



TUHH

BACHELOR'S THESIS

Optimizing an Augmented Reality Environment and Data Glove Integration for a Remote Inspection System

Author

Mohamad Nassif

Matr. -Nr.: 21967807

Hamburg, 08. January 2024

1st Examiner: **Prof. Dr.-Ing. Thorsten A. Kern**

2nd Examiner: **Fady Youssef, M. Sc**

Supervisor: **Fady Youssef, M. Sc**

at



INSTITUT
FÜR MECHATRONIK
IM MASCHINENBAU

Declaration of Autonomy

I declare, that the present bachelor thesis, examined by Prof. Dr.-Ing. Thorsten A. Kern, titled

Optimizing an Augmented Reality Environment and Data Glove Integration for a Remote Inspection System

was composed by myself and without using any resources other than those stated. I assure that when using external publications and sources, either by direct or indirect quotation, I explicitly marked them as such.

Neither this work nor any part if it has been turned in, in this or any similar form, to any examination office. I assure that the print version of this work matches the one included in the attached digital media.

Mohamad Nassif

Acknowledgements

I would like to express my gratitude to my thesis advisor, Fady Youssef, M.Sc for his patience, and support throughout the course of this research. I am equally grateful to Prof. Dr.-Ing. Thorsten A. Kern for his constructive criticism and valuable suggestions.

My heartfelt thanks go to my family, who have been my bedrock and a constant source of motivation. To my parents, Murhaf Nassif and Edna Collao, thank you for your unwavering belief in me and for standing by me at every juncture. To my brother, Hatem Nassif, BE, your guidance has been a beacon, illuminating my path. To my sister, Mouradi Nassif, your endless compassion has been a source of comfort.

I extend my deepest gratitude to Hussein Ibrahim, BE, whose wisdom and assistance have been invaluable. His determination and skills have been an inspiration, and his support was instrumental in this journey.

I would also like to extend my gratitude to Mahmoud Chamsiddin for his encouragement and support during the challenging moments of my journey.

The work presented in this paper is supported by the Institute for Mechatronics (iMEK) at the Hamburg University of Technology.

Abstract

Augmented reality (AR), a pivotal component of the fourth industrial revolution, is gaining significant traction in the industrial sector due to its potential to enhance safety measures and reduce workload. One promising application of AR technology is in remote inspection operations, where it can mitigate safety hazards and eliminate geographical constraints for experts. This paper presents a comprehensive effort to optimize an AR environment for an existing remote inspection system and fully integrate a data glove within this environment. Detailed solutions are provided to enhance the accuracy of the virtual representation of hand movements and improve its responsiveness to user input. The paper explains the process of overhauling the system's technical infrastructure, improving data transmission pathways, and exploring the potential of incorporating an interactive user interface. Data and statistical analysis are presented to validate the findings. The paper concludes with recommendations for future development, laying the groundwork for further innovation in this interdisciplinary research domain.

Table of Contents

List of Figures and Tables.....	vii
Chapter I: Introduction.....	1
1.1 Background and Justification.....	1
1.2 Objectives	1
1.3 Related Work	2
1.4 System Initial State	3
Chapter II: Conceptual Framework	5
2.1 Augmented/Mixed Reality.....	5
2.1.1 Unity	5
2.1.2 HTC Vive Pro	6
2.1.3 SteamVR.....	6
2.1.4 OpenXR and OpenVR	6
2.1.5 ZED mini-Camera.....	7
2.2 Pose Tracking.....	7
2.2.1 HTC Vive Tracker and Lighthouse System.....	8
2.2.2 Inertial Measurement Units (IMUs).....	8
2.2.3 Quaternions and Eulers	8
2.4 Data Transmission	9
2.4.1 Robot Operating System (ROS).....	9
2.4.2 ROS TCP and ROS#.....	9
2.4.3 ESP32 Microcontroller	10
2.4.4 Sampling Rate and Publishing Rate.....	10
Chapter III: Methodology	11
3.1 List of Requirements.....	11
3.2 System Structure	13
3.2.1 Black Box Model:	13
3.2.2 Functional Decomposition	14
Chapter IV: Development.....	17
4.1 Implementation	17
4.1.1 Software Infrastructure and Data Flow	17
4.1.2 Optimizing Hand Movement Tracking System	18

4.1.3 Integrating the User Interface	26
4.2 System Validation.....	27
4.2.1 Latency and Accuracy.....	27
4.2.2 User Evaluation.....	28
Chapter V: Findings and Discussion.....	29
5.1 Test Results	29
5.2 Questionnaire	31
Chapter VI: Conclusion and Future Works	32
References	33
Appendix A: Questionnaire	35
Appendix B: Scripts.....	39
Unity script for the hand	39
Unity script for publishing	42
Microcontroller main script	43
Microcontroller functions script	45
Microcontroller variables script.....	48

List of Figures and Tables

<i>Figure 1: The System's Initial State</i>	3
<i>Figure 2: The V-Model</i>	11
<i>Figure 3: Black Box Model</i>	13
<i>Figure 4: Functional Decomposition</i>	14
<i>Figure 5: System Structure</i>	16
<i>Figure 6: XR Plug-in Management in Unity</i>	17
<i>Figure 7: Publishing from Unity</i>	18
<i>Figure 8: The Data Glove</i>	18
<i>Figure 9: The previous approach for locking rotations</i>	19
<i>Figure 10: Pitch (Z) adjusting with the roll rotation(X)</i>	20
<i>Figure 11: Pitch and yaw (Y and Z) adjusting with the roll rotation(X)</i>	20
<i>Figure 12: The new approach of applying angles to an Object</i>	20
<i>Figure 13: SteamVR room setup</i>	21
<i>Figure 14: Functional hand tracking</i>	22
<i>Figure 15: Hand models from left to right: old, new with a realistic mesh, new with a transparent mesh</i>	23
<i>Figure 16: Data route within ROS</i>	24
<i>Figure 17: The new publishing rate</i>	25
<i>Figure 18: User Interface</i>	26
<i>Figure 19: Accuracy test process</i>	27
<i>Figure 20: Latency Chart when idle</i>	29
<i>Figure 21: Latency Chart when in motion</i>	29
<i>Figure 22: FPS counter</i>	30
<i>Table 1: Rotation angle conventions (from Bosch Sensortec)</i>	19
<i>Table 2 BNO055 operation modes overview (from Bosch)</i>	21

Chapter I: Introduction

1.1 Background and Justification

In an era characterized by widespread industrialization, the imperative for the maintenance and inspection of machinery has assumed heightened significance in ensuring safety and reliability. The contemporary technological landscape has facilitated the execution of these inspections remotely, mitigating the exposure of experts to potential hazards. In response to this imperative, the Institute of Mechatronics at the Hamburg University of Technology has embarked on a project aimed at developing a remote inspection system. This innovative system incorporates Augmented Reality (AR) technology and haptic feedback, employing a specialized haptic feedback glove capable of tracking finger movements.

This paper delves into the specific dimension of Augmented Reality, which is defined as a main aspect of the fourth industrial revolution [1]. The utilization of AR is underscored by its potential to immerse experts fully in the inspection scene, fostering a comprehensive understanding of the operational environment. Furthermore, the integration of AR lays the foundation for future advancements by introducing mixed reality (MR) through additional virtual elements and simulations that enhance the inspection process.

1.2 Objectives

This paper defines a set of comprehensive objectives aimed at guiding the development and evaluation of the AR component within the broader context of the remote inspection project. Each objective is strategically formulated to address critical facets, ensuring the creation of a robust and forward-looking AR environment.

a) Development of a Comprehensive Hand Movement Tracking System within the AR Environment

The primary objective of this task is to engineer a fully functional system dedicated to tracking and visualizing hand movements seamlessly within the AR scene. The success of this objective hinges upon the establishment of a robust connection between the user's physical interactions, facilitated by the aforementioned haptic feedback glove, and the corresponding visual representation within the AR environment. Extensive testing and refinement will be undertaken to ensure the precision and reliability of the hand movement tracking system, fostering a heightened sense of immersion for users.

b) Optimization of Latency and Error in Finger Tracking

This objective centers on the meticulous optimization of latency and error associated with finger tracking between the haptic feedback glove and the virtual interface. Addressing this challenge is imperative for enhancing the real-time responsiveness of the system, minimizing discrepancies between physical hand movements and their virtual counterparts.

c) Integration of Quality-of-Life Features: User Interface (UI) and Customization Options

In tandem with the technical advancements in hand movement tracking, this objective seeks to further augment the user experience through the incorporation of quality-of-life features. A UI will be designed and integrated into the AR scene, presenting users with intuitive controls and real-time feedback. Additionally, customization options will be implemented to allow users to tailor the appearance of the hand model based on their individual preferences.

d) Comparative Analysis of Data Flow Routes for Enhanced System Efficiency

This objective focuses on conducting a comprehensive analysis and comparison of the various routes for data flow within different components of the main project. The aim is to evaluate the efficiency and ease of use associated with different data flow architectures.

e) Ensuring Adaptability and Future-Proofing the Project

The final objective pertains to future-proofing the project by ensuring its adaptability to emerging technological developments. This involves designing the system with a modular architecture, facilitating the seamless integration of additional features and updates. Consideration will be given to industry standards, potential advancements in AR technology, and evolving user requirements.

1.3 Related Work

The project has seen active participation from numerous contributors, each advancing different aspects of its evolution. A key work [2] introduced the first functional haptic feedback glove, laying the groundwork for future improvements.

Subsequent enhancements focused on refining the Inertial Measurement Units [3] and incorporating a force feedback mechanism to improve the glove's functionality [4]. The integration of signals between system components was notably advanced in [5].

In the area of AR development, the project's initial AR environment was established by a single contributor. A prototype, outlined in [6] provided the foundation for integrating the haptic glove into the AR scene, setting the stage for further development. Progress continued with the creation of a system to replicate head movements [7], enhancing the project's immersive experience.

Moreover, a series of studies were undertaken to explore augmented reality implementation across various industries. A study [8] examines the use of an AR head-mounted display system in industrial quality inspection, finding that it enhances task performance, especially in complex tasks, and reduces mental workload. The research contributes empirical evidence to the field of industrial AR, demonstrating its positive effects on efficiency and user experience. Another paper [9] presents a usability case study on an AR head-mounted display for visual inspection tasks in manufacturing, revealing that while workload demands are manageable and

user reception is positive, interface improvements are needed to reduce frustration and enhance learning. It highlights ongoing human and technical challenges in AR implementation, suggesting that some issues can be resolved with targeted refinements, while others require broader industry efforts.

A recent paper [10] proposes an affordable, accurate sensor glove system that uses flex sensors for efficient motion tracking. The study evaluates the glove's performance and demonstrates its functionality in recording motion data for probabilistic movement models. This work significantly influenced our research, providing a valuable reference for our own sensor glove development. Another study that proved beneficial to this paper [11] introduces a low-cost, reliable, and easy-to-wear data glove equipped with 9-axis IMUs for capturing hand motions in medical evaluations. The glove's modular design allows for extensibility and compatibility with various microcontrollers, enhancing its stability and maintainability, and broadening its potential medical applications.

1.4 System Initial State

Before delving into the contributions of this paper, it is essential to outline the initial state of the project. At the outset, the AR environment demonstrated basic functionality but was plagued by several issues, rendering it unsuitable for a finalized product. Notably, virtual fingers failed to synchronize with the virtual hand as shown in Figure 1, resulting in visible glitches due to a fundamental mismatch between the IMUs and the virtual environment.

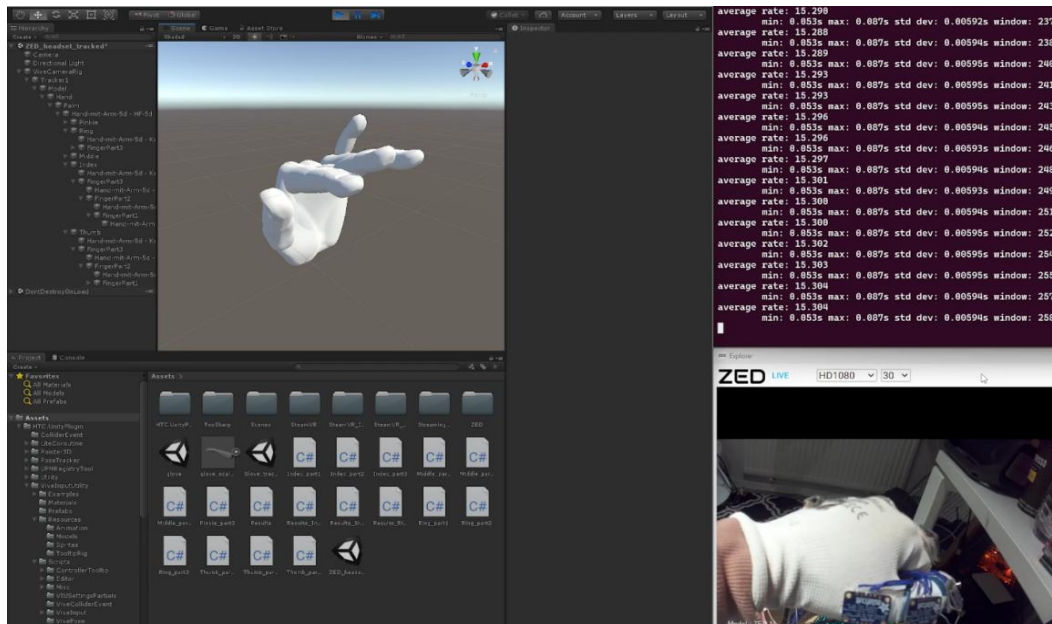


Figure 1: The System's Initial State

A significant drawback was the system's slow responsiveness to hand movements, a crucial requirement for this thesis. The observed sluggishness, with a publishing rate of only 15Hz (Figure 1), indicated suboptimal performance. Outdated software versions such as using Unity 2018 and reliance on numerous third-party packages further posed challenges. Some of these

third-party packages were outdated, hindering the project from leveraging new features and technologies in the rapidly evolving field.

Additionally, certain third-party packages were independently developed with uncertain support, raising the potential for conflicts among them. These challenges highlighted the need for a comprehensive reassessment and refinement of the project's technical foundations to overcome obstacles and align with contemporary standards and best practices in the dynamic field of Augmented Reality.

Chapter II: Conceptual Framework

2.1 Augmented/Mixed Reality

Augmented Reality and Mixed Reality (MR) represent two pivotal concepts in the realm of immersive technologies. These concepts, while distinct, share a common goal: to blend the physical and digital worlds in a manner that enhances user interaction and perception.

AR refers to the overlaying of digital information onto the user's real-world environment. This augmentation can take various forms, such as visual graphics, sound, or haptic feedback, and is typically achieved through the use of a smartphone or a head-mounted device. The primary objective of AR is to enrich the user's perception of the real world by integrating digital elements seamlessly into their environment.

MR, on the other hand, is a more advanced form of AR. It not only overlays but also anchors virtual objects to the real world, allowing users to interact with these objects as if they were physically present. This interaction can be as simple as viewing a virtual object from different angles by moving around it, or as complex as manipulating the object's shape and position in real-time.

In the context of this project, it can be argued that adding a user interface turns the system into mixed reality to some small degree but since the user interactions with the virtual elements are initially restricted to some essential functions such as exiting and calibrating, this paper will continue to consider AR as the main concept.

In the following sections, some key tools will be discussed in greater detail.

2.1.1 Unity

The main software used to create the AR environment is Unity3D. Unity3D, commonly referred to as Unity, is a robust and widely-used game development engine that facilitates the creation of interactive, real-time 3D content. Its versatility and user-friendly interface have made it a popular choice not only for game development but also for a variety of other applications, including virtual reality (VR), AR, and MR experiences, simulations, and visualizations.

Unity employs a scene-based design philosophy. Each scene represents a unique environment or level, and developers can populate these scenes with objects, characters, and effects to create interactive experiences. These elements can be manipulated using Unity's intuitive graphical interface or through scripts written in C#, a widely-used, object-oriented programming language. Unity's scripting API (Application Programming Interface) provides developers with a high degree of control over the behavior of their projects. Through scripting, developers can control the physics of the game world, manage user input, implement AI behaviors, and much

more. Unity's API is well-documented and supported by a large community of developers, making it a valuable resource for problem-solving and learning.

2.1.2 HTC Vive Pro

The HTC Vive Pro is a high-end virtual reality headset developed by HTC Corporation in collaboration with Valve Corporation. It is part of the SteamVR system and is designed to deliver a premium VR experience, with a focus on high resolution, precise tracking, and comfort [12].

The HTC Vive Pro also features an upgraded tracking system. Like the original Vive, it uses Valve's Lighthouse tracking system, but it supports the newer SteamVR Tracking 2.0 technology. This allows for larger play areas and improved tracking accuracy, which can be particularly beneficial for applications that require precise motion tracking.

2.1.3 SteamVR

SteamVR is a virtual reality system, with a suite of tools and services for VR applications, developed by Valve Corporation. It is a part of the Steam platform, a digital distribution platform for video games, and is primarily designed to work with the company's own hardware, such as the HTC Vive, Valve Index, and other OpenVR compatible devices. However, it also supports a wide range of other VR hardware, making it a versatile choice for VR development.

One of the key features of SteamVR is its tracking system, known as Lighthouse. This system uses Lighthouses that emit infrared light to track the position and orientation of VR devices with high precision and low latency. This makes it particularly suitable for applications that require accurate motion tracking, such as the hand tracking system developed in this project.

2.1.4 OpenXR and OpenVR

In the realm of virtual and augmented reality development, OpenXR and OpenVR are two significant APIs that facilitate the integration of hardware and software for immersive experiences.

OpenXR is an open standard for AR and VR platforms, developed by the Khronos Group, an industry consortium focused on creating open standard APIs [13]. The primary goal of OpenXR is to provide a unified interface for VR and AR applications, allowing developers to write code that is portable across a wide range of hardware devices without needing to use hardware-specific SDKs. This standard aims to simplify the development process and increase the interoperability of VR and AR software across different operating systems and devices. As OpenXR gains adoption, it is becoming the industry standard, offering broad support for various VR and AR devices [13]. What is also noteworthy about OpenXR is that it gives the option to bypass SteamVR completely reducing overhead [14] [15]. However, OpenXR currently has some limitations. Notably, it lacks support for Vive trackers, which are widely used in VR applications for precise motion tracking. Additionally, OpenXR is not yet fully

functional on Linux systems [16], which can be a significant drawback for developers working within Linux environments or targeting Linux as a platform for their VR/AR applications.

On the other hand, OpenVR is a platform-specific API developed by Valve Corporation, primarily for its SteamVR platform. OpenVR is closely associated with the HTC Vive hardware but also supports other VR devices. Unlike OpenXR, OpenVR is not an open standard, and while it provides comprehensive support for Vive trackers, it is less portable and requires more effort to support a wide range of VR hardware.

When developing VR or AR applications in Unity that require the use of Vive hardware, developers face a choice between these two APIs. If using OpenVR, an additional plugin is necessary to enable the use of Vive hardware within the Unity environment, this plugin acts as a bridge between Unity and the SteamVR platform, allowing developers to access the full range of features provided by the Vive trackers and controllers. In contrast, when using OpenXR with Unity, no extra plugins are required, as Unity has built-in support for this standard. This simplifies the development process by reducing the need for additional software layers and potentially streamlining the workflow. However, developers must weigh this convenience against the current limitations of OpenXR, such as the lack of support on Linux systems [16].

2.1.5 ZED mini-Camera

The ZED Mini camera, developed by Stereolabs, is a stereo camera designed to bring spatial perception to augmented/mixed reality applications. It is equipped with dual lenses that mimic human vision, capturing high-resolution stereo video and depth data. This depth perception is crucial for creating a convincing mixed reality experience, providing a 3D map of the environment for accurate placement and interaction of virtual objects within the real world.

In this project, the ZED Mini camera is used to capture the real-world environment. Its low-latency video pass-through allows for the seamless integration of real-world and virtual visuals, enhancing the realism of the augmented reality experience. The camera's compatibility with Unity3D, facilitated by Stereolabs' software development kit (SDK), simplifies its integration into the mixed reality application.

In summary, the ZED Mini camera is a key component in this project, providing the necessary visual inputs to create a cohesive and immersive augmented reality experience. Its advanced features and ease of integration with Unity3D make it an invaluable tool for blending the real and virtual worlds.

2.2 Pose Tracking

Pose tracking is a critical component in the realm of Augmented/Mixed Reality, particularly in applications that aim to replicate hand and finger movements. Various methods exist for tracking these movements, each with its own advantages and limitations. This project employs IMUs for finger tracking and the HTC Vive Tracker for hand pose tracking.

2.2.1 HTC Vive Tracker and Lighthouse System

The HTC Vive Tracker is used in this project to track the pose of the user's hand. This tracker is part of the SteamVR system and works in conjunction with the Lighthouse base stations to provide high-precision, low-latency tracking. The Lighthouse system uses a method known as "inside-out" tracking. The Lighthouses emit structured infrared light, which is detected by sensors on the Vive Tracker. By calculating the time it takes for the light to reach the sensors, the system can determine the tracker's position in 3D space [17]. This method provides a high level of accuracy and allows for large tracking areas.

2.2.2 Inertial Measurement Units (IMUs)

Nine Degrees of Freedom (9DOF) IMUs are sophisticated devices that capture detailed information about an object's motion and orientation in space. They are integral to a wide range of applications, from aviation and navigation to robotics and virtual reality.

A 9DOF IMU combines three key sensors: a three-axis accelerometer, a three-axis gyroscope, and a three-axis magnetometer. Each sensor contributes unique data that, when combined, provide a comprehensive picture of an object's movement and orientation.

In the context of this project, a 9DOF IMU from Bosch named BNO055 provides detailed, real-time data about the movements and orientation of the user's fingers. This data is processed by the ESP32 microcontroller, which translates the raw sensor data into a format that can be used to control the virtual hand in the augmented reality environment.

2.2.3 Quaternions and Eulers

Quaternions and Euler angles are both mathematical constructs used to represent rotations in three-dimensional space, but they differ significantly in their properties and applications.

Quaternions, represented by four numbers, are a type of complex number extension that are particularly useful for calculating rotations in three dimensions. They are less intuitive than Euler angles but have the advantage of avoiding the problem of gimbal lock, a situation where the axes of a three-axis rotation system align, causing a loss of a degree of freedom. Quaternions also allow for smooth interpolations of rotations, making them ideal for computer graphics and animation.

Euler angles, on the other hand, are a set of three angles that describe the orientation of a rigid body with respect to a fixed coordinate system. They are more intuitive to understand and use, as they directly correspond to rotations around the axes of the coordinate system. However, they are susceptible to gimbal lock and can result in more complex calculations for interpolations.

In summary, while Euler angles may be more straightforward to understand and implement, quaternions offer computational advantages and smoother transitions, making them the preferred choice for many applications in computer graphics and robotics.

2.4 Data Transmission

In this project, the Robot Operating System (ROS) serves as the central hub for data transmission, connecting different modules and facilitating their communication.

2.4.1 Robot Operating System (ROS)

ROS is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. ROS provides services designed for a heterogeneous computer cluster such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management [18]. ROS is particularly well-suited to this project due to its modular architecture, which allows for the integration of various software components with different functionalities. It also supports a wide range of programming languages, including Python and C++, which provides flexibility in software development.

ROS messages are the data structures used to exchange information between nodes. They are defined by the type of message and the data format. For example, the ROS package named `std_msgs` has messages of type “String” which consist of a string of characters. Other message packages for ROS have messages used for robot navigation or robotic sensors.

ROS topics are the “channels” through which messages are passed. Nodes publish messages to topics or subscribe to topics to receive messages. Each topic is strongly typed by the ROS message type used to publish to it, and nodes can only receive messages with a matching type. For example, a node might publish sensor data to a topic, and another node might subscribe to that topic to receive the sensor data.

2.4.2 ROS TCP and ROS#

ROS TCP is a Unity plugin that facilitates communication between Unity-based applications and the Robot Operating System (ROS). It is designed to enable a seamless data exchange using TCP/IP, which is a standard protocol for transmitting data over a network. It is maintained by Unity robotics hub. They also credit ROS# even though they have their own approach [19].

ROS# is a set of opensource software libraries and tools in C# for communicating with ROS from .NET applications [20]. It uses a `rosbridge` server for communicating with ROS, which uses JSON to allow communication. This approach requires a `rosbridge` server node to be running on a machine in the ROS environment, that receives data from Unity and republishes it as a ROS message. As such, all ROS communication to Unity first passes through this `rosbridge` node [21].

The distinctions between the two are analyzed in [21], significantly influencing the decision detailed in Section [4.1.1](#).

2.4.3 ESP32 Microcontroller

The ESP32 microcontroller, a highly integrated solution for Wi-Fi-and-Bluetooth IoT applications, is utilized in this project due to its robust functionality and cost-effectiveness. It is equipped with a dual-core processor, a large amount of memory, and a rich set of peripherals, making it capable of handling complex and demanding tasks.

In the context of this project, the ESP32 is employed to extract and process data from the IMUs. It plays a crucial role in the data pipeline, as it interprets the raw data from the IMUs, processes it, and then facilitates its transmission to ROS.

2.4.4 Sampling Rate and Publishing Rate

The concepts of sampling data and publishing data are closely related within the context of this project, and may have been used interchangeably in previous works. Hence, there is a need to define these two terms clearly to avoid misunderstandings in later chapters.

Sampling rate refers to the number of samples taken per second when converting a continuous analog signal into a discrete digital signal. This conversion occurs in an analog-to-digital converter within the microcontroller and is limited by the capabilities of this component.

In contrast, publishing rate in this work denotes the rate at which data is sent to the Robot Operating System (ROS) in the form of messages. Logically, the sampling rate must be equal to or higher than the publishing rate, as the data must first be sampled before it can be published. However, the publishing rate cannot exceed the sampling rate.

Chapter III: Methodology

In the pursuit of structuring the research process and fortifying the trajectory toward the research goal, the methodology adopted for this paper is the V model (Figure 2). In the context of this paper, the V model serves as a guiding framework, enabling a systematic exploration of the research objectives. By aligning the research activities with the sequential phases of the V model, this paper aims to enhance the robustness and reliability of the outcomes, ultimately contributing to the fulfillment of the research goal.

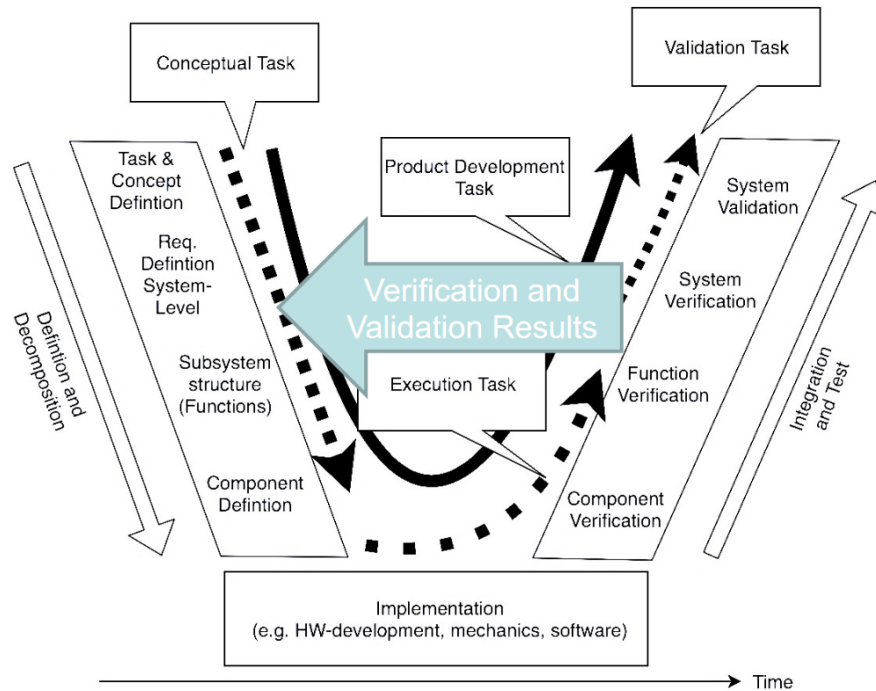


Figure 2: The V-Model

3.1 List of Requirements

In the initial phase of the methodology chapter, the foundational step involved the formulation of a comprehensive list of requirements aligned with the specified objectives of this paper. This catalog of requirements establishes explicit goals to be accomplished within defined parameters. The overarching aim guiding the creation of this list was the development of a system that not only meets the prescribed objectives but also embodies characteristics of adaptability, robustness, and reliability.

Index	Requirement	Description	Value	Unit	Rating	Further Notes
1	Latency	The time delay between a user's physical hand movement and the corresponding virtual representation within the AR environment.	<50	ms	Mandatory	[22]

Index	Requirement	Description	Value	Unit	Rating	Further Notes
2	Accuracy	The degree of disparity between the physical hand movements captured by the haptic feedback glove and their virtual counterparts in the AR environment.	<4.5	degree	Mandatory	[23]
3	Publishing Rate	The frequency at which the microcontroller captures and sends hand movement data per second.	>26	Hz	Mandatory	Nyquist Sampling Theorem
4	Frames Per Second (FPS)	The rate at which the AR environment refreshes and displays images to ensure smooth visual continuity.	≥ 90	fps	Mandatory	Should be equal or larger than the refresh rate of the HMD
6	Immersion	The extent to which the system engrosses the user in the AR scene, contributing to a heightened sense of presence and interaction.			Mandatory	Will be evaluated by users
7	Software	The version and suitability of the software utilized in the AR system, ensuring compatibility with cutting-edge features and technologies in the field.			Desirable	
8	Adaptability	The system's capacity to seamlessly integrate additional features and updates, ensuring adaptability to emerging technological developments.			Desirable	Consideration of modular architecture and standards.

3.2 System Structure

3.2.1 Black Box Model:

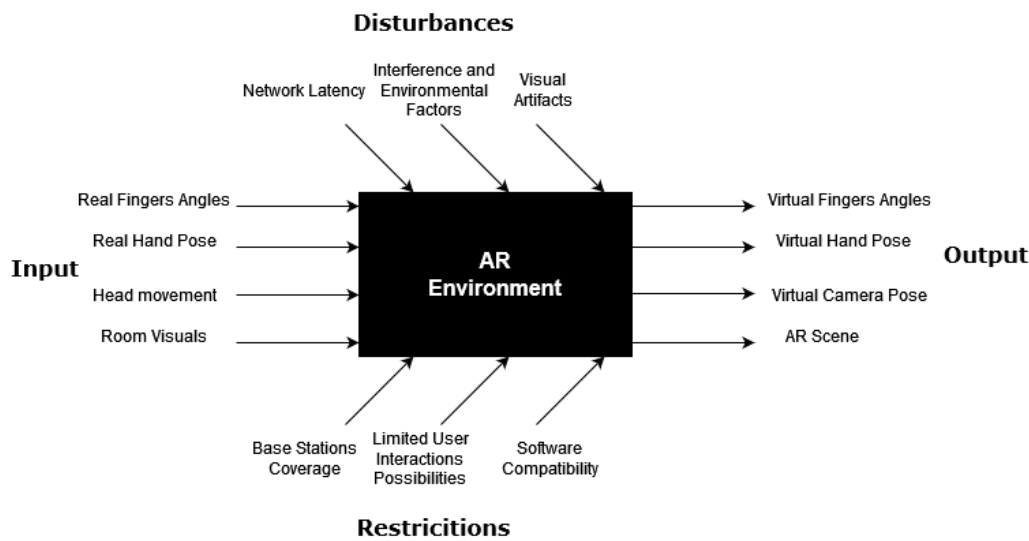


Figure 3: Black Box Model

The black box model depicted in Figure 3 characterizes the various inputs, disturbances, and restrictions affecting the system, along with the resulting outputs. The control and navigation of the AR scene rely on various data streams, including finger angles of rotation from the haptic feedback glove, pose data from the tracker and head-mounted device (HMD), and visual input from the camera.

For precise spatial localization of the tracker and HMD, two base stations, functioning as Infrared (IR) light emitters, are deployed. Ensuring optimal operation requires the devices to remain within the effective range of the base stations, with interference from external objects or intense lighting minimized for a smooth user experience.

The rotation data from IMU sensors on the haptic feedback glove follows a pipeline of devices and software to reach Unity software. Acknowledging that an extended route increases the risk of network disturbances, the robustness of this data transmission pathway is crucial.

Potential visual artifacts, arising from software glitches or data loss, pose challenges to the fidelity of the AR scene. The project's reliance on diverse software packages and plugins emphasizes the need for careful compatibility checks.

With no conventional controllers, the project demands the creation of unique interaction methods with virtual elements in the AR scene, likely through specific gestures and ray casts.

In its culmination, the entire process unfolds on the head-mounted device, highlighting the intricate nature of the project and the need for a comprehensive approach to address challenges and optimize functionalities.

3.2.2 Functional Decomposition

Functional decomposition is a systematic approach used to break down a complex system into smaller, more manageable functions or modules. This aids in a structured understanding of the system's functionality and facilitates the development and testing processes.

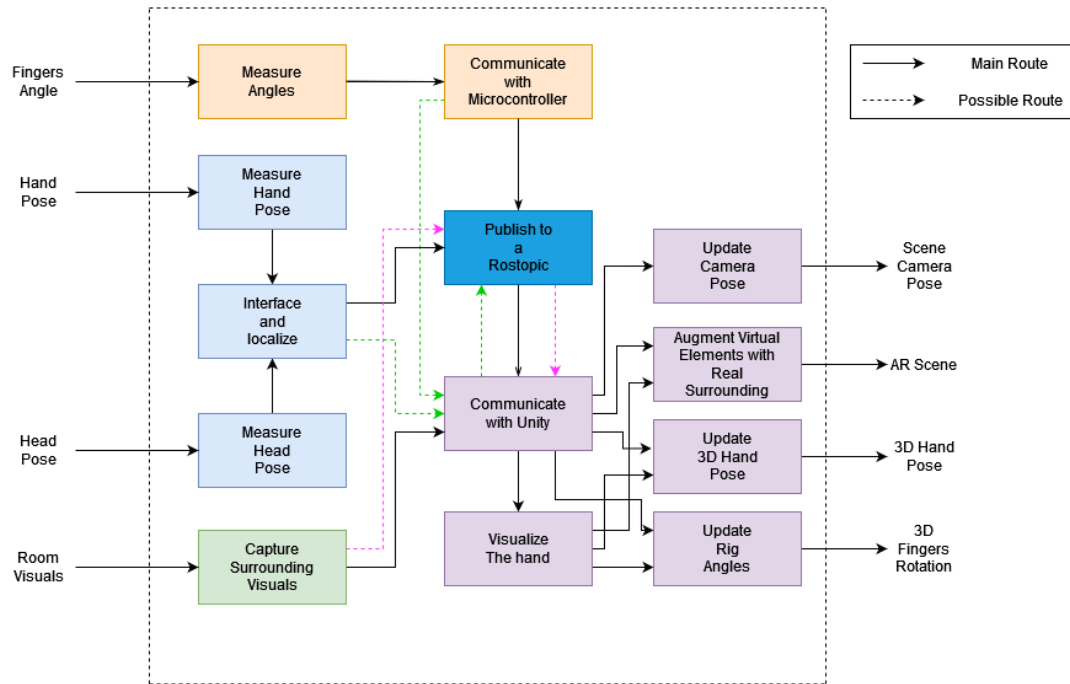


Figure 4: Functional Decomposition

Measuring Angles:

The assessment of finger rotation is presently conducted through the utilization of IMUs positioned near the joints. Alternative sensor types, including flex sensors, underwent testing, but IMUs emerged as the more preferable choice. The current IMU model in use is the BNO055, a widely adopted 9-axis sensor developed by Bosch. Another 9-axis sensor, the LSM9DS0, demonstrated proficiency in a comparable project [11]. The BMI160 is also noteworthy in discussions regarding data gloves [24].

An alternative method for tracking finger movement involves camera-based tracking, where markers are placed on the hand for recognition. In a study conducted in 2022 [25], a novel approach using genetic algorithms eliminated the need for physical markers, showcasing advancements in hand gesture recognition. Further advancements in this realm include the integration of machine learning and artificial intelligence to discern diverse and intricate hand gestures.

Close collaboration with the hardware team is essential to guarantee synchronization and compatibility between hardware and software components. Additionally, explorations into optimization alternatives were undertaken, including considerations such as the removal of the third IMU from the thumb while retaining the palm IMU for hand rotation. Another discussed

strategy involved connecting the IMUs in parallel, thereby reducing the number of multiplexers in the system.

Data Transfer:

Various methods were explored for publishing IMU data from the microcontroller to ROS. These approaches included consolidating all IMU data to one ROS topic after assigning IDs, serializing the data into a JSON document, or sending IMU data each to a topic. In an alternative route, IMU data could be transmitted from Unity to ROS instead.

For the extraction and publishing of Tracker and HMD pose data, external packages were employed. This published data serves varied components within the project. An additional pathway involves extracting pose data from Unity and subsequently publishing it to ROS eliminating these external packages.

Capturing the Real World:

While the eventual integration of the camera feed into ROS is planned, at present, it remains directly connected to the AR machine. The visual capture system employs the ZED M camera, selected for its superior image quality and precise depth-capturing capabilities. This choice is instrumental in ensuring accurate and high-quality visual data acquisition for the project.

Communicating with Unity:

In Unity, communicating with ROS topics can be achieved by creating a node through various plugins, such as ROS Bridge client, or ROS-TCP. These plugins establish a server-client connection between Unity and ROS for both publishing and subscribing (more in section [2.4.2](#)). The microcontroller's integration with Unity could also be facilitated through the independently developed package Ardity, establishing a direct connection and then publishing the IMU data to ROS thus reversing the route.

Visualizing the Hand:

The 3D model of the hand can be obtained through diverse methods, such as purchasing a model from the Unity Asset Store, utilizing 3D modelling software like Blender or Maya, or employing photogrammetry techniques. Notably, the approach in [6] involved the use of SolidCAD, a software tool for Computer-Aided Design (CAD), deviating from conventional industry practices. The resulting hand model featured separated sections to emulate a rig, a departure from the usual industry standard where animated 3D models are typically rigged to simulate a bone structure.

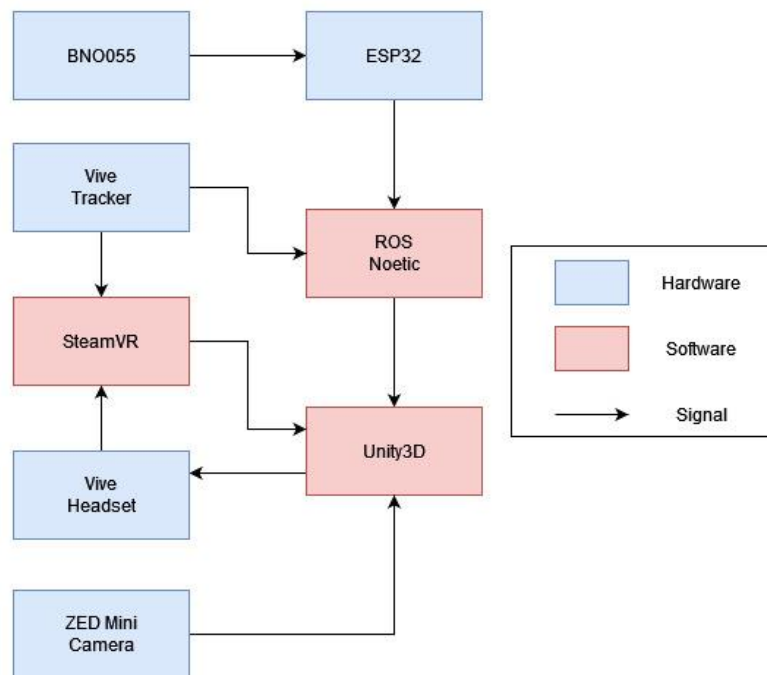


Figure 5: System Structure

Figure 5 illustrates the fundamental tools necessary for the system's operation. These tools are indispensable and, within the scope of this paper, are not subject to modification. However, the signal flow is indeed subject to potential alterations as shown in Figure 4. The subsequent chapter will delve into the development process, elucidating the rationale behind each decision made in relation to the different implementation methods outlined in this chapter.

Chapter IV: Development

4.1 Implementation

4.1.1 Software Infrastructure and Data Flow

As indicated in Section 1.4, the operational framework of the project initially involved Unity 2018. Pose data from the Vive tracker and headset were extracted through SteamVR, specifically OpenVR, using independently developed and outdated packages. More details on those packages can be found in [26] and [27]. An initiative to update the software infrastructure was undertaken to ensure future adaptability and leverage newer features.

In [6], the selection of Unity 2018 was influenced by compatibility issues with the SteamVR Unity plugin. The AR project development within Unity was facilitated by this plugin, as explained in Section 2.1.3. However, considering the continued developer backing and support for both tools in newer Unity versions, OpenXR, increasingly recognized as an industry standard for VR/AR projects in Unity (Section 2.1.4), was opted for. OpenXR is acknowledged within the community for its smooth operation and ease of use [16], also offering the option to bypass SteamVR, minimizing overhead [15] [14].

While certain challenges exist for the project with OpenXR, notably the absence of an official interaction profile for Vive trackers (partially addressed by using an independently developed profile) and the lack of OpenXR Unity package support on Linux platforms, these challenges are manageable. The decision to implement OpenXR in Unity was made under the context of working on Windows and utilizing WSL2 to install Ubuntu 20 and ROS Noetic. However, if the project is intended to run exclusively on Ubuntu, OpenVR would be the suitable choice instead of OpenXR. Regardless of the chosen XR tool, both OpenXR and OpenVR function effectively on newer Unity versions, influencing the decision to use Unity 2022.

For Unity to ROS communication, the conventional ROS# package was substituted with ROS TCP (Section 2.4.2), offering advantages in terms of speed and reliability elucidated in [21].

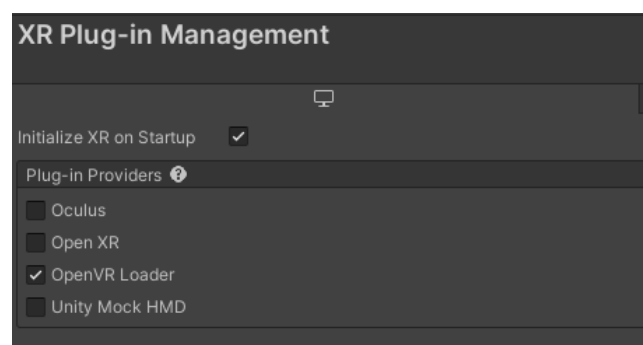


Figure 6: XR Plug-in Management in Unity

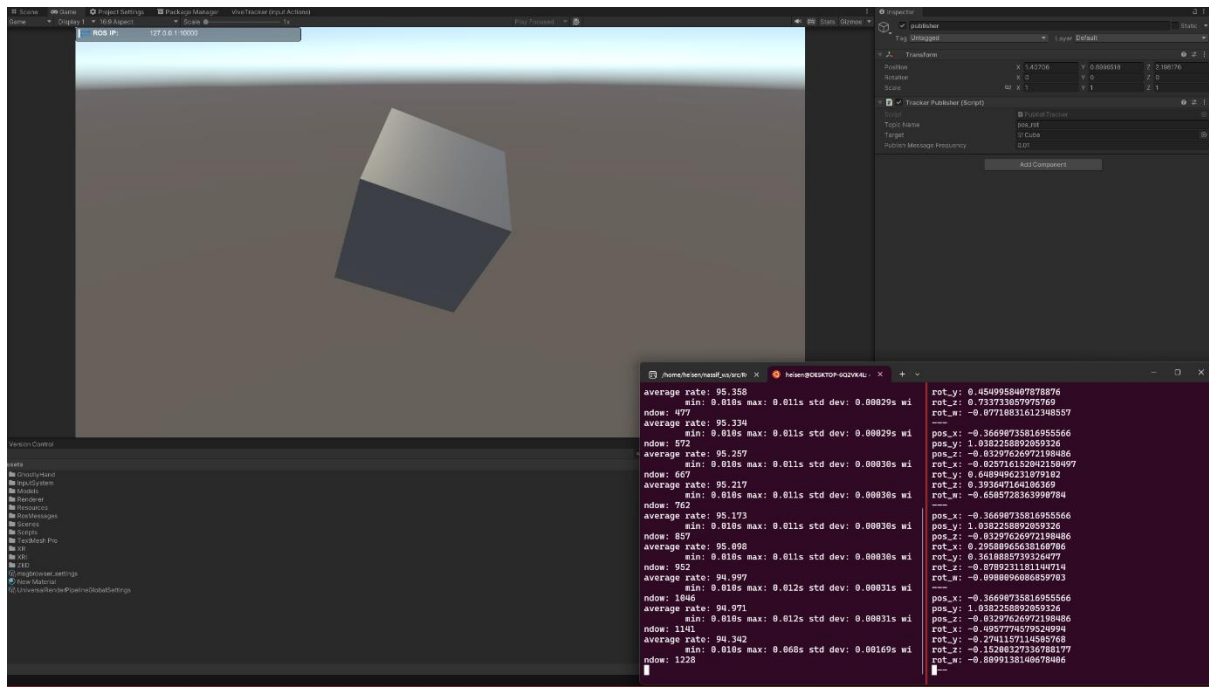


Figure 7: Publishing from Unity

Regarding the tracker and HMD pose data, an alternative data route was explored by transmitting pose data from Unity to ROS instead of the reverse, inversely reducing the publishing speed from 100Hz [26] to approximately 95Hz (Figure 7). However, this adjustment eliminates the need for extra third-party packages with numerous dependencies, contributing to an enhanced workflow and reduced overhead.

4.1.2 Optimizing Hand Movement Tracking System

Glove Integration

As explained in Section 1.4 and shown in Figure 1, the incorporation of the data glove into the Unity scene remains incomplete and riddled with errors. A comprehensive description of the data glove's current design is necessary to provide a clearer understanding of the situation. The glove is composed of 9 BNO055 breakout boards (without the pinkie finger), interconnected via standard wires to a primary PCB. This PCB facilitates the connection of the IMUs to a multiplexer and subsequently to the ESP32 Microcontroller.

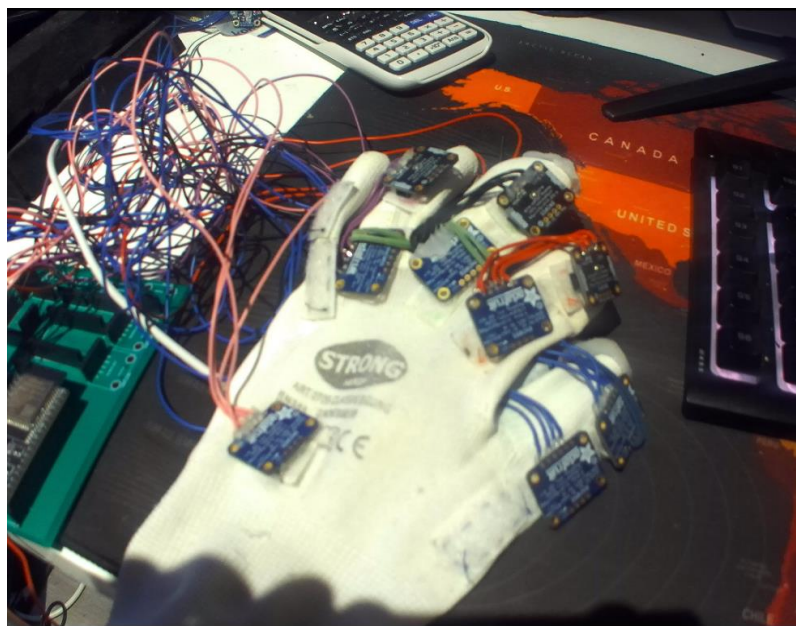


Figure 8: The Data Glove

The IMUs are affixed to a standard work glove using Velcro, a method that proves to be unstable and imprecise.

A significant issue identified is the incorrect movement of the virtual fingers when the data glove is used to make a fist. Instead of curling inward, the virtual fingers advance forward (Figure 1). By observing the angles within the Unity inspector, it becomes evident that the curling rotation, corresponding to the Z-axis in Unity coordinate frames, begins to decrease after reaching 90 degrees rather than continuing inward. This results in the virtual fingers rotating in the opposite direction after reaching 90 degrees, leading to a misalignment with the actual rotation of the data glove. A thorough examination of the IMU datasheet [28] reveals that this limitation is associated with the roll rotation of an IMU, which has a range of ± 90 degrees. This indicates that the curling of the fingers represents a roll rotation of the IMU. When a roll rotation reaches ± 90 degrees, the algorithms within the BNO055 readjust the other axes accordingly to simulate a continuous roll rotation. Therefore, if the IMU continued rotating after hitting ± 90 degrees the pitch and yaw axes will get an offset of 180 degrees to represent this rotation (Figure 11). Now the problem that occurs within Unity is with how the angles are applied and restricted. The previous strategy (Figure 9) to restrict the orientation of the fingers so they would rotate only around one axis, was to assign zeros to the other two axes. This meant that the aforementioned 180 degrees adjustment of the pitch and yaw axes will not occur in Unity and so when the roll rotation start increasing/decreasing after reaching ± 90 degrees, the object will change accordingly without adjusting the other two axes causing misalignment with the IMUs.

Table 1: Rotation angle conventions (from Bosch Sensortec)

Rotation Angle	Range (Android format)	Range (Windows format)
Pitch	+180° to -180° (turning clockwise decreases values)	-180° to +180° (turning clockwise increases values)
Roll	-90° to +90° (increasing with increasing inclination)	
Heading / Yaw	0° to 360° (turning clockwise increases values)	

The initial solution for this problem was to perform an axis remap, aligning the pitch rotation of the IMU, which has a range of ± 180 degrees, with the curling of the fingers. The pitch rotation causes no readjustment of the other two axes within the IMU and is more fitting to the curling of the fingers. This was but a half solution as it raised another critical issue concerning the rotation of the hand, which interfered with the rotation of the fingers.

```
transform.rotation = Quaternion.Euler(0, 0, msg.From<FLU>().eulerAngles.z);
```

Figure 9: The previous approach for locking rotations

As seen in Figure 10, the pitch rotation which is around Z is still adjusting and changing around 180 degrees when the IMU is rolled, which is exactly what happens when the hand itself is

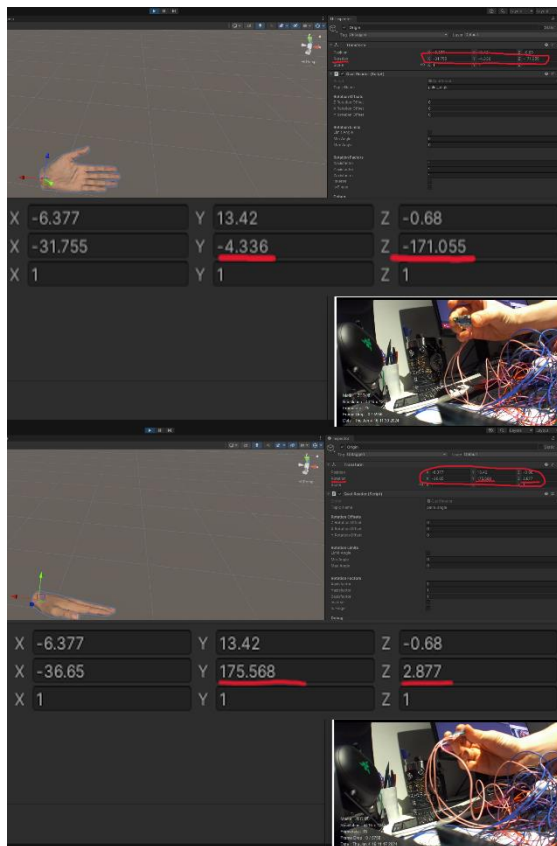


Figure 11: Pitch and yaw (Y and Z) adjusting with the roll rotation(X)

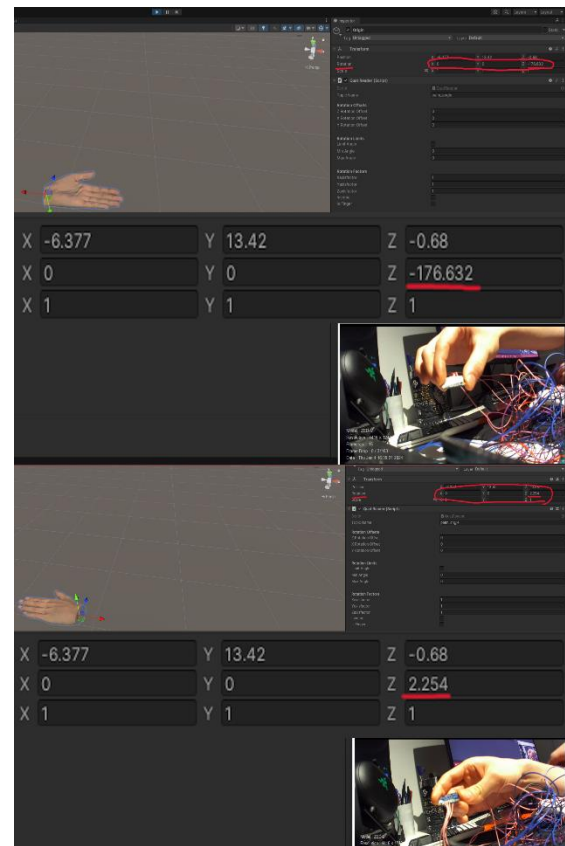


Figure 10: Pitch (Z) adjusting with the roll rotation(X)

turned. Since the other two axes are assigned zeros, there is no way for Unity to know that this actually is an adjustment related to the roll rotation, which means that the object will flip 180 degrees.

When the IMUs rotate in real space, the objects representing the fingers must retain this rotation around all axes in their global variable. This way Unity would be able to recognize axes adjustments made by the IMU. The restriction of rotation around the X and Y axes should only be applied locally, between the parent and child objects. This was precisely the approach taken to achieve alignment between hand rotation and finger rotation. The quaternions were applied to the global orientation of the fingers, and the local Euler angles were then accessed and had their X and Y axes reassigned to zero as shown in Figure 12.

```

/// <summary>
/// Handles the quaternion changed event and applies the rotation to the game object.
/// </summary>
/// <param name="msg">The quaternion message.</param>
1 reference
void QuaternionChanged(QuaternionMsg msg)
{
    transform.rotation = Quaternion.Euler(msg.From<FLU>().eulerAngles.x + xRotationOffset, msg.From<FLU>().eulerAngles.y + yRotationOffset, msg.From<FLU>().eulerAngles.z + zRotationOffset);
    Vector3 tempRotation = transform.localEulerAngles;
    tempRotation.x = 0;
    tempRotation.y = 0;
    transform.localEulerAngles = tempRotation;
}

```

Figure 12: The new approach of applying angles to an Object

This solution necessitated the use of an additional palm IMU to rotate the hand instead of the tracker. This IMU could have its axis easily remapped to match the fingers' IMUs, unlike the Vive tracker, thereby simplifying testing and debugging.

This leads us to another aspect, namely the frame of reference for the IMUs and tracker. The BNO055 offers various operational modes, allowing the user to select among the three types of sensors within the IMU: Accelerometer, Magnetometer, or Gyroscope. The fusion mode combines two or three of these types to derive orientation data.

Table 2 BNO055 operation modes overview (from Bosch)

Operating Mode		Available sensor signals			Fusion Data	
		Accel	Mag	Gyro	Relative orientation	Absolute orientation
	CONFIGMODE	-	-	-	-	-
Non-fusion modes	ACCONLY	X	-	-	-	-
	MAGONLY	-	X	-	-	-
	GYROONLY	-	-	X	-	-
	ACCMAG	X	X	-	-	-
	ACCGYRO	X	-	X	-	-
	MAGGYRO	-	X	X	-	-
	AMG	X	X	X	-	-
Fusion modes	IMU	X	-	X	X	-
	COMPASS	X	X	-	-	X
	M4G	X	X	-	X	-
	NDOF_FMC_OFF	X	X	X	-	X
	NDOF	X	X	X	-	X

The currently employed operation mode is the NDOF fusion mode, which combine all three sensors to determine orientation. This mode compensates for gyroscope drift and provides a high output rate, but the magnetometer is vulnerable to magnetic noise. The following is a note extracted from the BNO055 datasheet: “*The sensor fusion algorithm uses the accelerometer data to compensate for the gyroscope drift over time. When the device is in motion, the accelerometer data is temporarily ignored and the fusion relies on the gyroscope for pitch and roll. If the accelerometer data cannot be used for an extended period of time (e.g., due to vibration, or to constant movement), then this may cause the pitch and roll values to drift over time. Likewise, if the magnetometer data is detected to be distorted, it will automatically be ignored by the algorithm and this may cause the heading to drift over time (as in IMU mode)*”. A significant feature of the NDOF mode is its calculation of absolute

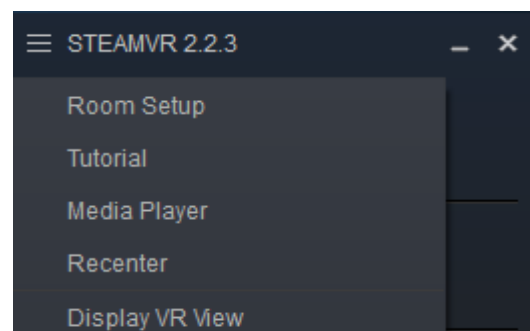


Figure 13: SteamVR room setup

orientation in relation to the Earth's magnetic pole, while the Vive Tracker computes relative orientation in relation to the setup rotation in SteamVR. This implies that the tracker and the fingers operate within different frames of reference, which can lead to misalignment if not addressed. Another mode for the IMU that obtains relative orientation was found to be highly unstable. The implemented solution varies depending on whether the tracker or the palm IMU is used for hand orientation. For the palm IMU: If the palm IMU is utilized, the hand and the fingers will naturally align, as they share the same frame of reference (the Earth's magnetic pole). However, this also creates a disconnection between the headset and the hand. The most straightforward countermeasure would be to alter the frame of reference of the Vive set by initializing room setup in SteamVR and aligning the reference with the North Pole. An alternative approach involves conducting a calibration phase at the outset. This calibration involves capturing the IMUs' quaternions at a specific moment and using this snapshot as the new reference by multiplying its inverse with the current quaternions like the following equation:

$$Quaternions_{relativ} = Quaternions_{reference}^{-1} * Quaternions_{current}$$

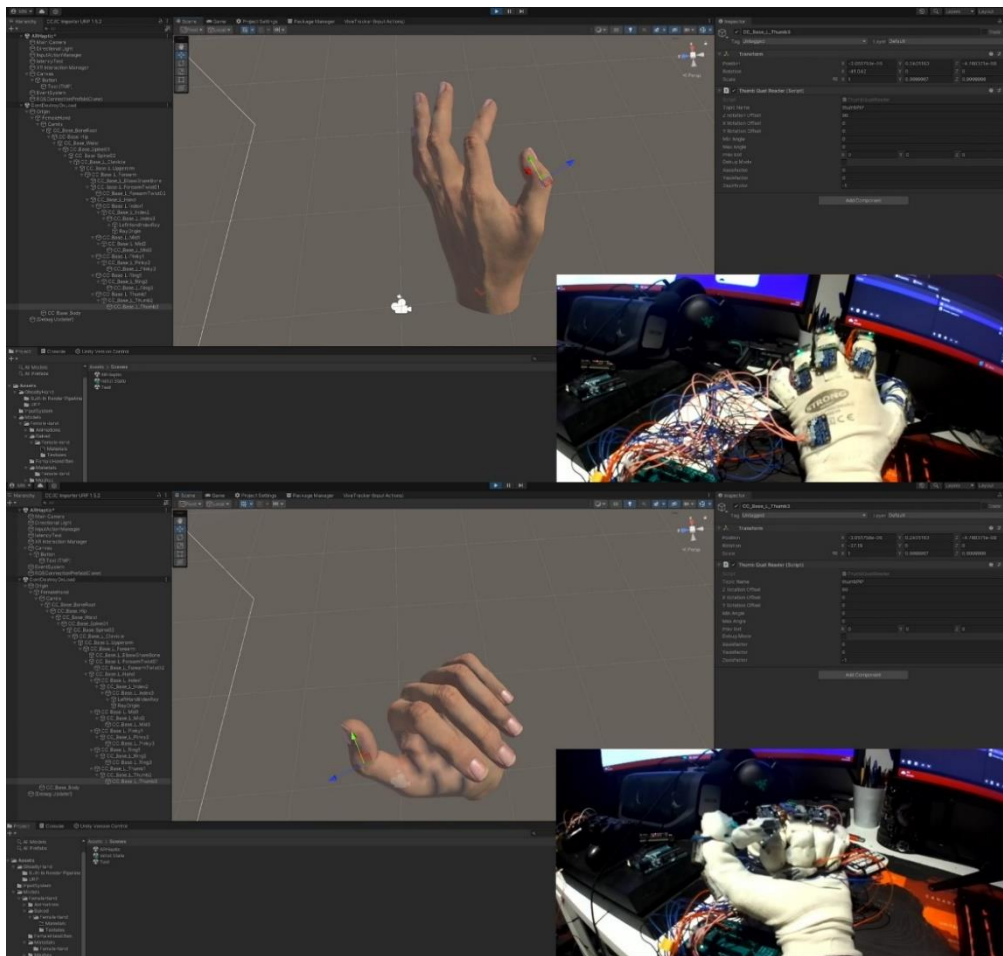


Figure 14: Functional hand tracking

For the tracker: There will be no disconnection between the tracker and the headset as they share the same frame of reference, but the physical zero orientation of the tracker could be challenging to mount on the glove. A similar approach to the palm IMU could be employed to address this, where the tracker's quaternions are captured and then used as a reference. This calibration could be performed within Unity by pressing a UI button. If the calibration data is needed elsewhere, the calibration phase could be conducted outside Unity, and a ROS service could be established. This service, when invoked, will capture a quaternion of a topic at a specific moment and return it to the caller.

With these implementations, the hand is now fully integrated and capable of complete rotation without compromising the rotation of the fingers as illustrated in Figure 14.

Hand model and visualization

The process of visualizing the hand within the Unity3D engine involved importing a 3D hand model, distinct from its predecessor in various aspects. This new model is constructed as a mesh of vertices and faces, meticulously forming the anatomical contours of a female hand. A realistic skin texture is applied to enhance its visual fidelity. Furthermore, an extra shader has been introduced, allowing users the flexibility to toggle between a realistic hand representation and a more translucent, ghost-like alternative. The potential for incorporating additional texture options remains an avenue for future exploration.

A pivotal divergence between the preceding and current hand models lies in the incorporation of a bone rig within the latter. This structural addition facilitates the simulation of a skeletal framework, contributing to the seamless and lifelike animation of hand movements. This strategic departure from the prior approach, which involved linking multiple objects, has been instrumental in mitigating issues of rigidity and susceptibility to visual glitches.

The implementation process involves affixing a singular script to the relevant joints. This script applies IMU quaternions, adhering to the parameters outlined in the glove integration section.

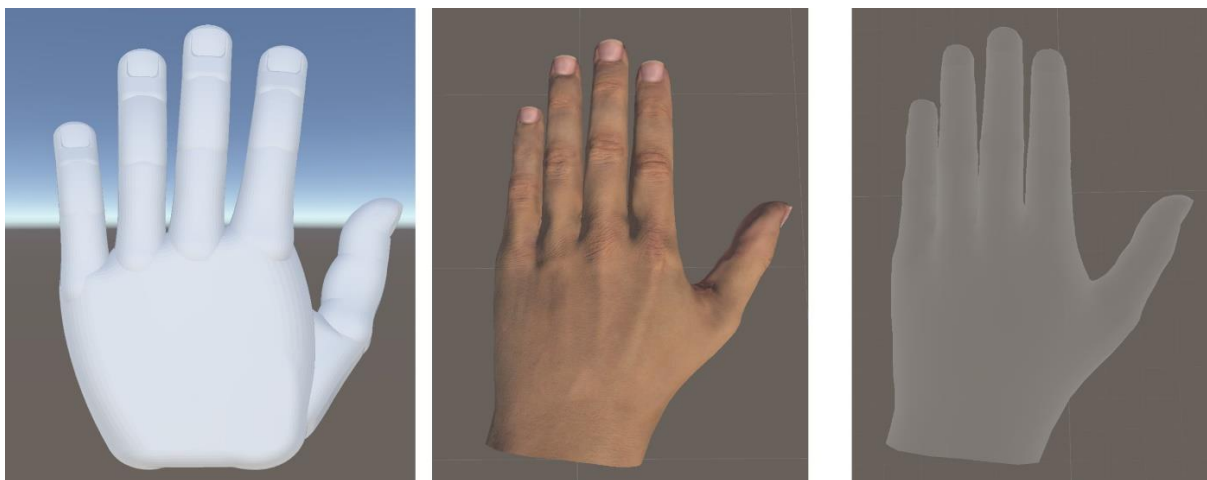


Figure 15: Hand models from left to right: old, new with a realistic mesh, new with a transparent mesh

Latency, publishing speed and codebase efficiency

The initial motivation for this thesis stemmed from the objective of enhancing the responsiveness of hand visualization to real hand movements. This objective is set to be accomplished by accelerating the data transmission rate and enhancing data flow. The following discussion will explain the code structure, outline the implemented changes, and provide rationale for their adoption.

Upon the microcontroller's initialization, the loop function commences. Within this function, the IMUs are sequentially accessed via their respective channels on the multiplexer. Upon accessing an IMU, its quaternion data is stored within a quaternion array. After populating the array with the quaternion data, each array element is published to an individual ROS topic, resulting in nine topics for nine IMUs. As detailed in Section 1.4, the publishing rate measured at 15Hz for each topic. Additionally, a Python script in ROS was needed for the project to launch. This script converts the quaternions to Euler angles and publishes them to new topics.

To determine the appropriate publishing rate, an estimation of the speed of finger movement is

required. The world record for typing speed is approximately 750 keystrokes per minute [29] or characters per minute (CPM), equating to roughly 12.5 Hz. While this is an extreme example and the average is below 10 Hz, a value of 13Hz will be assumed for conservative reasons as a proper estimation of rapid finger motion. According to the Nyquist–Shannon sampling theorem, to accurately capture and reconstruct a continuous analog signal into a digital form without any loss of information, the sampling rate must be at least twice the highest frequency present in the signal. Therefore, to accurately reconstruct the motion of the finger into a digital form, a sampling rate of at least 26Hz is required. This indicates that a 15Hz transfer rate would not be sufficient to accurately represent hand motions.

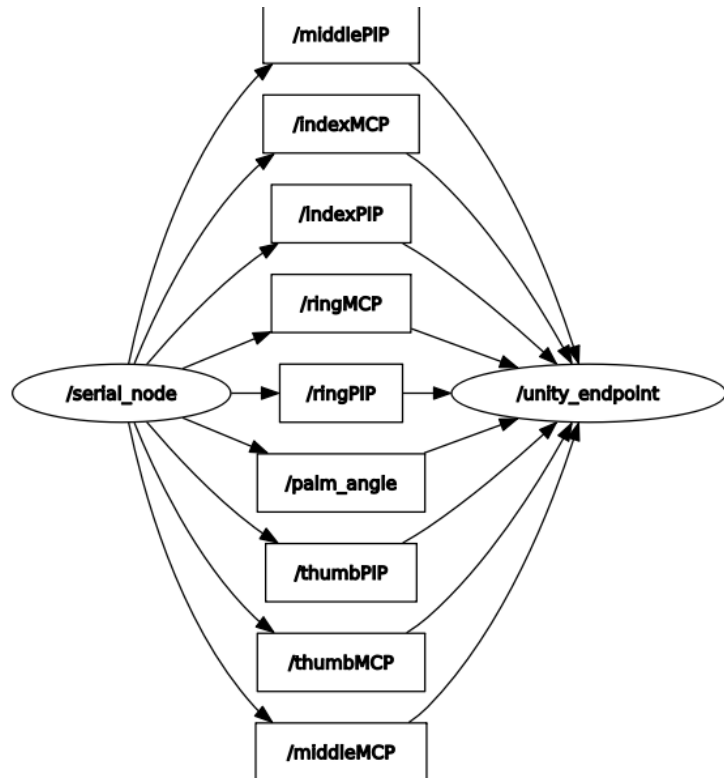
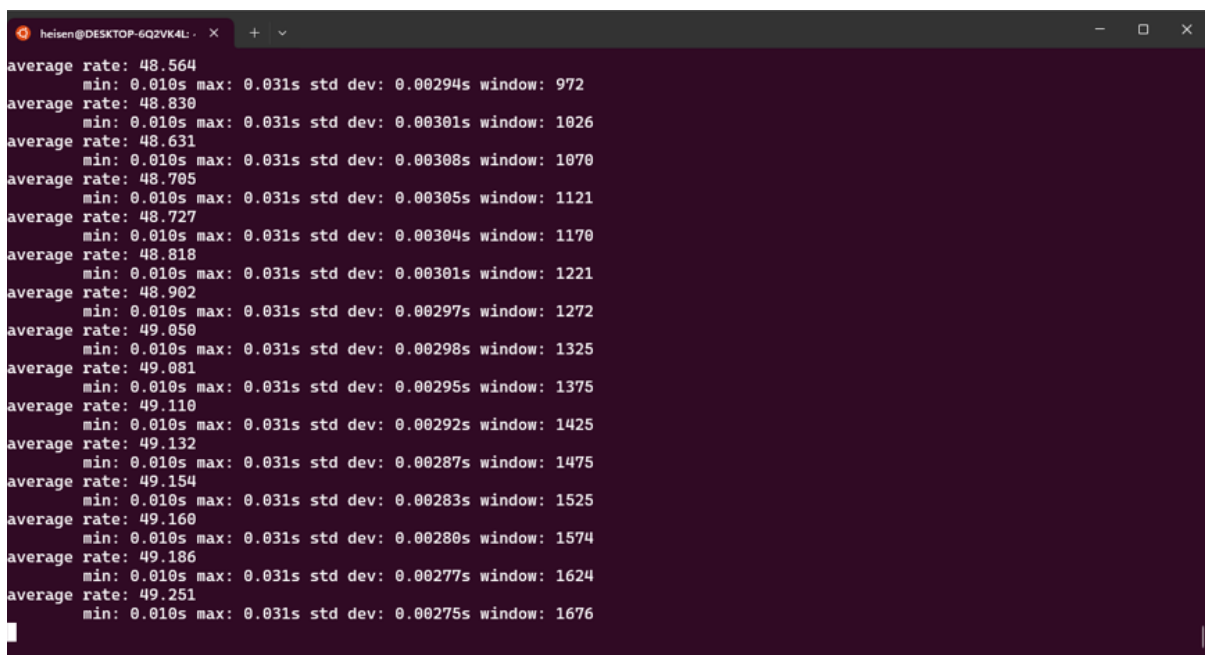


Figure 16: Data route within ROS

Tests were conducted to calculate latency and determine the publishing rate by measuring individual tasks within the code using the `millis()` function, outputting milliseconds since microcontroller initiation. The task of extracting quaternion data and saving it to the array averaged around 13ms, while publishing the quaternion array took approximately 6ms. Consequently, each iteration of the loop function should ideally take about 19ms, corresponding to a rate of approximately 52Hz. The realization that the data is instead reaching its topic at a rate of 15Hz raised the question of a bottleneck regarding the UART connection between the microcontroller and the machine hosting ROS and the inability to handle the volume of the data being transferred. To further substantiate this point, instead of sending quaternion messages which consist of four float64 variables, a new custom Vector3 message was created that consists of three unsigned int8 variables, for the purpose of sending rounded Euler data with a maximum error of 1 degree. This increased the publishing rate to about 25Hz. It was then determined that the Baud rate needed to be increased to account for the volume of the data. Increasing the Baud rate to around 700000 bps from the previous 57000 bps had the desired effect of increasing the publishing rate to 50Hz on average. Further increasing the Baud rate won't make a difference since the bottleneck now lies with reading of the sensors but according to the above estimated baseline of 26Hz, a result of 50Hz is sufficient.



```
heisen@DESKTOP-6QZVK4L: X + -
average rate: 48.564
min: 0.010s max: 0.031s std dev: 0.00294s window: 972
average rate: 48.830
min: 0.010s max: 0.031s std dev: 0.00301s window: 1026
average rate: 48.631
min: 0.010s max: 0.031s std dev: 0.00308s window: 1070
average rate: 48.705
min: 0.010s max: 0.031s std dev: 0.00305s window: 1121
average rate: 48.727
min: 0.010s max: 0.031s std dev: 0.00304s window: 1170
average rate: 48.818
min: 0.010s max: 0.031s std dev: 0.00301s window: 1221
average rate: 48.902
min: 0.010s max: 0.031s std dev: 0.00297s window: 1272
average rate: 49.050
min: 0.010s max: 0.031s std dev: 0.00298s window: 1325
average rate: 49.081
min: 0.010s max: 0.031s std dev: 0.00295s window: 1375
average rate: 49.110
min: 0.010s max: 0.031s std dev: 0.00292s window: 1425
average rate: 49.132
min: 0.010s max: 0.031s std dev: 0.00287s window: 1475
average rate: 49.154
min: 0.010s max: 0.031s std dev: 0.00283s window: 1525
average rate: 49.160
min: 0.010s max: 0.031s std dev: 0.00280s window: 1574
average rate: 49.186
min: 0.010s max: 0.031s std dev: 0.00277s window: 1624
average rate: 49.251
min: 0.010s max: 0.031s std dev: 0.00275s window: 1676
```

Figure 17: The new publishing rate

To accommodate the modifications detailed in the glove integration section, several lines of code were added. The code within the microcontroller was refined to enhance readability, and debugging is now confined to the ROS terminal, mitigating interference with the ROS connection caused by the serial monitor.

In conclusion, the optimization of the hand movement tracking system involved overcoming challenges in the integration of a data glove into the Unity scene. Addressing issues such as

instability in finger movement representation and problematic hand rotation, solutions were implemented, including an axis remap for alignment and a proper implementation of the IMU angles.

The visualization of the hand model in Unity3D saw significant improvements with the introduction of a realistic 3D hand model featuring a bone rig for lifelike animation. The ability to toggle between realistic and translucent representations adds versatility.

Efforts to enhance system efficiency and responsiveness involved a systematic analysis of finger movement speed, sampling rates, and communication bottlenecks. Strategic adjustments resulted in a substantial increase in the publishing rate, exceeding the baseline requirement of 26Hz.

4.1.3 Integrating the User Interface

The creation of a user interface is crucial to facilitate communication between the user and the system. The design and function of this interface are intended to be basic and minimal, serving as a blueprint for future development. The following section discusses the functionality and design of this interface. Essential functions such as the ability to exit the program, change settings, and access helpful information needed to be incorporated into the interface. Additional project-related functions, such as displaying sensor data, the ability to record voice notes, or accessing blueprints and designs, were considered. Some of these functions were implemented, while others were not due to time constraints, but they remain as ideas for future developers. As a result of progress in other areas, additional UI functions were added, such as a button to calibrate the reference angle and a button to switch between different textures.

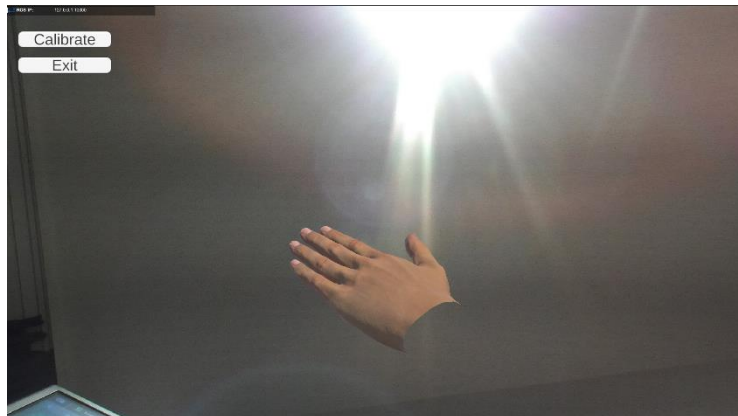


Figure 18: User Interface

In Unity, adding UI elements is relatively straightforward. However, the challenge lies in enabling the user to interact with them in an augmented reality environment. Given the project's lack of controllers for user input, a creative solution was required. To facilitate interaction with the UI elements, a ray caster was added to the tip of the index finger, and a collision box was added around the UI element, which is initialized as a world object. This setup allows for the detection of when the ray emitted from the index finger intersects with the UI elements. To enable the modification of the UI element (to press or drag), additional collision capsules were added to the thumb and index finger. Upon collision between them, the ray sends a signal to the UI element. This method allows the user to interact with UI elements without using external devices, but by making a specific hand gesture so the two collision capsules would collide.

4.2 System Validation

The implementations executed within this project required validation to be deemed successful, particularly in light of the extensive criteria established in Section 3.1. This section will explore various testing methodologies and explain the rationale behind the selected approaches.

4.2.1 Latency and Accuracy

As previously indicated, the latency and responsiveness of the hand tracking system were the primary motivations for this project. The latency of individual functions was assessed by employing the `millis()` timing function and comparing timestamps at the initiation and conclusion of each function. Yet, the latency of the connection itself still needed measurement and presented a challenge, as timestamps within the microcontroller could not be directly compared with those in Rospy or Unity due to misaligned time references. Consequently, a unified time reference was required across the different software environments. Initially, a clock node was implemented to publish the current Unix time, which is the number of nanoseconds since January 1, 1970. Nodes, such as the microcontroller or Unity, would subscribe to this clock and utilize its time for timestamping messages or for comparing received timestamps with the current time to determine latency. However, this method introduced a complication: the latency between the nodes and the clock node itself. Considering the following equation:

$$\begin{aligned} \text{latency} = & \text{timestamp}_{\text{received}} \\ & + \text{connection_latency} \\ & - (\text{timestamp}_{\text{sent}} \\ & + \text{connection_latency}) \end{aligned}$$

it becomes apparent that this method would only yield the latency between the timestamps if the connection latencies to the clock node were equal and thus cancelled each other out. Unfortunately, this was not the case, as the connection speed between Unity and the clock node was significantly faster and did not provide the necessary compensation. While this method might be advantageous for other applications and remains an available option, an alternative was chosen for this project.

The decision was made to utilize the `micros()` timing function within the ESP32 as the sole time reference for calculating the

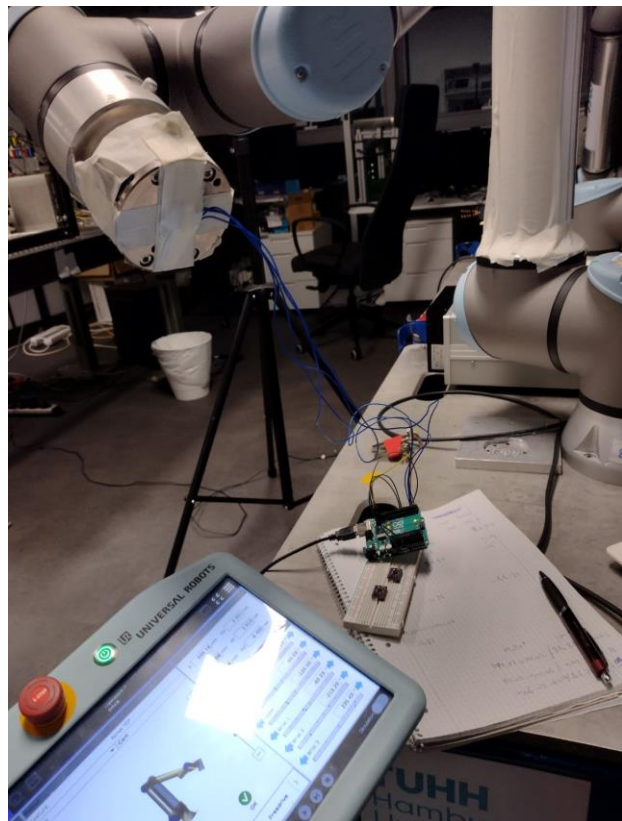


Figure 19: Accuracy test process

round-trip latency of the connection between the ESP32 and Unity. This was accomplished by sending a timestamped message from the microcontroller, republishing it from Unity upon receipt, receiving the message back within the microcontroller, and comparing its timestamp with the current time. The total latency was then determined as the sum of all individual latencies, as illustrated by the following equation:

$$Total\ latency = latency_{reading_sensors} + latency_{publishing} + \frac{latency_{roundtrip}}{2}$$

Accuracy tests were also performed to evaluate the precision of the IMUs. This was achieved by positioning the BNO055 on a universal robot and comparing the robot arm's angles with the sensor readings. Figure 19 showcases the testing process.

The findings will be further examined in Chapter V.

4.2.2 User Evaluation

While numerous facets of this project lend themselves to quantification, user experience remains inherently subjective, necessitating the execution of a user feedback survey. Ensuring a diverse pool of test participants was also crucial to account for a range of perspectives and experiences. The survey comprised the following questions, with responses ranging from 1 (strongly disagree) to 5 (strongly agree):

1. Have you previously used virtual reality applications?
2. Have you previously used augmented reality applications?
3. Is video gaming one of your primary hobbies?
4. Did you find the environment immersive?
5. Was the virtual hand responsive to your movements?
6. Were the visuals accurate representations of your gestures?
7. Did you find the setup comfortable and intuitive?
8. Additional comments?

The initial three questions aimed to assess the user's prior experience, which could significantly influence their perceptions and subsequent responses. The fourth question was designed to evaluate the user's level of engagement within the augmented reality environment. The purposes of questions five and six are self-evident. The seventh question sought feedback on the setup logistics, though it should be noted that the hardware used for this survey may be outdated, with more comfortable designs likely available in the future. The survey results will be further analyzed in Chapter V.

Chapter V: Findings and Discussion

5.1 Test Results

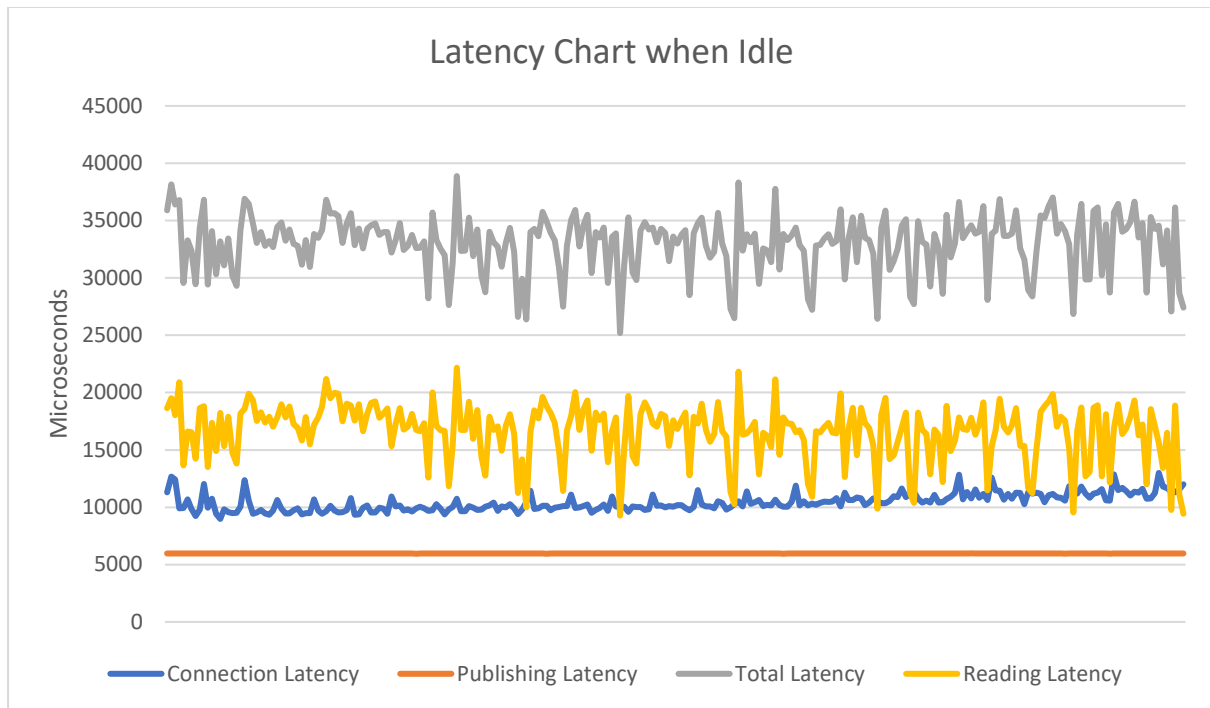


Figure 20: Latency Chart when idle

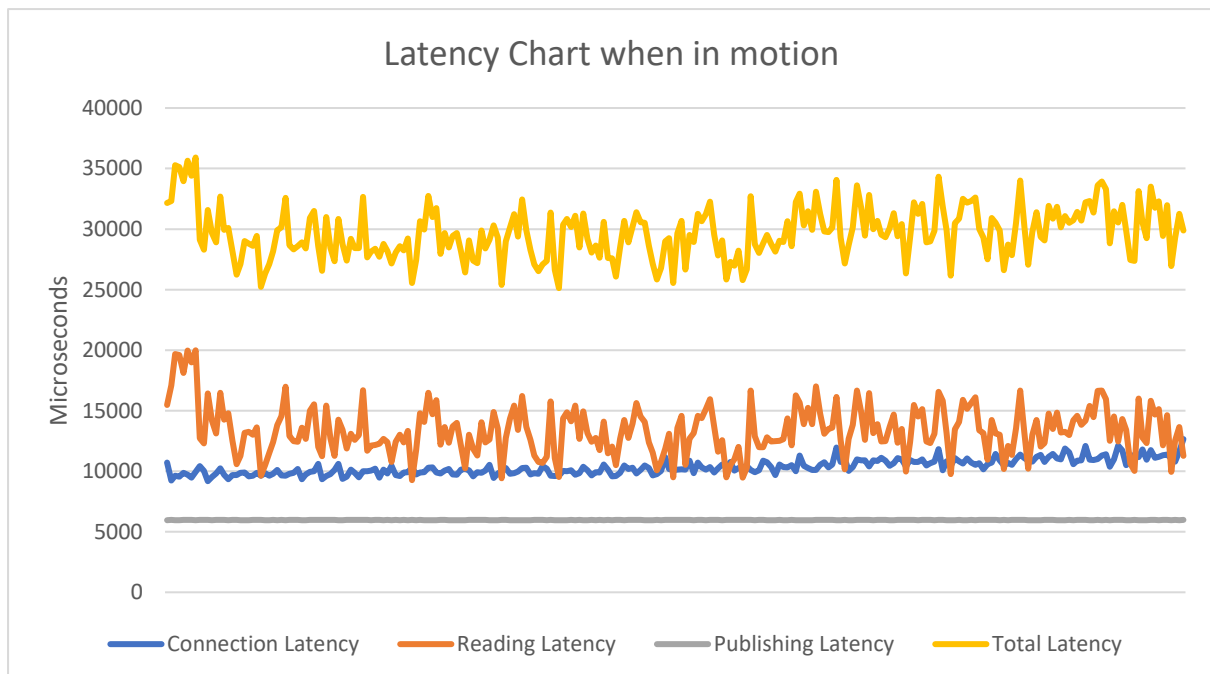


Figure 21: Latency Chart when in motion

Figure 20 and Figure 21 illustrate that the total latency averages approximately 30ms during motion and around 33ms in idle mode. The publishing latency remains relatively constant at around 6ms, and the connection latency from the microcontroller to Unity averages

5.2 Questionnaire

The evaluation phase yielded valuable insights into the system's operation and corroborated the findings of this study. Ten individuals participated in the testing phase, the majority of whom identified gaming as a primary hobby. As noted in Section 4.2.2, this suggests that these participants have some familiarity with computer graphics and can form independent opinions on visual fidelity. Only one participant had extensive experience with VR, considering it a major hobby. The remaining participants had limited VR exposure, with two having no prior experience. Interestingly, but not unexpectedly, all but one participant had no experience with augmented reality, with the exception having played the mobile game Pokemon Go. The participants responded to the evaluation questions as follows:

1. Did you find the environment immersive? Average score: 3.5/5
2. Was the virtual hand responsive to your movements? Average score: 4.7/5
3. Were the visuals accurate representations of your gestures? Average score: 4.2/5
4. Did you find the setup comfortable and intuitive? Average score: 2.9/5

The system received high scores for responsiveness and accuracy, with all responses ranging between 4 and 5, thus affirming the statistics and results of this study. The immersion score was moderate at 3.5, with many participants complimenting the realistic hand model but noting occasional jittering of the hand position, which disrupted the immersive experience. Some participants expressed a desire for more freedom of movement with the glove to enhance immersion, but this was not feasible due to the setup's wiring. This leads to the final question regarding comfort and intuition, which received the lowest average score of 2.9. Most participants felt their movements were restricted due to the numerous wires connected to the glove and the improper mounting of the tracker on the glove, which affected their comfort.

In conclusion, this testing phase offered a fresh perspective on the system, validated the results, and identified areas for future improvements.

Chapter VI: Conclusion and Future Works

This paper represents a comprehensive endeavor to optimize the AR environment and the hand tracking system. The system's infrastructure and data transmission pathways underwent a significant overhaul, resulting in substantial improvements in responsiveness and accuracy, and ensuring adaptability to future advancements. The restoration of the complete functionality of the data glove in the AR environment was achieved through a reconstructed approach for applying finger joint rotations. The user experience was further enhanced by the introduction of a rigged hand model and the option to toggle between different textures. An initial user interface was established, laying the groundwork for future development in this area.

With an average total latency of 30 milliseconds and a publishing rate of 50Hz, the baseline requirements were met if not exceeded. The positive responses from the user evaluation validate the system improvements achieved within the scope of the project. However, the survey also highlighted areas requiring further refinement, including mounting stability and intuitive interaction.

During the development process, several potential avenues for further investigation were identified. These include leveraging the dual-core feature of the ESP32 microcontroller to achieve parallel multitasking, thereby further enhancing the system's performance, specifically the extraction and transmission of the IMU data. Another potential area of exploration is the integration of camera-based tracking in addition to the glove, thus creating a multi-model system. Such a system was researched in [25], where a generative method (genetics algorithm) was implemented to recover hand pose using a data sensor glove for initialization.

One issue that impacted the comfort of the glove was the use of the Vive tracker, which proved challenging to mount due to its large size. Therefore, the development of a custom tracker for hand pose could be explored. This would also necessitate the development of a custom localization scheme to synchronize the device with the lighthouses. This and the ability to bypass SteamVR using OpenComposite and OpenXR would make the system more customized and efficient.

In conclusion, this thesis successfully fulfilled its central objectives of optimizing and integrating key components of the AR environment, while ensuring its forward compatibility. The outcomes provide a solid foundation for further innovation in this interdisciplinary research domain.

References

- [1] M. Rüßmann, "Industry 4.0: The Future of Productivity and Growth in Manufacturing Industries," April 2015. [Online]. Available: <https://shorturl.at/jtBH3>.
- [2] J. P. Theune, "Optimization, Control and Calibration of a Haptic Feedback Glove for Usage in a Remote Inspection System," May 2023. [Online]. Available: <https://shorturl.at/frstV>.
- [3] A. Strachatov, "Optimization and PCB Design of a Haptic Feedback Glove to be Used in a Remote Inspection System," July 2023. [Online].
- [4] O. Brahem, "Optimization of haptic glove force feedback mechanism to model material softness in the context of a remote inspection system," September 2023. [Online]. Available: <https://shorturl.at/enowR>.
- [5] M. ELkholy, "Enhancing a ROS digital twin and Implementing hardware interface between different componenets of a remote inspection system," July 2023. [Online]. Available: <https://shorturl.at/pvwAO>.
- [6] A. Aboelela, "Optimizing of VR/AR Environment and Augmenting Haptic Feedback Glove," July 2023. [Online]. Available: <https://shorturl.at/jnsy5>.
- [7] S. Ehab, "Design, Implementation, Control and Calibration of a Mechanism to Mimic the Human's Head Movement in a Remote Inspection System," May 2023. [Online]. Available: <https://shorturl.at/ipEMZ>.
- [8] L. C. T. N. Arne Seeliger, "Augmented reality for industrial quality inspection: An experiment assessing task performance and human factors," October 2023. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0166361523001355#b20>.
- [9] S. Howard, "Visual inspection with augmented reality head-mounted display: An Australian usability case study," March 2023. [Online]. Available: <https://onlinelibrary.wiley.com/doi/10.1002/hfm.20986>.
- [10] R. D. Robin Denz, "A high-accuracy, low-budget Sensor Glove for Trajectory Model Learning," December 2021. [Online]. Available: <https://ieeexplore.ieee.org/document/9659403>.
- [11] B.-S. Lin, "Design of an Inertial-Sensor-Based Data Glove for Hand Function Evaluation," March 2018. [Online]. Available: <https://www.mdpi.com/1424-8220/18/5/1545>.
- [12] J. Camp, "Review: HTC Vive Pro," May 2018. [Online]. Available: <https://www.wired.com/review/review-htc-vive-pro/>. [Accessed January 2024].
- [13] A. Wheeler, "Open XR: A Call for Standardization," March 2019. [Online]. Available: <https://www.engineering.com/story/open-xr-a-call-for-standardization>.
- [14] D. Heaney, "Virtual Desktop's New OpenXR Runtime Bypasses SteamVR To Boost Performance," November 2023. [Online]. Available: <https://www.uploadvr.com/virtual-desktops-vdxr-runtime/>.
- [15] "Steam VR vs OpenXR: Which Runtime is Best?," [Online]. Available: <https://pimax.com/steam-vr-vs-openxr-which-runtime-is-best/>.

- [16] Warwick, "OpenXR Vs OpenVR | Differences, Similarities, And Development Platforms That OpenXR And OpenVR Support," [Online]. Available: <https://sodeni.com/openxr-vs-openvr/>.
- [17] "SteamVR Lighthouse," [Online]. Available: <https://www.vrs.org.uk/virtual-reality-gear/motion-tracking/steamvr-lighthouse.html>.
- [18] AmandaDattalo, "ROSIntroduction," 2018. [Online]. Available: <http://wiki.ros.org/ROS/Introduction>.
- [19] "Unity-Robotics-Hub," [Online]. Available: <https://github.com/Unity-Technologies/Unity-Robotics-Hub/tree/main>. [Accessed 2023].
- [20] D. M. Bischoff, "ros-sharp," [Online]. Available: <https://github.com/siemens/ros-sharp>. [Accessed February 2023].
- [21] J. Allspaw, "Comparing Performance Between Different Implementations of ROS for Unity," 2023. [Online]. Available: <https://openreview.net/pdf?id=WH3yhsbBjj>. [Accessed 2023].
- [22] J.-P. Stauffert, "Latency and Cybersickness: Impact, Causes, and Measures. A Review," 2020. [Online]. Available: <https://www.frontiersin.org/articles/10.3389/frvir.2020.582204/full>. [Accessed 2023].
- [23] S. Esmaeili, "Detection of Scaled Hand Interactions in Virtual Reality: The Effects of Motion Direction and Task Complexity," March 2020. [Online]. Available: <https://ieeexplore.ieee.org/document/9089480>. [Accessed 2023].
- [24] C. Bagdas, "Data-glove based on IMU Sensors - Prototyp Version 2," January 2022. [Online]. Available: <https://tams.informatik.uni-hamburg.de/lectures/2021ws/seminar/tams/doc/bsc-slides-can-bagdas-data-glove.pdf>. [Accessed 2023].
- [25] L. Jiang, "A Model-Based System for Real-Time Articulated Hand Tracking Using a Simple Data Glove and a Depth Camera," October 2019. [Online]. Available: <https://www.mdpi.com/1424-8220/19/21/4680>. [Accessed 2023].
- [26] D. Arnett, "vive_tracker," 2017. [Online]. Available: https://github.com/moonwreckers/vive_tracker. [Accessed 2023].
- [27] robosavvy, "vive_ros," 2018. [Online]. Available: https://github.com/robosavvy/vive_ros. [Accessed 2023].
- [28] B. Sensortec, "BNO055," October 2021. [Online]. Available: <https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bno055-ds000.pdf>. [Accessed 2023].
- [29] "recordholdersrepublic," [Online]. Available: <https://www.recordholdersrepublic.co.uk/world-record-holders/43/barbara-blackburn.aspx>. [Accessed 2023].

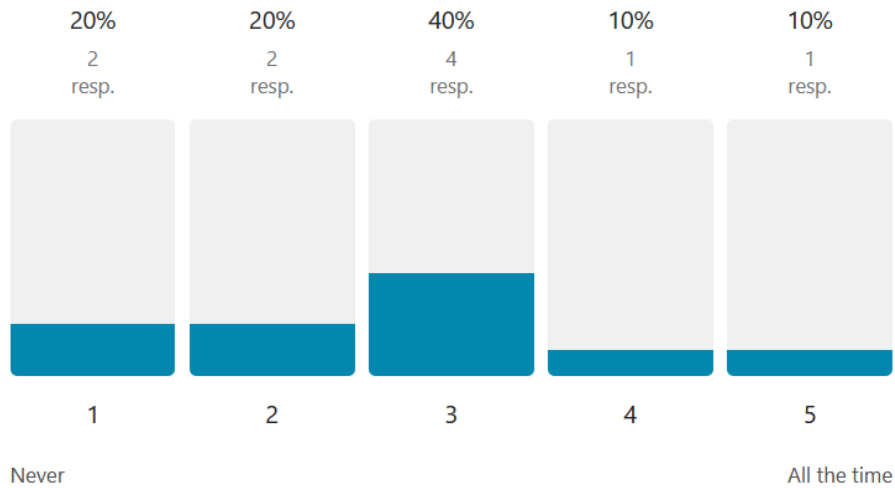
Appendix A: Questionnaire



Have you previously used virtual reality applications?

Avg. 2.7

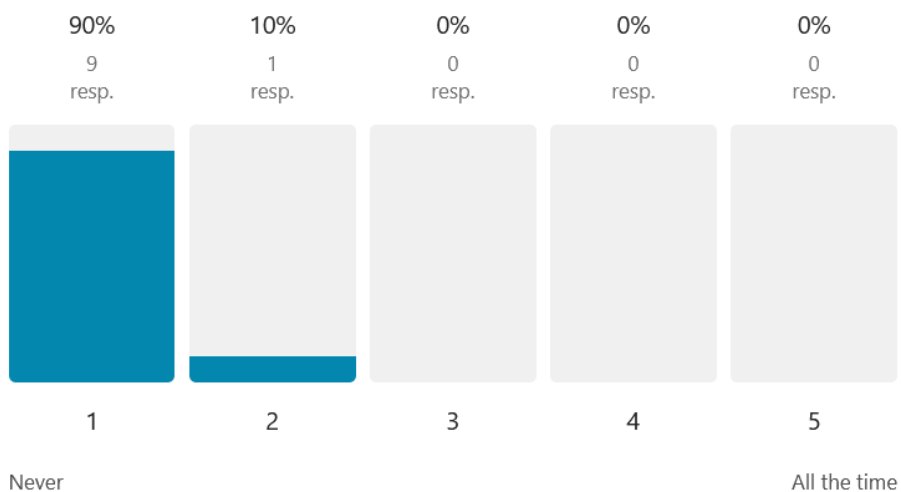
10 out of 10 people answered this question



Have you previously used augmented reality applications?

Avg. 1.1

10 out of 10 people answered this question

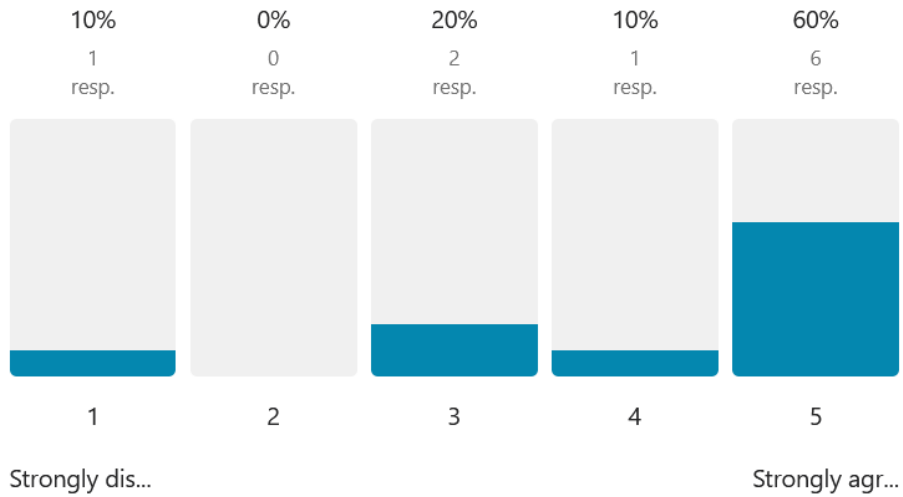




Is video gaming one of your primary hobbies?

Avg. 4.1

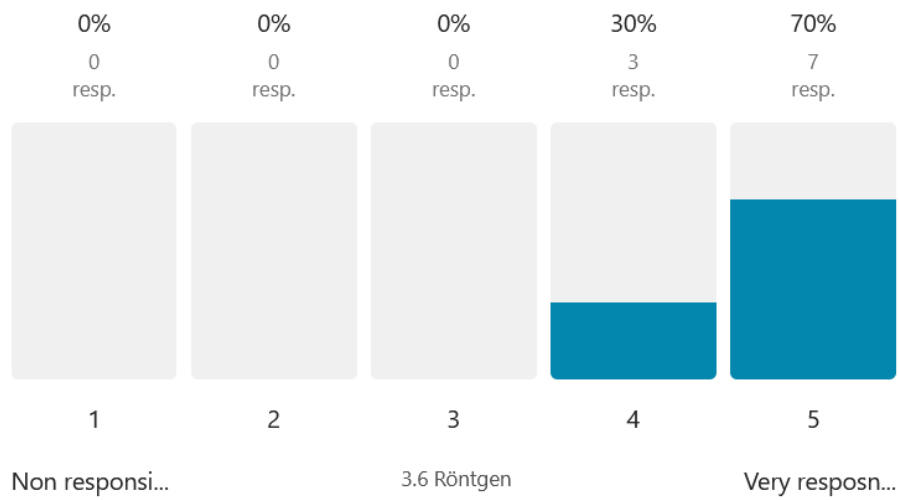
10 out of 10 people answered this question



How responsive was the virtual hand to your movements?

Avg. 4.7

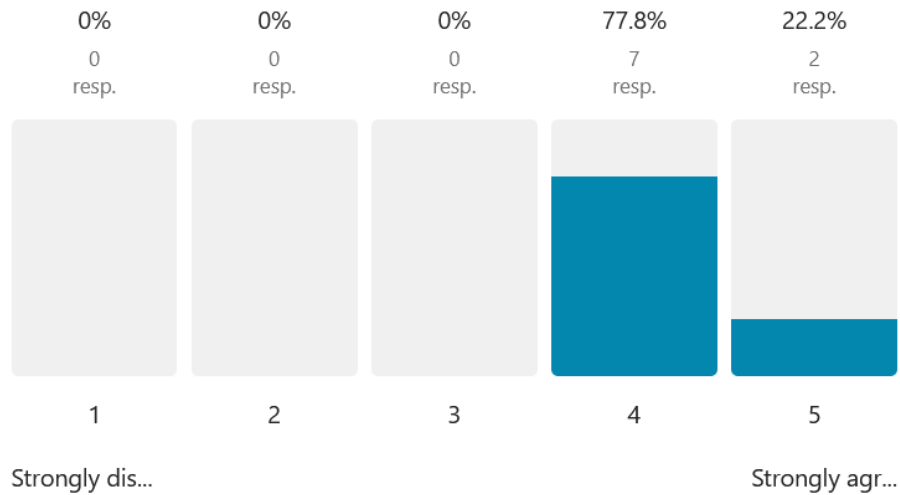
10 out of 10 people answered this question



5 Were the visuals accurate representations of your gestures?

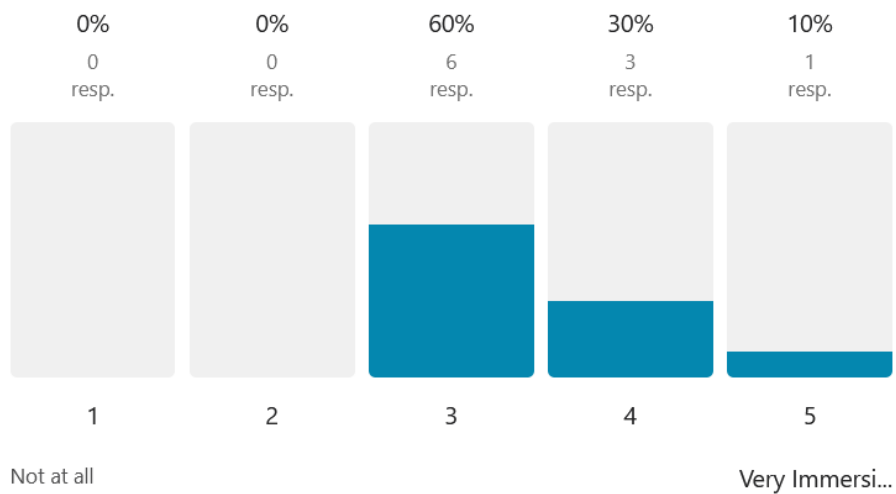
Avg. 4.2

9 out of 10 people answered this question

**6** Did you find the environment immersive?

Avg. 3.5

10 out of 10 people answered this question

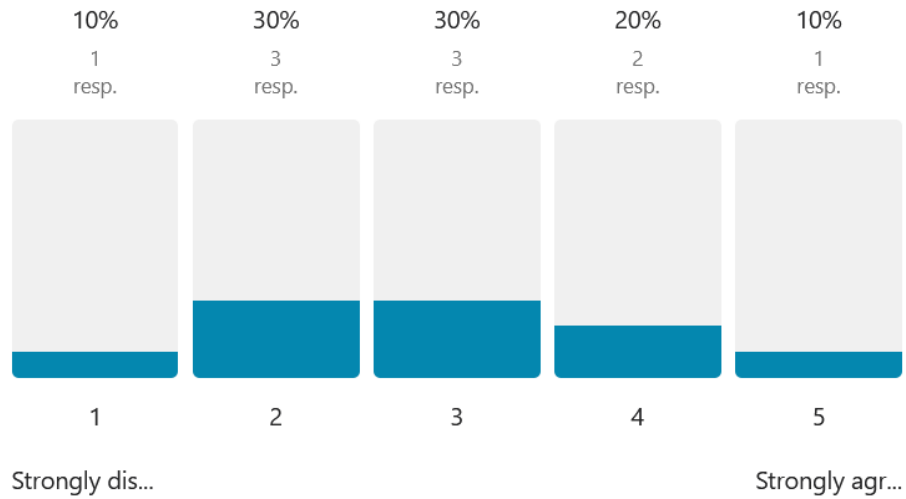




Did you find the setup comfortable and intuitive?

Avg. 2.9

10 out of 10 people answered this question



Appendix B: Scripts

Unity script for the hand

```
using UnityEngine;
using Unity.Robotics.ROSTCPConnector;
using Unity.Robotics.ROSTCPConnector.ROSGeometry;
using RosMessageTypes.Geometry;
using System;
using UnityEngine.InputSystem.XR;

/// <summary>
/// Reads quaternion data and applies rotations to a game object.
/// </summary>
public class QuatReader : MonoBehaviour
{
    /// <summary>
    /// The name of the topic to subscribe to for quaternion data.
    /// </summary>
    public string topicName = "finger1_first_angle";

    [Header("Rotation Offsets")]
    public float zRotationOffset = 0f;
    public float xRotationOffset = 0f;
    public float yRotationOffset = 0f;

    [Space(10)]
    [Header("Rotation Limits")]
    public bool limitAngle = false;
    public float minAngle = 0f;
    public float maxAngle = 0f;
    private Vector3 prevRot;

    [Space(10)]
    [Header("Rotation Factors")]
    public float Xaxisfactor = 0f;
    public float Yaxisfactor = 0f;
    public float Zaxisfactor = 1f;

    [Space(10)]
    [Header("Calibration")]
    private Quaternion quat_ref;
    private bool selfCalibrate = true;

    TrackedPoseDriver trackedPoseDriver;

    /// <summary>
```

```
    /// Gets a value indicating whether the object is currently self-
    calibrating.
    /// </summary>
    public bool SelfCalibrate => selfCalibrate;

    /// <summary>
    /// Gets or sets a value indicating whether the object should be
    calibrated.
    /// </summary>
    public bool Calibrate
    {
        get { return GlobalConfig.Instance.calibrate; }
        set { GlobalConfig.Instance.calibrate = value; }
    }

    public bool inverse = false;
    public bool isFinger = true;

    [Header("Debug")]
    [SerializeField] private bool debugMode = false;

    /// <summary>
    /// Subscribes to the quaternion topic and registers the calibration
    event.
    /// </summary>
    void Start()
    {
        ROSConnection.GetOrCreateInstance().Subscribe<QuaternionMsg>(topicName
, QuaternionChanged);
        GlobalConfig.calibrateEvent += CalibrateChanged;
    }

    /// <summary>
    /// Handles the quaternion changed event and applies the rotation to the
    game object.
    /// </summary>
    /// <param name="msg">The quaternion message.</param>
    void QuaternionChanged(QuaternionMsg msg)
    {
        transform.rotation = Quaternion.Euler(msg.From<FLU>().eulerAngles.x +
xRotationOffset, msg.From<FLU>().eulerAngles.y + yRotationOffset,
msg.From<FLU>().eulerAngles.z + zRotationOffset);
        if(isFinger)
            transform.localEulerAngles = LimitLocalRotation(limitAngle);
    }

    /// <summary>
```

```
    /// Handles the reset calibration event and updates the reference
    quaternion for calibration.
    /// </summary>
    /// <param name="msg">The quaternion message.</param>
    void HandleResetCalibration(QuaternionMsg msg)
    {
        if (Calibrate && selfCalibrate)
        {
            Debug.Log($"Resetting calibration {gameObject.name}");
            quat_ref = msg.From<FLU>();
            selfCalibrate = false;
        }
    }

    /// <summary>
    /// Calculates the difference between the current quaternion and the
    reference quaternion.
    /// </summary>
    /// <param name="msg">The quaternion message.</param>
    /// <returns>The adjusted quaternion based on rotation offsets.</returns>
    Quaternion GetAngleDiff(QuaternionMsg msg)
    {
        HandleResetCalibration(msg);
        Quaternion quat_curr = msg.From<FLU>();
        Quaternion quat_diff = Quaternion.Inverse(quat_ref) * quat_curr;
        return Quaternion.Euler(quat_diff.eulerAngles.x + xRotationOffset,
            quat_diff.eulerAngles.y + yRotationOffset, quat_diff.eulerAngles.z +
            zRotationOffset);
    }

    /// <summary>
    /// Limits the local rotation of the game object based on the specified
    angle limits.
    /// </summary>
    /// <param name="clampAngle">Indicates whether to clamp the angle within
    the limits.</param>
    /// <returns>The limited local rotation.</returns>
    Vector3 LimitLocalRotation(bool clampAngle = false)
    {
        Vector3 tempRotation = transform.localEulerAngles;
        if (clampAngle)
        {
            tempRotation.x = Mathf.Clamp(tempRotation.x, minAngle, maxAngle);
            if (transform.localEulerAngles.x < minAngle ||
transform.localEulerAngles.x > maxAngle)
                prevRot = transform.localEulerAngles;
            else
                tempRotation.x = prevRot.x;
        }
    }
}
```



```
        tempRotation.x = 0;
        tempRotation.y = 0;
        tempRotation.z *= Zaxisfactor;
        if (debugMode)
        {
            Debug.Log($"{gameObject.name}: rotation: {tempRotation}");
        }
        return tempRotation;
    }

    /// <summary>
    /// Handles the calibration changed event and updates the self-calibrate
    flag.
    /// </summary>
    /// <param name="value">The new calibration value.</param>
    void CalibrateChanged(bool value)
    {
        selfCalibrate = value;
    }
}
```

Unity script for publishing

```
using UnityEngine;
using Unity.Robotics.ROSTCPConnector;
using RosMessageTypes.UnityRoboticsDemo;
using Unity.Robotics.ROSTCPConnector.ROSGeometry;
using System.Runtime.InteropServices;

/// <summary>
/// 
/// </summary>
public class TrackerPublisher : MonoBehaviour
{
    ROSConnection ros;
    public string topicName = "pos_rot";

    // The game object
    public GameObject Target;
    // Publish the cube's position and rotation every N seconds
    public float publishMessageFrequency = 0.02f;

    // Used to determine how much time has elapsed since the last message was
    published
    private float timeElapsed;

    void Start()
    {

```

```
// start the ROS connection
ros = ROSConnection.GetOrCreateInstance();
ros.RegisterPublisher<PosRotMsg>(topicName);
}

private void Update()
{
    timeElapsed += Time.deltaTime;

    if (timeElapsed > publishMessageFrequency)
    {
        // Create a new message with the cube's position and rotation
        Target.transform.rotation = Random.rotation;
        PosRotMsg targetPos = new PosRotMsg(
            Target.transform.position.x,
            Target.transform.position.y,
            Target.transform.position.z,
            Target.transform.rotation.To<FLU>().x,
            Target.transform.rotation.To<FLU>().y,
            Target.transform.rotation.To<FLU>().z,
            Target.transform.rotation.To<FLU>().w
        );

        // Finally send the message to server_endpoint.py running in ROS
        ros.Publish(topicName, targetPos);

        timeElapsed = 0;
    }
}
}
```

Microcontroller main script

```
#include <functions.h>
#include <variables.h>

// void Callback(const std_msgs::Float64& msg){
//     float timediff = (micros() - msg.data);
//     timesSum += timediff;
//     timesCount++;
//     // String str = (String)timediff;
//     if(millis() - Update >= 1000){
//         Update = millis();
//         float average = timesSum / timesCount;
//         latency.data = average;
//         latency_pub.publish(&latency);
//         // String str = (String)average;
//         // nh.loginfo(str.c_str());
//     }
// }
```

```
//      timesSum = 0;
//      timesCount = 0;
//  }
//  // nh.loginInfo(str.c_str());
//  }
//  ros::Subscriber<std_msgs::Float64> sub("ruckflug", &Callback);
void setup(void){
    //Begin ROS node and set baud rate
    nh.getHardware()->setBaud(850000);
    nh.initNode();
    //Advertise ROS topics
    nh.advertise(indexMCP_pub);
    nh.advertise(indexPIP_pub);
    nh.advertise(middleMCP_pub);
    nh.advertise(middlePIP_pub);
    nh.advertise(ringMCP_pub);
    nh.advertise(ringPIP_pub);
    nh.advertise(thumbMCP_pub);
    nh.advertise(thumbPIP_pub);
    nh.advertise(palm_pub);
    // nh.advertise(float_pub);
    // nh.advertise(latency_pub);
    // nh.subscribe(sub);
    //Begin I^2C connection
    Wire.begin();
    //Set I^2C data rate to the maximum possible rate
    Wire.setClock(3400000UL);
    delay(1000);
    //Begin communication with the IMUs
    setupFingers();
    setupThumb();
    setupPalm();
}

void loop()
{
    if(millis() - lastUpdate >= 10){
        lastUpdate = millis();
        // before = micros();
        readOutIMUData();
        publishIMUData();
        // after = micros();
        // timediff = after - before;
        // timesSum += timediff;
        // timesCount++;
        // if(millis() - Update >= 1000){
        //     Update = millis();
        //     float average = timesSum / timesCount;
```

```
// latency.data = average;
// latency_pub.publish(&latency);
// // String str = (String)average;
// // nh.loginfo(str.c_str());
// timesSum = 0;
// timesCount = 0;
// }
// String str = (String)timediff;
// float_msg.data = micros();
// float_pub.publish(&float_msg);
// nh.loginfo(str.c_str());
}
nh.spinOnce();
}
```

Microcontroller functions script

```
#include <functions.h>
#include <variables.h>

void tcselect(uint8_t i) {
    if (i > 7) return;
    Wire.beginTransmission(TCAADDR);
    Wire.write(1 << i);
    Wire.endTransmission();
}

void setupFingers(){
    for(uint8_t i = 0; i < fingersAmount; i++){
        delay(20);
        tcselect(i);
        uint8_t a = 2 * i;
        uint8_t b = 2 * i + 1;
        if(!bno.begin()){
            String log = "Sensor " + (String)a + " not detected";
            nh.logerror(log.c_str());
            i = -1;
        }
        else{
            String log = "Sensor " + (String)a + " detected";
            nh.loginfo(log.c_str());
        }
        if(!bno2.begin()){
            String log = "Sensor " + (String)b + " not detected";
            nh.logerror(log.c_str());
            i = -1;
        }
        else{
            String log = "Sensor " + (String)b + " detected";
        }
    }
}
```

```
    nh.loginfo(log.c_str());
}
delay(20);
bno.setAxisRemap(Adafruit_BNO055::REMAP_CONFIG_P6);
bno2.setAxisRemap(Adafruit_BNO055::REMAP_CONFIG_P6);
bno.setAxisSign(Adafruit_BNO055::REMAP_SIGN_P6);
bno2.setAxisSign(Adafruit_BNO055::REMAP_SIGN_P6);
bno.setExtCrystalUse(true);
bno2.setExtCrystalUse(true);
// bno.setMode(adafruit_bno055_opmode_t::OPERATION_MODE_IMUPLUS);
// bno2.setMode(adafruit_bno055_opmode_t::OPERATION_MODE_IMUPLUS);
}
}
void setupThumb(){
    for(uint8_t i = 0; i < 2; i++){
        uint8_t a = 4 + i;
        tcaselect(a);
        while(!bno.begin()){
            String log = "Sensor T" + (String)a + " not detected";
            nh.logerror(log.c_str());
            delay(50);
        }
        String log = "Sensor T" + (String)a + " detected";
        nh.loginfo(log.c_str());
        delay(20);
        bno.setAxisRemap(Adafruit_BNO055::REMAP_CONFIG_P6);
        bno.setAxisSign(Adafruit_BNO055::REMAP_SIGN_P6);
        bno.setExtCrystalUse(true);
        // bno.setMode(adafruit_bno055_opmode_t::OPERATION_MODE_IMUPLUS);
    }
}
void setupPalm(){
    tcaselect(3);
    while(!bno.begin()){
        nh.logerror("Palm sensor not detected");
        delay(50);
    }
    nh.loginfo("Palm sensor detected");
    delay(20);
    bno.setAxisRemap(Adafruit_BNO055::REMAP_CONFIG_P6);
    bno.setAxisSign(Adafruit_BNO055::REMAP_SIGN_P6);
    bno.setExtCrystalUse(true);
    // bno.setMode(adafruit_bno055_opmode_t::OPERATION_MODE_IMUPLUS);
}
void readOutIMUData(){
    for(uint8_t i = 0; i < fingersAmount; i++){
        tcaselect(i);
        bno.getEvent(&event);
    }
}
```

```
    quat[2*i] = bno.getQuat();
    //second IMU
    bno2.getEvent(&event);
    quat[2*i+1] = bno2.getQuat();
}
readOutThumb();
tcaselect(3);
bno.getEvent(&event);
quat[11] = bno.getQuat();
// nh.loginfo("error");
}
void readOutThumb(){
    for(uint8_t i = 0; i < 2; i++){
        tcaselect(4 + i);
        bno.getEvent(&event);
        quat[8+i] = bno.getQuat();
    }
}
void publishIMUData(){
    indexMCP_msg.w = quat[0].w();
    indexMCP_msg.x = quat[0].x();
    indexMCP_msg.y = quat[0].y();
    indexMCP_msg.z = quat[0].z();
    indexMCP_pub.publish(&indexMCP_msg);
    indexPIP_msg.w = quat[1].w();
    indexPIP_msg.x = quat[1].x();
    indexPIP_msg.y = quat[1].y();
    indexPIP_msg.z = quat[1].z();
    indexPIP_pub.publish(&indexPIP_msg);
    middleMCP_msg.w = quat[2].w();
    middleMCP_msg.x = quat[2].x();
    middleMCP_msg.y = quat[2].y();
    middleMCP_msg.z = quat[2].z();
    middleMCP_pub.publish(&middleMCP_msg);
    middlePIP_msg.w = quat[3].w();
    middlePIP_msg.x = quat[3].x();
    middlePIP_msg.y = quat[3].y();
    middlePIP_msg.z = quat[3].z();
    middlePIP_pub.publish(&middlePIP_msg);
    ringMCP_msg.w = quat[4].w();
    ringMCP_msg.x = quat[4].x();
    ringMCP_msg.y = quat[4].y();
    ringMCP_msg.z = quat[4].z();
    ringMCP_pub.publish(&ringMCP_msg);
    ringPIP_msg.w = quat[5].w();
    ringPIP_msg.x = quat[5].x();
    ringPIP_msg.y = quat[5].y();
    ringPIP_msg.z = quat[5].z();
}
```

```
ringPIP_pub.publish(&ringPIP_msg);
thumbMCP_msg.w = quat[8].w();
thumbMCP_msg.x = quat[8].x();
thumbMCP_msg.y = quat[8].y();
thumbMCP_msg.z = quat[8].z();
thumbMCP_pub.publish(&thumbMCP_msg);
thumbPIP_msg.w = quat[9].w();
thumbPIP_msg.x = quat[9].x();
thumbPIP_msg.y = quat[9].y();
thumbPIP_msg.z = quat[9].z();
thumbPIP_pub.publish(&thumbPIP_msg);
palm_msg.w = quat[11].w();
palm_msg.x = quat[11].x();
palm_msg.y = quat[11].y();
palm_msg.z = quat[11].z();
palm_pub.publish(&palm_msg);
}
```

Microcontroller variables script

```
#include <variables.h>
#include <functions.h>

//Addresses of the Multiplexers
#define TCAADDR 0x70

//Define the IMUs and their adress
Adafruit_BNO055 bno = Adafruit_BNO055(55, 0x28);
Adafruit_BNO055 bno2 = Adafruit_BNO055(55, 0x29);

// Declare ROS node handle
ros::NodeHandle nh;

// Declare ROS messages and publishers
geometry_msgs::Quaternion indexMCP_msg;
geometry_msgs::Quaternion indexPIP_msg;
geometry_msgs::Quaternion middleMCP_msg;
geometry_msgs::Quaternion middlePIP_msg;
geometry_msgs::Quaternion ringMCP_msg;
geometry_msgs::Quaternion ringPIP_msg;
geometry_msgs::Quaternion thumbMCP_msg;
geometry_msgs::Quaternion thumbPIP_msg;
geometry_msgs::Quaternion palm_msg;
std_msgs::Float64 latency;
std_msgs::Float64 float_msg;
ros::Publisher latency_pub("latency", &latency);
ros::Publisher float_pub("hinflug", &float_msg);
```

```
ros::Publisher indexMCP_pub("indexMCP", &indexMCP_msg);
ros::Publisher indexPIP_pub("indexPIP", &indexPIP_msg);
ros::Publisher middleMCP_pub("middleMCP", &middleMCP_msg);
ros::Publisher middlePIP_pub("middlePIP", &middlePIP_msg);
ros::Publisher ringMCP_pub("ringMCP", &ringMCP_msg);
ros::Publisher ringPIP_pub("ringPIP", &indexPIP_msg);
ros::Publisher thumbMCP_pub("thumbMCP", &thumbMCP_msg);
ros::Publisher thumbPIP_pub("thumbPIP", &thumbPIP_msg);
ros::Publisher palm_pub("palm_angle", &palm_msg);

//event variables to save the data from the sensors
sensors_event_t event;
imu::Quaternion quat[12];
int fingersAmount = 3;

// external variables for the main loop
unsigned long lastUpdate = 0;
uint32_t sequence = 0;
unsigned long before = 0;
unsigned long after = 0;
unsigned long timediff = 0;
float timesSum = 0;
int timesCount = 0;
unsigned long Update = 0;
```