor Data Projection

To integrate the robot's visual perception directly into the mixed reality workspace, the system utilizes the `RosImageToMaterial.cs` component to project sensor streams into the environment.

This component subscribes to ROS 2 camera topics. Upon receiving a frame, it decodes the data into a Unity `Texture2D` and applies it to a material on a planar mesh attached to the model of the robot.

For the AR setup, this plane is positioned horizontally above the robot, projecting the camera stream onto the floor moving along with the robot. Similarly, in VR, this projection functions as a virtual dashboard, allowing the user to see the camera stream from a top-down perspective.

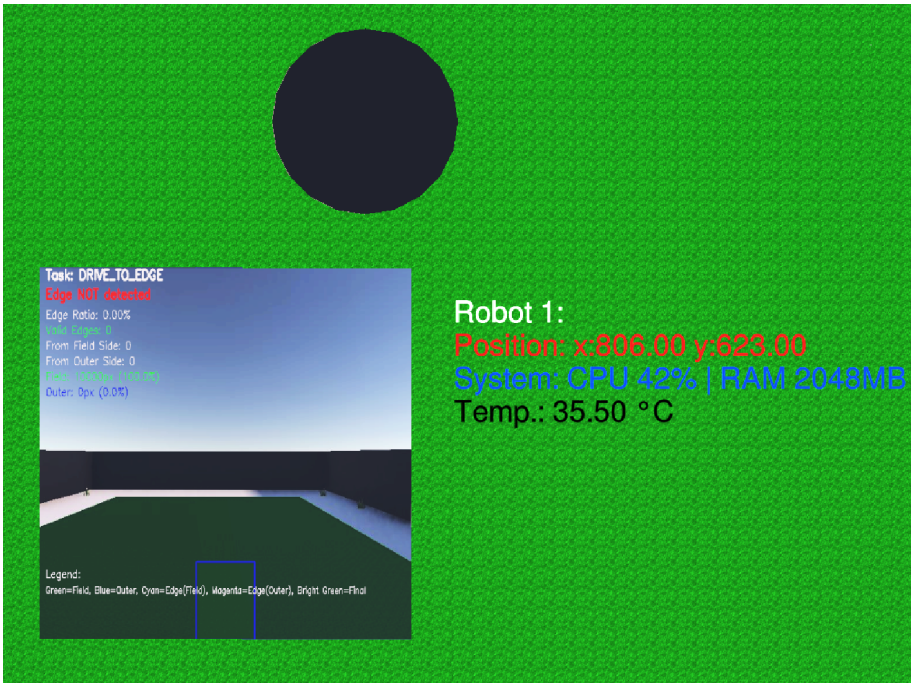


Figure 4.8: The Augmented Telemetry interface. A billboard displays system stats, while the `RosImageToMaterial.cs` component projects the live OpenCV debug feed onto a plane attached to the robot, visualizing the internal state of the perception stack.

4.5.4 Interactive Control Interfaces

The implementation utilizes a device-independent architecture, where the core logic relies on 3D raycasting rather than specific hardware events. This design allows the system to support both standard mouse inputs and 3D VR controllers interchangeably.

The `NavigationGoalController.cs` facilitates high-level control of the robot. In the desktop configuration, it casts a ray from the camera through the mouse cursor coordinates. When the ray intersects with the ground layer, the script calculates the target point and orientation based on the user's drag gesture. This data is serialized into a `geometry_msgs/PoseStamped` message and published to the `/goal_pose` topic, triggering the ROS 2 navigation stack. For Virtual Reality, this logic utilizes the `XR Ray Interactor`, enabling the operator to point at the virtual floor and dispatch the robot using the controller's trigger.

To support dynamic scenarios like Line Following, the user must be able to alter the environment at runtime. The `MouseSurfacePainter.cs` component enables users

to draw directly onto the terrain texture using a continuous raycast using a mouse or VR controller. When the user holds down the primary input, the script casts a ray from the camera or controller into the scene. If it hits the ground layer, it calculates the texture coordinates at the point of intersection and modifies the pixel buffer of the terrain texture, similar to the robot's `TrackPainter.cs`.

Beyond painting, the VR interface leverages the physics engine to allow direct object manipulation. Users can grab and move interactive objects using the VR controller's grip function. This allows for the manual reset of test scenarios or the introduction of dynamic obstacles (e.g., dropping a box in the robot's path) to validate the system's reactive planning capabilities.

In complex mixed reality scenarios, the density of visual information can become overwhelming. To manage this, the `VisibilityToggleManager.cs` (FR-18) allows users to selectively hide or reveal specific visualization aids attached to the robot. This component maps input actions to the rendering state of objects such as the `RobotInfoBillboard.cs`, the projected camera plane, light source or the robot's visual cylinder body. This feature allows operators to declutter the view when necessary.

4.6 Implementation of Robotic Applications

To validate the capabilities of the Mixed Reality Environment, the system hosts three distinct robotic applications. These applications utilize sensor data provided by the virtual environment to make decisions (FR-19), serving as practical demonstrations of the architecture's flexibility and reliability. They are presented below in order of increasing complexity:

- A **Line Following Application** that uses visual tracking to follow paths drawn by a human user in real-time.
- A **Logistics Application** that autonomously searches for, identifies and sorts colored objects within the environment and brings them to designated zones.
- A **Smart Farming Application** that autonomously explores a virtual field, identifies its boundaries and operates simulated tools to perform fieldwork tasks.

The high-level decision-making for all applications is managed using YASMIN (Yet Another State MachINe) [GRM+22], a ROS 2-native library that facilitates the creation of hierarchical, interruptible mission behaviors. This library organizes the logic of the robot into a graph of distinct states. This improves the readability of the control flow

and simplifies the debugging process, as the active state of the robot can be monitored and visualized in real-time using the YASMIN Viewer [GRM+22]. This structure ensures that the system can maintain control loops while transitioning between operational modes.

4.6.1 Navigation and Control Stack

The fundamental capability of movement is provided by the Nav2 stack [MMW+20b] (FR-17). To address the varying physical characteristics of the simulated versus the physical robot, the architecture employs a dual-configuration strategy. Two distinct parameter sets, `nav2_params_virtual.yaml` and `nav2_params_robot.yaml`, are maintained. While they share the same behavioral tree structure, they utilize different tuning for inflation radii, cost scaling and movement parameters to account for the physical robot's safety margins and movement characteristics. The appropriate parameter set is selected automatically at launch time based on the active system mode.

A critical design choice in the controller configuration is the use of the `RotationShimController` [MMW+20b] to wrap the `DWBLocalPlanner`. The underlying planner implements the Dynamic Window Approach (DWA) [FBT97]. This method discretizes the robot's control space into pairs of translational and rotational velocities (v, ω) and simulates the resulting trajectories [FBT97]. To ensure smooth motion, the DWA algorithm approximates these trajectories as circular arcs [FBT97]. It selects the optimal command by maximizing an objective function that balances progress toward the target, forward velocity and obstacle clearance [FBT97].

However, this reliance on arcs creates a limitation in the confined projection environment. The turning radius required to execute these continuous arcs often causes the robot's footprint to sweep into obstacles or cross the boundaries of the projection area.

To resolve this, the `RotationShimController` intercepts navigation requests before they reach the DWB planner [MMW+20b]. It continuously checks the angular deviation between the robot's heading and the path. If this deviation exceeds a threshold, the shim overrides the local planner and forces an in-place rotation [MMW+20b]. This enforced rotation before driving ensures the robot is safely aligned with its path before attempting forward motion, avoiding the collision risks associated with the arcs of the standard DWA.

To guarantee that the robot does not collide with virtual objects during autonomous operation, a *Collision Monitor* node runs in parallel with the navigation stack. Configured with a defined polygon representing the robot's footprint plus a safety margin, this node

monitors the virtual LiDAR stream of the robot directly. If an obstacle breaches the safety polygon of the robot, the monitor overrides the navigation stack and publishes a zero-velocity command to the motor controller, acting as a software-level emergency stop.

4.6.2 Modular Perception Stack

To interpret the visual data generated by the `CameraSensor` (FR-19), the robotic applications employ computer vision pipelines tailored to their operational requirements.

The Line Following application utilizes a standalone `line_perception_node.py` optimized for low latency. For the more complex Logistics and Smart Farming applications, a modular `perception_node.py` was developed. Rather than running all detection algorithms simultaneously, which would consume excessive computational resources, this node acts as a central dispatcher. It switches between specialized processors based on the current mission state defined by the state machine.

All processors inherit from a base class `base_processor.py` that standardizes the handling of `sensor_msgs/CompressedImage` messages and their conversion to OpenCV format [Bra00].

4.6.3 Application 1: Line Follower

[**Fuer jede Phase ein debug bild einbauen**] The Line Follower application operates in an interactive playground where a user, using the robot, a mouse, or VR controllers, can draw paths on the virtual ground in real-time. The robot must identify and track this line immediately.

This implementation utilizes the dedicated node `line_perception_node.py`:

- **Region of Interest (ROI) Cropping:** The image is cropped to the bottom 50%, focusing solely on the immediate path in front of the robot. This removes background noise and simplifies the processing required.
- **Line Fitting:** The processor applies a color threshold to isolate the high-contrast line drawn by the user. It then utilizes `cv2.fitLine` [Bra00] to fit a vector through the detected pixels to determine the path's heading.
- **Error Calculation:** From this vector, the system calculates two control metrics: the *lateral error* (horizontal distance of the line centroid from the image center) and the *angular error* (deviation of the line's heading from the vertical axis).

The debug view (Figure 4.9) visualizes these metrics in real-time. It overlays the detected line contour (green) and the calculated centroid (red dot) onto the camera feed, allowing the operator to verify the error terms driving the control loop.

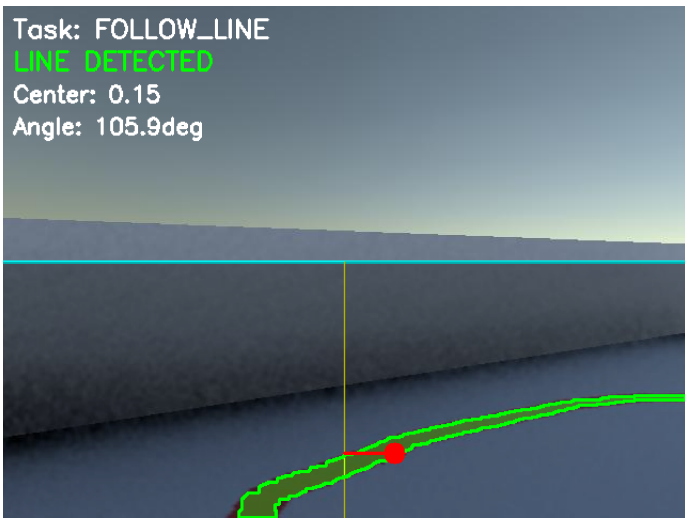


Figure 4.9: Debug view of the Line Follower. The system isolates the user-drawn path and fits a vector (blue line) to calculate lateral and angular errors for the PID controller.

The mission logic is defined in `mission_state_machine.py`. This application utilizes a hybrid control approach. It utilizes the Nav2 stack [MMW+20b] for global exploration but switches to a PID control loop for the primary task to ensure low-latency reactivity. The sequence proceeds as follows:

1. **FindLine:** The robot executes a spiral search pattern using the `find_line_action_server` and Nav2 stack to locate the drawn path.
2. **AlignToLine:** Once the line is detected, the robot transitions to the `AlignToLine` action. The robot rotates in place until the angular error returned by the perception node is minimized, ensuring the robot is parallel to the path before moving.
3. **FollowLine:** The robot engages a PID controller implemented in `follow_line_action_server.py`. This controller calculates velocity commands (`cmd_vel`) directly from the perception error values. If the line is erased or ends, the state machine transitions back to `FindLine` to search for a new path.

4.6.4 Application 2: Logistics

The Logistics application operates in a warehouse environment populated with colored transport boxes and corresponding projected delivery zones. The robot's objective is to autonomously search for items, physically attach them, identify the correct sorting destination and transport the payload.

The application employs the modular `perception_node.py`, switching between two specialized processors based on the mission phase:

1. **Box Detection (`BoxDetectionProcessor`):** This processor isolates colored objects from the background. It applies an HSV color mask to filter target colors and utilizes `cv2.findContours` [Bra00] to identify object blobs, filtering them by a minimum area threshold to reject sensor noise. The centroid of the largest valid contour is calculated using `cv2.moments` [Bra00] to provide a steering target.

2. **Zone Detection (`ZoneDetectionProcessor`):** This processor is tuned to detect the flat, projected delivery zones on the floor. While similar to box detection, it uses distinct HSV ranges and calculates the zone's center.

To assist in debugging, the perception node publishes an annotated video stream (Figure 4.10). In this example, the system draws bounding contours around valid targets and overlays classification labels (e.g., `RED BOX`) and the active search status directly onto the feed.

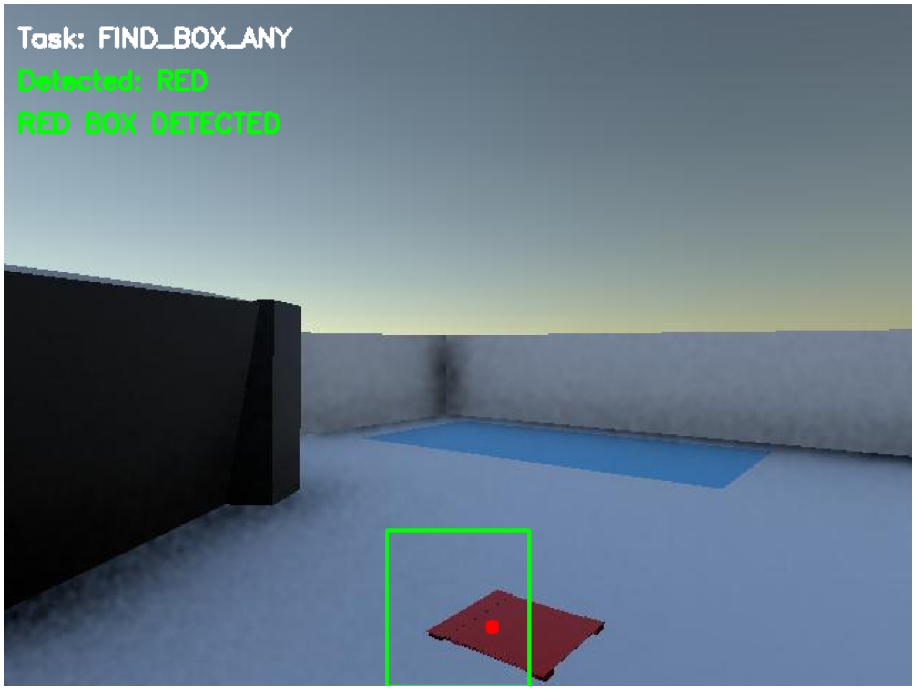


Figure 4.10: Debug view of the Logistics application. The processor identifies a red transport box in its ROI (green) and calculates the centroid (red dot).[**FALSCHEBOXBILD**]

The mission logic is defined in `mission_state_machine.py` as a YASMIN state machine [GRM+22] and implements a search, retrieve and deliver loop. This application uses the YASMIN blackboard [GRM+22] to implement spatial memory to learn the positions of the delivery zones.

1. **FindBox:** The robot executes the `find_box_action_server`. This generates a spiral pattern of waypoints using the Nav2 stack [MMW+20b] to efficiently cover the floor plan. The robot navigates these waypoints until the `BoxDetectionProcessor` identifies a valid target in the ROI.
2. **AttachBox:** Upon reaching the target, the robot performs a predefined approach. The `AttachBox` action sends a string command via the `/robot_command` topic to the Unity `AttachmentCommandManager.cs`, which locks the virtual box to the robot's chassis for transport.
3. **Spatial Memory Query:** Before searching for the destination, the application queries the blackboard. If the location of the matching colored delivery zone was discovered and cached during a previous traversal or was given by the user, the search phase is skipped.

4. **FindDeliveryZone / NavigateToZone:** If the location is unknown, the robot triggers `FindDeliveryZone` to execute a new spiral search. If the location is known, it uses `NavigateToZone` to drive directly to the stored coordinates.
5. **VerifyZoneColor (Recovery):** Upon arriving at the target, the robot executes the `VerifyZoneColor` action. This serves as a reliability check. If the camera view is obstructed (e.g., by a wall) or odometry drift has occurred, the robot performs a spin recovery maneuver. It rotates 90 degrees left and right to re-acquire the visual target before confirming the drop.
6. **DropBox:** Finally, the robot drives forward into the zone and triggers the detach command via the `/robot_command`, releasing the object into the sorting area.

4.6.5 Application 3: Smart Farming

This application represents the most complex use case, operating in a simulated farm with an agricultural field. The goal is to identify the field boundaries and position and sequentially apply three tools: a plow, seeder and harvester to the field.

The application utilizes four distinct perception processors depending on the active state:

1. **FieldDetection (`FieldDetectionProcessor`):** Determines if the robot is currently on arable land by calculating the ratio of field-colored pixels in the image using Hue Saturation Value thresholding.
2. **EdgeDetection (`EdgeDetectionProcessor`):** Used during the approach phase. It creates binary masks for both the field and its surrounding outer textures. By applying morphological dilation [Bra00] to the field mask and computing the bitwise AND with the outer mask, it identifies the boundary line where the two textures meet.
3. **EdgeAdjustment & EdgeFollowing:** Implemented via two processors (`EdgeAdjustmentProcessor` and `EdgeFollowingProcessor`), these modules are used for precise alignment and tracking. They utilize the Probabilistic Hough Transform (`cv2.HoughLinesP`) [Bra00] to find line segments along the detected boundary, calculating the robot's *lateral error* (distance from the line centroid to image center) and *angular error* (deviation from vertical).
4. **EquipmentDetection (`EquipmentDetectionProcessor`):** Active during tool search. It isolates specific tool colors using HSV masking and calculates the object's centroid to guide the docking approach.

A debug stream visualizes these detections on the `/perception/debug_image` topic. For example in figure 4.11, it overlays the ROI and the regression line (green) onto the camera feed to visualize the tracking performance.

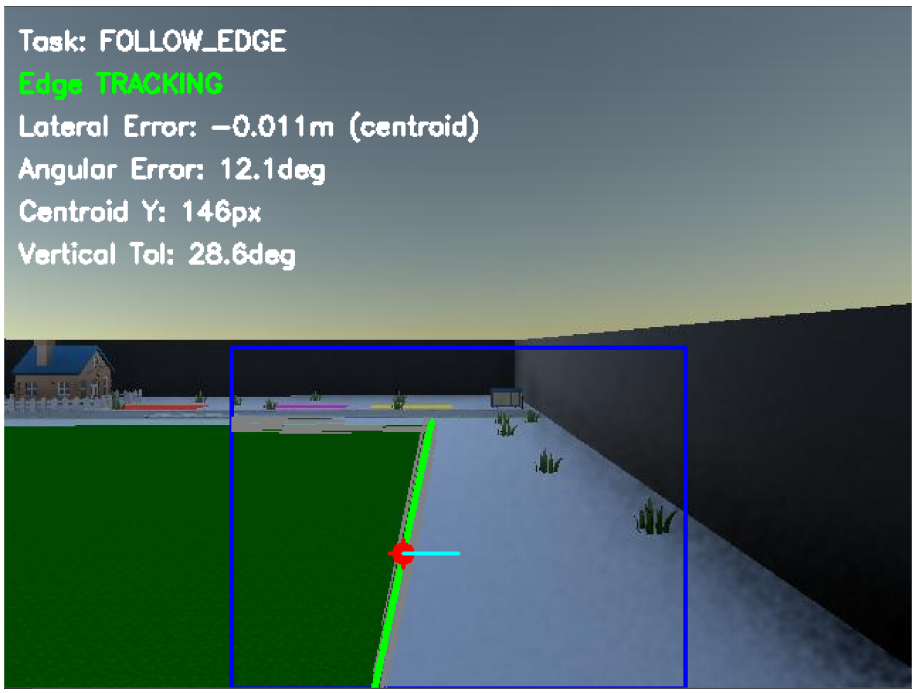


Figure 4.11: Debug view of the Smart Farming application during edge following. The processor identifies the boundary between the field (left) and the path (right). The green line represents the detected edge used for navigation, while the red dot indicates the centroid used to compute the lateral and angular errors displayed in the overlay.

The mission is governed by a YASMIN state machine [GRM+22] which executes two distinct phases:

Phase 1: Field Mapping

1. **CheckOnField:** The robot searches the field using a spiral search pattern using the Nav2 stack [MMW+20b] until it verifies it is inside the field boundaries.
2. **DriveToEdge:** The robot drives forward until the `EdgeDetectionProcessor` identifies a field boundary. It then executes a forward drive for a calibrated distance to account for the camera's blind spot, ensuring the robot's center of rotation is aligned with the edge.

3. **AdjustToEdge:** The robot rotates in place until the `EdgeAdjustmentProcessor` confirms that the edge line is vertical (angular error near zero) and centered.
4. **FollowEdge:** The robot circumnavigates the field by tracing its boundary. A Proportional Integral Derivative (PID) controller using the lateral error computes the required velocity commands and steers the robot. This action detects field corners by monitoring the robot's odometry. A sharp change in yaw (approx. 90 degrees) triggers the recording of the robot's current pose. If this pose is too close to a previously recorded corner, it is discarded. Once four corners are successfully recorded, the field boundary is defined.

Phase 2: Equipment Cycle

1. **FindEquipment:** The robot executes a spiral search pattern using the Nav2 stack [MMW+20b] to locate a specific tool.
2. **EquipTool:** Upon detection, the robot approaches the tool. The action triggers the Unity `AttachmentCommandManager.cs` via the `/robot_command` topic to kinematically lock the tool to the robot.
3. **GenerateCoveragePath:** Using the four corners identified in Phase 1, this action calculates a path that covers the entire field area.
4. **ExecuteCoverage:** The robot drives the generated path. During this phase, the tool equipped by the robot modifies the ground texture using raycasts.
5. **Return & Unequip:** The robot returns the tool to its origin and detaches it using the `/robot_command`. The state machine then loops back to search for the next tool in the sequence until all tools are used.

4.7 Implementation Synthesis

The development process described in this chapter has established the technical architecture of the Mixed Reality Environment. The system integrates the rendering and physics capabilities of the Unity engine with the decentralized communication infrastructure of ROS 2. This integration creates a bidirectional data pipeline: the physical robot transmits its state to the digital twin, while the simulation engine generates sensor data and physical feedback.

The architecture comprises several functional layers. The core layer handles the synchronization of the digital twin and the generation of virtual LiDAR and camera streams. Above this, the environmental layer manages dynamic elements, including

manipulable objects and modifiable surface textures. These automated features are complemented by the mixed reality interfaces, which provide visualization via floor projection and Virtual Reality control.

Three specific robotic applications were implemented: Line Following, Logistics, and Smart Farming. These applications serve as consumers of the data and interaction mechanisms provided by the environment. [**NOT FINAL**]With the software architecture defined and the functional modules integrated, the system is prepared for quantitative analysis. The following chapter addresses the measurement of technical performance metrics, including system latency, synchronization accuracy and resource utilization, to determine the operational limits of the platform.

5 Evaluation

This chapter evaluates the technical performance, scalability, and efficiency of the developed Mixed Reality Environment. The assessment focuses on how the system handles increasing complexity and compares the results to the previous VERA framework developed by Gehricke [Geh24].

5.1 Hardware and System Specifications

The performance of a real-time Mixed Reality simulation is heavily dependent on the underlying hardware, as it must simultaneously handle physics calculations, ROS 2 communication, and high-frequency stereoscopic rendering. The benchmarks for this thesis were conducted on a desktop workstation, while the reference measurements for the original VERA framework by Gehricke [Geh24] were performed on a high-performance laptop. The respective hardware configurations are summarized in Table 5.1.

Table 5.1: Comparison of hardware configurations used for evaluation.

Component	**Proposed System (Host)**	**VERA (Gehricke [Geh24])**
Processor (CPU)	Intel i7-8700K (up to 4.7 GHz)	Intel i9-14900HX
Graphics (GPU)	NVIDIA RTX 2070 Super (8 GB)	N/A (CPU-based rendering)
Memory (RAM)	16 GB DDR4	32 GB RAM
OS	Windows 11 / WSL 2	Ubuntu 24.04 (Native)

A critical distinction must be made regarding the rendering architecture. Gehricke's system relied entirely on the CPU for its 2D Pygame-based visualization, which introduces processing contention between logic and drawing. In contrast, the system presented here leverages a dedicated GPU for hardware-accelerated rendering and physics. This offloading is essential not only for the stereoscopic demands of the Virtual Reality interface but also for maintaining high frame rates in complex scenes.

5.2 Component Breakdown Analysis

To identify the computational cost of the system's core functions, a granular profiling analysis was conducted using a custom `PerformanceBreakdownLogger` script. This tool utilizes the Unity *ProfilerRecorder* API to capture the Main Thread execution time of three key subsystems: **Physics** (`Physics.Simulate`), **Rendering** (`Camera.Render`), and **Scripts** (`Update.ScriptRunBehaviour`). Data was recorded for 60 seconds across four test scenarios in both Digital Twin (DT) mode and Virtual Mode.

Figure 5.1 visualizes the distribution of CPU time, while Table 5.2 details the specific execution metrics and memory usage.

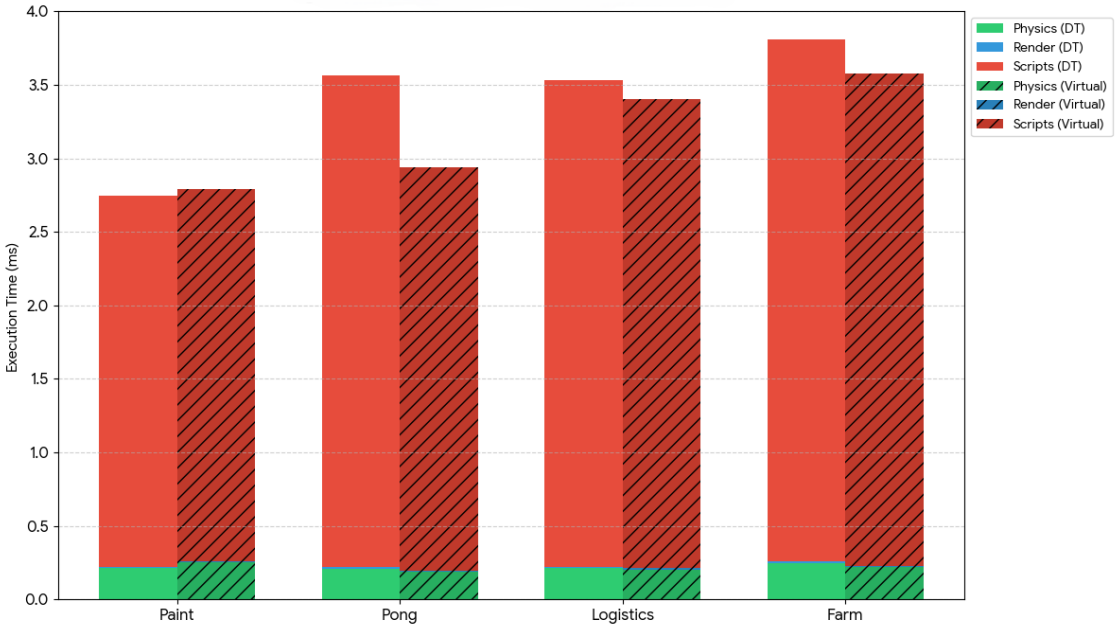


Figure 5.1: Component Breakdown Analysis illustrating the CPU execution time for Physics, Rendering, and Scripts across four scenarios in Digital Twin (DT) and Virtual modes.

The data reveals that **Scripts** account for the vast majority of the frame time, ranging from 2.5 ms in the *Paint* scenario to over 3.5 ms in the *Farm* scenario. This overhead is primarily driven by the C# logic required for ROS 2 serialization, message handling, and state management. Interestingly, in complex scenarios like *Pong* and *Farm*, the Digital Twin mode exhibits slightly higher script execution times (e.g., 3.55 ms vs. 3.35 ms in Farm) compared to Virtual Mode. This indicates that the overhead of

synchronizing the digital twin with external telemetry and processing coordinate transforms is computationally comparable to, or slightly more demanding than, running the internal simulation logic for the virtual robot.

Table 5.2: Average execution time and memory usage over a 60-second capture period.

Scenario	Mode	Physics (ms)	Render (ms)	Scripts (ms)	Memory (MB)
Paint	DT Mode	0.215	0.009	2.523	4374.05
	Virtual Mode	0.251	0.008	2.532	4643.03
Pong	DT Mode	0.210	0.012	3.342	4459.98
	Virtual Mode	0.187	0.008	2.746	4664.51
Logistics	DT Mode	0.212	0.008	3.313	4433.06
	Virtual Mode	0.204	0.008	3.189	4626.84
Farm	DT Mode	0.245	0.011	3.554	4379.40
	Virtual Mode	0.220	0.007	3.352	4632.64

Conversely, the **Physics** and **Render** components contribute negligibly to the CPU frame time (averaging $<$0.25 ms and $\approx$0.01 ms respectively). The extremely low CPU render times confirm that the rendering workload is successfully offloaded to the GPU. This is a significant architectural improvement over Gehricke's Pygame implementation, where rendering times averaged 3.75 ms on the CPU [Geh24].

5.3 Performance and Scalability Analysis

The scalability of the system was evaluated by measuring the Main Thread CPU time while incrementally increasing the number of interactive objects in the scene. To ensure reproducible and statistically stable results, the data was collected through an automated process: objects were spawned in fixed batches, followed by a one-second settling phase. This pause is critical to allow the physics engine to resolve initial contacts and allow rigid bodies to enter a "sleep" state, ensuring that the measurements reflect a stable environment. After this settling period, performance samples were collected over a one-second window to filter out singular frame spikes, such as those caused by background OS tasks or garbage collection.

The benchmarks were conducted across four configurations to isolate the costs of VR and dynamic physics. The resulting data is visualized in Figure 5.2, using a logarithmic scale to capture the wide variance in frame times.

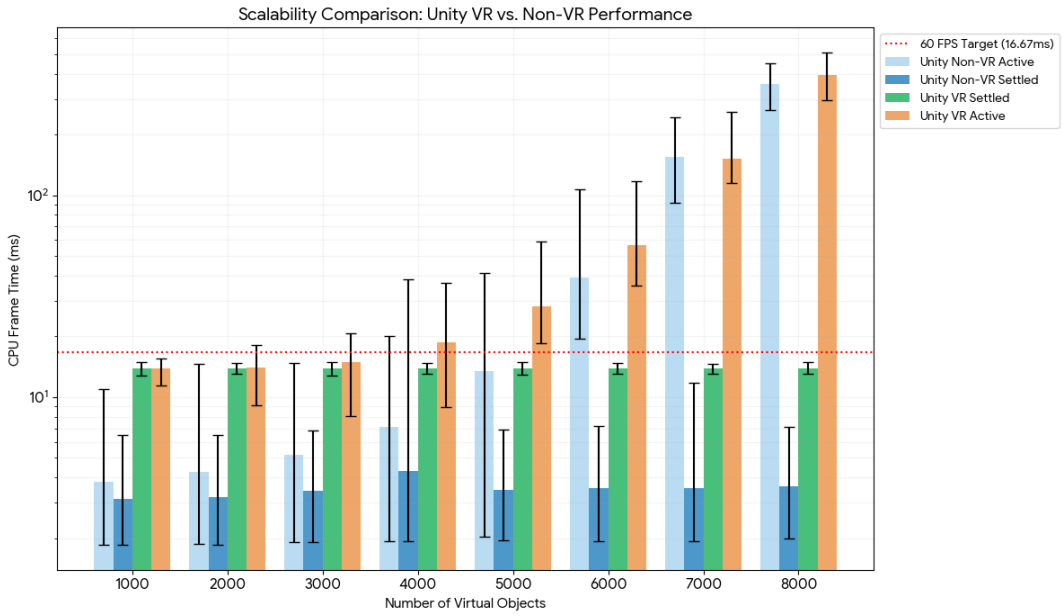


Figure 5.2: Scalability evaluation illustrating CPU frame time vs. object count. The 60 FPS target (16.67 ms) serves as the threshold for visual stability in Mixed Reality.

The *Non-VR Settled* mode serves as the baseline for the engine's rendering efficiency. In this mode, the system maintains a frame time between 3.12 ms and 3.68 ms for up to 9,000 objects. Gehricke [Geh24] reported an average update time of approximately 3.75 ms for his 2D Pygame visualizer, with spikes occurring around 11,250 objects due to rendering limitations. The proposed Unity-based architecture matches and slightly improves upon this baseline ($\approx$3.12 ms), despite rendering a full 3D environment with lighting and materials. This demonstrates the efficiency of static mesh batching in modern game engines compared to immediate-mode 2D drawing.

When activating Virtual Reality (*VR Settled*), the frame time stabilizes at approximately 13.8 ms. This constant overhead is characteristic of the stereoscopic rendering pipeline and the synchronization required for the 72 Hz display of the Meta Quest Pro. In this static state, the system successfully maintains the required 60 FPS threshold for up to 30,000 objects, significantly exceeding the limit observed in the original VERA framework.

However, the *Active Physics* scenarios reveal a computational bottleneck not present in the previous work, simply because the previous work did not simulate physics. Gehricke's collision checks were simple distance heuristics with linear scalability. In contrast, this system resolves mass, inertia, and friction using Nvidia PhysX. While rendering scales linearly, the cost of resolving thousands of simultaneous physical collisions increases exponentially. The system remains performant for up to 3,000 dynamic objects, but exceeds the 16.67 ms threshold beyond 5,000 objects.

5.4 Latency Analysis

In addition to frame rate stability, the system responsiveness was evaluated by measuring the visualization latency. This metric is defined as the time elapsed between the reception of a pose update from the ArUco tracking system and the completion of the rendered frame displaying that update. A custom `LatencyMonitor` script was utilized to capture timestamps across four distinct application scenarios, ranging from low-complexity environments to high-fidelity simulations: *Paint* (dynamic texture modification), *Pong* (fast-moving physics objects), *Logistics* (dynamic object spawning), and *Farm* (complex geometry and vegetation).

Figure 5.3 illustrates the recorded latency for both the AR Projection (Non-VR) and Virtual Reality (VR) modes. In the standard AR Projection mode, the system demonstrates high responsiveness, with average latencies ranging from 3.02 ms in the simple Paint scenario to 4.41 ms in the complex Farm environment.

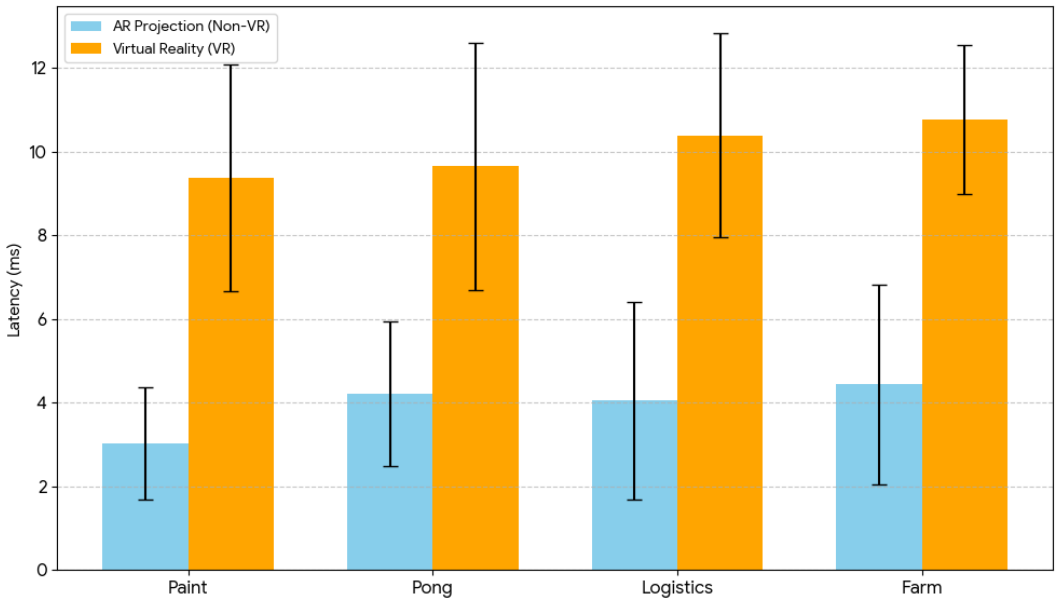


Figure 5.3: Visualization latency measured across four application scenarios. The chart compares the delay between pose reception and frame completion in AR Projection mode versus Virtual Reality mode. Error bars indicate the jitter (min/max range).

These results represent a significant improvement over the original VERA framework. Gehricke reported a collision-to-visualization latency of approximately 43.8 ms for an environment with 800 objects [Geh24]. Furthermore, Gehricke identified a critical scalability issue where the message queue for marker updates became saturated in environments with over 1,250 objects, leading to latencies spiking into the range of several seconds.

The proposed architecture eliminates this bottleneck. By utilizing the native `Ros2ForUnity` plugin, which communicates directly via the RCL layer without the overhead of Python callbacks used in the previous work, the system maintains a deterministic latency profile. Even in the complex *Farm* scenario, the Non-VR latency remains below 5 ms.

Activating Virtual Reality introduces a consistent latency overhead, increasing the average processing time to approximately 9.4 ms for Paint and up to 10.7 ms for the Farm scenario. This increase is attributed to the computational cost of stereoscopic rendering and the synchronization constraints of the VR headset. Despite this increase, the total visualization latency remains well below the critical 20 ms motion-to-photon threshold required to prevent motion sickness in VR, validating the system's suitability for immersive Robot-in-the-Loop interaction.

6 Conclusion

summary of the work and discussion of future research directions.

- erwähne wie der die virtuelle pyhsic in zukunkft auf den realen roboter übertragen werden kann (sim2real?)

List of Figures

Listings

Bibliography

[AA23] K. K. Alnowaiser and M. A. Ahmed. “Digital Twin: Current Research Trends and Future Directions”. In: *Arabian Journal for Science and Engineering* 48.2 (2023), pp. 1075–1095.

[AAR+19] Bulgaria Aleksandrov, Chavdar Acad, Bulgaria Rumenin, et al. “Review of hardware-in-the-loop -a hundred years progress in the pseudo-real testing”. In: 54 (Dec. 2019), pp. 70–84.

[ALV25] ALVR Team. *ALVR: Air Light VR – Open Source Remote Rendering for VR*. `https://github.com/alvr-org/ALVR`. GitHub repository. Accessed: 2025-12-15. 2025.

[AMV23] Hugo Araujo, Mohammad Reza Mousavi, and Mahsa Varshosaz. “Testing, Validation, and Verification of Robotic and Autonomous Systems: A Systematic Review”. In: *ACM Transactions on Software Engineering and Methodology* 32.2 (2023). ISSN: 1049-331X. DOI: 10.1145/3542945. URL: `https://doi.org/10.1145/3542945`.

[BM18] Mordechai Ben-Ari and Francesco Mondada. *Elements of Robotics*. Springer Open, 2018. ISBN: 978-3-319-62533-1. DOI: 10.1007/978-3-319-62533-1.

[Bra00] G. Bradski. “The OpenCV Library”. In: *Dr. Dobb’s Journal of Software Tools* (2000).

[BSH24] Rajmeet Bhourji, Lakmal Seneviratne, and Irfan Hussain. “A Deep Learning-Based Approach to Strawberry Grasping Using a Telescopic-Link Differential Drive Mobile Robot in ROS-Gazebo for Greenhouse Digital Twin Environments”. In: *IEEE Access* PP (Jan. 2024), pp. 1–1. DOI: 10.1109/ACCESS.2024.3520233. URL: `https://doi.org/10.1109/ACCESS.2024.3520233`.

[BVM20] Anja Babić, Goran Vasiljevic, and Nikola Miskovic. “Vehicle-in-the-Loop Framework for Testing Long-Term Autonomy in a Heterogeneous Marine Robot Swarm”. In: *IEEE Robotics and Automation Letters* PP (June 2020), pp. 1–1. DOI: 10.1109/LRA.2020.3000426.

[CCC+21] Giuseppina Lucia Casalaro, Giulio Cattivera, Federico Ciccozzi, et al. "Model-driven engineering for mobile robotic systems: a systematic mapping study". In: *Software and Systems Modeling* 21 (2021), pp. 19–49. DOI: 10.1007/s10270-021-00908-8.

[CCG+23] X. Cao, H. Chen, S. Y. Gelbal, et al. "Vehicle-in-Virtual-Environment (VVE) Method for Autonomous Driving System Development, Evaluation and Demonstration". In: *Sensors* 23.11 (2023), p. 5088. DOI: 10.3390/s23115088. URL: https://doi.org/10.3390/s23115088.

[CIR23] Enrique Coronado, Shunki Itadera, and Ixchel G. Ramirez-Alpizar. "Integrating Virtual, Mixed, and Augmented Reality to Human–Robot Interaction Applications Using Game Engines: A Brief Review of Accessible Software Tools and Frameworks". In: *Applied Sciences* 13.3 (2023). ISSN: 2076-3417. DOI: 10.3390/app13031292. URL: https://www.mdpi.com/2076-3417/13/3/1292.

[CKY20] Eunhee Chang, Hyun Taek Kim, and Byounghyun Yoo. "Virtual Reality Sickness: A Review of Causes and Measurements". In: *International Journal of Human–Computer Interaction* 36.17 (2020), pp. 1658–1682. DOI: 10.1080/10447318.2020.1778351.

[DBF+25] Juliana Dos Santos, Murilo Bicho, Tony Froes, et al. "Benchmarking Digital Twins for Tower Cranes: Isaac Sim vs. Gazebo". In: *IECON 2025 – 51st Annual Conference of the IEEE Industrial Electronics Society*. 2025, pp. 1–8. DOI: 10.1109/IECON58223.2025.11221823.

[DKK+21] Axel Diewald, Clemens Kurz, Prasanna Venkatesan Kannan, et al. "Radar Target Simulation for Vehicle-in-the-Loop Testing". In: *Vehicles* 3 (May 2021), pp. 257–271. DOI: 10.3390/vehicles3020016.

[DKQ+21] Xunhua Dai, Chenxu Ke, Quan Quan, and Kai-Yuan Cai. "RFlySim: Automatic test platform for UAV autopilot systems with FPGA-based hardware-in-the-loop simulations". In: *Aerospace Science and Technology* 114 (Apr. 2021), p. 106727. DOI: 10.1016/j.ast.2021.106727.

[DRC+17] Alexey Dosovitskiy, German Ros, Felipe Codevilla, et al. "CARLA: An Open Urban Driving Simulator". In: *1st Conference on Robot Learning (CoRL 2017)*. 2017. arXiv: 1711.03938.

[DSR+22] Maikol Drechsler, Varun Sharma, Fabio Reway, et al. "Dynamic Vehicle-in-the-Loop: A Novel Method for Testing Automated Driving Functions". In: *SAE International Journal of Connected and Automated Vehicles* 5 (June 2022), pp. 1–14. DOI: 10.4271/12-05-04-0029.

[EM21] Maria Engberg and Blair Macintyre. *Reality Media: Augmented and Virtual Reality*. Nov. 2021. ISBN: 9780262366250. DOI: 10.7551/mitpress/11708.001.0001.

[Epi19] Epic Games. *Unreal Engine*. Version 4.22.1. Apr. 25, 2019. URL: https://www.unrealengine.com.

[FBT97] D. Fox, W. Burgard, and S. Thrun. "The dynamic window approach to collision avoidance". In: *IEEE Robotics & Automation Magazine* 4.1 (Mar. 1997), pp. 23–33. DOI: 10.1109/100.580977.

[FCY25] Jose M. Flores Gonzalez, Enrique Coronado, and Natsuki Yamanobe. "ROS-Compatible Robotics Simulators for Industry 4.0 and Industry 5.0: A Systematic Review of Trends and Technologies". In: *Applied Sciences* 15.15 (2025). ISSN: 2076-3417. DOI: 10.3390/app15158637. URL: https://www.mdpi.com/2076-3417/15/15/8637.

[Foo13] Tully Foote. "tf: The transform library". In: *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*. Open-Source Software workshop. Apr. 2013, pp. 1–6. DOI: 10.1109/TePRA.2013.6556373.

[Geh24] P. Gehricke. "Virtual Environment for mobile Robotic Applications". Computer Engineering working group – Computer Science. Bachelor's thesis. Universität Osnabrück Neuer Graben/Schloss 49074 Osnabrück: Osnabrueck University, 2024.

[Gri15] Michael Grieves. "Digital Twin: Manufacturing Excellence through Virtual Factory Replication". In: (Mar. 2015).

[GRM+22] Miguel Á González-Santamarta, Francisco J Rodríguez-Lera, Vicente Matellán-Olivera, and Camino Fernández-Llamas. "Yasmin: Yet another state machine". In: *Iberian Robotics conference*. Springer. 2022, pp. 528–539.

[GTP25] P. Gehricke, M. Tassemeier, and M. Porrmann. "EMAROs: A Modular Autonomous Robot-Platform for Research and Education". In: *Robotics in Education (RiE 2025)*. Thessaloniki, Greece: Springer, Apr. 2025, pp. 533–544. DOI: 10.1007/978-3-031-98762-5_45. URL: https://doi.org/10.1007/978-3-031-98762-5_45.

[GV17] Michael Grieves and John Vickers. "Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems". In: Aug. 2017, pp. 85–113. ISBN: 978-3-319-38754-3. DOI: 10.1007/978-3-319-38756-7_4.

[HLT+19] Márton Horváth, Qiong Lu, Tamas Tettamanti, et al. “Vehicle-In-The-Loop (VIL) and Scenario-In-The-Loop (SCIL) Automotive Simulation Concepts from the Perspectives of Traffic Simulation and Traffic Control”. In: *Transport and Telecommunication Journal* 20 (Apr. 2019), pp. 153–161. DOI: 10.2478/ttj-2019-0014.

[Hu05] Xiaolin Hu. “Applying robot-in-the-loop-simulation to mobile robot systems”. In: *2005 IEEE International Conference on Robotics and Automation*. 2005, pp. 506–513. ISBN: 0-7803-9178-0. DOI: https://doi.org/10.1109/ICAR.2005.1507456.

[KH04] Nathan Koenig and Andrew Howard. “Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. Sendai, Japan, Sept. 2004, pp. 2149–2154.

[KYH+24] Seyed Mohamad Kargar, Borislav Yordanov, Carlo Harvey, and Ali Asadipour. “Emerging Trends in Realistic Robotic Simulations: A Comprehensive Systematic Literature Review”. In: *IEEE Access* 12 (2024), pp. 191264–191287. DOI: 10.1109/ACCESS.2024.3404881.

[KYS+25] Woojin Kwon, Jieun Yang, Seunghwa Song, et al. “Real-Time Digital-Twin-Based Cobot-Worker Collision Risk Prediction Using Unity, ROS, and UWB”. In: *IEEE Access* 13 (2025), pp. 85967–85978. DOI: 10.1109/ACCESS.2025.3569332. URL: https://doi.org/10.1109/ACCESS.2025.3569332.

[LMH+18] Jacky Liang, Viktor Makoviychuk, Ankur Handa, et al. “Gpu-accelerated robotic simulation for distributed reinforcement learning”. In: *Conference on Robot Learning*. PMLR. 2018, pp. 270–282.

[LWS+21] Jiewu Leng, Dewen Wang, Weiming Shen, et al. “Digital twins-based smart manufacturing system design in Industry 4.0: A review”. In: *Journal of Manufacturing Systems* 60 (July 2021), pp. 119–137. DOI: 10.1016/j.jmsy.2021.05.011.

[MCT21] Carlos Magrin, Gustavo Conte, and Eduardo Todt. “Creating a Digital Twin as an Open Source Learning Tool for Mobile Robotics”. In: *2021 Latin American Robotics Symposium (LARS), Brazilian Symposium on Robotics (SBR) and Workshop on Robotics in Education (WRE)*. Oct. 2021, pp. 13–18. DOI: 10.1109/LARS/SBR/WRE54079.2021.9605457.

[MFG+22] Steven Macenski, Tully Foote, Brian Gerkey, et al. “Robot Operating System 2: Design, architecture, and uses in the wild”. In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[Mic04] Olivier Michel. “Webots TM: Professional Mobile Robot Simulation”. In: *International Journal of Advanced Robotic Systems* 1.1 (2004), pp. 39–42. ISSN: 1729-8806.

[MK94] Paul Milgram and Fumio Kishino. “A Taxonomy of Mixed Reality Visual Displays”. In: *IEICE Transactions on Information Systems* E77-D.12 (Dec. 1994), pp. 1321–1329.

[MML+23] Steve Macenski, Tom Moore, David V. Lu, et al. “From the desks of ROS maintainers: A survey of modern & capable mobile robotics algorithms in the Robot Operating System 2”. In: *Robotics and Autonomous Systems* 168 (2023), p. 104493. ISSN: 0921-8890. DOI: 10.1016/j.robot.2023.104493. URL: https://www.sciencedirect.com/science/article/pii/S092188902300132X.

[MMW+20a] Steve Macenski, Francisco Martín, Ruffin White, and Jonatan Ginés Clavero. “The Marathon 2: A Navigation System”. In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020, pp. 2718–2725. DOI: 10.1109/IROS45743.2020.9341207.

[MMW+20b] Steven Macenski, Francisco Martín, Ruffin White, and Jonatan Ginés Clavero. “The Marathon 2: A Navigation System”. In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, Oct. 2020, pp. 2718–2725. DOI: 10.1109/IROS45743.2020.9341207. URL: https://doi.org/10.1109/IROS45743.2020.9341207.

[MRS24] Nuno Marques, Marco Rodrigues, and Diogo Sousa. “Driving Forward Mobile Robotics: A Digital Twin Architecture Case Study for AGVs Data-Driven Autonomy”. In: (Jan. 2024). DOI: 10.21203/rs.3.rs-3837578/v1. URL: https://doi.org/10.21203/rs.3.rs-3837578/v1.

[MTH22] F. Mihalič, M. Truntič, and A. Hren. “Hardware-in-the-Loop Simulations: A Historical Overview of Engineering Challenges”. In: *Electronics* 11.15 (2022), p. 2462. DOI: 10.3390/electronics11152462. URL: https://doi.org/10.3390/electronics11152462.

[MV20] Zhanat Makhataeva and Atakan Varol. "Augmented Reality for Robotics: A Review". In: *Robotics* 9.2 (Apr. 2020), p. 21. DOI: 10.3390/robotics9020021. URL: https://doi.org/10.3390/robotics9020021.

[PKP+20] Alexander Perzylo, Ingmar Kessler, Stefan Profanter, and Markus Rickert. "Toward a Knowledge-Based Data Backbone for Seamless Digital Engineering in Smart Factories". In: *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. 2020, pp. 164–171. DOI: 10.1109/ETFA46521.2020.9211943.

[PRR+20] L. Pérez, S. Rodríguez-Jiménez, N. Rodríguez, et al. "Digital Twin and Virtual Reality Based Methodology for Multi-Robot Manufacturing Cell Commissioning". In: *Applied Sciences* 10.10 (May 2020), p. 3633. DOI: 10.3390/app10103633. URL: https://doi.org/10.3390/app10103633.

[Rob21] Robotec.AI. *ROS2 For Unity*. GitHub repository. 2021. URL: https://github.com/RobotecAI/ros2-for-unity.

[Rob24] Robotec.AI. *ROS2 For Unity*. https://github.com/RobotecAI/ros2-for-unity. Accessed on May 26, 2025. 2024.

[SKG+24] Maulshree Singh, Jayasekara Kapukotuwa, Eber Lawrence Souza Gouveia, et al. "Unity and ROS as a Digital and Communication Layer for Digital Twin Application: Case Study of Robotic Arm in a Smart Manufacturing Cell". In: *Sensors* 24.17 (2024). ISSN: 1424-8220. DOI: 10.3390/s24175680. URL: https://www.mdpi.com/1424-8220/24/17/5680.

[SKG+25] Maulshree Singh, Jayasekara Kapukotuwa, Eber Lawrence Souza Gouveia, et al. "Comparative Study of Digital Twin Developed in Unity and Gazebo". In: *Electronics* 14.2 (2025). ISSN: 2079-9292. DOI: 10.3390/electronics14020276. URL: https://www.mdpi.com/2079-9292/14/2/276.

[SPD+21] P. Stączek, J. Pizoń, W. Danilczuk, and A. Gola. "A Digital Twin Approach for the Improvement of an Autonomous Mobile Robots (AMR's) Operating Environment—A Case Study". In: *Sensors* 21.23 (2021), p. 7830. DOI: 10.3390/s21237830. URL: https://doi.org/10.3390/s21237830.

[Spr25] Julia Spradau. “Echtzeitfähige Bildverarbeitung für das Tracking mobiler Roboter”. Computer Engineering working group – Computer Science. Bachelor’s thesis. Universität Osnabrück Neuer Graben/Schloss 49074 Osnabrück: Osnabrueck University, 2025.

[SSW21] Richard Skarbez, Missie Smith, and Mary Whitton. “Revisiting Milgram and Kishino’s Reality-Virtuality Continuum”. In: *Frontiers in Virtual Reality* 2 (Mar. 2021). DOI: 10.3389/frvir.2021.647997. URL: https://doi.org/10.3389/frvir.2021.647997.

[Uni23] Unity Technologies. *Unity*. Version 2023.2.3. Game development platform. 2023. URL: https://unity.com/.

[Uni25] Unity Technologies. *Unity Documentation*. https://docs.unity.com/en-us. Accessed: 2025-12-15. 2025.

[VTS21] Balázs Varga, Tamas Tettamanti, and Zsolt Szalay. “System Architecture for Scenario-In-The-Loop Automotive Testing”. In: *Transport and Telecommunication Journal* 22 (Apr. 2021), pp. 141–151. DOI: 10.2478/ttj-2021-0011.

[WPC+22] Michael Walker, Thao Phung, Tathagata Chakraborti, et al. *Virtual, Augmented, and Mixed Reality for Human-Robot Interaction: A Survey and Virtual Design Element Taxonomy*. 2022. arXiv: 2202.11249 [cs.RO]. URL: https://arxiv.org/abs/2202.11249.

[YMZ20] Yuanlin Yang, Wei Meng, and Shiquan Zhu. “A Digital Twin Simulation Platform for Multi-rotor UAV”. In: *2020 7th International Conference on Information, Cybernetics, and Computational Social Systems (ICCSS)*. 2020, pp. 591–596. DOI: 10.1109/ICCSS52145.2020.9336872. URL: https://doi.org/10.1109/ICCSS52145.2020.9336872.

Declaration of Authorship

I hereby declare that I have written this thesis independently and without unauthorized assistance. I have not used any sources or aids other than those indicated. All passages taken verbatim or in spirit from the works of other authors have been properly acknowledged and cited.

Osnabrück, 23 December 2025

Jonas Kittelmann