

**Mixed Reality Human Robot Interaction Interface using Hand Tracking
and Stereo Vision for Unknown and Hazardous Environments**

by

Damith Tennakoon

A thesis submitted to the faculty of graduate studies in partial fulfillment of the
requirements for the degree of

Master of Science

Department of Earth and Space Science and Engineering
York University
Toronto, Ontario

August 2025

© Damith Tennakoon, 2025

Abstract

Many industries employ human workers to perform hazardous and unsafe tasks that result in injuries and fatalities. Some of these tasks include on-site monitoring of radioactive environments, collisions with heavy machinery on construction sites, or performing repetitive and arduous tasks for long durations. Some industries have integrated robotic systems, by means of automation and teleoperation, to mitigate these health and safety risks. The main challenges faced in remote robotic teleoperation systems are a lack of depth perception, immersion of the robot’s workspace, and unintuitive control input interfaces. Robotics have been proven to improve task efficiency, product quality, and workplace safety in many industries, though it has posed a challenge to be integrated into unstructured and dynamic environments. Our research aims to develop a generalized immersive robot arm teleoperation interface to enable the deployment of robotics quicker into real-world applications. The main objective is the development of an intuitive Human-Robot Interaction (HRI) system that integrates the precision, strength, and deployability of robots while leveraging the decision-making, experience, and creativity of human operators. The methodology consists of mounting the ZEDM stereo camera to the end effector of a 6-axis robot arm. The stereo images are streamed to the Meta Quest 2 virtual reality (VR) head-mounted display (HMD), providing a First-Person Video (FPV), with human-like vision, of the robot’s workspace. Hand tracking libraries are used to track and map the operator’s hands into the FPV view. Localizing the tracked hand models with the end effector enables omnidirectional motion control and gripper controls, via gesture recognition, of the manipulator. In-situ data are spatially mapped to the stereoscopic

view, constructing a mixed-reality (MR) HRI interface. The results demonstrate a higher depth awareness of the robot’s workspace and high control stability compared to conventional monocular video feedback systems that are coupled with keyboard input controls. Specifically, the mixed reality interface shows a 39% time reduction in target tracking experiments and has over $5\times$ more stability in trajectory control experiments relative to conventional teleoperation methods. The developed MR HRI interface facilitates the use of robotics in unknown and hazardous applications, enabling workers to safely complete high-risk tasks remotely using HMDs and hand gestures.

Dedication

This is dedicated to my parents, for their endless sacrifice, and belief in me.

Acknowledgements

I would like to thank my supervisor, Dr. Mojgan Jadidi, for her guidance, support, and contributions in shaping this research. Her expertise and insights were invaluable throughout the course of this master's degree.

I would also like to thank my committee member, Dr. Liam Butler, for his feedback on my research progress and insightful tips on how to improve the analysis of my work.

I also extend my sincere appreciation to my examiners, Dr. Sunil Bisnath and Dr. John Moores for taking the time to assess my research work and provide invaluable feedback and insights.

Finally, I am grateful to everyone that has and continue to support me through this journey - academically, professional, and personally.

Table of Contents

Abstract	iii
Dedication	iv
Acknowledgements	v
List of Tables	x
List of Figures	xiv
Abbreviations	xvi
1 Introduction	1
1.1 Motivation of Research	1
1.2 Problem Statement	3
1.3 Research Objectives	5
1.4 Research Contributions	7
1.5 Structure of Thesis	8
2 Related Works	9
2.1 Robotics in Hazardous Environments	9
2.2 Depth Perception	15
2.3 Conventional Teleoperation Systems	16
2.4 Mixed Reality in Robotics	17
2.5 Summary of Gap Analysis	19
3 Methodology	21
3.1 Stereo Cameras and Head-Mounted Displays	23
3.2 Robot Kinematics	27
3.3 Simulation Development	30
3.3.1 Simulation Environment and Architecture	31

3.3.2	Robot Arm Modelling	32
3.3.3	Inverse Kinematics and Control Integration	34
3.3.4	Stereo Camera and HMD Emulation	36
3.3.5	Hand Tracking and Gesture Recognition	39
3.3.6	Mixed Reality Teleoperation Interface	45
3.4	Hardware Development	46
3.4.1	Hardware Components and Architecture	47
3.4.2	Teleoperation Modes	50
3.4.3	System Communication Pipeline	59
3.5	Experiment Setup	62
3.5.1	Quantitative Experiment Design	63
4	Results and Discussion	67
4.1	Trajectory Analysis	67
4.2	Distance Minimization Analysis	69
4.3	Jerk Profile Analysis	71
4.4	General Trends and Comparative Analysis	75
4.4.1	Task Completion Time Comparison	75
4.4.2	Jerk Magnitude Comparison	81
4.5	Limitations	87
4.5.1	Sources of Error	87
4.5.2	Research Limitations	88
4.6	Applications of Developed System	89
5	Conclusion and Recommendations	92
5.1	Summary of Findings	92
5.2	Impact of the Research	94
5.3	Future Work	95
5.3.1	Short-Term Improvements	95
5.3.2	Long-Term Extensions	96
	References	97
	Appendix A: Graphs and Plots of All Participants	104
A.1	Trajectory Plots	104
A.2	Distance Minimization Plots	108
A.3	Jerk Profile Plots	117

Appendix B: User Interface Segment Code - C#	128
B.1 User Input Classes	128
B.2 Communication Classes	135
B.3 Data Acquisition and Rendering Classes	141
Appendix C: Robot Hardware Code - ROS-Python	150
C.1 Communication Scripts	150
C.2 Control Code	152

List of Tables

3.1	ZED Mini stereo camera specifications	27
4.1	Average task completion times (in seconds) for each control mode and target	70
4.2	Average task completion times and standard deviations for Target 1 in seconds.	75
4.3	Average task completion times and standard deviations for Target 2 in seconds.	76
4.4	Average task completion times and standard deviations for Target 3 in seconds.	77
4.5	Table summarizes the main results of the ANOVA analysis performed for task completion times across the three control modes for each target	78
4.6	Pairwise mean differences in task completion time for Target 1. Significance determined using the computed LSD value 3.1 s	78
4.7	Pairwise mean differences in task completion time for Target 2. Significance determined using the computed LSD value 2.2 s	78
4.8	Pairwise mean differences in task completion time for Target 3. Significance determined using the computed LSD value 2.5 s	79
4.9	Efficiency gains in task completion times against the various control modes. Specifically compares the effectiveness when combining stereo vision and gesture-based control against conventional control/vision systems. Also compares the effect of only using stereo vision against conventional vision systems.	81
4.10	Average jerk magnitudes (mm/s^3) and standard deviations for Target 1	82
4.11	Average jerk magnitudes (mm/s^3) and standard deviations for Target 2	82
4.12	Average jerk magnitudes (mm/s^3) and standard deviations for Target 3	83
4.13	Summary of one-way ANOVA results for jerk magnitudes across the three control modes for each target.	84

4.14	Pairwise mean differences in jerk magnitudes for Target 1. Significance determined using the computed LSD value 3629 mm/s³	84
4.15	Pairwise mean differences in jerk magnitudes for Target 2. Significance determined using the computed LSD value 4590 mm/s³	84
4.16	Pairwise mean differences in jerk magnitudes for Target 3. Significance determined using the computed LSD value 4610 mm/s³	84
4.17	Average jerk magnitude ratios between the various control modes. . .	86

List of Figures

3.1	Flowchart diagram showing the three main segments in the proposed teleoperation architecture and each of their functions.	22
3.2	Diagram depicting the functionality of a stereo camera and how depth is perceived [43]	23
3.3	Functional diagram showcasing how the ZED SDK captures images, processes the data, and exposes data to the user [49].	26
3.4	Robot kinematic chain setup for a 6-axis serial robot arm depicting the D-H parameter convention [50].	28
3.5	Developed simulation environment in the Unity engine for a 4-DOF robot arm.	33
3.6	The parametric CAD model of the designed gripper in Fusion 360 shown on the left, and it's integration in the robotic simulation environment, within the Unity Engine, on the right.	34
3.7	Real-time 4-DOF robot arm inverse kinematic simulation workflow diagram.	36
3.8	Simulated head tracking 2-axis servo actuated gimbal system for the ZEDM stereo camera.	37
3.9	HMD head tracking system during simulation run-time.	38
3.10	Hand tracking joint configuration convention [60].	40
3.11	Relative distances between the tip of the thumb and index fingers. . .	42
3.12	Relative distances of the tip of the middle, index, and pinky fingers relative to the wrist.	42
3.13	Relative distances for all joints required for the composite direction control vector, where peaks of the plot represent the resting hand positions and troughs represent an invoked gesture.	43
3.14	Mixed reality teleoperation interface in simulation, showing the stereo camera feed from the robot arm's end effector displayed in the headset, with real-time hand tracking and gesture control enabling intuitive arm control.	46

3.15	Hardware system integration architecture flowchart	47
3.16	Description of the robot arm and setup (left) and diagram of the robot kinematic chain (right).	50
3.17	(a) Image of the operator wearing HMD, providing input in the MR interface, using hand gestures. (b) Operator's view as depicted in the VR HMD (FPV of the robot's workspace). (c) Robot arm at starting position. (d) Robot arm moving towards target using the hand tracking data. (e) User refining trajectory with gesture corrections. (f) Robot arm achieving the target position.	51
3.18	Third-person interface consisting of a real-time digital twin of the robot arm and a point cloud generated from the stereo camera.	56
3.19	The third-person control system allowing the operator to manipulate the position and orientation of the end effector using the digital twin model of the robot arm with hand tracking.	58
3.20	Block diagram of the three different data encryption processes designed for MR interface.	61
3.21	Experiment setup – the end effector is moved in free space toward a cube target. The position of the end effector is recorded continuously for analysis.	64
4.1	Trajectory of the robot arm's end effector, for multiple runs, using the (a) Mixed Reality Hand Control method, (b) Mixed Reality Keyboard Input method, and (c) Computer Video Keyboard Input method . . .	69
4.2	Distance minimization plots for participant 4. Each subplot corresponds to a target cube: (a) Target 1, (b) Target 2, and (c) Target 3. Each line represents a trial using one of the three control methods. . .	71
4.3	Jerk profiles for Student 4 across all control types and target cube positions. Each row corresponds to a different target, and each column corresponds to a different control mode. Coloured lines represent filtered jerk magnitude across trials; dashed horizontal lines indicate average jerk.	74
4.4	Grouped bar chart comparing average completion times across all targets and control modes.	79
4.5	Bar chart comparing the average jerk magnitudes across the various targets and control modes.	85

A.1	Student 1 - Trajectory of the robot arm's end effector, for multiple runs, using the (a) Mixed Reality Hand Control method, (b) Mixed Reality Keyboard Input method, and (c) Computer Video Keyboard Input method	104
A.2	Student 2 trajectories plots.	104
A.3	Student 3 trajectories plots.	105
A.4	Student 4 trajectories plots.	105
A.5	Student 5 trajectories plots.	105
A.6	Student 6 trajectories plots.	106
A.7	Student 7 trajectories plots.	106
A.8	Student 8 trajectories plots.	106
A.9	Student 9 trajectories plots.	107
A.10	Student 10 trajectories plots.	107
A.11	Distance minimization plots for Student 1. Each subplot corresponds to a target cube: (a) Target 1 (right), (b) Target 2 (centre), and (c) Target 3 (left). Each line represents a trial using one of the three control methods.	108
A.12	Student 2 distance minimization plots.	109
A.13	Student 3 distance minimization plots.	110
A.14	Student 4 distance minimization plots.	111
A.15	Student 5 distance minimization plots.	112
A.16	Student 6 distance minimization plots.	113
A.17	Student 7 distance minimization plots.	114
A.18	Student 8 distance minimization plots.	115
A.19	Student 9 distance minimization plots.	116
A.20	Student 10 distance minimization plots.	117
A.21	Jerk profiles for Student 1 across all control types and target cube positions. Each row corresponds to a different target, and each column corresponds to a different control mode. Coloured lines represent filtered jerk magnitude across trials; dashed horizontal lines indicate average jerk.	118
A.22	Student 2 jerk profile plots.	119
A.23	Student 3 jerk profile plots.	120
A.24	Student 4 jerk profile plots.	121
A.25	Student 5 jerk profile plots.	122
A.26	Student 6 jerk profile plots.	123
A.27	Student 7 jerk profile plots.	124

A.28 Student 8 jerk profile plots.	125
A.29 Student 9 jerk profile plots.	126
A.30 Student 10 jerk profile plots.	127

Abbreviations

AR Augmented Reality

AUV Autonomous Underwater Vehicle

DOF Degree of Freedom

FK Forward Kinematics

FPV First-Person View

HMD Head-Mounted Display

HRC Human-Robot Collaboration

HRI Human-Robot Interaction

IK Inverse Kinematics

IMU Inertial Measurement Unit

MR Mixed Reality

OBC Onboard Board Computer

OTF Off-the Shelf

PCD Point Cloud

ROS Robot Operating System

SBC Single Board Computer

SDK Software Development Kit

TCP Transmission Control Protocol

UART Universal Asynchronous Receiver/Transmitter

UAV Unmanned Aerial Vehicle

UDP User Datagram Protocol

URDF Unified Robot Description Format

VR Virtual Reality

ZEDM ZED Mini

Chapter 1

Introduction

1.1 Motivation of Research

Today, many industries employ human workers to perform hazardous and unsafe tasks, resulting in injuries and fatalities. For instance, the Architecture, Engineering, and Construction (AEC) sector, specifically the construction industry, exposes on-site workers to high concentrations of particulate matter, collisions with heavy machinery, and falls-off high altitude sites. Other industries such as the nuclear sector require workers to survey and monitor radioactive environments, where high exposure times have led to serious health consequences. However, in some industries such as the Automotive sector, the integration of automation and robotics has been a proven solution in reducing health and safety risks.

A wide range of industries continue to adopt robotic solutions to improve task efficiency, productivity, product quality, and worker safety. The manufacturing industry uses robotic manipulators to automate assembly lines. These robotic applications highlight the benefit of performing repetitive, hazardous, and time-constrained tasks that would otherwise be difficult and prone to error by human workers. Not only are robotic systems very efficient, but they are also highly competent in precision-seeking tasks, as made evident by their applications in the medical sector. Specifically, the deployment of robots in the performing of surgical procedures has resulted in minimally invasive procedures, as well as higher precision than its human surgeon counterpart.

Robotic systems also excel at running uninterrupted work, which is evident in applications in the agricultural field. Autonomous robotic swarms perform tasks such as planting, harvesting, and monitoring the health of crops with minimal human supervision [1].

Modern day robotics has enabled humans to explore and operate in unknown and inhospitable environments that would otherwise be inaccessible. Robotic exploration is conventionally achieved with real-time remote telepresence, through automated or teleoperated robotic manipulators. In situations that pose a risk to human operators, such as underwater construction, maintenance of nuclear power plants, and repairs to orbital spacecraft, robotic systems that can be intuitively controlled from a distance have ensured operator safety and precision [2]. Despite advances in robotics, existing teleoperation methods lack operator situational awareness, intuitive control, and a generalized control system necessary for a wide variety of challenging applications.

Automation and robotics are clear to provide many benefits; however, one of the main challenges that arises is the ability to rapidly deploy these solutions into unknown and hazardous environments. The mentioned benefits of robotic technologies are demonstrated effectively in structured environments; however, the advantages slowly reduce when applied to complex and unstructured environments that expect high dexterity non-repetitive task executions. The issues of deployability is specifically highlighted in the field of nuclear disaster robotics, with teleoperated robots that are capable of handling radiation environments, restricted access, and precise control in hazardous environments. To deploy robots in these types of environments, existing robotic systems must be heavily modified to effectively perform meaningful tasks [3]. Such setbacks are the main reason for the lack of implemented robotic solutions in the AEC sector. A study demonstrated that the successful implementation of robotics in performing construction tasks requires the adaptation of robots to the unpredictable and dynamic nature of construction sites. Existing applications of construction robotics consist of robot arms fixed-mounted onto mobile manipula-

tors that perform a single task, within a controlled setting. These types of robots pose a significant risk to on-site workers as the robot performs pre-programmed tasks without human-robot collaboration (HRC) [4]. To solve this problem, attempts have been made to train robots, using machine learning techniques and real-world datasets. One of the limiting factors for this approach is the lack of available quality training data for construction applications [5]. The combination of current robotics deployment methods further shows the need for a generalized multipurpose robotic system that would allow rapid integration into various applications. An intuitive interface enabling robotic manipulation, with telepresence, can allow operators to utilize their decision-making, experience, and problem solving skills, while taking advantage of the precision, strength, dexterity, and safety offered by robotics [6].

1.2 Problem Statement

Teleoperated robot arms excel in environments that pose a risk to human access, either a health and safety risk or a physical impracticality. As discussed in the previous section, teleoperated robot arms enable manipulation and inspection of objects in the field. However, the effectiveness of teleoperation is highly dependent on the ability to accurately and intuitively control the manipulator. Conventional robotic teleoperation interfaces struggle at this point, primarily due to their reliance on traditional visual and control input systems. Typically, these control input systems are keyboards, joysticks, and touchscreen devices. They operate by mapping the input, which is usually binary or planar, to position and orientation movements of the robot's end effector. These types of control systems require a lot of training for the operator to learn and are also slow, providing unsteady trajectories to the target pose. The visual feedback component system of common robotics interfaces consists of a fixed-mounted monocular (single-lens) camera mounted near the robot's workspace, which is streamed in real-time to a conventional computer display. One of the main issues with this type of visual feedback is that it suffers from the absence of depth perception.

Without true 3D visualization or an immersive feedback system, operators have a difficult time estimating distances, defining spatial relationships, or measuring relative orientation of target objects [7]. As a result, the operator’s cognitive load increases as they must mentally reconstruct a 3D understanding of the robot’s workspace using 2D images. The high cognition load, coupled with the rudimentary control input system, may lead to inefficient and error-prone operation, especially in the context of high dexterous task manipulation [8]. One of the emerging solutions to this challenge in robotics is the integration of mixed reality (MR) technologies, coupled with stereo vision systems, using head-mounted displays (HMDs) to provide full immersion and intuitive control of robotic systems. MR systems enable for true 3D visual feedback, first-person immersion, and novel interaction systems such as gesture-based control. The immersion is often achieved through projecting stereo images, taken from a stereo camera onboard the robotic manipulator, onto the HMD. These systems provide the operator with an increased sense of spatial awareness and depth perception [9, 10]. However, even with the advent of MR technology in robotic systems, several research gaps remain. The combination of stereo vision, MR interfaces, and gesture-based control has not been thoroughly explored in the context of robotic arm teleoperation, specifically as a means of a generalized system in the context of hazardous and unstructured environments. There is also a need to quantitatively evaluate and compare these types of control systems to conventional control setups to assess their usability, performances, and efficiency. In the case there is a significant measurable efficiency gain, it will dictate whether such control methods are worth implementing in real-world scenarios. The following points summarize the core limitations and challenges associated with existing robotic teleoperation systems:

1. Monocular video feedback systems prevent user immersion into the robot’s workspace, causing misjudgment of the spatial relationships, resulting in inefficient task execution.

2. Conventional control input systems, such as keyboards, joysticks, or touch-screens, prevent intuitive and direct 3D end effector control, making complex movements cognitively demanding.
3. The need for quantitative comparison of the effects of MR technologies in robotics with conventional teleoperation interfaces to determine their feasibility in real-world applications.

The limitations defined above motivate the need for an immersive, intuitive, and efficient robotic teleoperation interface that increases user performance and is a generalized system that is applicable for various applications. The following section outlines the research objectives, which address the issues defined here, through the use of MR, stereo vision, and hand tracking technologies.

1.3 Research Objectives

To understand how MR technologies are impacting robotic teleoperation systems, it is important to first introduce the concepts of augmented reality (AR) and virtual reality (VR). AR is the process in which information, in the form of 3D graphics, is overlaid onto the physical world. AR systems often contain a user input system to interact and manipulate 3D models within the hybrid environment [11]. VR takes this concept a step further by completely immersing the user into a virtual environment, where haptic feedback controllers are used to navigate and interact with the 3D space [12]. Head-mounted displays (HMD) are used to achieve this type of immersion. HMDs consist of a left and right display for each eye, each with adaptable lenses, enabling the brain to reconstruct 3D perception, through the combination of the two images, mimicking human depth perception [13]. Mixed reality fuses the concepts of AR and VR, enabling the user to view the physical world and the virtual world depending on the user's preference [14]. The MR environment is achieved using stereo camera systems, that generate high fidelity stereo images, localized with the user's

environment. The images are processed and re-projected onto the Liquid Crystal Displays (LCD) of the HMD [15]. Specifically, stereo cameras consist of two identical monocular cameras, separated by a fixed distance known as the baseline, that render two slightly different perspectives of the target scene. The disparity between the images then allows the extraction of depth information using triangulation techniques [16]. The combination of stereo camera systems with MR hardware enables full immersion into various workspaces, especially advantageous in robotic applications. A study conducted in [7] that compared conventional monocular camera video feedback teleoperation with stereoscopic VR teleoperation showed the stereoscopic systems yielded significantly faster task completion times for maneuvering applications, in the context of unmanned ground vehicles (UGV). Further experiments conducted using the coupled technologies has shown to increase depth cues, user performance, and usability, compared to conventional teleoperation methods, in the field of mobile robotics [9].

This research aims to utilize the advantages of stereo vision cameras and MR systems, coupled with hand tracking technologies, to develop an intuitive hands-free teleoperation interface for a 6-axis robot arm, in the context of hazardous off-site applications. Listed below are the specific research objectives:

1. Develop a novel mixed reality application which provides the operator with a first-person view, with true depth awareness, of the robot’s workspace.
2. Implement hand tracking and gesture detection algorithms to guide the end effector of a robot arm using intuitive hand gestures.
3. Devise experiments that measure the efficiency gain of completing target-tracking tasks for the proposed interface against conventional control interfaces.

1.4 Research Contributions

The results showcased a significant increase in depth perception, situational awareness, and control stability using the proposed MR interface, compared to conventional teleoperation systems. Specifically:

1. Demonstrated a **39%** decrease in time reduction for target tracking tasks, compared against conventional methods.
2. **5×** increase in control stability of the end effector, relative to conventional teleoperation methods.
3. Instantaneous depth perception via mixed reality and stereo vision.
4. Ease of control of the end effector, in 3D space, from hand tracking and gesture recognition.

In addition, multiple papers have been written on this research and presented in notable academic conferences:

1. **Tennakoon, D., Jadidi, M., and RazaviAlavi, S.**, “*A Mixed Reality Human Robot Interaction Interface Using Hand Tracking and Stereo Vision for Construction Applications*,” in Proceedings of the *European Conference on Computing in Construction (EC3 2024)*, vol. 5, Chania, Greece: European Council on Computing in Construction, July 2024. doi: 10.35490/EC3.2024.275

Recognition: Runner-Up Best Paper Award; Keynote Speaker at EC3 2024.

2. **Tennakoon, D., Jadidi, M., and RazaviAlavi, S.**, “*Development of an Intuitive Mixed Reality Human Robot Interaction Interface for Construction Applications*,” in Proceedings of the *Transforming Construction with Off-site Methods and Technologies (TCOT 2024)* Conference, University of New Brunswick, Fredericton, Canada, June 2024. <https://shorturl.at/rpV2T>

Recognition: Best Paper Award at TCOT 2024.

3. **Automation in Construction Journal:** Invited by the journal to extend both conference papers into a full journal paper, currently in the process of being published.

1.5 Structure of Thesis

The structure of the remainder of the thesis is outlined as follows:

- **Chapter 2 - Related Works:** Presents a literature review of robotics in hazardous or unknown environments. It also introduces the underlying issues of depth perception found in robotics as well as existing teleoperation interfaces.
- **Chapter 3 - Methodology:** Introduces the technical concepts behind stereo vision, mixed reality hardware, and robot kinematics. The chapter also documents the development of the simulation environment, hardware integration, and the overall system development process.
- **Chapter 4 - Experiment Setup:** Details the experiment design, data collection process, and planned methods for data analysis.
- **Chapter 5 - Results and Discussion:** Presents the collected data in the form of plots and tables, with detailed analysis and metrics. It also discusses the impact of the findings to real-world applications.
- **Chapter 6 - Conclusion:** Summarizes the key outcomes of the research, reflects on the impact, and highlights potential directions for future work.

Chapter 2

Related Works

2.1 Robotics in Hazardous Environments

Robotics have been, and continue to be applied in many fields and applications. Robotics can refer to various types of mechanisms; they are found in space applications, construction scenes, nuclear operations, and search and rescue operations, to name a few. The robots applied in these domains may vary from specialized robots designed to complete a very specific task, such as the Canadarm2 on the International Space Station (ISS), or more off-the shelf (OTF) robots such as 4-wheeled differential drive rovers adapted for vacuuming everyday households. The advantage that robotic systems provide is they enable high-risk tasks to be completed without the need for human operations. Depending on the mechanisms of the robot, they are able to traverse inhospitable terrains and environments, perform actions that are tasked for humans, and have high strength, dexterity, and accuracy.

For instance, decommissioning of nuclear facilities requires the handling of radioactive material. Conventionally, human operators would be required to go in the field, wearing protective clothing, with the risk of exposure to locate, handle, and transport these hazardous materials in unknown, unstructured, and unsafe environments. In the 2011 Fukushima disaster, it was determined that the risk to human operators was too high, thus, robots were rapidly developed to replace their human counter-part. Specifically, the Nuclear Safety Technology Centre (NUSTEC) developed robots that

could monitor the environment of the structure in which the accident had occurred. Further, the Japan Atomic Research Institute (JAERI) developed a swarm of surveillance robots that could monitor dose levels caused by beta and gamma radiation [17]. These types of robot drastically reduced the need for humans to physically enter and monitor these hostile environments. To perform tasks such as debris removal, complex teleoperated manipulators were designed and deployed. Further, [18] have developed a dual-arm handling robot purposed for nuclear emergency tasks. The high-degree of freedom (DOF) dual arm system, mounted on a mobile platform, is able to traverse nuclear sites, remove obstacles, handle waste liquid barrels, carry heavy loads, and precisely handle long tubes and pipes for repairs. These robotic solutions drastically reduce the need for human exposure in radiation environments, to perform dangerous and life threatening tasks.

Another useful application of robotics in hazardous environments is in remote underwater and deep-sea operations. These applications make use of a specialized category of robotics known as Remotely Operated Vehicles (ROVs) that are powered via a physical tether, and are controlled by operators on ships. Many instances of ROVs have been researched and deployed, one instance being the Cable-controlled Undersea Recovery Vehicle (CURV), developed by the Royal Navy in 1973, which performed a groundbreaking deep-sea recovery operation in which it rescued two trapped sailors in a submersible, at a depth of 470 m below the ocean surface. Today, underwater robotics has drastically improved the ability for underwater construction, specifically in the oil and gas sector. ROVs have enabled the construction of sub-sea oil and gas developments at depths that exceed the reach of human divers. These robots further reduce the need for constant human operations as they are able perform routine monitoring, inspections, and repairs in these high pressure environments, with high precision for long durations. A growing area of research in underwater robotics is the development of Autonomous Underwater Vehicles (AUVs). AUVs have the ability to be deployed for long durations, with no input from human-operators. These robots

are equipped with a multitude of sensors that enable them to be used for various deep-sea applications, that would otherwise be too costly or unachievable with human divers. Using high frequency sonar and high resolution camera systems, they can collect detailed data and create real-time maps of the seabed [19].

One of the areas where the advantage of robotic deployment in inaccessible environments is highlighted is in the space robotics domain. The space environment, whether in low-earth orbit or on extraterrestrial planets, is harsh, dangerous, and extremely costly for human exploration. Therefore, it is one of the most well-researched areas in robotic applications. Space robotics is grouped into three main classes: (1) orbital robotics, (2) asteroid robotics, and (3) planetary robotics [20]. A significant Canadian contribution to the class of orbital robotics, that is currently operational, is the Canadarm-2, which is a high DOF robotic manipulator that was designed to construct, repair, and maintain the ISS. The robotic arm has docking capabilities to attach various space vehicles and move them accurately and safely. With an extremely dexterous end effector, the arm is able to mount astronauts and transport them to various external parts of the ISS for performing extravehicular activities (EVAs) [21]. Currently, human planetary exploration is not achievable as it is complex, costly, and dangerous. However, it has been proven that unmanned robotic missions can be sent to planets at much lower cost with no danger to human operators. An ideal example of this is NASA's Curiosity rover mission, which landed on Mars in 2012. The rover, equipped with a multitude of science instruments, was dedicated to the goal of assessing the potential for Mars harbouring life. The instruments included a spectrometer and an imager attached to a robotic arm, a chemical camera, environment sensors, and an analytical laboratory instrument, to name a few. The rover, along with its many instruments, continues to be carefully teleoperated from Earth to make discoveries about the Martian geology and atmosphere [22]. The advent of interplanetary robotic missions has enabled the expansion of human knowledge that would otherwise be unreachable through human exploration. Though the technology

required to place humans on planetary bodies like Mars is being developed and will be possible in the very near future, there are some planets that are completely out of human reach and can only be explored via robotic solutions. One of the long-time goals for space exploration has been to understand the atmosphere and current conditions of Venus. With an average temperature of 450 °C and 90 atmospheres of pressure, comprised primarily of carbon-dioxide, Venus is one of the most inhospitable planets in the solar system. Past missions have shown that it is possible to send robots to Venus, as demonstrated by the Russian Venera-7 mission, which landed probes on the Venusian surface that lasted approximately 2 hours. Currently, robotic missions are underway that will be capable of traversing the Venusian surface for 50 days, as well as solar-powered aircraft for surveying the atmosphere. These missions are of high priority for the US Academies of Sciences, as they will provide insights on the greenhouse effects that made Venus inhospitable, which would be beneficial to understanding the climate here on Earth [23].

The aforementioned space robotics applications require a lot of pre-planning prior to task execution. An area of robotics that demands real-time decision making and quick task completion is that of search and rescue robotics. There are many categories of robotics that fall under this field, a significant one being that of quadruped robots. These types of robots are platforms, actuated via multi-degree freedom legs, for mobility. A state-of-the-art quadruped mobile robot, named Momaro, was developed for the Defence Advanced Research Projects Agency (DARPA) robotics competition. The developed robot was equipped with a 4-DOF leg design, each with an actuated wheel system, and two 7-DOF robotic arms attached to the main platform, allowing it to perform high dexterous tasks and navigate complex terrain. The teleoperated quadruped demonstrated it's ability to open doors, turn valves, and cut holes in drywall [24]. In many search and rescue scenarios, direct teleoperation may not be feasible. The NuBot was designed to operate with a semi-autonomous control system, enabling for less input from human operators and more in-situ decision-making,

specifically for urban search and rescue scenarios. The NuBot is a differential drive mobile robot that was able to successfully demonstrate its capabilities of simultaneous localization and mapping (SLAM), victim identification, path planning, and semi-autonomous mobility [25]. These robotic systems are more effective when the search workload can be off-loaded to other systems. The rapid growth of Unmanned Aerial Vehicles (UAV) provides this advantage in the context of search and rescue. UAVs have the capability to explore vast areas of terrain much easier than ground robots. Though limited by payload capacity and battery capacity, semi-autonomous drones have been deployed to search for victims in remote environments. The authors of [26] present a UAV system that is designed for search and rescue of snow covered avalanche survivors in mountainous terrains. The system is able to autonomously fly while the operator can focus on locating victims, using a wide variety of onboard sensors.

One of the many benefits of robotic systems is its strength and accuracy, which is highlighted in the Automation, Engineering, and Construction (AEC) sector where on-site workers are expected to perform arduous, repetitive, and dangerous tasks. The integration of robotics into these fields have greatly reduced health and safety risks associated with modern day construction. One of the ways this is being implemented is the development of specialized robots for key construction tasks. The authors of [27] present a 6-DOF robot arm that can be mounted onto scaffolding, enabling the robot to perform high-precision tasks, with various end effector tools, that would otherwise be a fall risk for on-site workers. Precision is a fundamental requirement in any construction-related task, specifically noted in welding applications. Welding is not only physically taxing but also slow due to its hazardous nature. Additionally, welding is a task that requires extreme precision and long-term experience. Computerized welding is a new form of technology that takes advantage of machine learning to monitor, track, and control welding parameters. These systems, coupled with robotic arms equipped with welding tools, are able to minimize workers' exposure, increase weld

quality, decrease production times, and lower labour costs [28]. Another challenge in construction is ensuring that concrete is poured accurately and evenly according to the design specifications. Varying worker experience often leads to inconsistent placement which affects the final structure. To tackle this challenge, advanced sensor for robotic arms have been developed, such as laser scanners and stereo cameras, which increases accuracy and consistency in concrete extrusion. These technologies help to minimize material waste, subsequently reducing the overall project costs and promoting sustainable construction practices [29]. Excavation is a critical component of any large-scale construction project, which conventionally requires highly trained operators to dig trenches and lay foundations. Often, these tasks take months to complete and are highly susceptible to human error, which may compromise the final structure. Advancements made in autonomous excavators have enabled machines to perform the same tasks as their human counterpart with greater speed and precision, following the design specifications as a blueprint. These systems are able to accurately understand their working environments through the use of LiDAR sensors, which can create 3D digital twins of their surroundings. Coupled with machine learning algorithms, the excavators are able self-drive and excavate with minimal errors. Real-world deployment of these systems have resulted in significantly shorter project timelines compared to traditional methods, which has led to more efficient construction [30]. As structures age, maintenance and repairs become necessary, at a high cost. These costs increase the longer it takes to address the faults. To mitigate this problem, robotic systems are being deployed into the field for proactive inspection and monitoring, enabling for early fault detection. UAVs equipped with high resolution cameras and advanced object detection software are deployed into tunnels, bridges, and buildings to identify cracks and structural wear, both internally and externally. These robotic systems provide data from vantage points that are unreachable through conventional human inspection methods [31]. These types of applications enable high risk tasks to be executed by replaceable robotic systems, allowing for a more safe,

efficient, and sustainable construction environment for on-site workers.

2.2 Depth Perception

One of the challenges in modern day robotic teleoperation is the inherent lack of depth perception of the robot’s workspace conveyed to the operator. The root cause of this perception mismatch is the use of monocular cameras to extract depth information. The conventional approach for providing vision in robotics is to stream a monocular camera view to a conventional computer screen, forcing the operator to infer 3D spatial information from a 2D image [32]. The need to infer depth from perspective shifts, including lighting and shadow changes or variations in object size, inherently increases cognitive load and forces continuous camera adjustments, resulting in high-risk trial-and-error teleoperation. As a result, depth inference decreases further in unknown environments and conditions, such as estimating the depth of a rubble pile, where colours overlap and the scene is disorganized [33]. Further, it is quite challenging for humans to infer depth in virtual environments (VE), as is the case when using a computer screen to visualize a robot’s workspace, than it is to infer depth in the physical world [34]. Another conventional system used for 3D depth extraction in robotics are sensors, the most common one being the Light Detection and Ranging (LiDAR) sensor. LiDAR systems are a type of time-of-flight (ToF) sensor which determines depth by measuring the elapsed time for a transmitted ray of light to return to the sensor. LiDAR systems are used to create dense real-time point clouds and are often used in industry for various applications. Although LiDAR can be used for depth perception, it has many limitations as the accuracy of the depth map is highly dictated by environmental factors. In outdoor environments with non-ideal weather conditions such as clouds, heavy rain, fog, or intense sunlight, the accuracy of depth estimations can be reduced [35]. Indoor environments bring their own challenges such as obstacles blocking the transmitted light rays, resulting in incomplete 3D representations of the environment. These issues are often overcome

through sensor fusion, where data collected from additional sensors such as radar, ultrasonic sensors, and cameras provide a more complete and comprehensive digital map. One of the unavoidable setbacks of LiDAR is that they are designed to function under the assumption that all objects in their environment are diffuse objects. Reflective objects such as mirrors and transparent objects such as windows distort incoming light, resulting in ambiguous depth computations and is dangerous as incorrect depth estimations or no depth estimations, in the case of windows, result in unexpected collisions for robotic systems [36].

2.3 Conventional Teleoperation Systems

Another major component that inhibits robotic applications in hazardous environments is the lack of efficient and intuitive control interfaces. Conventional teleoperation systems often consist of a feedback controller, such as joysticks and keyboards. Although these systems can be used efficiently with a lot of training, learning them is extremely unintuitive. These types of control systems require the operators to use a lot of cognition in mapping 2-dimensional planar inputs into 3D movements. With the added overhead of iteratively perceiving depth through 2D images, the end effector cannot be moved in 3-dimensions, rather constraining the movement to first a planar motion, followed by the vertical displacements [37]. A study conducted by [38] compared the trajectories of drone flight paths when teleoperating the system using conventional 2D gamepads against 3D hand tracking using a Leap Motion sensor. The results of the experiment showed that the trajectories were much smoother and natural when using directional gestures compared to the conventional gamepad system. It was determined that the more intuitive control method would reduce cognitive loads and operational errors. In robotics, conventional systems like gamepads and joysticks lead to the end effector trajectories being jagged and slow, which is not only inefficient, but also increases the risk of collisions with objects in the robot's workspace. These control input systems makes teleoperation a hazard as it is not

easily applicable to generalized applications. The effect of unintuitive control is amplified as teleoperation not only requires positional movements but also control of orientation (roll, pitch, yaw), coupled with the dependency of end effector speed. Often, the pose of the robotic system, which is the combination of position and orientation, is required to be set together, rather than consecutively, which is quite a challenge to achieve with conventional forms of teleoperation, such as keyboards and joysticks, as specific actions are mapped to a single button or an axis. The rudimentary controls fundamentally prevent the pose from being computed in a single instance, which drastically slows down operations and can be dangerous. Further, conventional systems cannot easily adapt to the various types of end effectors. For instance, in the case where the end effector is a prismatic gripper, inputs (buttons on a joystick or keys on a keyboard) must be predefined to achieve fixed states of the gripper. For example, a specific key is set to close the gripper, another key set to open the gripper, and another to variably control the gripper. The binary input system is challenging and inefficient, as the operator will need to memorize each binding. As the complexity of the end effector increases, the effectiveness of these conventional teleoperation systems diminish.

2.4 Mixed Reality in Robotics

During the past few years, much research and development has accumulated specifically on leveraging Augmented Reality (AR), Virtual Reality (VR), and Mixed Reality (MR) technologies to effectively improve human-robot interaction and collaboration. The study by Puljiz et al. [39] proposes and implements an Human Robot Interaction (HRI) interface to teleoperate an industrial grade robot arm, utilizing the Microsoft HoloLens AR Head Mounted Display (HMD). The approach presented utilizes the built-in spatial mapping of the HoloLens to identify the target robot arm, and its end effector, within the operator’s physical space which is achieved by storing a previously known model of the robot arm and matching the known model with the detected mesh

from the spatial mapping output. The known mesh and the HMD are localized with the detected robot arm in the physical space, and hand detection techniques are used to translate the end effector. The designed interface enables the teleportation of a robot arm without the need for physical interaction with the manipulator.

The study by Yim et al. [40] emphasized the importance of complete immersion into the robot’s workspace through means of stereo vision. They implement a method to teleoperate a 6-DOF robot arm by fixed-mounting an Intel RealSense stereo camera to the robot arm’s platform, with a view facing down and towards the end effector. The stereo images are streamed to a Meta Quest 2 VR HMD to replicate the robot’s workspace. Hand held controllers are then localized with the robot arm to manipulate its end effector. The developed HRI interface enables a robot to be controlled from a safe distance with human-like perception.

The work conducted by Chang et al. [41] tackles on the ongoing challenge of effective robot workspace visualization and perception by their design of a 2D/3D multi-view teleoperated robot arm. Specifically, they have devised an approach to view a robot’s workspace via a First-Person View (FPV) real-time 3D point cloud (PCD) model, from a stereo vision camera mounted to the end effector of the manipulator, and a 2D view from a global RGB camera. The multi-modal visual feedback system effectively enabled for the visualization of the direct workspace, via the PCD, as well as the global workspace, via the RGB camera. The developed MR interface allowed the user to switch between the two visual systems in their HMD, using a handheld controller, providing a comfortable teleoperation process.

The study performed by Ulloa et al. [42] highlights the need for more immersive telepresence systems as the use of conventional interfaces does not provide immersion and creates a degree of mistrust when performing dexterous tasks. The system proposed by these authors consists of teleoperating a 6-DOF robot arm, mounted to a quadruped, by constructing a virtual model of the arm using the Unity engine. Incorporating the Microsoft HoloLens HMD, a 3D sphere is placed at the end effec-

tor of the robot arm’s digital twin equivalent. As the user interacts and moves the target object within the virtual workspace, data is transmitted to a set of ROS nodes that compute inverse kinematics solutions, enabling motion of the end effector. The HRI system aims to construct a one-to-one digital twin of the robot’s workspace, to provide the operator with higher depth perception and confidence in control.

Many other similar approaches have been proposed and implemented showcasing positive results towards achieving a natural HRI interface. However, these HRI systems do not come without their own setbacks. With the implementation of the HoloLens AR approach by Puljiz et al. [39], though the interface provides robot arm teleoperation without the need for physical interaction with the manipulator, it is constrained, as it requires the operator to be in the physical workspace of the robot arm. The constraint limits the control bandwidth and deployability, as the operator needs to have a direct line-of-sight (LoS) of the robot arm for teleoperation. In the case of the immersive stereo vision teleoperated robot arm developed by Yim et al. [40], many safety issues arise when the user’s view is obstructed by the robot arm itself or other objects, within the robot’s workspace, due to the vision system being fixed mounted. The authors Chang et al. [41] are on the right track with their multi-view MR HRI interface, however, the cognition load required for the operator increases when frequently needing to switch between a 3D view and a fixed 2D view; the operator will need significant training to be comfortable localizing their coordinate system with the robot arm’s coordinate frames, in the two views. Many lessons were taken away from all the aforementioned approaches and this has aided the development of the HRI interface developed in this research.

2.5 Summary of Gap Analysis

Modern day teleoperated robotic systems use advanced sensors, visualization systems, and a wide variety of user interfaces. These systems have been deployed into real-world hazardous applications and have proven to have many setbacks. One core

challenge is the ability to provide the operator with accurate depth awareness of the robot’s workspace. Conventional monocular video-based methods fail to provide true depth perception, which result in decreased task completion efficiency and contributes to unstable control behaviour. Further, one of the most common methods in teleoperating robotic systems is through the use of keyboard-based input systems. Although simple to implement, these systems are unintuitive and poorly suited for dynamic or high-risk environments where dexterity is a high priority. An emerging solution to this challenge is the integration of mixed reality, which is designed to provide increased telepresence into remote workspaces through stereo vision. A key setback of mixed reality in robotics is that they lack full integration with intuitive gesture-based control, or comprehensive validation against conventional teleoperation systems. These gaps highlight the need for a teleoperation interface that combines stereo vision and natural control inputs, and it benchmarked against conventional control interfaces. Addressing this research gap will allow robotic systems to be deployed more rapidly into real-world hazardous and unstructured environments.

Chapter 3

Methodology

The devised system architecture for this mixed reality robot arm teleoperation interface consists of three main segments: Robotic Manipulator, Mixed Reality Interface, and Remote Teleoperation. The robotic manipulator segment consists of a high DOF robotic arm, where the stereo camera system is attached to its end effector. The left and right images from the camera feed is transmitted to the Mixed Reality Interface segment which renders a 3D immersive view of the robot's workspace. The VR headset's onboard sensors are then used to track the operator's hands in real-time. Gesture recognition algorithms enable the operator to provide control inputs to the robot arm, which is transmitted to the Remote Teleoperation segment. The remote segment converts the high-level control inputs into low-level commands, as per the robot arm's control architecture, which moves the end effector iteratively. The devised architecture is summarized diagrammatically in Figure 3.1.

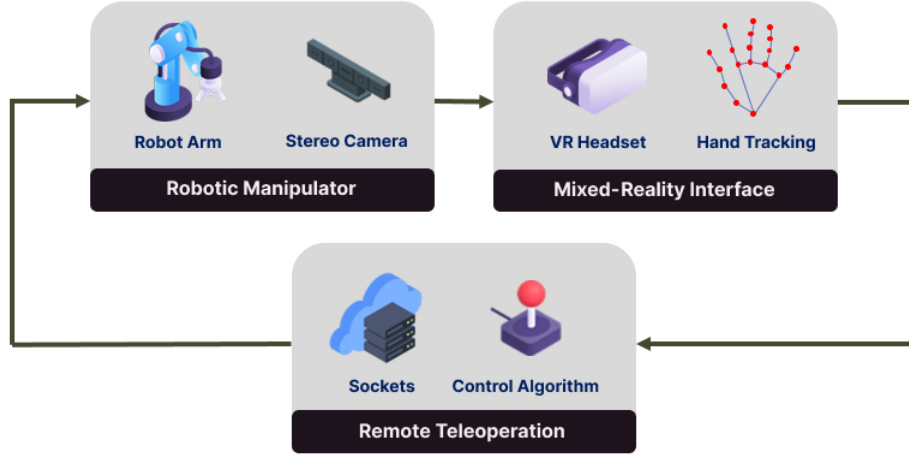


Figure 3.1: Flowchart diagram showing the three main segments in the proposed teleoperation architecture and each of their functions.

The development of the proposed MR interface followed a sequential process that begins with simulating the system and gradually advancing toward real-world implementation and evaluation. The workflow started with a simulation environment where inverse kinematics, stereo camera emulation, and hand gesture control were developed. Once the proposed system was validated to be feasible, the hardware integration process began using the physical robot arm: (1) implementation of basic keyboard teleoperation, (2) integration of stereo camera with the VR headset, and (3) incorporation of hand-tracking to provide direct end effector control in first-person view. Once this baseline hardware integration was completed, additional features were added to complete the interface such as gripper controls and a third-person digital twin viewer. Lastly, data collection experiments were performed to measure the effectiveness of the developed system relative to conventional methods.

3.1 Stereo Cameras and Head-Mounted Displays

One of the main components used in this research is a stereo camera, and its purpose is to provide depth information about the robot's workspace. Prior to understanding its integration with head-mounted displays (HMD), it is important to understand how depth data is extracted using stereo vision. The hardware of a stereo camera consists of two spatially separated imaging sensors along the horizontal axis. These two sensors capture synchronized images of the same scene from slightly different perspectives. Then, utilizing methods of triangulation, the depth is computed using the disparity between the corresponding pixels in the two images. The geometry of stereo vision is presented in Figure 3.2.

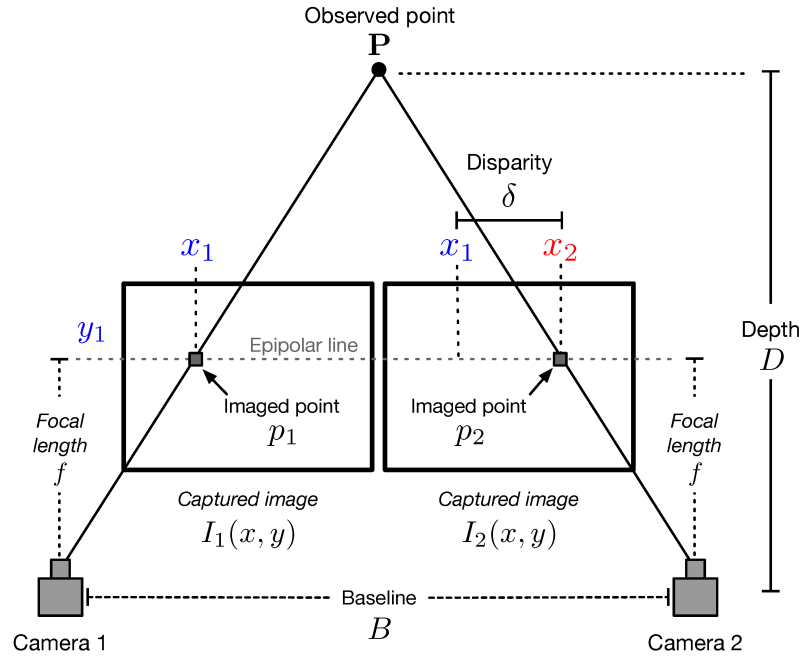


Figure 3.2: Diagram depicting the functionality of a stereo camera and how depth is perceived [43]

The main process for extracting depth is triangulation. Given the disparity between the corresponding pixels in the left and right images, denoted as δ , along with the baseline, B , the focal length, f , the depth, D , of a point in 3D space, can be computed using the following formula:

$$D = \frac{f \times B}{\delta} \quad (3.1)$$

The above equation assumes that the images have been rectified, where the epipolar lines are aligned on the horizontal axis, which reduces the disparity estimation to a one-dimensional search problem. The disparity is defined as the horizontal shift between the corresponding pixels in the stereo image pair:

$$\delta = |x_1 - x_2| \quad (3.2)$$

One of the core challenges in stereo imaging is stereo matching, or disparity computation. Stereo matching aims at finding the corresponding pixels between two images. Common algorithms consist of computing a matching cost using methods such as the sum of absolute differences and normalized cross-correlation, followed by cost aggregation, disparity optimization, and post-processing [44]. The result is what is commonly referred to in machine vision as a disparity map. The disparity map encodes the relative shift for each pixel, which can be converted into a dense depth map using the above triangulation technique. Modern day techniques utilize deep learning to improve matching accuracy in challenging conditions such as low texture, repetitive patterns, or occlusions [45]. These machine learning techniques that learn end-to-end disparity estimation from data can significantly outperform conventional stereo matching algorithms.

One of the crucial elements for any form of teleoperation is the ability to provide the operator with intuitive and immersive visual feedback. To this end, stereo camera systems are commonly integrated with HMDs to reconstruct the robot’s 3D workspace in real-time which is achieved by projecting the synchronized image pairs from a stereo camera onto the left and right eyes of the user, recreating natural 3D perception of the workspace.

The underlying concept of stereoscopic vision in HMDs is to replicate the binocular

disparity that humans naturally use to perceive depth. Specifically, humans perceive in 3D by observing the world through two slightly offset perspectives which is caused by the separation between their eyes. The separation is commonly referred to as the inter-pupillary distance (IPD), and is a key concept in mixed reality. The brain has been trained to merge these two viewpoints to perceive a single 3D image that contains depth. Stereo cameras mimic this effect by capturing two simultaneous images from horizontally offset lenses. These images are then streamed to the corresponding displays in the HMD. With correct alignment and synchronization, it creates the illusion of depth through binocular parallax [46, 47].

The rendering process involves the following procedure:

1. **Stereo Imaging:** The stereo camera is mounted onto the end effector of a mobile manipulator which captures the left and right images simultaneously.
2. **Viewport Mapping:** The left and right captured images are rendered to the corresponding displays of the VR HMD.
3. **Low-Latency Streaming:** Images must be rendered with minimal latency and high fidelity to ensure the user does not experience any discomfort during operation.
4. **Synchronization:** Ensure the left and right images are displayed simultaneously to prevent stereo mismatch which can lead to user disorientation and overall discomfort.

Conventional teleoperation systems use monocular video feedback as a means of depth perception which is intrinsically inefficient as depth is inferred using cues such as shading, motion parallax, or perspective distortion. The low depth awareness results in ambiguous perception of 3D environments, due to a high cognitive load requirement, which reduces task efficiency and increases error. With stereo imaging and HMD integration, operators can perceive depth instantly with little mental

processing [48].

The stereo camera system used for this research is the ZED Mini (ZEDM) designed by StereoLabs¹. The camera is compact, provides high-resolution stereo images, and is specifically designed for applications in robotics, AR/VR, and mixed reality. The camera is capable of dual 2K resolution with a baseline of 63 mm, which mimics the average human IPD for depth estimation and immersion. The camera is also adaptable as it provides streaming capabilities of 60 frames per second (FPS) at lower resolutions (720p), which enables for real-time performance in robotic teleoperation applications. The ZEDM was chosen over conventional depth sensors such as the Intel RealSense or Microsoft Azure Kinect due to its robust integration with 3D engines such as Unity² and Unreal engine³, through well-designed and supported System Developer Kits (SDK). The SDK provided by Stereo Labs exposes real-time access to stereo images, depth maps, point clouds, and camera pose tracking. Table 3.1 shows the specifications of the ZEDM stereo camera. Figure 3.3 shows how the stereo camera functions from capturing images, performing internal depth and pose estimations, and how the user can interface with this data.

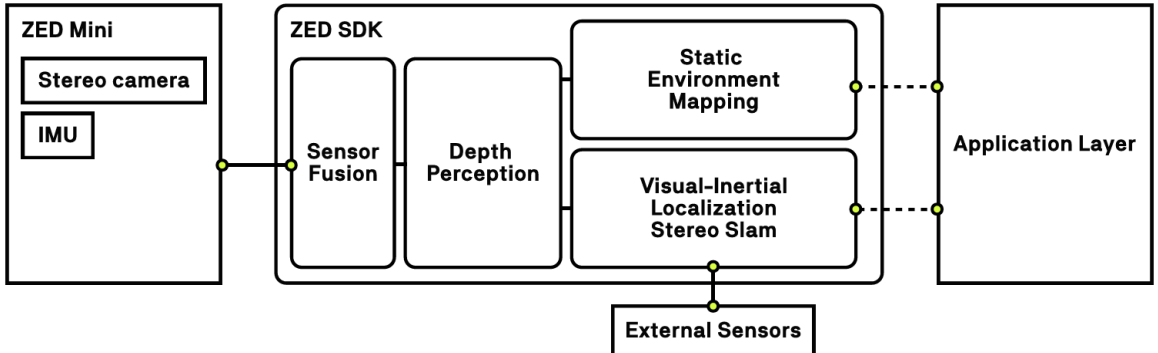


Figure 3.3: Functional diagram showcasing how the ZED SDK captures images, processes the data, and exposes data to the user [49].

Another key aspect of choosing this camera was due to its intended design of

¹StereoLabs. About Us. <https://www.stereolabs.com/en-ca/about>. Accessed June 26, 2025.

²Unity Technologies. <https://unity.com>. Accessed June 26, 2025.

³Epic Games. *Unreal Engine*. <https://www.unrealengine.com>. Accessed June 26, 2025.

Output Resolution	2x (2208x1242) – 15fps 2x (1920x1080) – 30fps 2x (1280x720) – 60fps 2x (672x376) – 100fps
Field of View	Max. 102°(H) x 57°(V) x 118°(D)
RGB Sensor Type	1/3" 4MP CMOS
Active Array Size	2688x1520 pixels per sensor (4MP)
Focal Length	3.06mm (0.12") – f/2.0
Shutter	Electronic synchronized rolling shutter
Interface	USB 3.0 Type-C port

Table 3.1: ZED Mini stereo camera specifications

mixed reality augmentation. The selected camera, unlike conventional stereo cameras, is intended to be mounted to the front of a virtual reality headset, allowing the user to visualize the physical world in stereo while superimposing virtual elements through passthrough AR features. The design concept is aligned with the project’s goal of an MR-based human-robot interaction system, where the robot’s physical workspace needs to be visualized in real-time with natural human vision as well as superimposed digital data. It also includes a host of additional features readily accessible to developers such as real-time spatial mapping, visual-inertial odometry, spatial anchoring, and occlusion management within mixed reality.

3.2 Robot Kinematics

One of the core concepts in robotics is kinematics, which describes the study of motion without considering forces. In robotic manipulation, specifically for serial-link robot arms, kinematic modelling is a key step in understanding how the joints

of a robot relates to the position and orientation of its end effector in 3D space. Robotic manipulators are composed of links and joints, commonly connected in a serial configuration. Each joint invokes a degree-of-freedom (DOF) which is commonly a rotational (revolute) or translational (prismatic) joint. The joints increase the robots ability to position its end effector in space. A robot arm is typically represented using a kinematic chain, which reduces the physical setup of the arm to its links, joints, and coordinate frames as depicted in Figure 3.4.

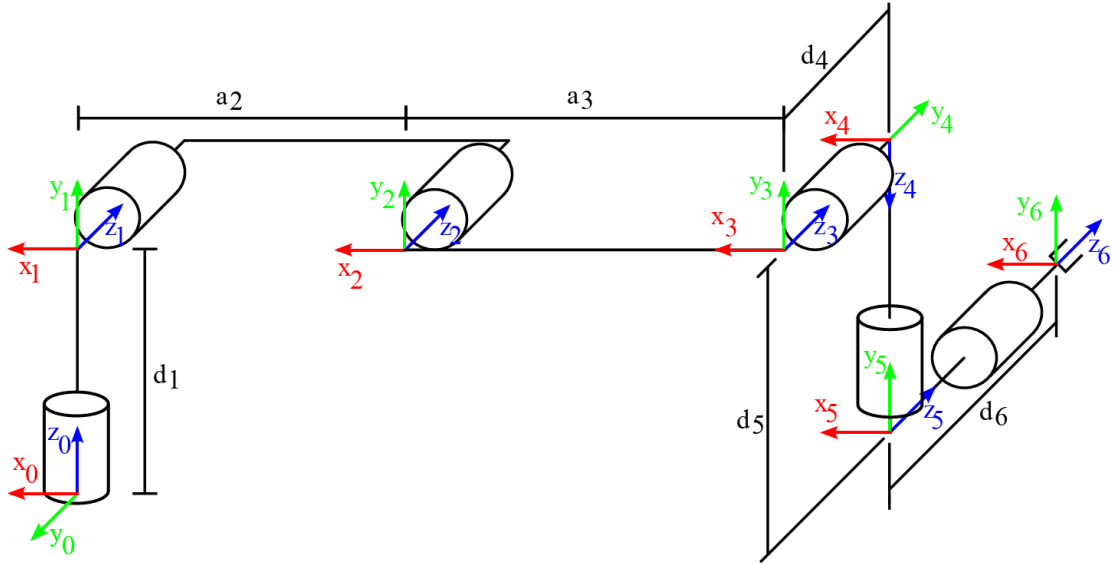


Figure 3.4: Robot kinematic chain setup for a 6-axis serial robot arm depicting the D-H parameter convention [50].

Forward kinematics refers to the process of determining the position and orientation of the end effector given the robot's current joint angles, which is a key process in robot motion planning and is represented using homogeneous transformation matrices. A robot arm with six rotational joints requires the setup of six of these matrices, which will be multiplied to retrieve the final transform representing the end effector's position and orientation in the base frame [51]. The mathematical process is simplified using the Denavit–Hartenberg (D-H) convention, which standardizes the way transformations are defined between adjacent links using the following four parameters:

1. θ_i : joint angle (varies for revolute joints).
2. d_i : link offset (varies for prismatic joints).
3. a_i : link length (constant).
4. α_i : link twist (constant).

Each transformation matrix $\mathbf{T}_i$ from link $i - 1$ to i is represented as:

$$\mathbf{T}_i = \begin{bmatrix} \cos \theta_i & -\sin \theta_i \cos \alpha_i & \sin \theta_i \sin \alpha_i & a_i \cos \theta_i \\ \sin \theta_i & \cos \theta_i \cos \alpha_i & -\cos \theta_i \sin \alpha_i & a_i \sin \theta_i \\ 0 & \sin \alpha_i & \cos \alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.3)$$

From here, the total forward kinematic solution is computed by:

$$\mathbf{T}_0^n = \prod_{i=1}^n T_i \quad (3.4)$$

Inverse kinematics refers to the process of determining the required joint parameters to achieve a desired end effector pose. Inverse kinematics is significantly more challenging than forward kinematics since mapping pose to joint space is nonlinear, calculations result in multiple, non, or even an infinite number of solutions, and the joint limits and singularities must be taken into consideration. For a serial robot with simple geometry, closed-form inverse kinematic solutions can be derived, which takes the form of solving geometric equations and using trigonometric identities to compute joint angles. Analytical solutions to this challenge are fast and interpretable, however, only applies to specific robot types. For more general manipulators, numerical methods are used, the most common being the Jacobian-based iterative algorithm [52]. In this method, the Jacobian matrix J defines the linear relationship between joint velocities $\dot{\theta}$ and end effector velocities $\dot{x}$:

$$\dot{x} = J(\theta) \cdot \dot{\theta} \quad (3.5)$$

Where the Jacobian is defined as the rate of change of the end effector position with respect to the joint angles:

$$J(\theta) = \begin{bmatrix} \frac{\partial x}{\partial \theta_1} & \frac{\partial x}{\partial \theta_2} & \dots & \frac{\partial x}{\partial \theta_n} \\ \frac{\partial y}{\partial \theta_1} & \frac{\partial y}{\partial \theta_2} & \dots & \frac{\partial y}{\partial \theta_n} \\ \frac{\partial z}{\partial \theta_1} & \frac{\partial z}{\partial \theta_2} & \dots & \frac{\partial z}{\partial \theta_n} \end{bmatrix} \quad (3.6)$$

From here, a small change in joint angles, $\Delta\theta$, is computed to move the end effector closer to the desired pose:

$$\Delta\theta = J^+ \cdot \Delta x \quad (3.7)$$

In the above formula, J^+ represents the pseudo-inverse of the Jacobian and Δx is the error between the current and desired end effector position [53]. With this, the joint angles are iteratively updated, with a learning rate α until the end effector pose converges to the desired pose within a threshold:

$$\theta_{k+1} = \theta_k + \alpha\Delta\theta \quad (3.8)$$

$$(\mathbf{x}_{\text{current}} - \mathbf{x}_{\text{target}})^2 \approx 0 \quad (3.9)$$

3.3 Simulation Development

The following subsections will discuss the simulation environment, its specifications, systems, and packages included that make it an ideal choice for the system validation process. It also describes in detail concepts used for developing the robot arm 3D model that is used for the simulation, the programming and mathematical processes implemented to develop motion planning solutions, and physics software developed to interact with the robot's environment. Following this will be a detailed description on the communication pipelines used and the development process of the user control interface.

3.3.1 Simulation Environment and Architecture

The simulation environment used for this project was the Unity Engine. Unity is a cross-platform game development engine that enables developers to create realistic video games, simulations, and mixed reality applications. The proposed system was achieved using 3D models, graphics, and programming logic using the C# object-oriented programming language. Although many robotic simulation platforms exist, including but not limited to Gazebo, NVIDIA Isaac Sim, and Webots, Unity provides a bundle of mixed reality development interfaces, such as built-in support for OpenXR tools, as well as the ability to develop real-time physics based simulations. Unlike other robotic simulation environments, Unity provides robust UI/UX tools for creating interactive control interfaces and visual overlays which is a key component for HRI research. Another strong reason for choosing this engine was Unity’s built-in support for real-time physics simulation through the NVIDIA PhysX 3.4 engine that enable for stable and turnable rigid-body dynamics, collision detection, and configurable joints. These features make it possible to simulate robotic kinematic chains, gripper mechanics, and interaction with physical objects in the simulated workspace.

The design of the HRI system focuses on creating an immersive, hands-free, and safe interface for teleoperating a robot arm to perform hazardous tasks. To simulate this system effectively, several core features must be developed. First, a real-time simulated robot arm, that can receive a target position and orientation and compute the necessary joint angles through inverse kinematics to reach that pose, was required. Second, a stereo camera system must be simulated to replicate human-like vision of the robot’s workspace, and it must be integrated with a VR headset to provide immersive visual feedback to the operator. Third, algorithms must be developed for real-time hand tracking and gesture recognition, which enables intuitive control inputs from the user. Lastly, the control inputs must be mapped into the robot’s local coordinate system to enable accurate and responsive teleoperation of the robot

arm in real-time.

3.3.2 Robot Arm Modelling

The first step in designing a real-time simulated robot arm was selecting a 3D model that represented a real-world industrial grade robotic manipulator, namely the robots developed by KUKA. The KUKA robot arms are ideal for simulation as they are the standard for performing hazardous tasks, often found in construction applications. The model used for this simulation was retrieved from the open-source 3D modelling Sketchfab website⁴. The 3D model features a 4-DOF robot arm, in FBX format. The robot arm model underwent many modifications to organize the multi-component mesh into a structured hierarchical set of objects, where each rotating joint, connecting each of the links together, was aligned relative to a global coordinate system. In addition to the robot arm model, the environment around the arm was designed to enable the robot arm to perform hazardous tasks. To achieve this, 3D objects representing wooden pallets, along with bricks, were placed in the simulation workspace. Collision physics were added to these components to make them interactable during run-time. Figure 3.5 shows the devised simulated environment.

⁴Sketchfab. *Robot Arm Model*. <https://sketchfab.com/3d-models/robotic-arm-daaffeda139a441cbb3ca9c835045fb4>

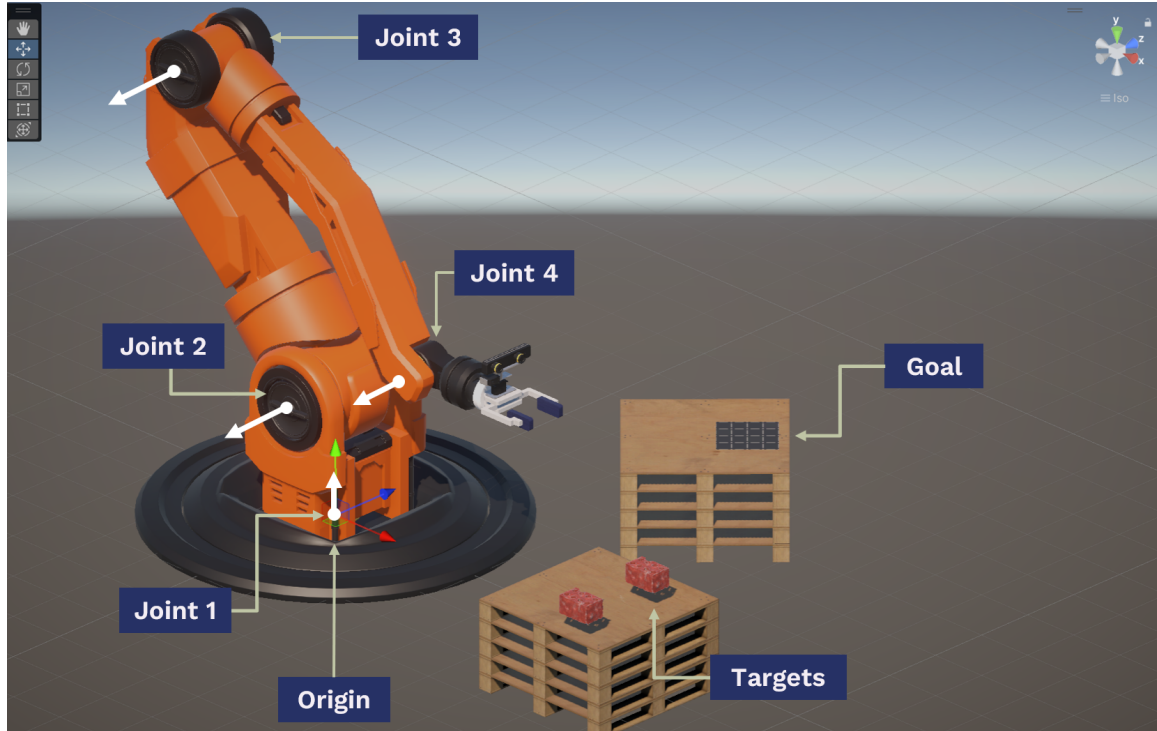


Figure 3.5: Developed simulation environment in the Unity engine for a 4-DOF robot arm.

To manipulate objects within the robot's workspace, a gripper system had to be simulated. To achieve this, a parametric model of a prismatic gripper was designed using the Autodesk Fusion 360 CAD software. A prismatic gripper is a type of robot gripper that simulates a two-finger setup. They are one of the most commonly used grippers in industrial applications, are easy to manufacture, and are cost-effective. Once designed, the model was imported into the simulation environment in Unity and attached to the end effector of the robot arm. A gripper controller script was written that enables the system to interact with its environment, including detecting collisions with components and the ability to pick-and-place objects in the simulation workspace. The designed gripper solution is depicted in Figure 3.6.

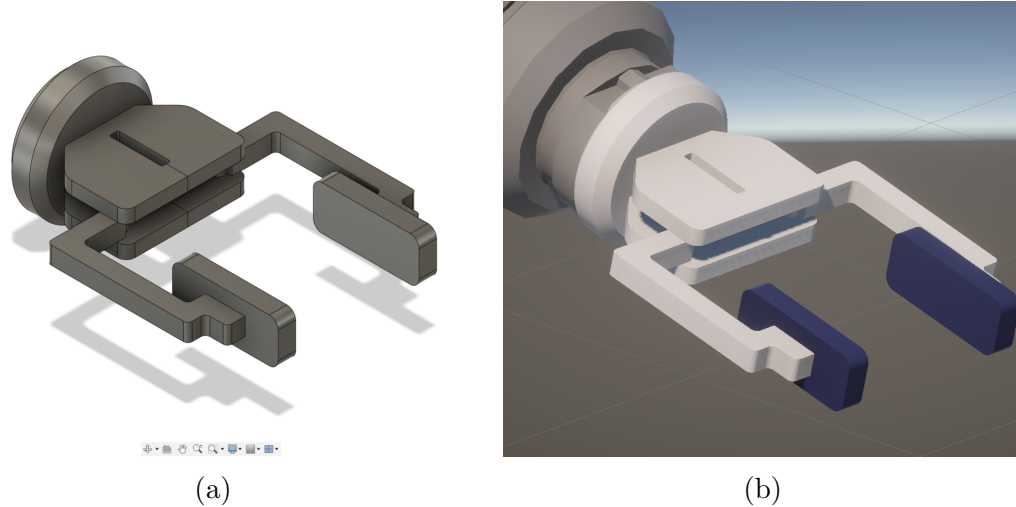


Figure 3.6: The parametric CAD model of the designed gripper in Fusion 360 shown on the left, and its integration in the robotic simulation environment, within the Unity Engine, on the right.

This constructed 3D model represents the visual rendering of the robot arm, along with collision dynamics. However, when performing costly mathematical computations like inverse kinematics, smaller and more interoperable representations of the robot arm are required. When simulating robot arms, their kinematic chain can be represented using a Unified Robotics Description Format (URDF) file. A URDF file is a low-level description file, written in the Extensible Markup Language (XML), which describes the geometry, mobility, and their corresponding relationships, of a robotic manipulator [54]. As will be discussed in more detail in the next section, URDF files are required as input parameters for open-source inverse kinematic solvers, as they enable the system to initialize the robot’s kinematic chain and its joint limits.

3.3.3 Inverse Kinematics and Control Integration

To actuate the robot arm, an inverse kinematic solver is required. Many open-source inverse kinematic solvers exist and simplify the process of retrieving target joint angles. Without these libraries, computing inverse kinematics becomes cumbersome, as changes made to the robot arm’s kinematic chain will require the derivation of new analytical closed-form solutions. For this application, the IKPy library was used.

IKPy is a Python library that performs inverse kinematics solutions efficiently, given a kinematic chain, as a URDF file, a target position, and orientation [55]. A Python script was written that imports the constructed URDF file of the 4-DOF robot arm used in the simulation which enables the system to initialize the kinematic chain of the robot arm. To connect the Python inverse kinematic scripts to the Unity simulation environment, a Transmission Control Protocol (TCP) server-client system was designed. TCP is a data link protocol that enables the transmission and reception of messages between devices, over an internet connection. A three-way handshake connection between two devices enables the host server to send packets of information to the client, where the client transmits a message back to the server, confirming data reception. TCP's error handling capability ensures reliable point-to-point communication, without data loss [56]. Python TCP server scripts which integrate the inverse kinematic solvers, and the corresponding C# client scripts, were developed to establish a connection between the computing server and the simulation environment client. During run-time, a Transform object placed in the robot's workspace acts as the target position and orientation input system. In Unity, a Transform object is an empty object that does not contain a visual rendering system; rather only position and orientation data. C# scripts developed retrieve the target position and orientation and format them into a transmittable message that is encoded using the UTF-8 character encoding standard, which is then transmitted to the Python server. The server decodes this message using the same format, and with the initialized kinematic chain, computes an inverse kinematic solution given the input pose. The system returns a set of joint angles, which is transmitted back to the Unity client system. Within the simulation environment, a Joint Space Controller script was designed that parses the received joint angle data, and using linear interpolation, moves the joint angles of the robot arm model from their current angle to the target angle, over time. The developed pipeline is summarized in a functional diagram, as depicted in Figure 3.7.

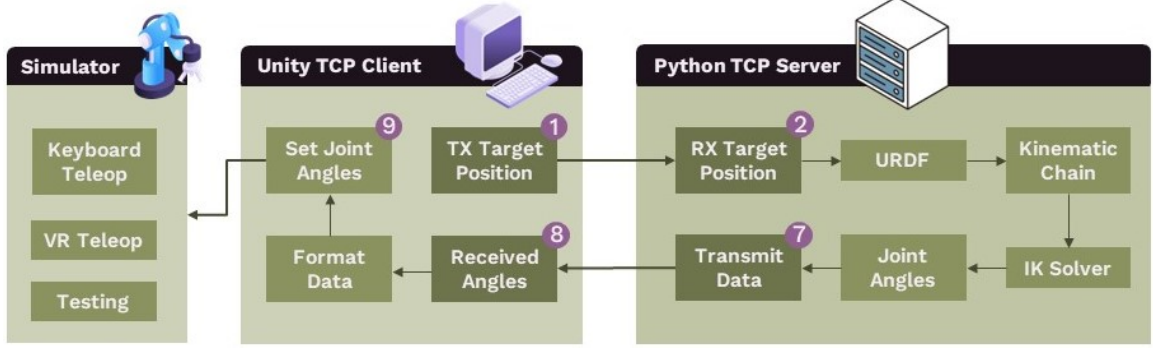


Figure 3.7: Real-time 4-DOF robot arm inverse kinematic simulation workflow diagram.

With the devised system, the 3D model of the robot arm is now able to move to various positions and orientations in the simulation environment as the target Transform object is repositioned at run-time. The developed system completes the core functionality of the robotic simulation. The next steps include developing the stereo camera systems followed by the mixed reality interface.

3.3.4 Stereo Camera and HMD Emulation

One of the core components of the teleoperation interface is simulating the stereo camera, to enable a first-person view (FPV) of the robot's workspace. In order to visualize the output of the stereo camera from its location, an open-source CAD model of the ZEDM camera, made available by Stereo Labs, was integrated into the simulation workspace. The camera was positioned on the end effector, the gripper, of the robot arm. To visualize the camera's perspective in a VR headset, the predefined VR interaction class, provided by OpenXR, was instantiated and placed at the same position and orientation of the camera model. The integrated class enables the camera's perspective to be rendered to the left and right displays of the HMD. However, even though with this current setup, the camera will move with the end effector during run-time, it only provides a forward-facing view of the robot's workspace. To increase operator awareness, a method was developed to actuate the camera's orientation based on the operator's head movements. The camera system was mounted

onto a 2-axis servo actuated gimbal system, as depicted in Figure 3.8.

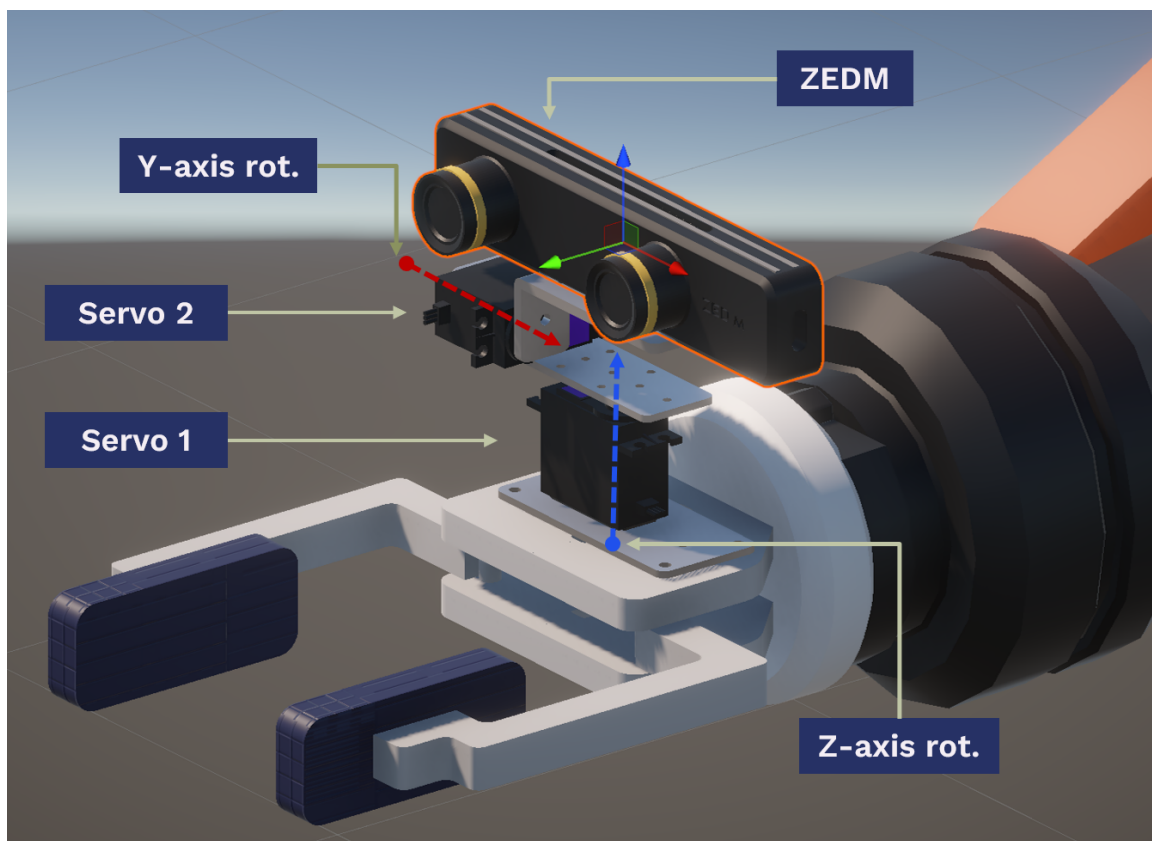


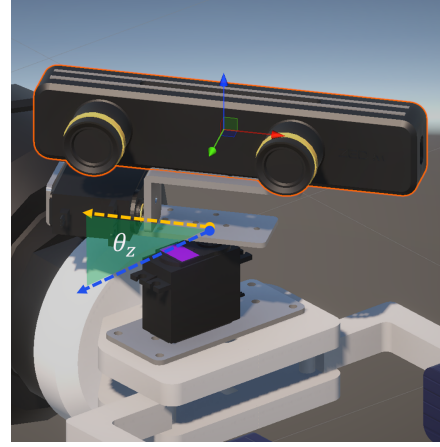
Figure 3.8: Simulated head tracking 2-axis servo actuated gimbal system for the ZEDM stereo camera.

This gimbal system consists of two simulated MG996R servo motors, one mounted on top of the other, with the bottom servo motor providing a yaw rotation (rotation around z-axis) and the top servo motor providing pitch rotation (rotation around the y-axis). It's important to note that these rotations are described with respect to the right-hand Cartesian coordinate system, however Unity follows a left-hand coordinate system where the conventional up-direction is the y-axis. The HMD has onboard IMU (Inertial Measurement Unit) sensors that continuously measure the position and orientation of the headset. As the operator looks around the simulated environment, the Euler angles of the headset are retrieved, and the corresponding roll and yaw angles are set as the target angles for the two servo motors. The setup enables the camera rig to track the head movements of the operator, in real-time,

allowing them to look around the robot’s simulated workspace, as shown in Figure 3.9. Natural head-tracking is of significance as MR wearable technology is prone to creating discomfort for users, which occurs when the physical movements of the operator do not align with their sensory input systems (vision, sound, touch). As a result, constructing a one-to-one human-like vision system was a key priority.



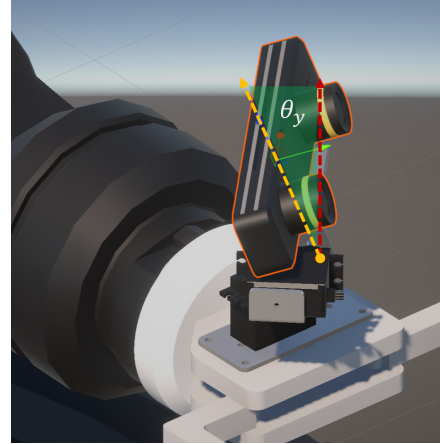
(a) Operator turning their head right, rotating about the z-axis by angle θ_z



(b) 2-axis gimbal rotating around the z-axis servo by the corresponding input yaw angle.



(c) Operator turning their head upwards, rotating about the y-axis by angle θ_y



(d) 2-axis gimbal rotating around the y-axis servo by the corresponding input pitch angle.

Figure 3.9: HMD head tracking system during simulation run-time.

With this system in place, the operator now has a mode to visualize the robot’s workspace in a first-person view with human-like perception, through the use of stereo

images and a VR headset. The next key component of this system is providing the ability for the operator to provide input control intuitively.

3.3.5 Hand Tracking and Gesture Recognition

The hand control system developed for this application was intended to behave as an intuitive method to provide control commands to the robot arm using gesture recognition techniques. The core functionality of this system depends on real-time hand tracking of the operator using the built-in cameras of the HMD. Most modern VR headsets, especially those developed by Meta (e.g., Quest 2 and Quest Pro), are equipped with multiple front-facing cameras that perform inside-out tracking. The cameras on the Meta Quest 2, used for this project, are monochrome infrared (IR) which are used for headset positional tracking, environment spatial mapping, and controller tracking. Additionally, they enable markerless hand tracking by capturing images of the user's hands and utilizing computer vision and machine learning algorithms to estimate the 3D pose of the hands in real-time [57]. Within Unity Engine, these hand tracking systems are enabled using the OpenXR plugin, which provides access to VR runtime APIs with hand tracking features across multiple hardware platforms. Specifically, the plugin provides an interface to retrieve joint pose data for each hand, which includes 3D positions and orientations for 21 key points, as illustrated in Figure 3.10. The hand tracking data are real-time and dynamic, as it is updated per frame, allowing for the extraction of high-frequency joint positions. Although a very intuitive interaction system, the accuracy and robustness of hand tracking are influenced by the following factors:

1. **Lighting conditions:** Low-light or overexposed environments will degrade the IR image quality resulting in less reliable tracking.
2. **Hand orientation:** Obscure hand poses or occlusions, such as one hand covering the other, will result in loss of tracking and jitter.

3. **Field of view:** The user's hands must be within the headset's camera field-of-view for continuous tracking.
4. **Hardware constraints:** While the Meta Quest 2 provides reliable tracking accuracy, newer headsets like the Quest 3 and Pro feature improved sensor arrays and depth sensing technology for increased tracking accuracy [58].

Studies reviewing the capabilities of the Meta Quest 2 HMD show that it can achieve a hand position tracking error of 1.1 cm, given optimal environment conditions, with a joint angle error of 9.6°, and a temporal delay of 45.0 ms [59].

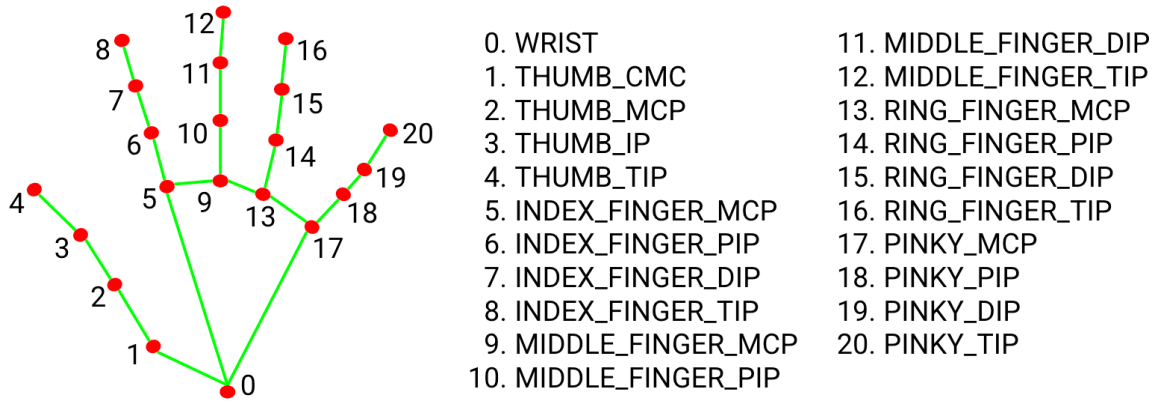


Figure 3.10: Hand tracking joint configuration convention [60].

The next stage in developing hand control was implementing reliable gesture recognition algorithms. The goal was to ensure that the robot arm only responds to intentional user commands and avoid unintended movement caused by natural hand motion during tracking and operation. Intent recognition was achieved by devising two distinct hand gestures: a pinch gesture and a grip (or fist) gesture. The pinch gesture enables direction control input of the end effector. When it has been detected, the algorithm begins by computing a 3D directional vector, which is denoted as $\mathbf{r}_{\mathbf{O}\mathbf{H}}$, between a fixed point, $\mathbf{r}_{\mathbf{O}}$ which is an offset applied from the HMDs origin, and the current position of the tracked hand model, denoted as $\mathbf{r}_{\mathbf{H}}$. The vector defines the direction the user intends the robot arm's end effector to move towards, as represented

by Equation 3.10. However, the tracking system is prone to false positives when the user unintentionally forms a pinch-like gesture during normal hand movement. Therefore, to increase reliability, an additional condition was introduced to form a more intentional composite gesture. The composite gesture combines a pinch gesture, between the thumb and index finger, while simultaneously detecting a grip gesture invoked by the middle, ring, and pinky fingers. The combination of gestures ensures that the robot arm only moves when the user performs a pinch-and-fist gesture with their right hand, which adds a layer of intent. The gesture allows for intent recognition since it requires coordination across multiple fingers. Once the gesture has been detected by the system, it automatically computes the unit vector direction, denoted as $r_{\hat{OH}}$, from $\mathbf{r}_{\vec{OH}}$, as shown in Equation 3.11. The unit vector is then multiplied by a constant scaler, C , which determines the movement speed of the robot arm per update, and scaled by ΔT , the elapsed simulation time between frames, to ensure time-consistent updates. The resulting direction vector is summed with the current position of the end effector $\mathbf{T}_t$, yielding a new target position $\mathbf{T}_t$ which is input to the inverse kinematics solver. The full mathematical solution is described in the following equations:

$$\mathbf{r}_{\vec{OH}} = \mathbf{r}_{\vec{H}} - \mathbf{r}_{\vec{O}} = \begin{bmatrix} x_H - x_O \\ y_H - y_O \\ z_H - z_O \end{bmatrix} \quad (3.10)$$

$$r_{\hat{OH}} = \frac{\mathbf{r}_{\vec{OH}}}{|\mathbf{r}_{\vec{OH}}|} = \frac{\langle x_H - x_O, y_H - y_O, z_H - z_O \rangle}{\sqrt{(x_H - x_O)^2 + (y_H - y_O)^2 + (z_H - z_O)^2}} \quad (3.11)$$

$$\mathbf{T}_{t+1} = \mathbf{T}_t + C\Delta t \times r_{\hat{OH}} = \begin{bmatrix} x_t + C\Delta t \times r_{\hat{OH}_x} \\ y_t + C\Delta t \times r_{\hat{OH}_y} \\ z_t + C\Delta t \times r_{\hat{OH}_z} \end{bmatrix} \quad (3.12)$$

To classify gestures, deterministic threshold-based algorithms were developed. These algorithms relied on the spatial relationships between the tracked joints provided by

the OpenXR hand tracking system. Specifically:

1. **Figure 3.11** shows the distance between joints 4 and 8 during a pinch gesture. When the hand is open, the distance remains above a defined threshold and as the user pinches, the distance quickly decrease.
2. **Figure 3.12** illustrates the distances between joints 12, 16, and 20 to the wrist (joint 0) during a grip gesture. As the hand closes, these distances contract below a second threshold.
3. **Figure 3.13** highlights the composite gesture detection, showing both pinch and grips components occurring simultaneously over-time.

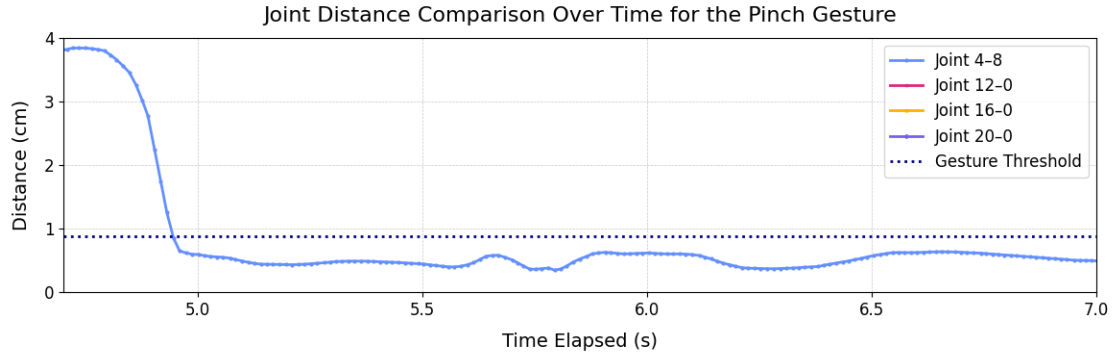


Figure 3.11: Relative distances between the tip of the thumb and index fingers.

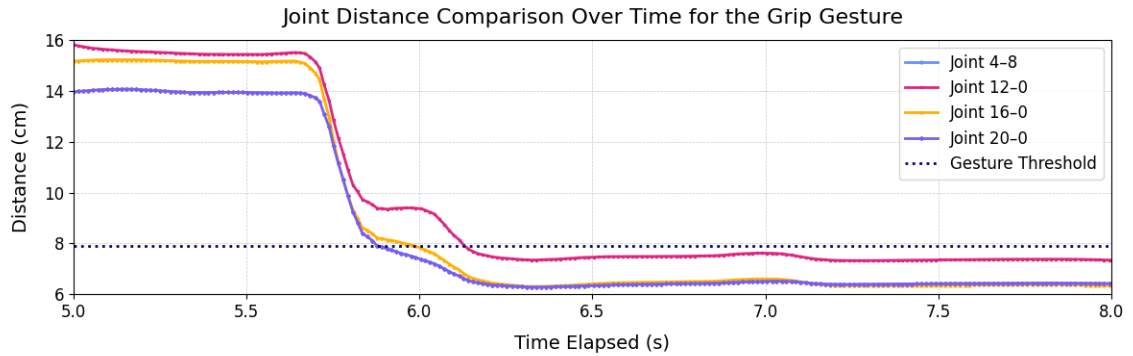


Figure 3.12: Relative distances of the tip of the middle, index, and pinky fingers relative to the wrist.

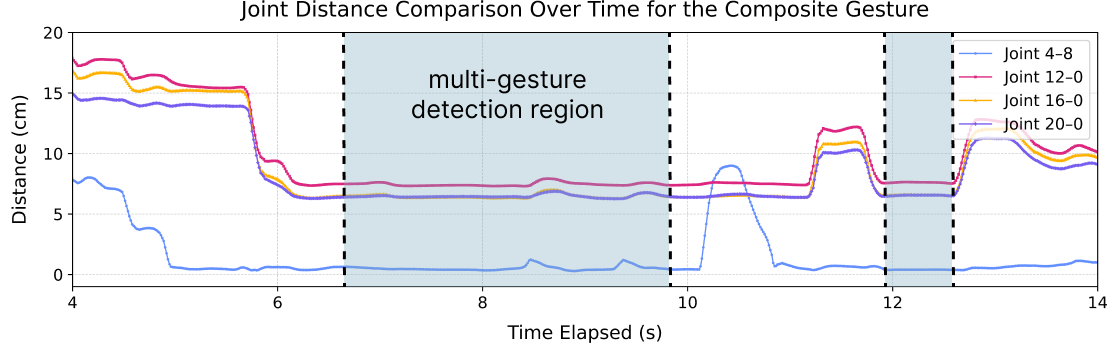


Figure 3.13: Relative distances for all joints required for the composite direction control vector, where peaks of the plot represent the resting hand positions and troughs represent an invoked gesture.

In addition to enabling intuitive omnidirectional control of the robot arm using directional hand gestures, it was necessary to develop a method to interact with objects in the simulated workspace. Object interaction was achieved by developing a gesture-based gripper control system, allowing users to intentionally open and close the prismatic gripper mounted to the robot arm’s end effector. The algorithm developed for invoking this control logic was based on recognition of repeated grip gestures (i.e., closed fist of the right hand) under a short time window. Further, the system extended the definition of a grip to include the simultaneous closure of four fingers: index, middle, ring, and pinky, relative to the wrist joint. These distances are continuously monitored in real-time and thresholds were determined using the same procedure as per the previous gesture recognition algorithms. The algorithm functioned by detecting how many times a grip gesture was invoked within a one-second window. Specifically:

1. **Two consecutive grip gestures** within the one-second time window triggers a **gripper open** command.
2. **Three consecutive grip gestures** within the one-second time window triggers a **gripper close** command.

To prevent gesture misclassifications due to holding the hand in a closed form, a flag is maintained to ensure that only new grip instances increment the count. Each time a new grip is detected, the timer is reset to allow further gesture detection with the current recognition window. Once the timer completed, the counter is evaluated to determine the appropriate gripper command. The devised system ensures deliberate and reliable control of the robot arm's gripper. The devised algorithm is summarized in pseudo-code in Algorithm 1.

Algorithm 1 Gripper Gesture Detection Algorithm

```

1: Initialize:  $gripCount \leftarrow 0$ ,  $gripTimer \leftarrow 0$ ,  $gripWindow \leftarrow 1.0$ ,  $isGripping \leftarrow$ 
   false
2: while simulation is running do
3:   Compute distances:
4:      $d_{8,0} \leftarrow \text{distance}(\text{index\_tip}, \text{wrist})$ 
5:      $d_{12,0} \leftarrow \text{distance}(\text{middle\_tip}, \text{wrist})$ 
6:      $d_{16,0} \leftarrow \text{distance}(\text{ring\_tip}, \text{wrist})$ 
7:      $d_{20,0} \leftarrow \text{distance}(\text{pinky\_tip}, \text{wrist})$ 
8:   if  $d_{8,0}, d_{12,0}, d_{16,0}, d_{20,0} < \text{gripThreshold}$  then
9:     if not  $isGripping$  then
10:       $isGripping \leftarrow \text{true}$ 
11:       $gripCount \leftarrow gripCount + 1$ 
12:       $gripTimer \leftarrow gripWindow$ 
13:    end if
14:  else
15:     $isGripping \leftarrow \text{false}$ 
16:  end if
17:  if  $gripTimer > 0$  then
18:     $gripTimer \leftarrow gripTimer - \Delta t$ 
19:    if  $gripTimer \leq 0$  then
20:      if  $gripCount = 2$  then
21:        open gripper:  $handControlData[6] \leftarrow 1.0$ 
22:      else if  $gripCount = 3$  then
23:        close gripper:  $handControlData[6] \leftarrow 0.0$ 
24:      end if
25:      Reset:  $gripCount \leftarrow 0$ 
26:    end if
27:  end if
28: end while

```

3.3.6 Mixed Reality Teleoperation Interface

The final teleoperation interface integrates real-time stereo vision, gesture-based control, and inverse kinematics into a cohesive simulation system, that allows users to control a robot arm with natural hand movements. The interface was designed to replicate a first-person, immersive control experience, enabling operators to teleoperate a high-degree of freedom robotic manipulator in hazardous environments without the need for conventional input devices.

To summarize the developed simulation, the operator is able to wear a VR HMD that captures the robot’s workspace with natural binocular video through the Unity XR systems. The two axis servo-actuated gimbal enables the operator to freely look around their environment in order to gain a higher sense of situational awareness. By reconstructing the robot’s workspace from the end effector’s perspective, the interface allows the user to perceive the scene as if looking through the robot’s ”eyes”. Hand tracking is performed using the OpenXR framework and is fully integrated into the simulation environment. The user’s hands are tracked in 3D space using the built-in spatial mapping IR cameras of the HMD, where the 3D joint coordinates of both hands are extracted per frame. To invoke robot movement, a deterministic gesture recognition system detects a composite gesture formed by a simultaneous pinch and grip of the right hand. The gesture invokes an algorithm that computes a normalized direction vector that is used to generate a target pose for the robot arm. The computed unit vector is passed to an inverse kinematics solver, which calculates the required joint angles to achieve the pose, and the robot arm is moved accordingly and iteratively. To provide interaction with the simulation environment, a secondary gesture control system was implemented for gripper actuation. By making a full hand grip gesture, within a fixed time window, the user can command the gripper to open or close. Two consecutive grips result in an open command, while three result in a close command. The method ensures that gripper commands are intentional, which

reduces the risk of accidental object manipulation.

Overall, the simulated mixed reality teleoperation interface enables for immersive control of a robot arm, which increases depth perception by allowing the user to instantly perceive depth, instead of requiring operators to develop 3D mental models from 2D images. It also removes the rudimentary keyboard or joystick input system, through the user of hand tracking and gesture recognition. A visual of the interface in operation during the simulation run-time is shown in Figure 3.14.

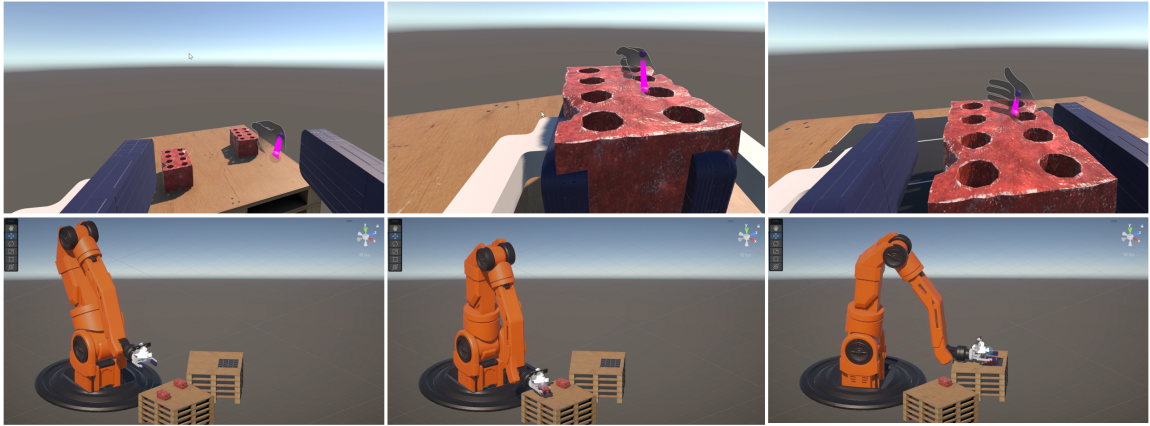


Figure 3.14: Mixed reality teleoperation interface in simulation, showing the stereo camera feed from the robot arm’s end effector displayed in the headset, with real-time hand tracking and gesture control enabling intuitive arm control.

3.4 Hardware Development

The following subsections document the process of transforming the simulated software, iteratively, into a hardware solution. The section will begin by defining the overall devised system architecture, as well as detailed a description on the specific hardware used for this research. It also describes key communication pipelines used and developed to achieve real-time robot arm motion. Then, the various visual modalities designed for this interface will be discussed, as well as a description of the mathematical processes used to create the visual renderings and control algorithms.

3.4.1 Hardware Components and Architecture

Once the proposed teleoperation system was validated in the simulation environment, hardware integration followed. The goal of the hardware integration phase was to replicate the teleoperation architecture proven in the simulation and deploy it on a physical robot arm to evaluate its real-world feasibility and performance. The developed MR teleoperation interface consists of multiple hardware and software systems, which are categorized into three main segments: the User Interface, the Middleware, and the Robot Arm. The system is summarized in a high-level block diagram, as shown in Figure 3.15.

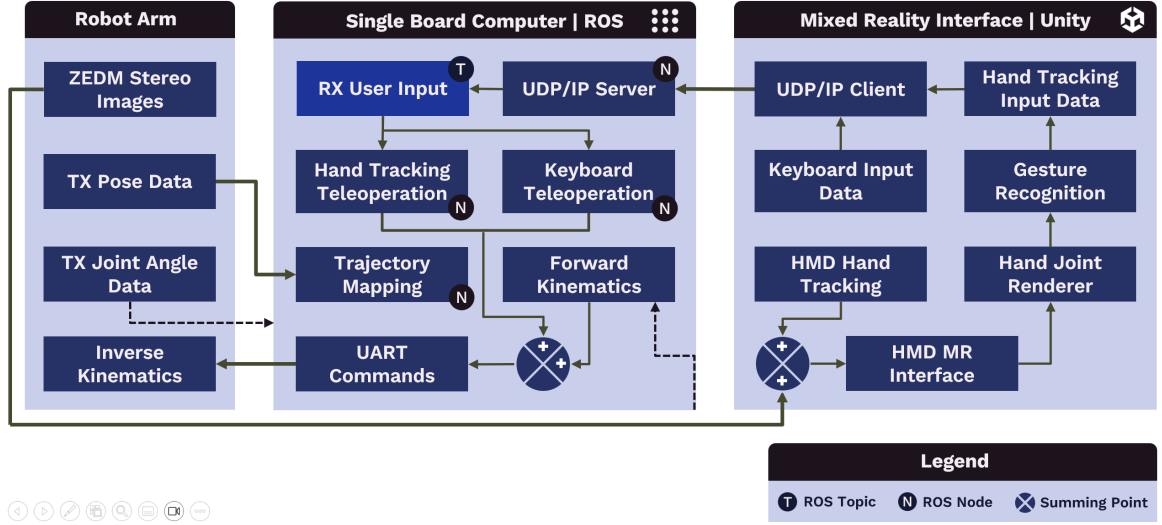


Figure 3.15: Hardware system integration architecture flowchart

User Interface Segment

The objective of the User Interface (UI) segment is for capturing user input and providing real-time mixed reality visualization. The system builds on the software designed for the simulation, in that it runs on the Unity Universal Render Pipeline (URP) engine and is executed on a Lenovo Legion Pro 5i laptop that is equipped with a 13th Gen Intel Core-i5-13500HX CPU, NVIDIA GeForce RTX 4050 GPU, and 16 GB of RAM. The previous section highlighted the various streaming resolutions and speeds of the ZEDM stereo camera used for this application. For the hardware in-

tegration, the stereo camera is set to run at HD1080 resolution at 30 FPS, yielding a horizontal FOV of 66° and a vertical FOV of 40° . The setting provides high fidelity images of the robot's workspace whilst providing a stable frame-rate for user comfort. The resolution was chosen by a trial-error basis, to determine the most comfortable user setting, whilst prioritizing frame rate and compute power. The camera is connected via USB-C from the robot arm to the main laptop for high-speed data transmission. The gesture recognition algorithms developed in the simulation component were integrated into the mixed-reality environment. The output of this process is sent to the Middleware segment.

Middleware Segment

The Middleware segment consists of a Raspberry Pi 4B single-board computer (SBC) running ROS2 Humble on the Ubuntu 22.04 operating system (OS). The SBC serves as a lightweight and capable computation node to bridge the connection between the Unity user interface with the physical robot arm. To achieve this communication bridge, a UDP/IP (User Datagram Protocol) socket-based protocol is used. The Unity client connects and continuously transmits control packets to a ROS2 UDP server running on the SBC. These packets contain the various control data, such as hand tracking directional vectors and gripper commands which are parsed and published using the ROS topic system. The Robot Operating System (ROS) is an ecosystem that provides a middleware solution for modular robotic system development. ROS topics abstract away hardware communication and allow different nodes to subscribe and publish control or sensor data asynchronously [61]. The communication architecture enables modularity and future scalability, and it is widely used in industry for robotic software and hardware development.

Robot Arm Segment

The research project used the myCobot 320 M5Stack, a 6-DOF collaborative robot arm developed by Elephant robotics, as depicted in Figure 3.16. The myCobot 320 was chosen because it is designed for research applications due to its open-source support and ease of integration with ROS. The arm provides interfaces for joint space and Cartesian space control, with a maximum reach of 350 mm. The arm contains two microcontrollers, each dedicated for various types of computations: an ESP32 responsible for controlling individual joints and an M5Stack core that processes high-level commands and Universal Asynchronous Receiver/Transmitter (UART) commands. The Raspberry Pi SBC communicates with the M5Stack controller using serial UART to send inverse kinematic commands, which the robot arm interprets and executes. Attached to the end effector of this arm is a prismatic gripper, also developed by Elephant Robotics, which was mounted using developed in-house 3D printed components. The factory brackets used for the robot arm were insufficient to mount both the gripper and the stereo camera, thus custom mounts were designed using Autodesk Fusion 360 and the Elegoo Neptune 3 Pro Fused Deposition Modelling (FDM) printer. The gripper is controlled using gesture recognition algorithms as devised previously: two consecutive grips close the gripper, while three consecutive grips open it. The gestures enable the user to not only reach for targets, but also interact with them through a full hands-free interface.

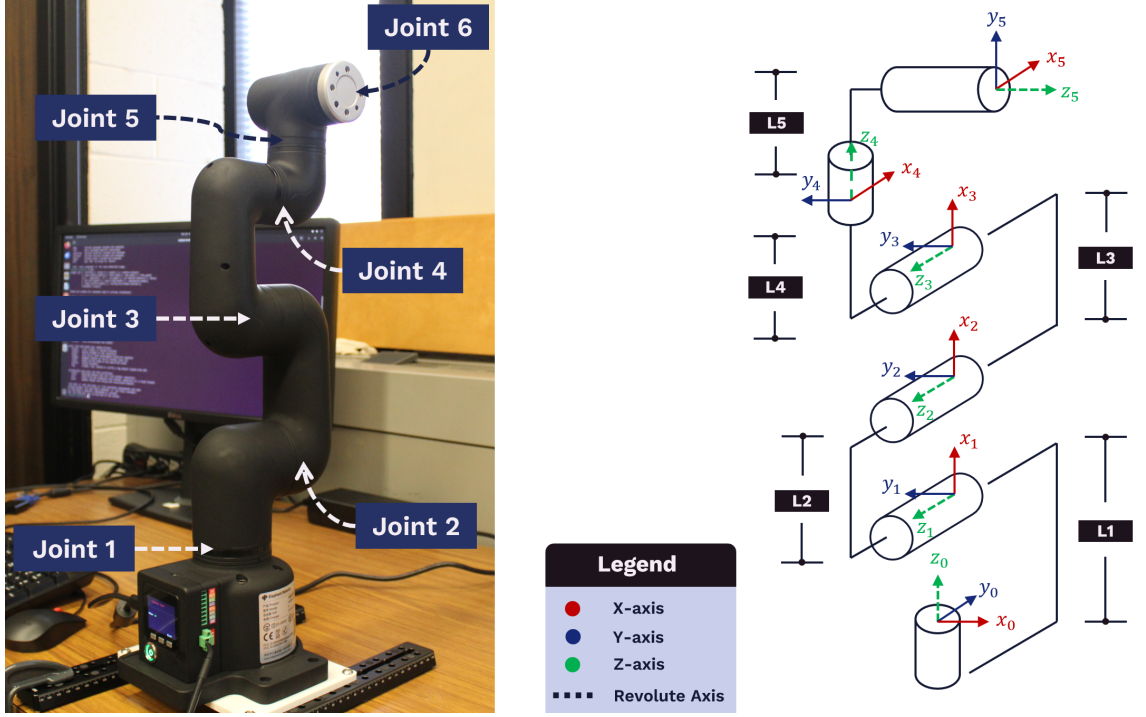


Figure 3.16: Description of the robot arm and setup (left) and diagram of the robot kinematic chain (right).

3.4.2 Teleoperation Modes

First-Person MR Hand Controller

The MR teleoperation interface developed in the hardware integration phase offers the operator two control modes, each designed to increase situational awareness and allow safe operation in hazardous manipulation tasks. The main instance of this is the First-Person MR Hand Tracking Control mode (FPV). In this mode, real-time stereo video from the ZEDM camera is streamed directly to the Meta Quest 2 headset, where the left and right camera views are mapped to the corresponding displays in the headset. The result is a stereoscopic view of the robot's workspace with human-like depth perception, which conventional teleoperation interfaces using 2D monitors cannot achieve. Even using 3D digital twins displayed on standard screens, users must infer depth cues, which adds cognitive overhead and increases hazards during operation due to spatial errors. Onward, the MR FPV control mode

overlays 3D hand skeleton models, representing the real-time tracked hand data, onto the stereo images. The 3D models give the operator the feeling that their own hands are directly interacting with robot's environment, which is also unachievable with conventional means of teleoperation. In this mode, the operator can point their hand in the direction they want the end effector to move in, using the composite pinch-grip gesture, as shown in Figure 3.17. The system interprets the direction vector from a fixed anchor point within the MR subspace, to the index fingertip of the right hand, which consequently maps it into a velocity command for the robot arm in Cartesian space.

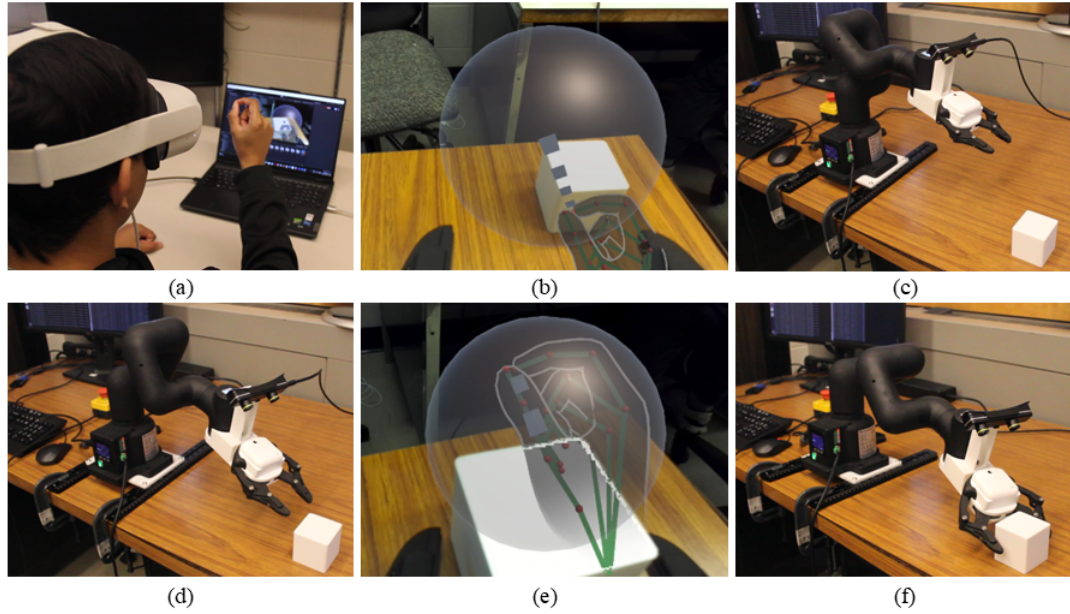


Figure 3.17: (a) Image of the operator wearing HMD, providing input in the MR interface, using hand gestures. (b) Operator's view as depicted in the VR HMD (FPV of the robot's workspace). (c) Robot arm at starting position. (d) Robot arm moving towards target using the hand tracking data. (e) User refining trajectory with gesture corrections. (f) Robot arm achieving the target position.

The control mode was first validated in the simulation phase, and then extended to the physical hardware. One of the challenges that arose in the real-world implementation was the difference in coordinate systems between the Unity engine and general robotics standards. Specifically, the Unity engine uses a left-handed coordi-

nate system that defines the conventional up-direction as the y-axis, which is different from the coordinate frame used in robotics which defines a right-handed system with the up-direction defined as the z-axis. To account for this discrepancy, a coordinate transformation is derived, which is used to transform the Unity direction unit vector $\hat{r}_{OH}$. The process is explained mathematically in Equation 3.13. In this transformation calculation, the constant C is a user-defined gain which controls the Euclidean distance that the robot arm should move per iteration, in millimetres.

$$\vec{\mathbf{T}}_{t+1} = \vec{\mathbf{T}}_t + C \cdot R \cdot \hat{r}_{OH} = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix} + C \cdot \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} r_{\hat{O}H_x} \\ r_{\hat{O}H_y} \\ r_{\hat{O}H_z} \end{bmatrix} \quad (3.13)$$

The MR FPV control mode also includes gesture recognition to operate the gripper, using the consecutive grip recognition software developed in the simulation phase which enables the operator to move the robot arm in 3D space using hand tracking and manipulate objects in the robot's workspace with simple hand gestures.

Third-Person Global Space Controller

The second control mode developed for the MR teleoperation system is the Third-Person Digital Twin MR Hand Tracking Control mode. The objective of this mode is to increase operator situational awareness from a spatial perspective, enabling safer and deliberate task execution. While the MR FPV mode provides accurate depth perception of the workspace, it limits the operator's awareness of the full configuration of the robot's kinematic chain. Specifically, FPV view occlude key information, such as joint link orientations or proximity to surrounding structures. The occluded view creates a safety hazard when the robot is operating near equipment or tight environments, where unobserved collisions may occur. To address this limitation, a digital twin of the physical robot arm and its environment is created within the MR subspace. The digital twin is achieved by using a CAD model of the myCobot 320 M5 arm, developed by Elephant Robotics, which is integrated into the MR scene.

The model is scaled to 1:1 real-world proportions, and is driven by a custom joint space controller which synchronizes the virtual model with the physical robot arm in real-time. The synchronization is achieved via ROS nodes running on the SBC that continuously fetch the current joint state parameters of the physical robot arm and transmit them to the Unity instance over UDP. In the Unity engine, this data is parsed and applied to the joint space controller of the CAD model, allowing the digital twin to reflect the actual pose on every frame update. The interface enables the user to observe the full arm configuration relative to its environment from an external vantage point.

To provide context to the surrounding workspace, a real-time voxel-based point cloud is generated using RGB-D data from the ZEDM stereo camera. As described in Section 3.1, stereo vision computes depth using triangulation across left and right images. The ZED SDK utilizes GPU-based processing to compute per-pixel depth maps from stereo disparity. In Unity, a custom algorithm was implemented that extracts RGB values and depth estimates from the camera’s output using the ZED SDK. Since per-pixel depth extraction and rendering is computationally expensive, a pixel-stride parameter was invoked to the algorithm to skip a fixed number of pixels during extraction. The resulting RGB-D data is organized into a structured array and is used to generate a point cloud via Unity’s built-in particle rendering system. Each rendered point is visualized as a small cube, known as a voxel, where the resolution of the point cloud is determined by the voxel size. A lower voxel size produces a higher resolution point cloud but increases rendering cost. Based on experimental performance and testing, a pixel stride of 3 and a voxel resolution of 0.01 meters were selected to balance fidelity and frame rate. Further, an inverse linear interpolation method was used to convert the pixel-depth maps into a heat map, where warmer colours represent objects closer to the camera and cooler colours for objects further away. The generated point cloud is dynamically rendered each frame from the end-effector of the digital twin, allowing the user to view the robot arm in context with

its environment. The code designed for this system is summarized in Algorithm 2 and Figure 3.18 shows a real-time rendering of the third-person controller.

Algorithm 2 Point Cloud Depth Extraction and Voxel Rendering

Input: ZED Mini stereo camera image stream, point cloud material from GPU

2: **Output:** Real-time RGB-D point cloud rendered via Unity particle system
Initialize ZED camera and allocate CPU-side point cloud buffer

4: Define stride s , max depth d_{max} , min depth d_{min} , max point count N_{max}
Create empty lists **PointPositions** and **PointColours**

6: **procedure** UPDATEPOINTCLOUD
 Clear **PointPositions**, **PointColours**

8: Retrieve 4-channel (XYZRGBA) point cloud image from ZED camera
 for $y = 0$ to height step s **do**

10: **for** $x = 0$ to width step s **do**
 if **PointPositions**.length $\geq N_{max}$ **then**

12: **break**
 end if

14: $(X, Y, Z, A) \leftarrow \text{getPointCloudValue}(x, y)$
 $P \leftarrow (X, Y, Z)$

16: range $\leftarrow \|P\|$
 if $d_{min} < \text{range} < d_{max}$ **then**

18: **if** heatmap disabled **then**
 Decode RGB color from packed alpha channel A

20: Append P to **PointPositions**
 Append decoded color to **PointColours**

22: **else**
 Normalize depth to $[0, 1]$

24: Convert to HSV color (red to blue)
 Append P to **PointPositions**

26: Append heatmap color to **PointColours**
 end if

28: **end if**

30: **end for**
 end procedure

32: **procedure** RENDERVOXELPARTICLES
 if Length of **PointPositions** equals **PointColours** **then**

34: **for** $i = 1$ to **PointPositions**.length **do**
 Set particle i position to **PointPositions** $[i]$

36: Set particle i color to **PointColours** $[i]$
 Set particle i size to predefined voxel size

38: **end for**
 Update Unity particle system with new particle array

40: **end if**
 end procedure

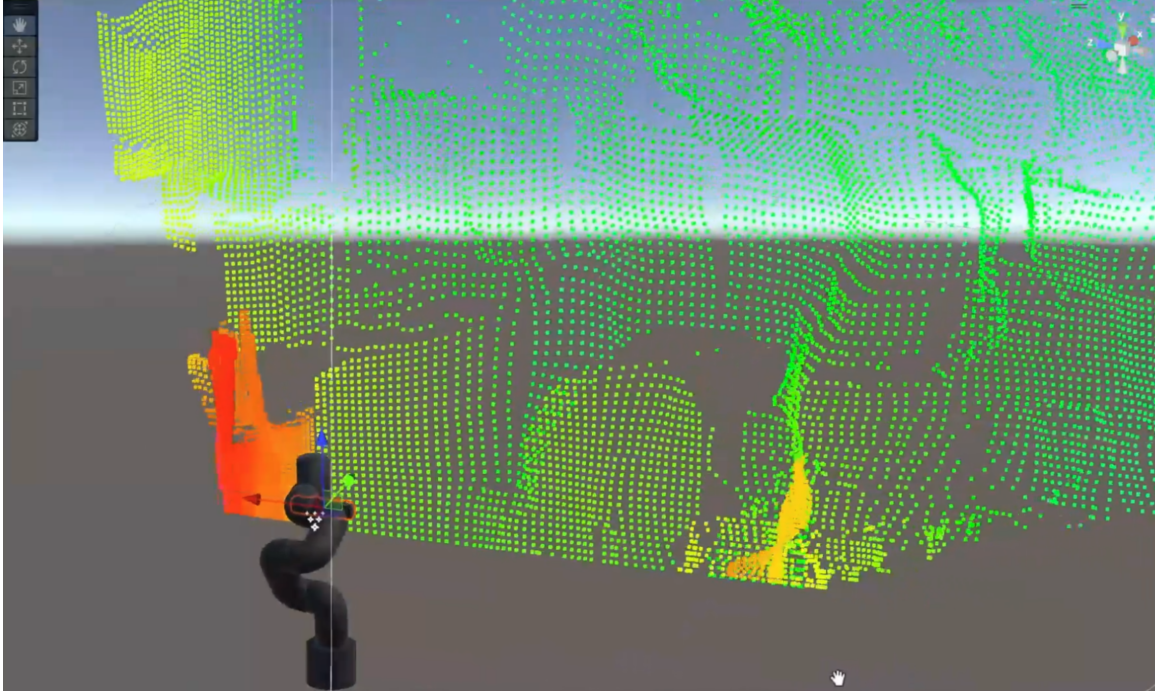


Figure 3.18: Third-person interface consisting of a real-time digital twin of the robot arm and a point cloud generated from the stereo camera.

The gesture recognition algorithm was extended in the digital twin environment, enabling the operator to teleoperate the robot arm for an external view. To achieve this, the previously designed pinch gesture was utilized. The objective was to enable positional control of the end effector whilst a grip gesture enables for both position and orientation control. For both instances of control, a target cube mesh is positioned at the end effector of the robot arm's digital twin. When the cube is grabbed using a pinch gesture and moved to a new location in the virtual environment, the position vector $\vec{\mathbf{T}}_{\mathbf{t}+1}$ relative to the robot's base coordinate system is computed, in the MR subspace. The relative position vector is transmitted to the ROS server which performs arithmetic by multiplying with the transformation matrix to convert to the robot's coordinate frame as previously. To enable orientation control of the end effector, first the target cube object's rotation is retrieved and stored as a quaternion, $\mathbf{q}_t$. The quaternion is transmitted to the ROS server and is converted into Euler angles, namely to roll, pitch, and yaw denoted by (ϕ, θ, ψ) . Following this, the current Euler

angles of the robot arm are retrieved, denoted by $(c_\phi, c_\theta, c_\psi)$. Each of the corresponding angles is summed and wrapped between $[0^\circ, 360^\circ]$. The mathematical operation is detailed in the following equations and Figure 3.19 showcases this control mode during operation.

$$\mathbf{q_t} = [x, y, z, w] \in \mathbb{R}^3 \quad (3.14)$$

$$\begin{aligned} \phi &= \arctan 2(2(wx + yz), 1 - 2(x^2 + y^2)) \\ \theta &= \arcsin(2(wy - zx)) \end{aligned} \quad (3.15)$$

$$\begin{aligned} \psi &= \arctan 2(2(wz + xy), 1 - 2(y^2 + z^2)) \\ \begin{bmatrix} r_\phi \\ r_\theta \\ r_\psi \end{bmatrix} &= \begin{bmatrix} c_\phi + \phi \\ c_\theta + \psi \\ c_\psi - \theta \end{bmatrix} \text{mod} 360^\circ \end{aligned} \quad (3.16)$$

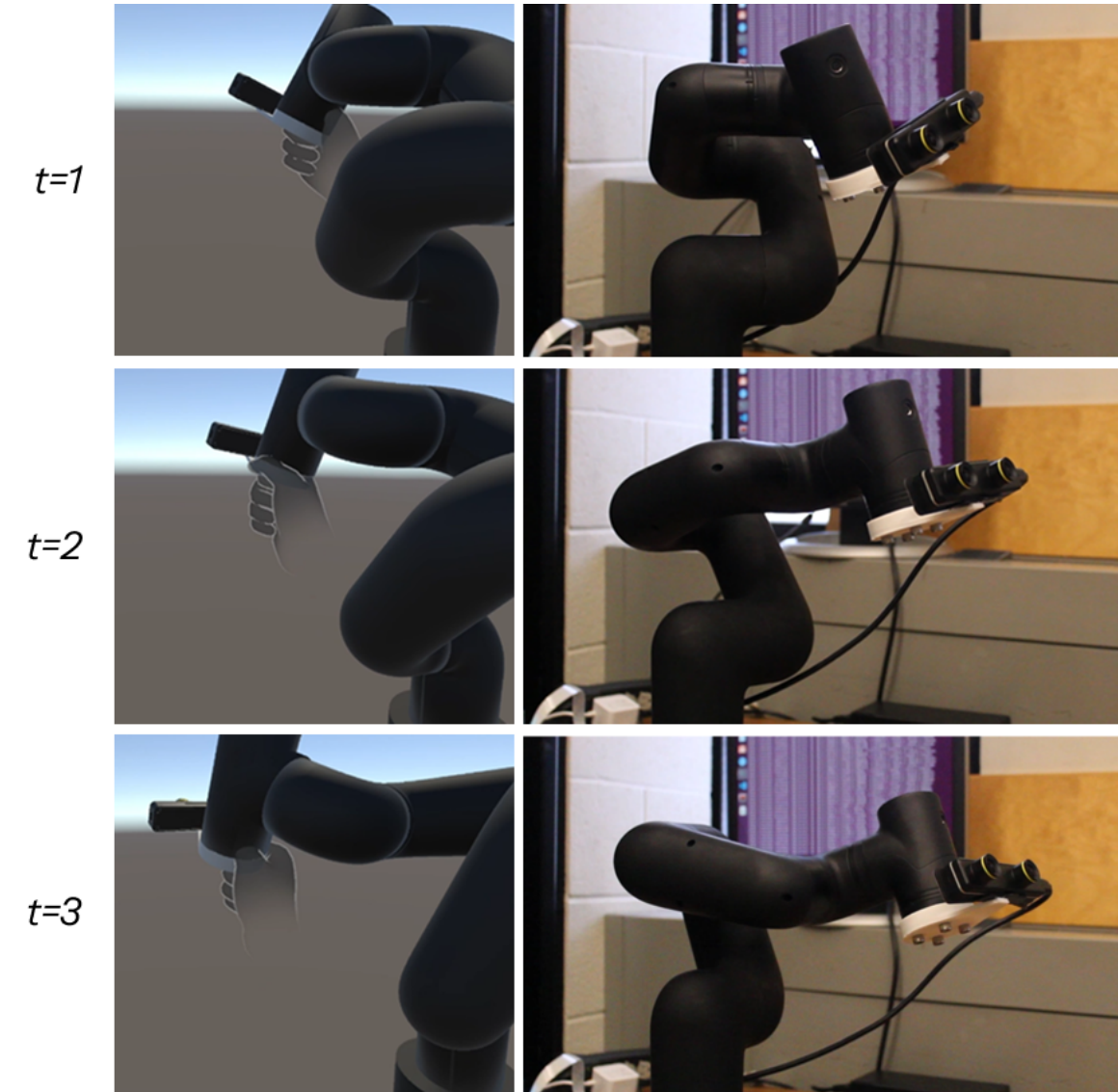


Figure 3.19: The third-person control system allowing the operator to manipulate the position and orientation of the end effector using the digital twin model of the robot arm with hand tracking.

Computer Video Keyboard Control

In addition to the MR control interface, as described in the previous two subsections, a conventional teleoperation system was devised, namely a Computer Video Feedback Keyboard Controller. The objective of this control mode was to serve as a baseline for comparison with the developed mixed reality teleoperation interface. Keyboard teleoperation is one of the most widely used input paradigms in robotics research

and remote operations, due to its simplicity, low implementation cost, and hardware accessibility [62]. In this control mode, discrete key presses are mapped to movements along the robot’s degrees of freedom, in Cartesian space. The input system enables the operator to manually move the end effector along the X , Y , and Z axes through defined increments per key press. To reconstruct this control system, a standard RGB video feed was provided by extracting the video stream from a single lens of the ZEDM stereo camera. The feed was displayed within the Unity engine interface on a conventional laptop screen, which effectively simulates a monocular camera view from the end effector. The operator relies purely on this 2D video feedback to perceive depth and spatial information, which is inherently limited compared to stereoscopic or immersive views.

The control inputs were implemented by developing custom C# scripts that map specific keyboard keys to robot motion commands. In this implementation, the arrow keys of the keyboard control the end effector along the X - Y plane. The **M** and **N** keys provide vertical displacement control, and the **Q** and **W** keys are used to control the open / close states of the gripper. The setup replicates the conventional mode of teleoperation found in robotics, which is used as a benchmark to evaluate the operational advantages of the proposed MR teleoperation interface.

3.4.3 System Communication Pipeline

The main architectural difference between the simulation and hardware implementations of the teleoperation interface is the communication protocol used to transmit the control data. Within the simulation, a TCP/IP socket-based protocol was used, where as in the physical implementation, a UDP/IP protocol was implemented. The protocol was chosen due to the trade-offs between the two communications protocols and their responsiveness in real-time robotic applications. TCP is a connection-based protocol that prioritizes reliability, ordered, and error-checked data transmission. It functions using a three-way handshake communication system, where a key process

includes retransmitting lost data packets to ensure data integrity. Although this is a secure and reliable method for general purpose communications and applications, it adds overhead in latency and blocking issues that are undesirable for responsive robot control in low-latency systems. UDP (User Datagram Protocol) is a connectionless protocol that transmits packets without confirmation of reception or transmit order. The system lacks reliability, but significantly reduces transmission latency, thus making it ideal for real-time systems where speed is critical, rather than data integrity [63]. One of the biggest concerns in robotics is safety and this can only be achieved using a UDP system. For instance, if the operator stops issuing commands and a TCP connection is still resolving previous unacknowledged packets, the robot arm may continue executing outdated commands, potentially resulting in collisions or unsafe trajectories. With UDP, outdated packets are dropped, which ensure only the latest commands are executed, yielding a more safe mode for teleoperation.

Control Data Encoding and Transmission

The hardware integrated teleoperation system consists of three types of control modes: First-Person MR Hand Tracking Control, Third-Person Digital Twin MR Hand Tracking Control, and Conventional RGB Video with Keyboard Control. All control inputs are captured and parsed within the Unity environment. Depending on the selected control mode, different data encoding schemes were devised to transmit control commands via the UDP client to a ROS server running on the Raspberry Pi 4B SBC. The communication encryption structure for the transmitted data begins with a three-character control code that identifies the control type:

1. **HTC (Hand Tracking Control):** Includes six floating point values representing left and right hand target positions, a boolean for the gripper state, and two floats for yaw and pitch of the user’s head orientation.
2. **GRC (Global Robot Control):** Used in third-person MR mode, includes

three floats for target end-effector position and four for orientation (quaternion).

3. **Keyboard Control:** No prefix code; only a single string denoting the pressed key.

All these values are formatted using comma-separated values (CSV) and encoded in UTF-8 (Unicode Transformation-8-bit) prior to transmission. The encoded message is sent over UDP to the Raspberry Pi, which hosts the receiving end of the system. The custom designed data encryption process is depicted in Figure 3.20.

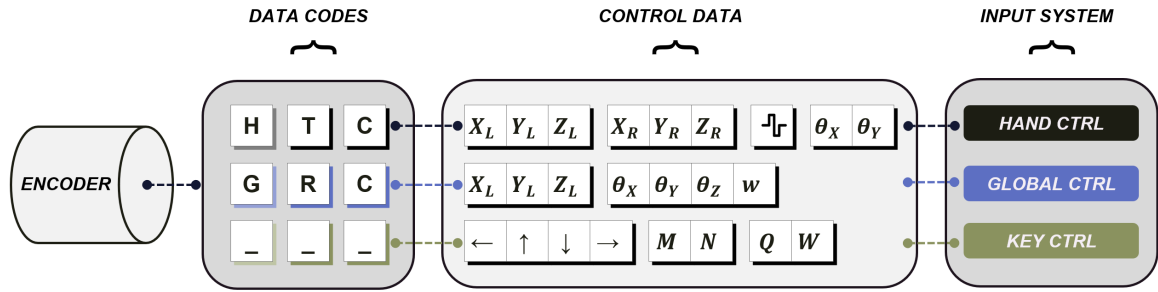


Figure 3.20: Block diagram of the three different data encryption processes designed for MR interface.

ROS-Based Decoding and Command Execution

The SBC hosting the UDP server node is responsible for receiving control data. Once a UDP data packet is received on the server node, it uses UTF-8 to convert the message to a string object. The intermediate string message is then published to a ROS topic named "Raw Data". ROS uses a publisher-subscriber architecture which enables asynchronous data processing. From there, custom ROS nodes subscribe to this "Raw Data" topic and process the message only if the appropriate control code has been detected:

1. The **hand_tracking_ee_ctrl** ROS node handles FPV control data. It parses the position and orientation data, structures the string in a multi-float array object, computes inverse kinematics solutions, and transmits commands over UART to the robot arm's M5Stack microcontroller.

2. The **global_robot_ctrl** ROS node performs similar processing for the digital twin control mode.
3. The **keyboard_ee_ctrl** ROS node maps string keys directly to manipulate an initialized and continuous pose array.

Each ROS control node functions using event-driven callbacks which trigger when new data appear on the "Raw Data" topic. Internal to the nodes, logic is written that converts structured string data into floating-point arrays, applies filtering and transformations, and generates executable joint or cartesian-space commands. All ROS nodes are initialized at runtime on the SBC, but only one control node executes commands based on user selection in the Unity interface.

3.5 Experiment Setup

Following the development and validation of the MR teleoperation interface in the simulation environment, the objective was to evaluate its performance in a real-world setting. Initial runs in the simulation demonstrated that the MR interface is more immersive and intuitive compared to traditional computer video keyboard-based teleoperation methods. With the successful integration of the hardware segment, experiments were designed to quantitatively measure the effectiveness of this immersive interface. The main idea behind employing stereo vision, coupled with VR HMDs, was to provide operators with true depth perception of the robot's workspace, which is absent in conventional 2D video feedback systems. Additionally, integrating real-time hand tracking in the MR subspace provided a new layer of spatial awareness, allowing for 3D control which is not offered in conventional interfaces. To evaluate the contributions of these features, the experiment was designed to isolate two main factors: (1) the improvement in depth-perception offered through stereo vision relative to conventional monocular video feedback, and (2) the benefits of intuitive 3D control enabled by real-time hand tracking. Comparison of these features aims to

understand the degree to which MR technology improves teleportation performance in terms of efficiency, accuracy, and control fluidity in robotics.

3.5.1 Quantitative Experiment Design

The main objective of the experiment was to evaluate and compare the effectiveness of the proposed MR teleoperation interfaces with the conventional teleoperation method. The design of the experiment was dictated by the robot's physical workspace, which is 350 mm. A task was set in which an operator was required to drive the end effector of the robot arm toward three fixed target positions, using each of the teleoperation modes. The target positions were defined by placing 3D printed cubes (dimensions: $5 \times 5 \times 5$ cm), each with a mass of 26 g at three fixed locations within the workspace, where each position represented unique spatial orientations from the robot's neutral pose. To ensure consistency across all tests, the robot arm was always initialized to a fixed starting pose, which was shared across all ROS control nodes through a defined pose array. Target 1 was placed in a far-right position from the robot's starting pose ($x=29, y=-324, z=102$). The second was positioned in front of the end effector ($x=76, y=-317, z=102$), and the third was located to the far-left ($x=144, y=-310, z=102$), all relative to the robot's base coordinate frame in units of millimetres. These positions ensure that each introduced unique challenges in depth perception and trajectory planning. The altitude remained the same for all cubes as it maximized the range required to reach the target, allowing for longer experiment run-times. For each cube position, the experiment was repeated using the following three teleportation modes:

1. Mixed Reality Hand Control: The operator wears the HMD and uses the hand control system to move the end effector to the target in 3D space.
2. Mixed Reality Keyboard Control: The operator wears the HMD and uses the conventional keyboard inputs to move the end effector, isolating the effects of depth perception.

3. Computer-Video Keyboard Control: The operator uses real-time feedback from the monocular video streamed to the laptop and provides control using the keyboard input system.

Three trials were conducted, for each control mode and for each target, which resulted in 9 trails and datasets per target position and 27 overall. To track the trajectory of the robot arm during run-time, a Unity C# script was written that recorded the x , y , and z positions of the end effector, along with the elapsed time, in the form of a CSV file. The devised experiment setup is shown in Figure 3.21 for a single target position.

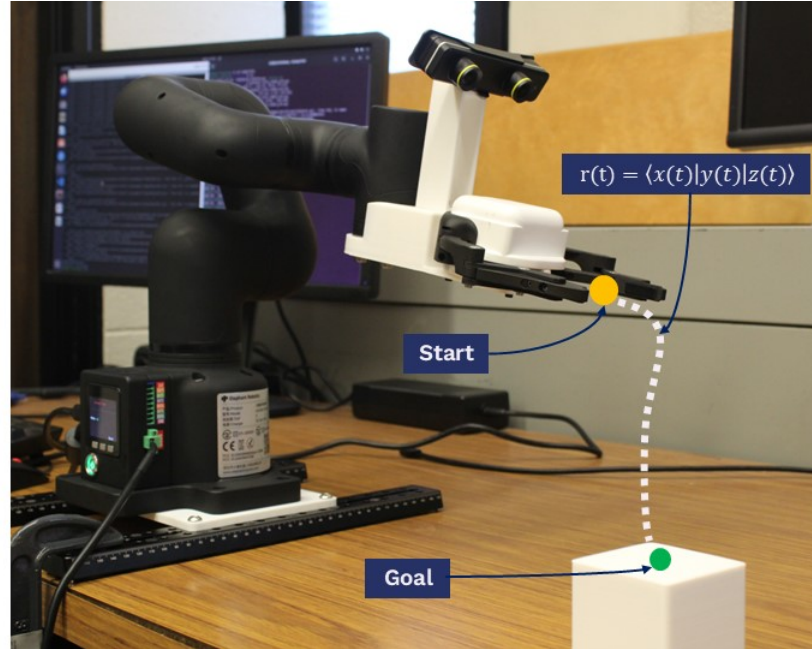


Figure 3.21: Experiment setup – the end effector is moved in free space toward a cube target. The position of the end effector is recorded continuously for analysis.

The experiment was conducted by 10 students with little or no prior experience in robotics or remote operations. Each participant received approximately 10 minutes of training per control mode before conducting their runs. The training time was determined by allowing a few students to use the teleoperation interfaces where it was determined only 2-3 minutes was required for them to learn how to use each

system. The 10 minute limit for this experiment ensured that in the worst-case scenario, each participant had enough time to learn the system. From the collected data, several forms of analysis were conducted. 3D trajectory plots were created to visualize the spatial motion profile and intuitiveness of the control system. Distance minimization curves were constructed to plot the end effector distance to the target over time to assess the efficiency and rate of convergence, as defined in Equation 3.17. Here, (x_t, y_t, z_t) represents the target location and $(x(t), y(t), z(t))$ represents the current end effector position at time t .

$$D(t) = \sqrt{(x(t) - x_t)^2 + (y(t) - y_t)^2 + (z(t) - z_t)^2} \quad (3.17)$$

Lastly, a jerk analysis was performed to gather numerical data on the control stability of each control mode. Jerk is defined as the third derivative of position with respect to time, or the time derivative of acceleration, as defined below:

$$\mathbf{p}(t) = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix}, \quad \mathbf{j}(t) = \frac{d^3 \mathbf{p}(t)}{dt^3} \quad (3.18)$$

$$j(t) = \|\mathbf{j}(t)\| = \sqrt{\left(\frac{d^3 x}{dt^3}\right)^2 + \left(\frac{d^3 y}{dt^3}\right)^2 + \left(\frac{d^3 z}{dt^3}\right)^2} \quad (3.19)$$

Equation 3.18 provides the analytical definition of jerk which can be converted into a numerical solution using the following formula, for discrete data sampled at variable intervals:

$$\mathbf{j}[n] \approx \frac{\mathbf{a}[n] - \mathbf{a}[n - \Delta n]}{\Delta t}, \quad j[n] = \|\mathbf{j}[n]\| \quad (3.20)$$

However, due to a high sampling frequency, of about 10 ms, and the millimetre-scale motion of the arm, the jerk values are highly sensitive to noise. To mitigate this, the Savitzky-Golay filter was applied to the jerk dataset. The filter preserves local features

and trends in the data while removing high-frequency noise by means of fitting a low-degree polynomial over a sliding window using least squares regression [64, 65]. The filter functions by representing the original jerk time series as $\{j_1, j_2, \dots, j_n\}$, then the smoothed jerk $\tilde{j}_k$ at time k is computed as:

$$\tilde{j}_k = \sum_{i=-m}^m c_i \cdot j_{k+i} \quad (3.21)$$

where c_i define the filter coefficients for a window length of $2m + 1$ and polynomial order d . The method is commonly used in signal processing to extract patterns in noisy data [65]. The smoothened jerk magnitudes were averaged for each trial and grouped by control mode and target position. The resulting values, alongside the average task completion times and total distances travelled, were summarized in bar plots for quantitative comparison. The jerk metrics provides a layer of insight regarding the fluidity of control which is a key factor for teleoperation systems where precision and user intent mapping are vital.

Chapter 4

Results and Discussion

This chapter presents the results of the experiment that evaluates the performance of the developed MR teleoperation system compared to the conventional control modes. The results are structured by analysis type, encompassing trajectory analysis, distance minimization, and motion dynamics. Given the repeated experimental procedure across multiple participants, each analysis includes a representative set of plots from a single participant. The graphs were selected for clarity and legibility, as the remaining plots exhibit similar trends and would otherwise result in redundant visual data. The complete set of graphs for all participants are included in Appendix A for completeness and reproducibility. In addition, generalized metrics such as average task completion times and average jerk magnitude are extracted across all participants and presented as summary tables and comparative bar charts. These results are then discussed to identify performance trends and gather insights regarding the control modes.

4.1 Trajectory Analysis

Figure 4.1 presents the end effector trajectories executed by a single participant while attempting to reach each of the three target cube positions. The three subplots correspond to the three control modes: (a) MR Hand Control, (b) MR Keyboard Control, and (c) Computer Video Keyboard Control. Each trajectory represents one trial,

with three trials being performed per target. The figure was selected as a representative sample due to the visual clarity and consistency of the behavioural trends. Although the participants yielded unique trajectories, the overall characteristics across control modes were consistent. Therefore, to improve readability, only one participant's plots are shown in the main body of this chapter, while full results for all participants can be found in Appendix A.1-A.3.

In Figure 4.1a), the trajectories resulting from the MR Hand Control method show smooth, continuous motions toward the target cubes. Across trials, the paths are consistently directed from the initial pose toward the goal with minimal oscillation, correction, or hesitation. The observation suggests that the use of stereo vision and FPV MR immersion provides strong depth cues and intuitive motion mapping, allowing users to command the robot arm in a natural and fluid manner.

Figure 4.1b) displays the trajectories using the MR Keyboard Control interface. These flight paths, while still moderately regulated, show more correctional movements and angular transitions relative to the hand control mode. The non-linear behaviour is to be expected as the keyboard input system introduces discrete control steps rather than continuous 3D motion. However, the use of the MR stereo vision interface still provides an additional layer of spatial awareness, which mitigates harsh corrective behaviour.

Figure 4.1c) shows the trajectories using the conventional Computer video Keyboard Control interface. These flight paths are significantly more jagged and discontinuous, with frequent changes in direction and increased deviation from direct paths to the targets. These trends indicate that the operator has a difficult time estimating relative spatial positions using only monocular camera feedback projected on a 2D screen. The absence of true depth likely forces operators to make repeated adjustments and visual re-evaluations, thus resulting in the erratic trajectories.

Overall, the trajectory analysis highlighted a clear trend in that the incorporation of stereo vision and MR depth cues enables smoother, more direct, and consistent

control of the robot arm. The contrast between these control modes showcases the cognitive and perceptual benefits of MR based teleoperation for robotics.

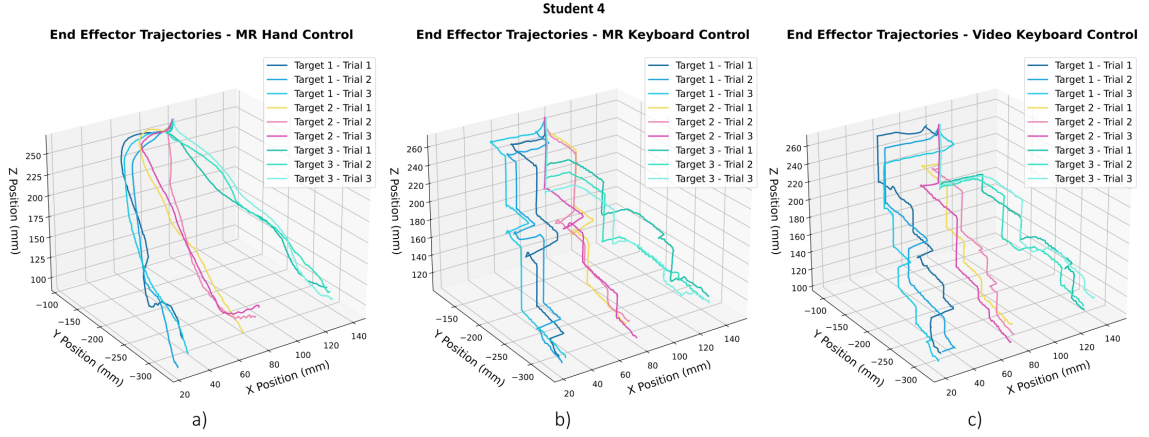


Figure 4.1: Trajectory of the robot arm's end effector, for multiple runs, using the (a) Mixed Reality Hand Control method, (b) Mixed Reality Keyboard Input method, and (c) Computer Video Keyboard Input method

4.2 Distance Minimization Analysis

As per the trajectory analysis, Figure 4.2 shows the corresponding participant's distance minimization plots, using the same dataset. These plots present how the distance between the end effector and the target is minimized with respect to time, for all three targets. These graphs provide insights into task completion times as well as show control behaviour such as corrections and overshooting, which can be extracted from the rate of change of the lines. Each subplot in the figure corresponds to a different target cube position, specifically: (a) the far-right target, (b) the centre-front target, and (c) the far-left target. For each control type, three lines represent the three independent trials conducted per target per target.

In subplot (A) of Figure 4.2, which corresponds to Target 1, the MR Hand Control interface shows a smooth and consistent reduction in distance for all three runs. The slopes of these lines are uniform and monotonic, showing continuous and decisive movement without the need for significant corrections or stalls. The MR Hand control mode achieved an average task completion time of approximately 17.3 seconds.

In contrast, the MR Keyboard Control plots show less consistent slopes. Some portions of the graph demonstrate flatter slopes or brief plateaus, suggesting moments when the operator paused or made correctional inputs. The average completion time using this control mode was 23.8 seconds for Target 1, which shows a decrease in operational efficiency compared to the hand tracking control.

For the Computer Video Keyboard Control mode, the curves show even more erratic trends. Frequent stalls, reversals (temporary increases in distance), and a general slower convergence toward the target can be observed. These patterns highlight the lack of intuitive depth perception and limited spatial feedback inherent in conventional monocular teleoperation systems. The average time to reach Target 1 using this method was 25.5 seconds, which is the slowest among the three control modes. The observed trend remains consistent across the other two target positions, as seen in subplots (b) and (c) of Figure 4.2. Table 4.1 summarizes the average completion times for this participant.

Control Mode	Target 1 (Right)	Target 2 (Front)	Target 3 (Left)
MR Hand Control	17.3	16.0	14.5
MR Keyboard Control	23.8	20.8	21.4
Computer Video + Keyboard	25.5	21.0	21.9

Table 4.1: Average task completion times (in seconds) for each control mode and target

The results of these distance minimization plots highlight that the MR Hand Control interface consistently enabled faster task execution across all target positions. The MR Keyboard Control interface, although benefiting from stereo vision, suffered from the limited degrees of freedom that is imposed through keyboard-based inputs. Lastly, the Computer Video Keyboard Control system showed the least efficiency, due to both limited input control and the absence of immersive depth feedback.

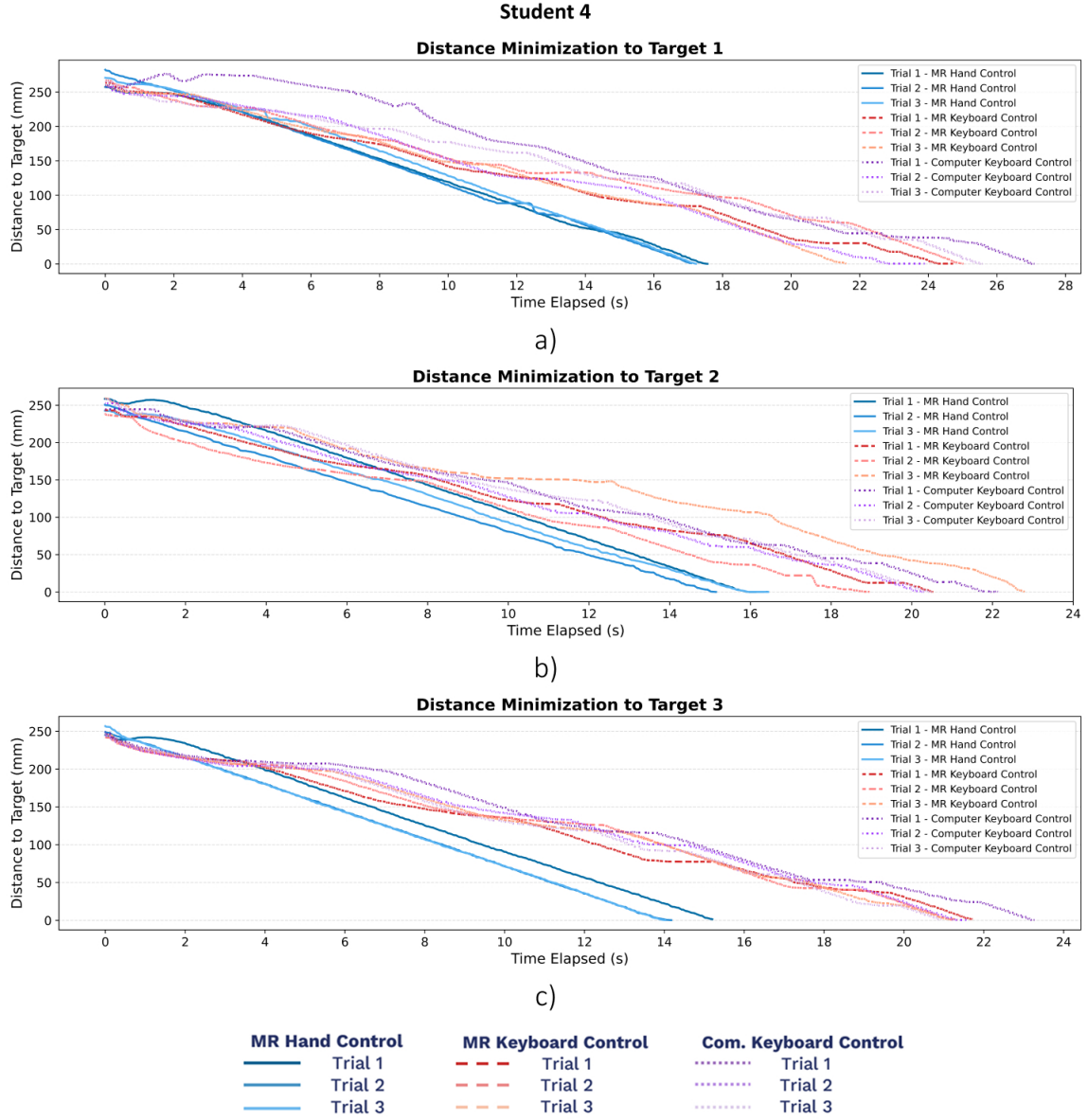


Figure 4.2: Distance minimization plots for participant 4. Each subplot corresponds to a target cube: (a) Target 1, (b) Target 2, and (c) Target 3. Each line represents a trial using one of the three control methods.

4.3 Jerk Profile Analysis

Figure 4.3 presents the jerk profiles corresponding to the same student and set of experiments used in the previous analysis. These plots visualize the time derivatives of acceleration, which are referred to as jerk, for each trial across the three different control modes and target positions. Jerk is an important metric for assessing motion

smoothness and trajectory continuity, as it provides insight into abruptness or hesitations in motion that may not be completely captured by the distance-based analyses. Each subplot in Figure 4.3 represents one control mode for a specific target. The columns represent control modes: (i) MR Hand Control, (ii) MR Keyboard Control, and (iii) Computer Video Keyboard Control. The rows represent the three target cube positions: (a) Target 1, (b) Target 2, and (c) Target 3. Each plot contains three coloured lines representing the filtered jerk magnitude for each trial, and a horizontal dark dashed line showing the average jerk magnitude for that control type and target.

For Target 1, shown in subplots (ai) through (aiii), the MR Hand Control interface, (ai), shows low and consistent jerk magnitudes across all three runs. The graph suggests that the acceleration is smooth and is a continuous motion requiring little abrupt changes. The average jerk for this mode was 3518 mm/s^3 .

Interestingly, the MR Keyboard Control mode for Target 1 (aii) displays comparable jerk behaviour, with a mean value of 3656 mm/s^3 . Although the distance minimization profiles in the previous section showed more erratic trends for the keyboard interface, the jerk profiles show that when combined with an immersive MR subspace, the operator is still able to execute smooth trajectories despite the binary nature of keyboard input.

However, the Computer Video Keyboard Control mode (aiii) results in much noisier jerk profiles, with sharp spikes and abrupt changes in magnitude across trials. The average jerk for this condition was $29,883 \text{ mm/s}^3$, which is nearly an order of magnitude greater than the MR-based control methods. The finding underscores the impact of limited depth perception and non-intuitive feedback systems when teleoperating through a conventional 2D interface. These trends remain consistent across the other targets, as well as participants. For Target 2, the average jerk values were:

- **MR Hand Control:** 3379 mm/s^3
- **MR Keyboard Control:** 3680 mm/s^3

- **Computer Keyboard Control:** 20,293 mm/s³

And for Target 3, the average values were:

- **MR Hand Control:** 3499 mm/s³
- **MR Keyboard Control:** 3572 mm/s³
- **Computer Keyboard Control:** 20,333 mm/s³

These results highlight two key observations. First, the MR Hand Control mode consistently facilitates smooth motion across all tasks. Second, even through the MR Keyboard interface introduces discrete directional inputs, its immersion and depth awareness greatly reduces abrupt movement patterns, especially when compared to the conventional computer-video based system. The results show the strength of the jerk as a quantitative indicator of motion quality and teleoperation control.

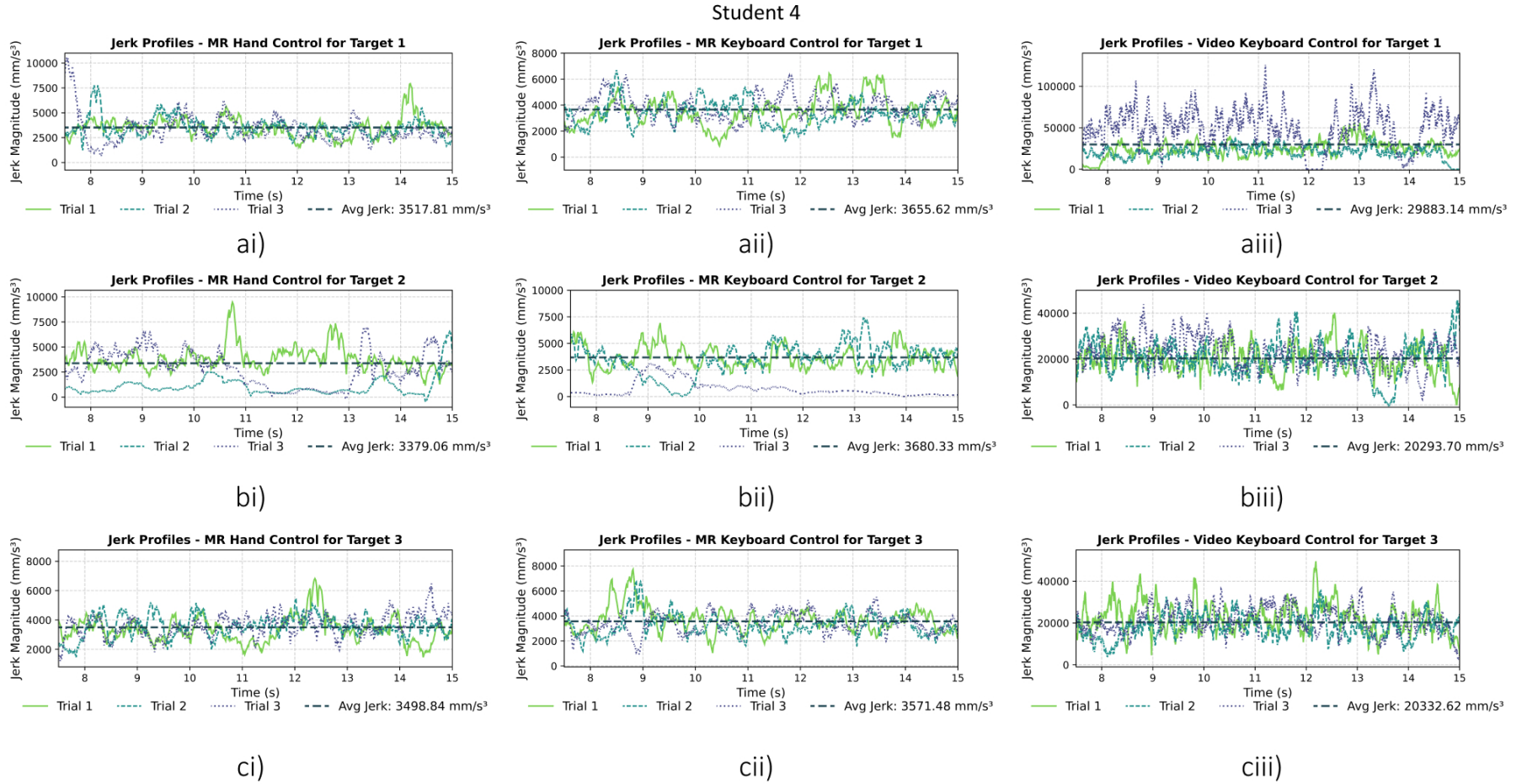


Figure 4.3: Jerk profiles for Student 4 across all control types and target cube positions. Each row corresponds to a different target, and each column corresponds to a different control mode. Coloured lines represent filtered jerk magnitude across trials; dashed horizontal lines indicate average jerk.

4.4 General Trends and Comparative Analysis

4.4.1 Task Completion Time Comparison

To gain a full understanding of the efficiency of the proposed mixed reality teleoperation interface, the average task completion times were computed for all students across the three target positions. The collected data is summarized over each participant, in Tables 4.2 through 4.4. The mean values and standard deviations for each control method is shown in the last row of each table. In this data, lower completion times represent more efficient and intuitive control.

Student	MR Hand	MR + Keyboard	Video + Keyboard
1	15.9	32.7	33.7
2	16.7	22.8	22.7
3	16.3	24.5	24.6
4	17.3	23.8	25.5
5	17.2	22.6	23.5
6	15.9	32.7	33.7
7	16.4	24.1	27.9
8	15.6	28.8	26.4
9	15.0	22.8	22.3
10	15.4	34.1	31.3
Average $\pm$ Std. Dev.	16.0 $\pm$ 0.9	26.5 $\pm$ 4.3	26.8 $\pm$ 3.9

Table 4.2: Average task completion times and standard deviations for Target 1 in seconds.

Student	MR Hand	MR + Keyboard	Video + Keyboard
1	15.2	23.2	23.6
2	15.4	20.6	20.0
3	16.0	23.0	20.9
4	16.0	20.8	21.0
5	14.6	21.0	19.7
6	14.1	24.2	25.4
7	13.4	22.4	24.8
8	14.4	22.9	26.7
9	14.0	18.4	20.2
10	14.6	27.7	28.5
Average $\pm$ Std. Dev.	14.8 $\pm$ 0.9	22.4 $\pm$ 2.5	23.1 $\pm$ 3.2

Table 4.3: Average task completion times and standard deviations for Target 2 in seconds.

Student	MR Hand	MR + Keyboard	Video + Keyboard
1	14.5	24.1	30.6
2	15.1	20.1	20.5
3	14.6	21.2	20.0
4	14.5	21.4	21.9
5	13.9	19.8	22.9
6	14.6	27.1	25.8
7	13.5	26.4	24.5
8	13.6	23.2	24.6
9	13.6	20.4	18.8
10	13.6	26.1	28.9
Average $\pm$ Std. Dev.	14.2 $\pm$ 0.6	23.0 $\pm$ 2.8	23.9 $\pm$ 3.8

Table 4.4: Average task completion times and standard deviations for Target 3 in seconds.

The task completion data tables show that the mean task completion times for the Mixed Reality Hand Control method was not only significantly smaller than the other modes, but also has a standard deviation less than 1 second for each target. The analysis show that the control method is consistently more efficient when compared with the Mixed Reality Keyboard and Computer Video Keyboard control methods. To validate whether task completion times differed significantly across control modes, a one-way analysis of variance (ANOVA) was performed separately for each target. The results, summarized in Table 4.5, provide statistical evidence of differences in performance among control methods.

The ANOVA results indicated statically significant differences in task completion times across control modes for all targets ($p < 0.001$ for each case). To identify which specific control method differed from another, a post-hoc least significant difference (LSD) analysis was performed. The computed LSD values were 3.1 s, 2.2 s, and 2.5

Target	df (Between, Within)	MS (Within)	F	p-value	Significant?
1	2, 27	11.33	33.07	< 0.001	Yes
2	2, 27	5.58	38.27	< 0.001	Yes
3	2, 27	7.58	38.07	< 0.001	Yes

Table 4.5: Table summarizes the main results of the ANOVA analysis performed for task completion times across the three control modes for each target

s for Targets 1, 2, and 3, respectively. Subsequently, the pairwise mean differences between the control modes were compared using these LSD thresholds to determine significance, as shown in Tables 4.6-4.8. A pairwise mean difference value is defined to be significant if it is greater than its respective LSD value.

Comparison	Mean Difference (s)	Significant (Yes/No)
MR Hand – MR Keyboard	10.4	Yes
MR Hand – Video Keyboard	10.8	Yes
MR Keyboard – Video Keyboard	0.3	No

Table 4.6: Pairwise mean differences in task completion time for Target 1. Significance determined using the computed LSD value **3.1 s**

Comparison	Mean Difference (s)	Significant (Yes/No)
MR Hand – MR Keyboard	7.6	Yes
MR Hand – Video Keyboard	8.3	Yes
MR Keyboard – Video Keyboard	0.7	No

Table 4.7: Pairwise mean differences in task completion time for Target 2. Significance determined using the computed LSD value **2.2 s**

The results of the post-hoc LSD comparisons provide consistent trends across all three targets. For each target, the Mixed Reality Hand Control method shows a statistically significant improvement in task completion time relative to both the Mixed Reality Keyboard and the standard Computer Video Keyboard control methods. The results of the analysis also show that there is no significant difference between the

Comparison	Mean Difference (s)	Significant (Yes/No)
MR Hand – MR Keyboard	8.8	Yes
MR Hand – Video Keyboard	9.7	Yes
MR Keyboard – Video Keyboard	0.9	No

Table 4.8: Pairwise mean differences in task completion time for Target 3. Significance determined using the computed LSD value **2.5 s**

Mixed Reality Keyboard and Computer Video Keyboard control methods, in the context of task completion time. These results show that hand tracking significantly improves task completion efficiency while using mixed reality over standard monocular visual feedback has no impact in this category. A bar chart, as shown in Figure 4.4, was created showing these overall trends in task efficiency.

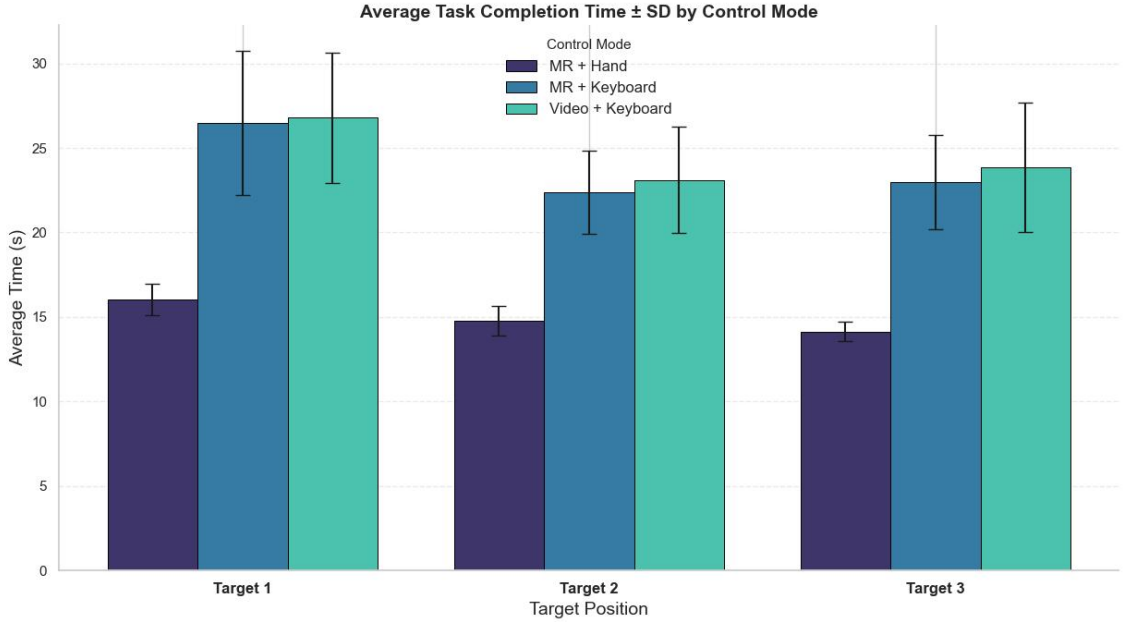


Figure 4.4: Grouped bar chart comparing average completion times across all targets and control modes.

To gain deeper understanding of each control mode, the average task completions times, for each of the three targets, were further analyzed to compute the overall efficiency gains. Table 4.9 summarizes these computed efficiency gains across the various

control modes. To quantify the efficiency gains of the mixed reality interface, the percentage reduction in task time was computed for two key comparisons: (1) between the MR Hand Control and MR Keyboard Control modes, and (2) between the MR Hand Control and the conventional Computer Video Keyboard Control mode. The comparison isolates the benefits of combining 3D gesture-based input with immersive depth perception. The efficiency gains were computed using Equation 4.1. In this formula, T_{base} is the average task completion time for the baseline method (either MR Keyboard or Computer Video) and $T_{\text{MR Hand}}$ represents the task completion time using the MR Hand Control method.

$$\text{Percentage Reduction} = \frac{T_{\text{base}} - T_{\text{MR Hand}}}{T_{\text{base}}} \times 100 \quad (4.1)$$

Analyzing the average efficiency gains of the MR Hand Control system shows that it outperforms the MR Keyboard control mode by **37%** and the Computer Video Keyboard by **39%**. These reductions were not only large in magnitude but also statistically significant according to the LSD tests. The analysis clearly shows the benefits of combining stereo cameras with gesture-based control input systems, as it significantly improves task completion times. In contrast, the MR Keyboard control mode showed only a **3%** improvement over the Computer Video Keyboard control method, and the LSD tests confirmed that this reduction was not statistically significant. The findings indicate that the use of stereo vision alone does not yield a meaningful efficiency gain in terms of task completion time. However, as will be discussed in the following section, there is still quite a significant difference in control stability when using these two visual feedback systems. To summarize this analysis, stereo vision and HMD fusion technology, in the context of robotic task completion times, yields a significant improvement only when combined with intuitive hand gesture-based control input systems.

Comparison	Target 1	Target 2	Target 3
MR Hand vs. MR Keyboard	39%	34%	38%
MR Hand vs. Video Keyboard	40%	36%	41%
MR Keyboard vs. Video Keyboard	1%	3%	4%
Average Reduction	—	—	—
MR Hand vs. MR Keyboard		37%	
MR Hand vs. Video Keyboard		39%	
MR Keyboard vs. Video Keyboard		3%	

Table 4.9: Efficiency gains in task completion times against the various control modes. Specifically compares the effectiveness when combining stereo vision and gesture-based control against conventional control/vision systems. Also compares the effect of only using stereo vision against conventional vision systems.

4.4.2 Jerk Magnitude Comparison

A similar analysis was performed to quantify the general trends in control stability, by averaging the jerk profiles across the different teleoperation interfaces, as well as computing the standard deviation values. Tables 4.10 through 4.12 show the average jerk magnitudes over each sample and control mode, where a single table represents a target position. In this data, lower jerk magnitudes present greater control stability and smoothness in teleoperation. Additionally, the term "Video Keyboard" in these tables is an abbreviation for the Computer Video Keyboard Control mode.

Student	MR Hand	MR Keyboard	Video Keyboard
1	3699	2982	15687
2	3326	3569	24815
3	3473	3487	20961
4	3518	3656	29883
5	3375	4856	18057
6	3869	3099	35441
7	3797	3496	17412
8	3780	3295	16516
9	3044	4434	21177
10	3839	3084	14064
Average $\pm$ Std. Dev.	3572 $\pm$ 271	3596 $\pm$ 605	21401 $\pm$ 6818

Table 4.10: Average jerk magnitudes (mm/s³) and standard deviations for Target 1

Student	MR Hand	MR Keyboard	Video Keyboard
1	3640	3249	19487
2	2816	3637	22749
3	3646	3668	21722
4	3379	3680	20294
5	4207	3625	20530
6	3788	3275	43798
7	4109	3915	17545
8	4120	3634	13033
9	3698	4474	20277
10	4158	3396	12572
Average $\pm$ Std. Dev.	3756 $\pm$ 432	3656 $\pm$ 352	21201 $\pm$ 8647

Table 4.11: Average jerk magnitudes (mm/s³) and standard deviations for Target 2

Student	MR Hand	MR Keyboard	Video Keyboard
1	3510	3395	15745
2	3430	3588	21577
3	3613	3713	23395
4	3499	3572	20333
5	4052	4414	16559
6	3600	3205	43285
7	4174	3470	17661
8	3427	3897	14797
9	3754	4176	20812
10	4035	3524	12001
Average $\pm$ Std. Dev.	3709 $\pm$ 280	3695 $\pm$ 369	20616 $\pm$ 8690

Table 4.12: Average jerk magnitudes (mm/s³) and standard deviations for Target 3

The jerk profile tables show that the mean jerk magnitudes for both the Mixed Reality Hand and Mixed Reality Keyboard control methods have much lower magnitudes than the Computer Video Keyboard control method. The data also show a common trend of the standard deviations being lower for the mixed reality systems when compared with the conventional visual feedback system. To statistically validate whether the jerk magnitudes significantly varied across the control modes, ANOVA analysis was performed separately for each target. The results, summarized in Table 4.13, provide statistical evidence of differences in performance among the control modes.

The ANOVA results indicated statically significant differences in jerk magnitudes across control modes for all targets ($p < 0.001$ for each case). To identify which specific control method differed from another, a post-hoc LSD analysis was performed. The computed LSD values were 3629 mm/s³, 4590 mm/s³, and 4610 mm/s³ for Targets 1, 2, and 3, respectively. Subsequently, the pairwise mean differences between the

Target	df (Between, Within)	MS (Within)	F	p-value	Significant?
1	2, 27	1.56×10^7	67.65	< 0.001	Yes
2	2, 27	2.50×10^7	40.77	< 0.001	Yes
3	2, 27	2.52×10^7	37.78	< 0.001	Yes

Table 4.13: Summary of one-way ANOVA results for jerk magnitudes across the three control modes for each target.

control modes were compared using these LSD thresholds to determine significance, as shown in Tables 4.14-4.16.

Comparison	Mean Difference (mm/s ³)	Significant (Yes/No)
Video Keyboard – MR Hand	17829.24	Yes
Video Keyboard – MR Keyboard	17805.44	Yes
MR Keyboard – MR Hand	23.80	No

Table 4.14: Pairwise mean differences in jerk magnitudes for Target 1. Significance determined using the computed LSD value **3629 mm/s³**

Comparison	Mean Difference (mm/s ³)	Significant (Yes/No)
Video Keyboard – MR Hand	17444.38	Yes
Video Keyboard – MR Keyboard	17545.06	Yes
MR Keyboard – 100.69	23.80	No

Table 4.15: Pairwise mean differences in jerk magnitudes for Target 2. Significance determined using the computed LSD value **4590 mm/s³**

Comparison	Mean Difference (mm/s ³)	Significant (Yes/No)
Video Keyboard – MR Hand	16907.15	Yes
Video Keyboard – MR Keyboard	16921.05	Yes
MR Keyboard – 13.89	23.80	No

Table 4.16: Pairwise mean differences in jerk magnitudes for Target 3. Significance determined using the computed LSD value **4610 mm/s³**

The results of the post-hoc LSD comparisons provided consistent trends across all three targets. For each target, both the Mixed Reality Hand and keyboard control modes show a significant improvement in jerk magnitude reduction, compared with the standard Computer Video Keyboard control method. The results of this analysis also show there is no significant difference between the Mixed Reality Hand and Mixed Reality Keyboard control modes. These findings indicate that using mixed reality as a visual feedback system helps to significantly lower jerk magnitudes, thus increasing control stability in robotic teleoperation, compared to the standard monocular visual feedback system. A bar graph was generated, as shown in Figure 4.5, which provides a visual representation of these findings.

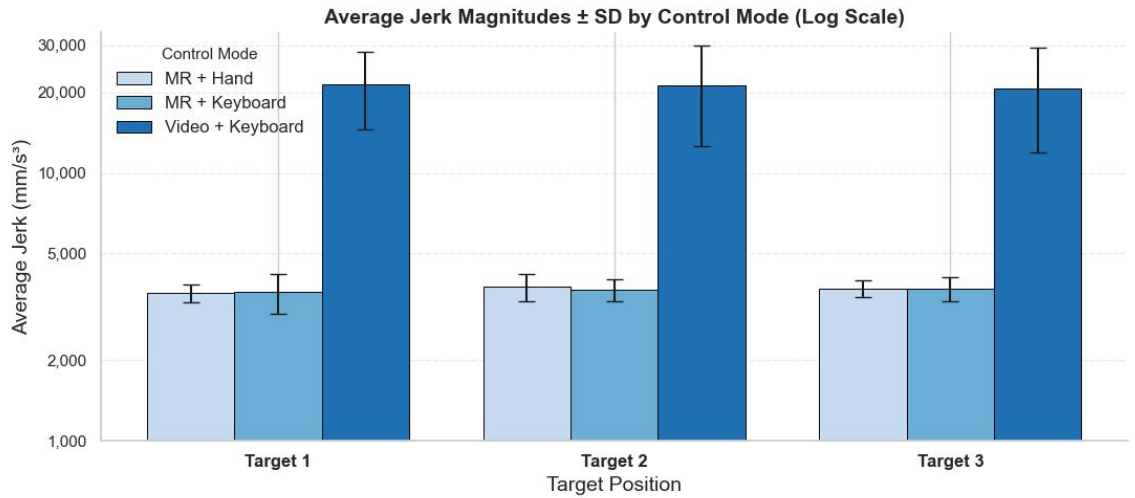


Figure 4.5: Bar chart comparing the average jerk magnitudes across the various targets and control modes.

A deeper analysis was performed on the jerk magnitudes for the various control types, to empirically measure differences in control stability and trajectory smoothness. Table 4.17 highlights these findings as it shows the ratios between the average jerk magnitudes for the conventional Computer Video Keyboard Control mode to both MR Control modes. Additionally, the ratio between the MR Hand and MR Keyboard control modes were computed. The findings show that across all three targets, the average jerk magnitude of the Computer Video Keyboard Control mode was

more than **5 times higher** than for both MR-based systems. To reiterate, a higher jerk magnitude represents less control stability suggesting that the conventional teleoperation system is much less intuitive to teleoperate than the mixed reality systems. The lack of depth perception causes trajectory instability, which forces the operator to make abrupt corrective movements. The distinction becomes clearer when examining the ratio between the conventional control mode and the MR Keyboard control mode, which is **5.8**. The ratios suggest that even when using the keyboard as a control input system, the trajectories are still over 5 times more stable and smooth using stereo vision and mixed reality as a visual feedback system over monocular video images streamed to a computer screen. Both these findings have been proven to be significant from the LSD analysis, which clearly highlights the positive effects of using mixed reality headsets for robotic teleoperation applications. Lastly, the MR Keyboard Control mode is only **1 $\times$** higher in jerk compared with the MR Hand Control mode. The ratio shows that regardless of whether using a rudimentary control input system such as a keyboard or an intuitive one such as hand tracking gesture-based control, the control stability is quite similar due to the advent of mixed reality depth perception.

Target	Video/MR Hand	Video/MR Keyboard	MR Keyboard/MR Hand
Target 1	6.0 $\times$	6.0 $\times$	1.0 $\times$
Target 2	5.6 $\times$	5.8 $\times$	1.0 $\times$
Target 3	5.6 $\times$	5.6 $\times$	1.0 $\times$
Average	5.7$\times$	5.8$\times$	1.0$\times$

Table 4.17: Average jerk magnitude ratios between the various control modes.

4.5 Limitations

Although the devised teleoperation interface showed significant improvements in task completion and control stability experiments, several factors constrain the reliability of the results. The constraints can be divided into two categories. The first category involves the various sources of error, which are derived from hardware and software issues, including sensor noise, actuation inaccuracies, and data sampling limitations. The second category arises from broader research limitations that relate to the scope of the study, experimental design, and applicability of the findings to other hazardous environments. The following subsections will discuss these two aspects in detail to provide context of the boundaries of the current work.

4.5.1 Sources of Error

The experiments conducted yielded very positive results, which indicate a heightened sense of depth perception, due to the use of the stereo camera and HMD, as well as more decisive and fluent control of the end effector through the intuitive hand tracking control system. To familiarize, the experiment consisted of placing three target cubes, at various positions in the robot's workspace, and giving participants the task of driving the robot arm's end effector to the target cube. When the end effector collided with the cube, the experiment was stopped. To collect data on the robot arms trajectory, the robot's internal M5Stack OBC was used to retrieve the position of the end effector, at a high frequency.

One of the sources of error that may limit the accuracy of the observed results is the pose estimations conducted by the robot arm. The robot arm's servo motors require frequent calibration to ensure they can accurately measure the angle of each of the six joints. During long run-times and with a heavy end effector load, this accuracy can diminish. However, this would be a very minuscule error estimation in the order of millimetres. To overcome this, it was ensured that before each participant began

the experiment, the robot arm was correctly calibrated by aligning the servo motors to their neutral position, followed by executing a servo movement test to ensure the servo's return to their base angle. The calibration is also recommended by Elephant Robotics, prior to use of the arm.

Further, during the experiment, as the robot arm's end effector moves to the target, eventually colliding with the target cube, the experiment is stopped. As the target cubes were $4.5 \times 4.5 \times 4.5$ cm, the tip of the end effector could collide at any point of the front of the cube, leaving space for different collision points that terminate the experiment. To minimize this error, participants were directed to aim for the centre of the cube, which reduces the error per trial. Errors of approximately ± 1 second may occur due to the extra distance the end effector travels. However, as this experiment was not an accuracy test, rather more focused on understanding control fluidity and general depth awareness, this source of error is not significant.

4.5.2 Research Limitations

This developed Mixed Reality teleoperation interface consisted of two control modes; (1) a FPV Hand Control mode, and (2) a third-person digital twin hand control mode. One of the setbacks of the third-person control mode was a low resolution point cloud. The stereo camera's per-pixel depth computation system is designed to be highly accurate but at moderate distances. Within the experimental setup conducted, which was limited by the robots 350 mm range, objects near the arm, at about 15-20 cm, were difficult to measure the distance to. Therefore, the point cloud that was generated often times included objects further away, thus making the robot arm's near workspace difficult to visualize. For this reason, the third-person control mode was not used in the experiment conducted. Additionally, this shows the need for sensor fusion, where adding more sensitive depth cameras, such as LiDAR or Infrared (IR) sensors, may increase the density of the point cloud near the robot's workspace for better operator awareness.

As seen from the results, the MR Hand Control interface performs each task much faster and more steadily than all other control interfaces provided. However, the measurement of the true efficiency gain was limited by the physical reach of robot arm. It is hypothesized that the MR Hand Control system should be significantly more efficient than the conventional systems, as was the case in the simulation component. However, as the experimental targets were relatively close together, due to the limited robot workspace, the measurements between the various control modes did not differentiate as much as they could with a larger robot arm reach.

Lastly, in this research, the stereo camera feed was streamed to the HMD over a USB-C connection. However, in real-world teleoperation systems, often the video feed is streamed over an internet connection, if possible. Otherwise, it would be connected via a long USB connection. The ZEDM camera is designed to stream at very low latency, approximately 30 FPS, at HD1080 resolution, natively. However, in order to interface with the ZED SDK, a GPU is required. As the SBC used for this application was a Raspberry Pi 4B, which does not have a dedicated GPU, it cannot install the CUDA drivers necessary to interface with the ZEDM camera to stream the images to the Unity instance wirelessly. Therefore, the feed was streamed directly through a USB-C connection to the main computer.

4.6 Applications of Developed System

The findings of this research demonstrate the potential for MR technology to address some of the underlying issues found in conventional robotic teleoperation systems. As mentioned in the introductory chapter, some of the key challenges in current day robotics is the lack of depth perception and unintuitive control systems that make teleoperation cognitively demanding and inefficient. However, the results demonstrated in this experiment support the idea that combining immersive stereo vision with natural hand tracking significantly improves teleportation. Specifically, the **37%** and **39%** task completion time reductions, relative to MR Keyboard and Computer

Video Keyboard Control respectively, show that there is a significant efficiency gain through intuitive gesture-based interfaces. The increase in efficiency is very applicable in time sensitive operations such as remote maintenance, nuclear waste cleanup, or search and rescue. Additionally, the jerk analysis showed a significant improvement in control stability within MR-based interfaces. Specifically, there was upwards of a $5\times$ increase in trajectory smoothness compared to conventional monocular video-based systems. Such functionality is extremely useful when interacting with delicate objects or confined spaces, including underwater and off-site construction sites, orbital repairs, and surveying applications. One of the main highlights of this analysis was the comparison between the MR Keyboard Control and Computer Video Keyboard Control which demonstrated that even without natural gesture input, the presence of stereo vision alone, using HMDs, reduces erratic control behaviour. The finding indicates that mixed reality systems can slowly be adopted into existing systems to take immediate advantage of the performance benefits, even before integrating immersive control systems. From the perspective of scalability, the developed MR interface has the ability to be adapted into a wide range of existing robotic platforms and use-cases. By taking advantage of the modular software frameworks used and designed in this research (Unity and ROS-based systems), this interface can be reconfigured for different robotic arms and control tasks. Additionally, as the MR interface runs on the OpenXR framework, it can be applied to a wide variety of VR headsets. Further development of this teleoperation system can expand its applications towards bimanual manipulation, multi-robot collaboration (swarm robotics), or semi-autonomous operations. Such upgrades will enable the deployment of this system into sectors such as space exploration, advanced manufacturing, and infrastructure inspection. Specifically, to successfully deploy the developed interface into an area such as construction sites to support precise material handling, added sensors, allowing for denser point cloud generation, and the deployment onto a mobile platform would be required. In the context of nuclear environments, radiation-hardened hardware and robust com-

munication protocols would be necessary. For the application of space environments, latency tolerant teleoperation as well as lightweight hardware, to minimize payload mass, would need to be adopted. These considerations show that although the developed system shows strong experimental validation, deployment into real-world hazardous settings require for it to be outfitted with targeted hardware and software modifications.

Chapter 5

Conclusion and Recommendations

5.1 Summary of Findings

This section summarizes the main findings of this research project. To recall, the main objectives of this research was to develop a novel mixed reality application that provides the operator with a first-person view, with true depth awareness, of the robot's workspace. Among the project goals was the implementation of hand-tracking and gesture-detection algorithms for robot arm control. The last objective was to devise experiments that would measure the efficiency gain of completing target-tracking tasks for the proposed system against conventional interfaces.

The target tracking experimental data was gathered and trajectory plots were generated to visually inspect the paths taken for the various control interfaces. The inspected trajectory graphs showed smooth paths, that were more consistent with a straight line, for the MR Hand Control mode compared with MR Keyboard and Computer Video Keyboard Control modes. It highlighted the first signs of elevated control stability provided by the intuitive hand tracking input system. To further analyze these effects and to compute efficiency, a distance minimization analysis was performed, plotting the Euclidean distance to the target over time. These graphs show the overall time to complete each task, for each of the control modes. The results of this analysis show that the MR Hand Control system achieves a time reduction of **37%** compared to the MR Keyboard Control mode and **39%** compared to the

Computer Video Keyboard Control mode, which highlights the added benefits of combining immersive stereo depth perception with intuitive gesture-based control. The metric achieves the desired objective of the research of improving teleoperation efficiencies through the design of a novel mixed reality application. Specifically, this data shows a direct correlation between stereo vision systems to the depth awareness provided to the operator. Further, a comparison was performed between the MR Keyboard Control mode and the Computer Video Keyboard Control mode which yielded a **3%** improvement. The finding suggested that although stereo video provides perceptual benefits, its impact is less effective with rudimentary input systems, from a task completion time standpoint. It should be noted that these experiments were conducted for operators that have little to no experience with robotic teleoperation systems. The participants were also only subject to 10 minutes of training prior to task execution, which shows that in the context of untrained operators, the MR Hand control method significantly outperforms conventional keyboard and monocular visual feedback teleoperation methods.

To gain deeper insights on the effects of control input systems to trajectory stability, an analysis of the jerk profiles was computed for each control mode. Jerk is the time-derivative of the robot arm’s acceleration, and it provides insights into motion smoothness and trajectory continuity. The results of this analysis showed that for each of the three targets, the average jerk values for the Computer Video Keyboard Control mode were consistently more than **5 times higher** than for both MR-based systems, which shows the negative affects caused by the absence of stereo vision, forcing the operator to make abrupt corrective movements. In contrast, the MR Keyboard Control mode was only **1×** higher in jerk compared to the MR Hand Control mode, which showed that although keyboard input introduces a low degree of freedom in movement, the depth awareness provided by the MR environment has a significant impact on dampening erratic control behaviour. Combing this finding with the low percentage change in task completion times between the MR Keyboard

Control and Computer Video Keyboard Control, it shows that although keyboard control results in less direct paths to the targets, the trajectory is much more stable and decisive than when stereo camera systems are not used for depth perception. The finding achieves the secondary objective of this research which was to successfully implement hand tracking and gesture detection algorithms to improve robot arm control.

5.2 Impact of the Research

The research demonstrated that utilizing stereo cameras, with its integration of mixed reality headsets, and intuitive hand control input system significantly increase task completion efficiency, control stability, depth awareness, and situational awareness in remote robotic teleoperation. It also enables complex robotic systems to be deployed in real-world scenarios much faster, as the training complexity for operators is significantly reduced. The main reason for this is due to the simplistic and natural mode of providing control inputs via hand tracking as well as the ability to instantly perceive depth with stereo images through a head-mounted display.

These findings are specifically impactful for industries that heavily rely on remote robotic operations. Areas such as construction, space robotics, underwater construction, and nuclear waste cleanup / monitoring can use this technology to reduce the hazards imposed on human workers, utilizing their experience and knowledge to complete tasks more efficiently, through a remote environment. The simplistic control nature of this system enables robotics to enter into new applications without prior experience in robotic operations.

The devised teleoperation system is designed to be extremely modular, requiring three main components: (1) Standalone Single Board Computer, (2) Laptop with a dedicated GPU, (3) a Meta Quest 2 VR HMD, and (4) a ZED Mini stereo camera. Additionally, the code written to teleoperate the myCobot 320 M5 is extensible to a wide range of robot arms currently available, due to the development being conducted

using the Robot Operating System (ROS).

5.3 Future Work

To further extend the capabilities of this research, and to make it more extensible to real-world applications, several steps can be taken. These future directions are categorized into short-term implementations and long-term visions. The short-term actionables build directly on the current system while the long-term ones aim to broaden the scope of the research and its impact.

5.3.1 Short-Term Improvements

- **Integration of a dedicated GPU SBC:** The current SBC used for this research is a Raspberry Pi 4B, which limits video streaming performance. Upgrading to the NVIDIA Jetson Nano would enable real-time internet streaming of the stereo images using network protocols and the built-in ZED SDK, which require access to a dedicated GPU and CUDA drivers [66].
- **Development of hardware gimbal system:** The simulated 2-axis servo actuated gimbal system should be implemented in the hardware integration component. Utilizing the GPIO pins of the Jetson Nano would allow the stereo camera to track the user's head orientation, which will improve situational awareness, as proven in the simulation segment.
- **Upgrading the HMD:** Upgrading the current VR HMD from the Meta Quest 2 to the Quest Pro can significantly improve the precision of hand tracking, due to an array of improved spatial mapping sensors, enabling for more precise control of the end effector.

5.3.2 Long-Term Extensions

- **Deployment of the system in hazardous field environments:** The current system can be extended by mounting the robot arm to a unmanned ground vehicle (UGV), where it can be tasked with generating a digital twin of an unknown or inaccessible environment using the stereo camera system's point cloud generation tool.
- **Semi-autonomous assistant algorithms:** Develop machine learning models that enable the system to recognize repetitive task executions, such as pick-and-place of objects, utilizing machine vision, to reduce teleoperation workload of the operator.
- **Sensor fusion:** Add LiDAR sensors, with a shorter focal length, and fuse the generated depth data with the stereo cameras point cloud extraction tool to construct more detailed real-time digital maps.

References

- [1] R. D. S. G. Campilho and F. J. G. Silva, “Industrial process improvement by automation and robotics,” *Machines*, vol. 11, no. 11, 2023, ISSN: 2075-1702. DOI: 10.3390/machines11111011. [Online]. Available: <https://www.mdpi.com/2075-1702/11/11/1011>.
- [2] B. Fu *et al.*, “Towards virtualized welding: Visualization and monitoring of remote welding,” in *2014 IEEE International Conference on Multimedia and Expo (ICME)*, 2014, pp. 1–6. DOI: 10.1109/ICME.2014.6890297.
- [3] R. Smith, E. Cucco, and C. Fairbairn, “Robotic development for the nuclear environment: Challenges and strategy,” *Robotics*, vol. 9, no. 4, 2020, ISSN: 2218-6581. DOI: 10.3390/robotics9040094. [Online]. Available: <https://www.mdpi.com/2218-6581/9/4/94>.
- [4] C. Brosque, E. Galbally, O. Khatib, and M. Fischer, “Human-robot collaboration in construction: Opportunities and challenges,” in *2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*, 2020, pp. 1–8. DOI: 10.1109/HORA49412.2020.9152888.
- [5] M. Zhang, R. Xu, H. Wu, J. Pan, and X. Luo, “Human-robot collaboration for on-site construction,” *Automation in Construction*, vol. 150, p. 104812, 2023, ISSN: 0926-5805. DOI: <https://doi.org/10.1016/j.autcon.2023.104812>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0926580523000729>.
- [6] D. Mukherjee, K. Gupta, L. H. Chang, and H. Najjaran, “A survey of robot learning strategies for human-robot collaboration in industrial settings,” *Robotics and Computer-Integrated Manufacturing*, vol. 73, p. 102231, 2022, ISSN: 0736-5845. DOI: <https://doi.org/10.1016/j.rcim.2021.102231>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0736584521001137>.
- [7] B. Illing, M. Westhoven, B. Gaspers, N. Smets, B. Brüggemann, and T. Mathew, “Evaluation of immersive teleoperation systems using standardized tasks and measurements,” in *2020 29th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*, 2020, pp. 278–285. DOI: 10.1109/RO-MAN47096.2020.9223497.

- [8] S. R. Ellis, B. D. Adelstein, and R. B. Welch, "Kinesthetic compensation for misalignment of teleoperator controls through cross-modal transfer of movement coordinates," *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 46, no. 17, pp. 1551–1555, 2002. DOI: 10.1177/154193120204601705. eprint: <https://doi.org/10.1177/154193120204601705>. [Online]. Available: <https://doi.org/10.1177/154193120204601705>.
- [9] M. Wonsick and T. Padir, "A systematic review of virtual reality interfaces for controlling and interacting with robots," *Applied Sciences*, vol. 10, no. 24, 2020, ISSN: 2076-3417. DOI: 10.3390/app10249051. [Online]. Available: <https://www.mdpi.com/2076-3417/10/24/9051>.
- [10] L. Meng, J. Liu, W. Chai, J. Wang, and M. Q.-H. Meng, "Virtual reality based robot teleoperation via human-scene interaction," *Procedia Computer Science*, vol. 226, pp. 141–148, 2023, Proceedings of International Conference on Biomimetic Intelligence and Robotics, ISSN: 1877-0509. DOI: <https://doi.org/10.1016/j.procs.2023.10.648>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S187705092301815X>.
- [11] X. Han, Y. Chen, Q. Feng, and H. Luo, "Augmented reality in professional training: A review of the literature from 2001 to 2020," *Applied Sciences*, vol. 12, no. 3, 2022, ISSN: 2076-3417. DOI: 10.3390/app12031024. [Online]. Available: <https://www.mdpi.com/2076-3417/12/3/1024>.
- [12] P. A. Suri, M. E. Syahputra, A. S. H. Amany, and A. Djafar, "Systematic literature review: The use of virtual reality as a learning media," *Procedia Computer Science*, vol. 216, pp. 245–251, 2023, 7th International Conference on Computer Science and Computational Intelligence 2022, ISSN: 1877-0509. DOI: <https://doi.org/10.1016/j.procs.2022.12.133>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1877050922022116>.
- [13] T. Pladere, E. Svarverud, G. Krumina, S. J. Gilson, and R. C. Baraas, "Inclusivity in stereoscopic xr: Human vision first," *Frontiers in Virtual Reality*, vol. 3, 2022, ISSN: 2673-4192. DOI: 10.3389/frvir.2022.1006021. [Online]. Available: <https://www.frontiersin.org/journals/virtual-reality/articles/10.3389/frvir.2022.1006021>.
- [14] T. S. Sievers, B. Schmitt, P. Rückert, M. Petersen, and K. Tracht, "Concept of a mixed-reality learning environment for collaborative robotics," *Procedia Manufacturing*, vol. 45, pp. 19–24, 2020, Learning Factories across the value chain – from innovation to service – The 10th Conference on Learning Factories 2020, ISSN: 2351-9789. DOI: <https://doi.org/10.1016/j.promfg.2020.04.034>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2351978920310726>.
- [15] J. de Souza and R. Tartz, "Visual perception and user satisfaction in video see-through head-mounted displays: A mixed-methods evaluation," *Frontiers in Virtual Reality*, vol. 5, 2024, ISSN: 2673-4192. DOI: 10.3389/frvir.2024.1368721. [Online]. Available: <https://www.frontiersin.org/journals/virtual-reality/articles/10.3389/frvir.2024.1368721>.

- [16] E. Dandil and K. K. Çevik, “Computer vision based distance measurement system using stereo camera view,” in *2019 3rd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, 2019, pp. 1–4. DOI: 10.1109/ISMSIT.2019.8932817.
- [17] S. Kawatsuma, M. Fukushima, and T. Okada, “Emergency response by robots to fukushima-daiichi accident: Summary and lessons learned,” *Industrial Robot: An International Journal*, vol. 39, no. 5, pp. 428–435, 2012, ISSN: 0143-991X. DOI: 10.1108/01439911211249715. [Online]. Available: <https://doi.org/10.1108/01439911211249715>.
- [18] S. Chen *et al.*, “A dual-arm handling robot for nuclear emergency tasks,” in *2023 2nd International Conference on Robotics, Artificial Intelligence and Intelligent Control (RAIIC)*, 2023, pp. 7–12. DOI: 10.1109/RAIIC59453.2023.10280845.
- [19] R. Bogue, “Underwater robots: A review of technologies and applications,” *Industrial Robot: An International Journal*, vol. 42, no. 3, pp. 186–191, 2015, ISSN: 0143-991X. DOI: 10.1108/IR-01-2015-0010. [Online]. Available: <https://doi.org/10.1108/IR-01-2015-0010>.
- [20] K. Yoshida, “Achievements in space robotics,” *IEEE Robotics & Automation Magazine*, vol. 16, no. 4, pp. 20–28, 2009. DOI: 10.1109/MRA.2009.934818.
- [21] R. Mamen, “Applying space technologies for human benefit; the canadian experience and global trends,” in *International Conference on Recent Advances in Space Technologies, 2003. RAST '03. Proceedings of*, 2003, pp. 1–8. DOI: 10.1109/RAST.2003.1303381.
- [22] A. R. Vasavada, “Mission overview and scientific contributions from the mars science laboratory curiosity rover after eight years of surface operations,” *Space Science Reviews*, vol. 218, no. 3, p. 14, 2022, ISSN: 1572-9672. DOI: 10.1007/s11214-022-00882-7. [Online]. Available: <https://doi.org/10.1007/s11214-022-00882-7>.
- [23] G. A. Landis, “Robotic exploration of the surface and atmosphere of venus,” *Acta Astronautica*, vol. 59, no. 7, pp. 570–579, 2006, ISSN: 0094-5765. DOI: <https://doi.org/10.1016/j.actaastro.2006.04.011>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0094576506001767>.
- [24] M. Schwarz *et al.*, “Nimbro rescue: Solving disaster-response tasks with the mobile manipulation robot momaro,” *Journal of Field Robotics*, vol. 34, no. 2, pp. 400–425, 2017. DOI: <https://doi.org/10.1002/rob.21677>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/rob.21677>. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21677>.
- [25] C. Wong, E. Yang, X.-T. Yan, and D. Gu, “An overview of robotics and autonomous systems for harsh environments,” in *2017 23rd International Conference on Automation and Computing (ICAC)*, 2017, pp. 1–6. DOI: 10.23919/ICOnAC.2017.8082020.

- [26] E. Z. Mario Silvagni Andrea Tonoli and M. Chiaberge, “Multipurpose uav for search and rescue operations in mountain avalanche events,” *Geomatics, Natural Hazards and Risk*, vol. 8, no. 1, pp. 18–33, 2017. DOI: 10.1080/19475705.2016.1238852. eprint: <https://doi.org/10.1080/19475705.2016.1238852>. [Online]. Available: <https://doi.org/10.1080/19475705.2016.1238852>.
- [27] S. Dritsas and G. S. Soh, “Building robotics design for construction,” *Construction Robotics*, vol. 3, no. 1, pp. 1–10, 2019, ISSN: 2509-8780. DOI: 10.1007/s41693-018-0010-1. [Online]. Available: <https://doi.org/10.1007/s41693-018-0010-1>.
- [28] B. Wang, S. J. Hu, L. Sun, and T. Freiheit, “Intelligent welding system technologies: State-of-the-art review and perspectives,” *Journal of Manufacturing Systems*, vol. 56, pp. 373–391, 2020, ISSN: 0278-6125. DOI: <https://doi.org/10.1016/j.jmsy.2020.06.020>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0278612520301102>.
- [29] S. Zhao *et al.*, “Application and development of autonomous robots in concrete construction: Challenges and opportunities,” *Drones*, vol. 6, no. 12, 2022, ISSN: 2504-446X. DOI: 10.3390/drones6120424. [Online]. Available: <https://www.mdpi.com/2504-446X/6/12/424>.
- [30] J. Kim, D. eun Lee, and J. Seo, “Task planning strategy and path similarity analysis for an autonomous excavator,” *Automation in Construction*, vol. 112, p. 103108, 2020, ISSN: 0926-5805. DOI: <https://doi.org/10.1016/j.autcon.2020.103108>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0926580519305989>.
- [31] S. Halder and K. Afsari, “Robots in inspection and monitoring of buildings and infrastructure: A systematic review,” *Applied Sciences*, vol. 13, no. 4, 2023, ISSN: 2076-3417. DOI: 10.3390/app13042304. [Online]. Available: <https://www.mdpi.com/2076-3417/13/4/2304>.
- [32] L. Thomas and C. Wickens, “Effects of display frames of reference on spatial judgments and change detection,” p. 75, Sep. 2000.
- [33] R. Murphy, “Human-robot interaction in rescue robotics,” *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 34, no. 2, pp. 138–153, 2004. DOI: 10.1109/TSMCC.2004.826267.
- [34] D. R. Lampton, D. P. McDonald, M. Singer, and J. P. Bliss, “Distance estimation in virtual environments,” *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 39, no. 20, pp. 1268–1272, 1995. DOI: 10.1177/154193129503902006. eprint: <https://doi.org/10.1177/154193129503902006>. [Online]. Available: <https://doi.org/10.1177/154193129503902006>.
- [35] A. Carballo *et al.*, “Libre: The multiple 3d lidar dataset,” in *2020 IEEE Intelligent Vehicles Symposium (IV)*, 2020, pp. 1094–1101. DOI: 10.1109/IV47402.2020.9304681.

- [36] X. Zhao, Z. Yang, and S. Schwertfeger, “Mapping with reflection - detection and utilization of reflection in 3d lidar scans,” in *2020 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, 2020, pp. 27–33. DOI: 10.1109/SSRR50563.2020.9292595.
- [37] Z. Hou, W. He, and S. W. and, “Effect of 2d/3d visual and haptic cues for fine operation tasks with varying difficulties,” *International Journal of Human-Computer Interaction*, vol. 41, no. 5, pp. 3004–3013, 2025. DOI: 10.1080/10447318.2024.2329386. eprint: <https://doi.org/10.1080/10447318.2024.2329386>. [Online]. Available: <https://doi.org/10.1080/10447318.2024.2329386>.
- [38] D. Lee, H. Kim, H. Yoon, and W. Lee, “Usability comparison between 2d and 3d control methods for the operation of hovering objects,” *Drones*, vol. 7, no. 8, 2023, ISSN: 2504-446X. DOI: 10.3390/drones7080520. [Online]. Available: <https://www.mdpi.com/2504-446X/7/8/520>.
- [39] D. Puljiz, E. Stöhr, K. S. Riesterer, B. Hein, and T. Kröger, “General hand guidance framework using microsoft hololens,” in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019, pp. 5185–5190. DOI: 10.1109/IROS40897.2019.8967649.
- [40] L. S. Yim *et al.*, “Wfh-vr: Teleoperating a robot arm to set a dining table across the globe via virtual reality,” in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2022, pp. 4927–4934. DOI: 10.1109/IROS47612.2022.9981729.
- [41] Y.-W. Chang, S.-T. Hsu, H.-H. Yeh, and Y.-C. Liu, “An intuitive human-manipulator interface with mixed reality,” in *2023 23rd International Conference on Control, Automation and Systems (ICCAS)*, 2023, pp. 629–634. DOI: 10.23919/ICCAS59377.2023.10316764.
- [42] C. Cruz Ulloa, D. Domínguez, J. Del Cerro, and A. Barrientos, “A mixed-reality tele-operation method for high-level control of a legged-manipulator robot,” *Sensors*, vol. 22, no. 21, 2022, ISSN: 1424-8220. DOI: 10.3390/s22218146. [Online]. Available: <https://www.mdpi.com/1424-8220/22/21/8146>.
- [43] V. H. Diaz-Ramirez, M. Gonzalez-Ruiz, V. Kober, and R. Juarez-Salazar, “Stereo image matching using adaptive morphological correlation,” *Sensors*, vol. 22, no. 23, 2022, ISSN: 1424-8220. DOI: 10.3390/s22239050. [Online]. Available: <https://www.mdpi.com/1424-8220/22/23/9050>.
- [44] D. Scharstein, R. Szeliski, and R. Zabih, “A taxonomy and evaluation of dense two-frame stereo correspondence algorithms,” in *Proceedings IEEE Workshop on Stereo and Multi-Baseline Vision (SMBV 2001)*, 2001, pp. 131–140. DOI: 10.1109/SMBV.2001.988771.
- [45] J.-R. Chang and Y.-S. Chen, *Pyramid stereo matching network*, 2018. arXiv: 1803.08669 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1803.08669>.

- [46] J. E. Cutting, “How the eye measures reality and virtual reality,” *Behavior Research Methods, Instruments, & Computers*, vol. 29, no. 1, pp. 27–36, 1997, ISSN: 1532-5970. DOI: 10.3758/BF03200563. [Online]. Available: <https://doi.org/10.3758/BF03200563>.
- [47] P. Milgram and F. Kishino, “A taxonomy of mixed reality visual displays,” *IEICE Trans. Information Systems*, vol. vol. E77-D, no. 12, pp. 1321–1329, Dec. 1994.
- [48] P. Milgram, S. Zhai, D. Drascic, and J. Grodski, “Applications of augmented reality for human-robot communication,” in *Proceedings of 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '93)*, vol. 3, 1993, 1467–1472 vol.3. DOI: 10.1109/IROS.1993.583833.
- [49] Stereolabs, *Zed mini stereo camera*, Accessed: 2025-06-07, 2024. [Online]. Available: <https://www.stereolabs.com/en-ca/store/products/zed-mini>.
- [50] R. Keating and N. J. Cowan, *Ur5 inverse kinematics*, 2016. [Online]. Available: https://tianyusong.com/wp-content/uploads/2017/12/ur5_inverse_kinematics.pdf.
- [51] X. Huang, L. Kong, and G. Dong, “Modeling and compensation of motion errors for 6-dof robotic manipulators,” *Applied Sciences*, vol. 11, no. 21, 2021, ISSN: 2076-3417. DOI: 10.3390/app112110100. [Online]. Available: <https://www.mdpi.com/2076-3417/11/21/10100>.
- [52] J. Karpińska, K. Tchoń, and M. Janiak, “Approximation of jacobian inverse kinematics algorithms: Differential geometric vs. variational approach,” *Journal of Intelligent & Robotic Systems*, vol. 68, no. 3, pp. 211–224, 2012, ISSN: 1573-0409. DOI: 10.1007/s10846-012-9679-4. [Online]. Available: <https://doi.org/10.1007/s10846-012-9679-4>.
- [53] I. Agustian, N. Daratha, R. Faurina, A. Suandi, and Sulistyaningsih, “Robot manipulator control with inverse kinematics pd-pseudoinverse jacobian and forward kinematics denavit hartenberg,” *CoRR*, vol. abs/2103.10461, 2021. arXiv: 2103.10461. [Online]. Available: <https://arxiv.org/abs/2103.10461>.
- [54] M. Ivanou, S. Mikhel, and S. Savin, “Robot description formats and approaches: Review,” in *2021 International Conference "Nonlinearity, Information and Robotics" (NIR)*, 2021, pp. 1–5. DOI: 10.1109/NIR52917.2021.9666120.
- [55] P. Manceron, *Ikpy*, version v3.4, Aug. 2024. DOI: 10.5281/zenodo.13256291. [Online]. Available: <https://doi.org/10.5281/zenodo.13256291>.
- [56] J. Bohuslava, J. Martin, and H. Igor, “Tcp/ip protocol utilisation in process of dynamic control of robotic cell according industry 4.0 concept,” in *2017 IEEE 15th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, 2017, pp. 000 217–000 222. DOI: 10.1109/SAMI.2017.7880306.
- [57] Meta, *Hand tracking overview*, Online, <https://developer.oculus.com/documentation/unity/unity-handtracking/>, 2021.

- [58] A. Casile *et al.*, “Quantitative comparison of hand kinematics measured with a markerless commercial head-mounted display and a marker-based motion capture system in stroke survivors,” *Sensors*, vol. 23, no. 18, 2023, ISSN: 1424-8220. DOI: 10.3390/s23187906. [Online]. Available: <https://www.mdpi.com/1424-8220/23/18/7906>.
- [59] D. Abdulkarim *et al.*, “A methodological framework to assess the accuracy of virtual reality hand-tracking systems: A case study with the meta quest 2,” *Behavior Research Methods*, vol. 56, no. 2, pp. 1052–1063, 2024, Epub 2023 Feb 13. DOI: 10.3758/s13428-022-02051-8.
- [60] Google, *Mediapipe hand landmarker*, https://ai.google.dev/edge/mediapipe/solutions/vision/hand_landmarker, Accessed: 2025-06-07, 2023.
- [61] J. Kerr and K. Nickels, “Robot operating systems: Bridging the gap between human and robot,” in *Proceedings of the 2012 44th Southeastern Symposium on System Theory (SSST)*, 2012, pp. 99–104. DOI: 10.1109/SSST.2012.6195127.
- [62] D. Kent, C. Saldanha, and S. Chernova, “A comparison of remote robot teleoperation interfaces for general object manipulation,” in *2017 12th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, 2017, pp. 371–379.
- [63] G. Xylomenos and G. Polyzos, “Tcp and udp performance over a wireless lan,” in *IEEE INFOCOM ’99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320)*, vol. 2, 1999, 439–446 vol.2. DOI: 10.1109/INFOCOM.1999.751376.
- [64] A. Savitzky and M. J. E. Golay, “Smoothing and differentiation of data by simplified least squares procedures.,” *Analytical Chemistry*, vol. 36, no. 8, pp. 1627–1639, 1964. DOI: 10.1021/ac60214a047. eprint: <https://doi.org/10.1021/ac60214a047>. [Online]. Available: <https://doi.org/10.1021/ac60214a047>.
- [65] R. W. Schafer, “What is a savitzky-golay filter? [lecture notes],” *IEEE Signal Processing Magazine*, vol. 28, no. 4, pp. 111–117, 2011. DOI: 10.1109/MSP.2011.941097.
- [66] A. A. Süzen, B. Duman, and B. Şen, “Benchmark analysis of jetson tx2, jetson nano and raspberry pi using deep-cnn,” in *2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*, 2020, pp. 1–5. DOI: 10.1109/HORA49412.2020.9152915.

Appendix A: Graphs and Plots of All Participants

A.1 Trajectory Plots

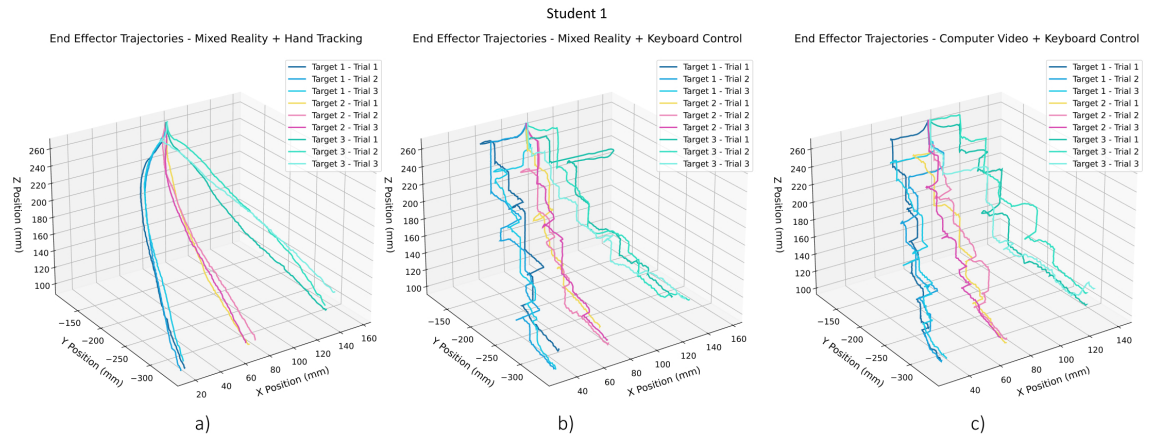


Figure A.1: Student 1 - Trajectory of the robot arm's end effector, for multiple runs, using the (a) Mixed Reality Hand Control method, (b) Mixed Reality Keyboard Input method, and (c) Computer Video Keyboard Input method

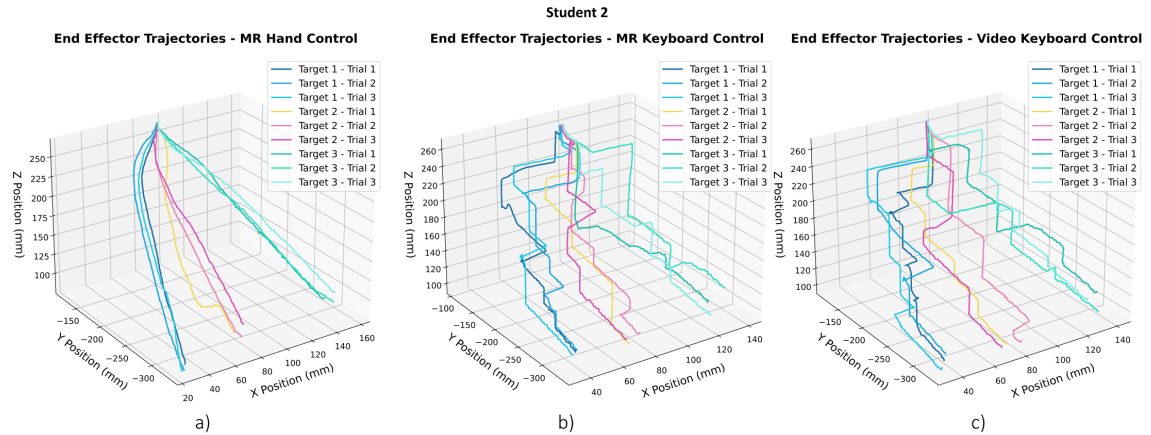


Figure A.2: Student 2 trajectories plots.

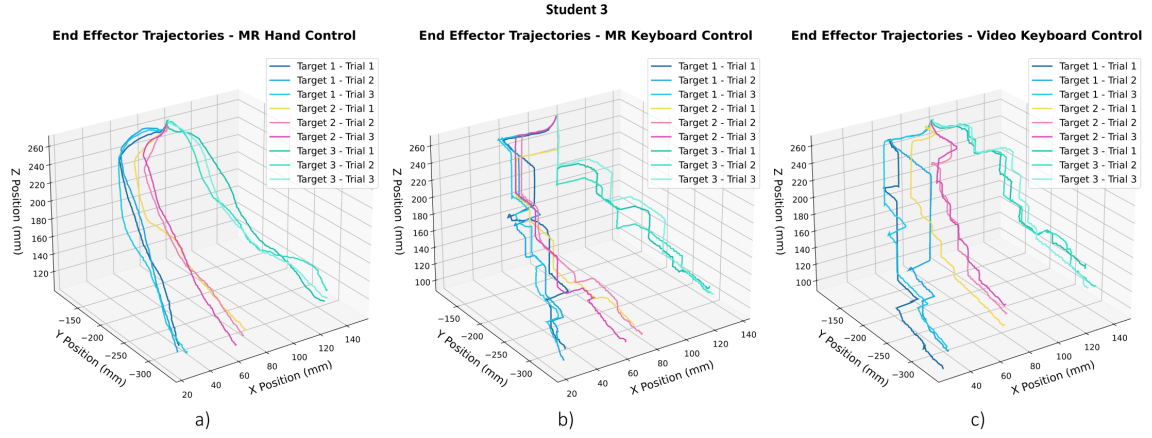


Figure A.3: Student 3 trajectories plots.

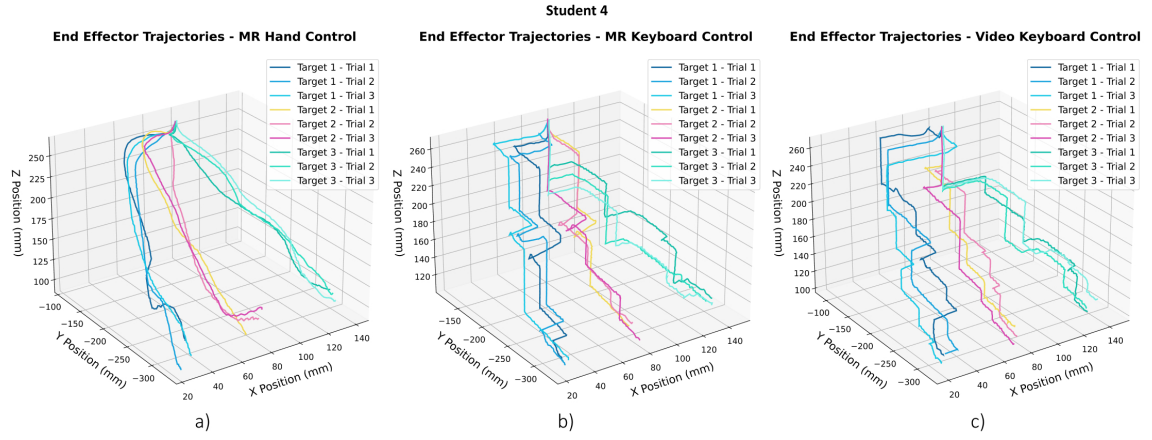


Figure A.4: Student 4 trajectories plots.

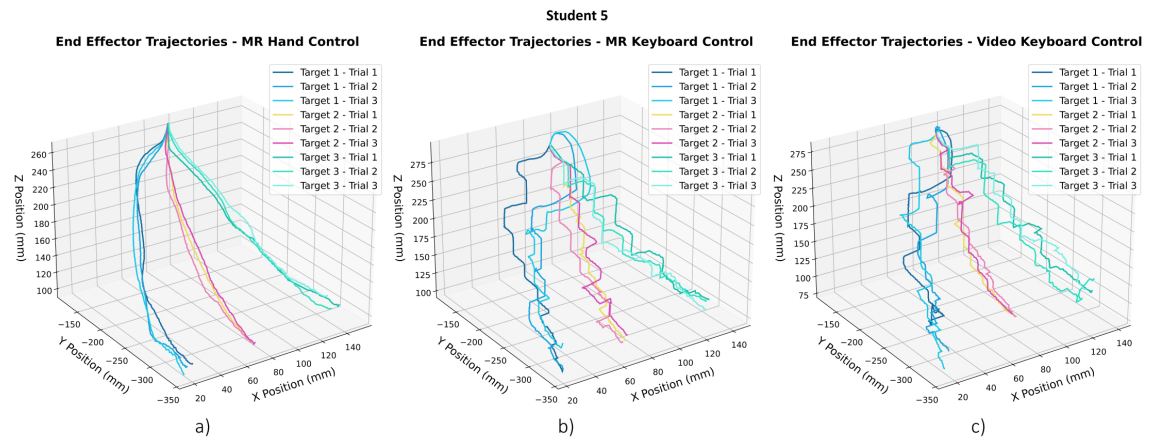


Figure A.5: Student 5 trajectories plots.

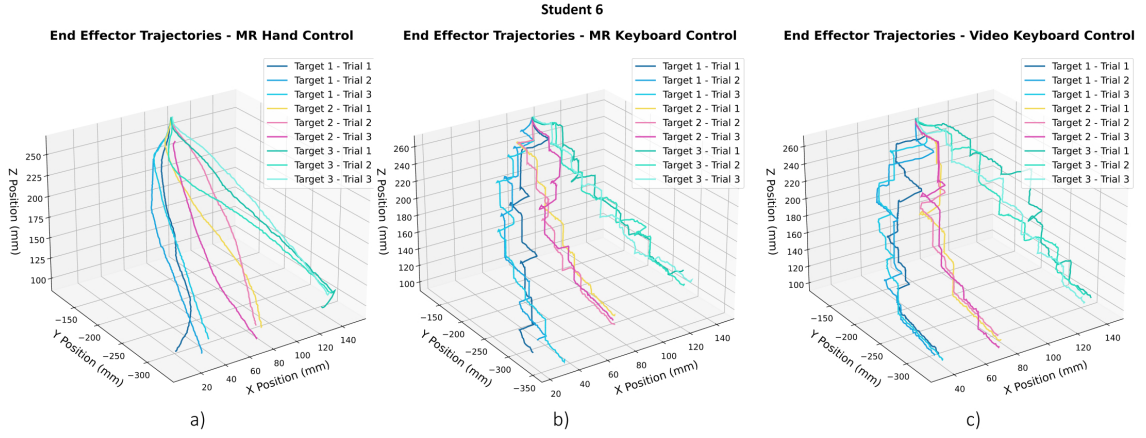


Figure A.6: Student 6 trajectories plots.

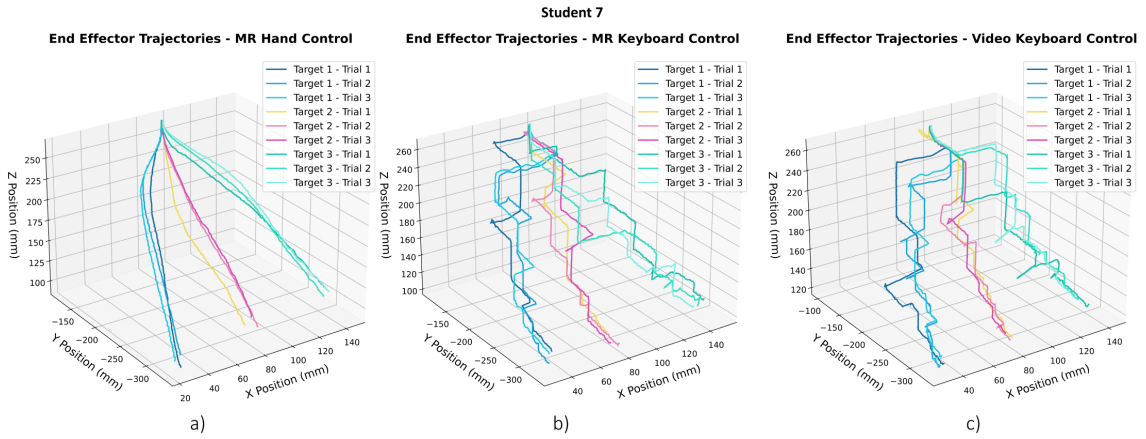


Figure A.7: Student 7 trajectories plots.

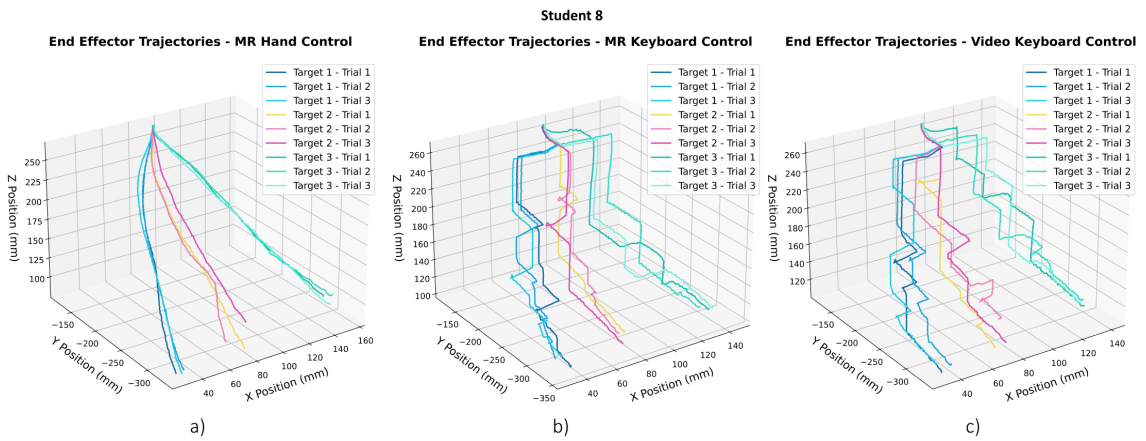


Figure A.8: Student 8 trajectories plots.

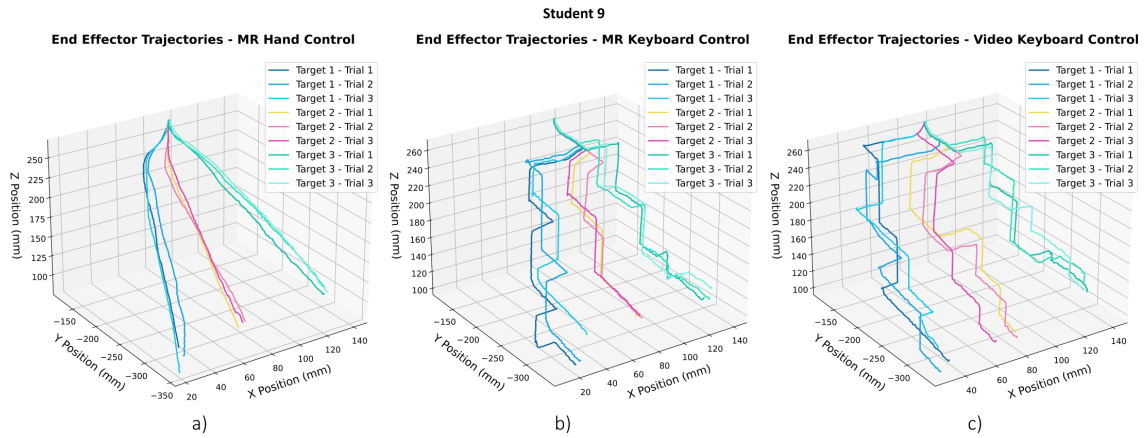


Figure A.9: Student 9 trajectories plots.

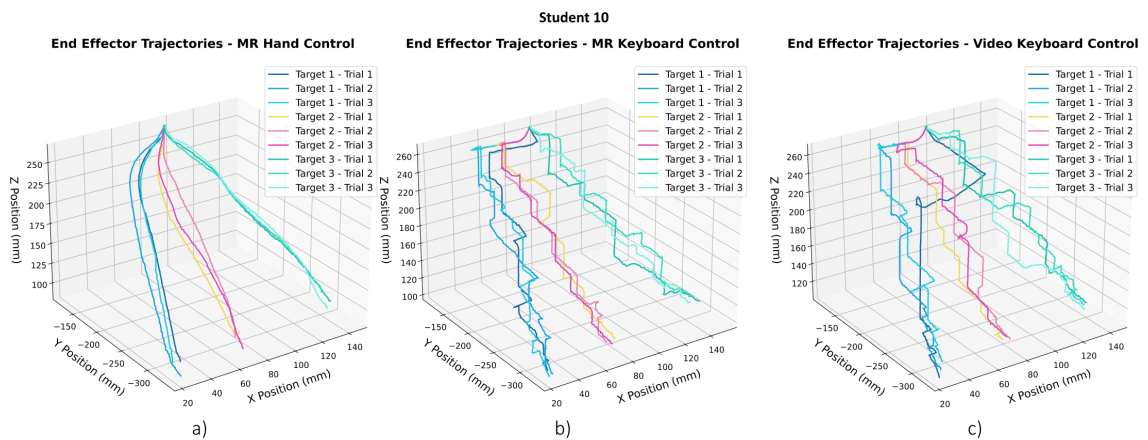


Figure A.10: Student 10 trajectories plots.

A.2 Distance Minimization Plots

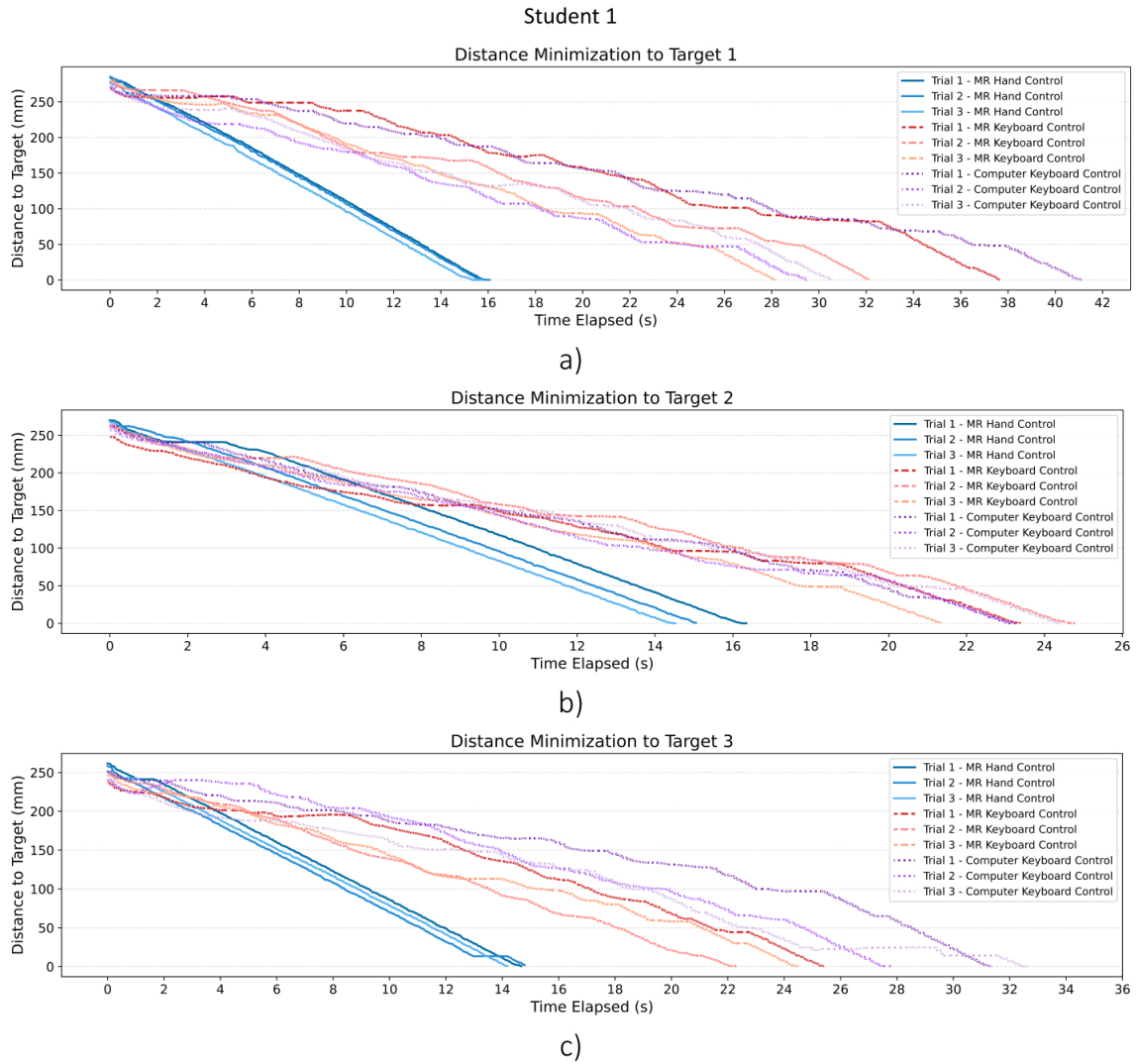


Figure A.11: Distance minimization plots for Student 1. Each subplot corresponds to a target cube: (a) Target 1 (right), (b) Target 2 (centre), and (c) Target 3 (left). Each line represents a trial using one of the three control methods.

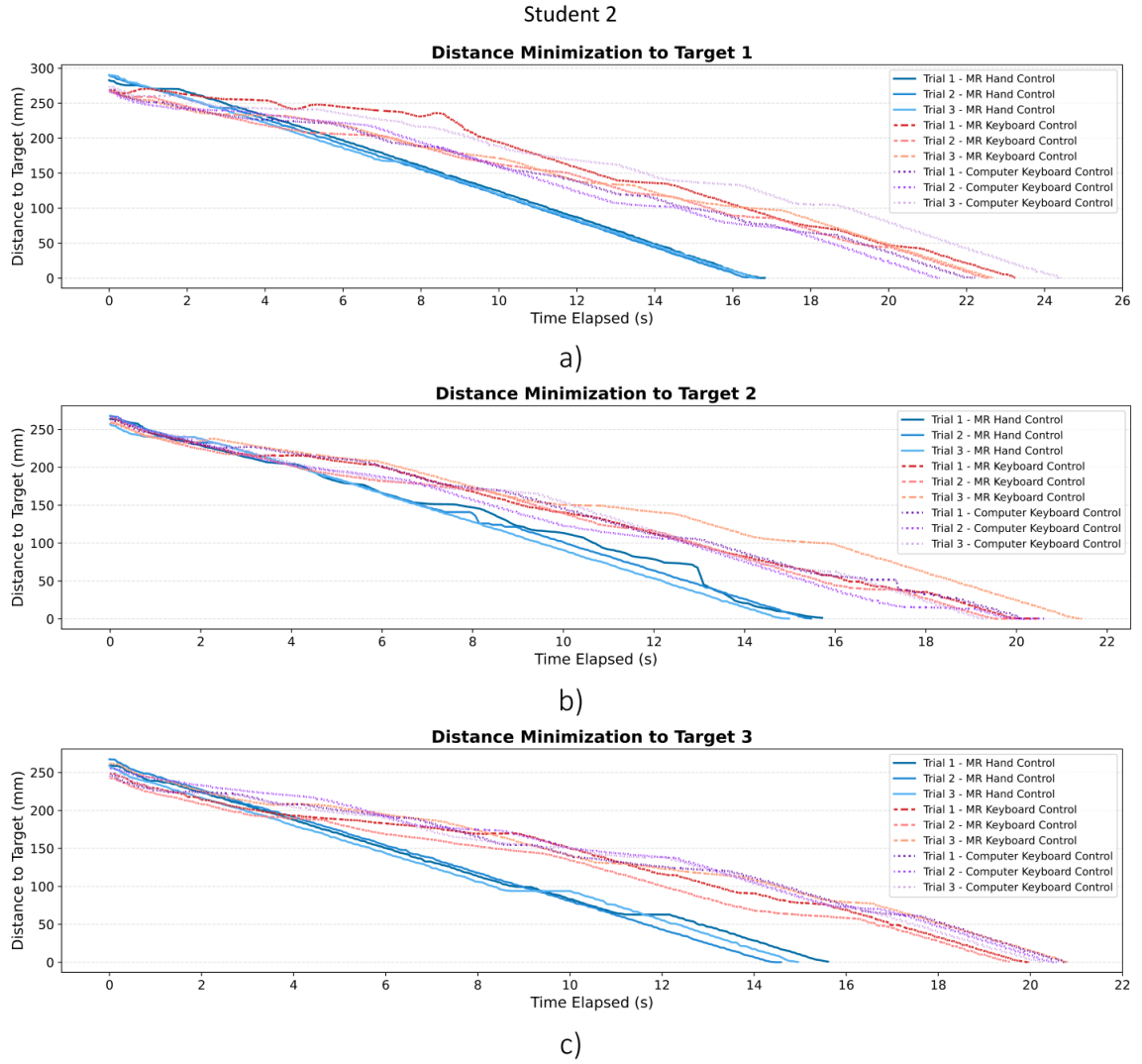


Figure A.12: Student 2 distance minimization plots.

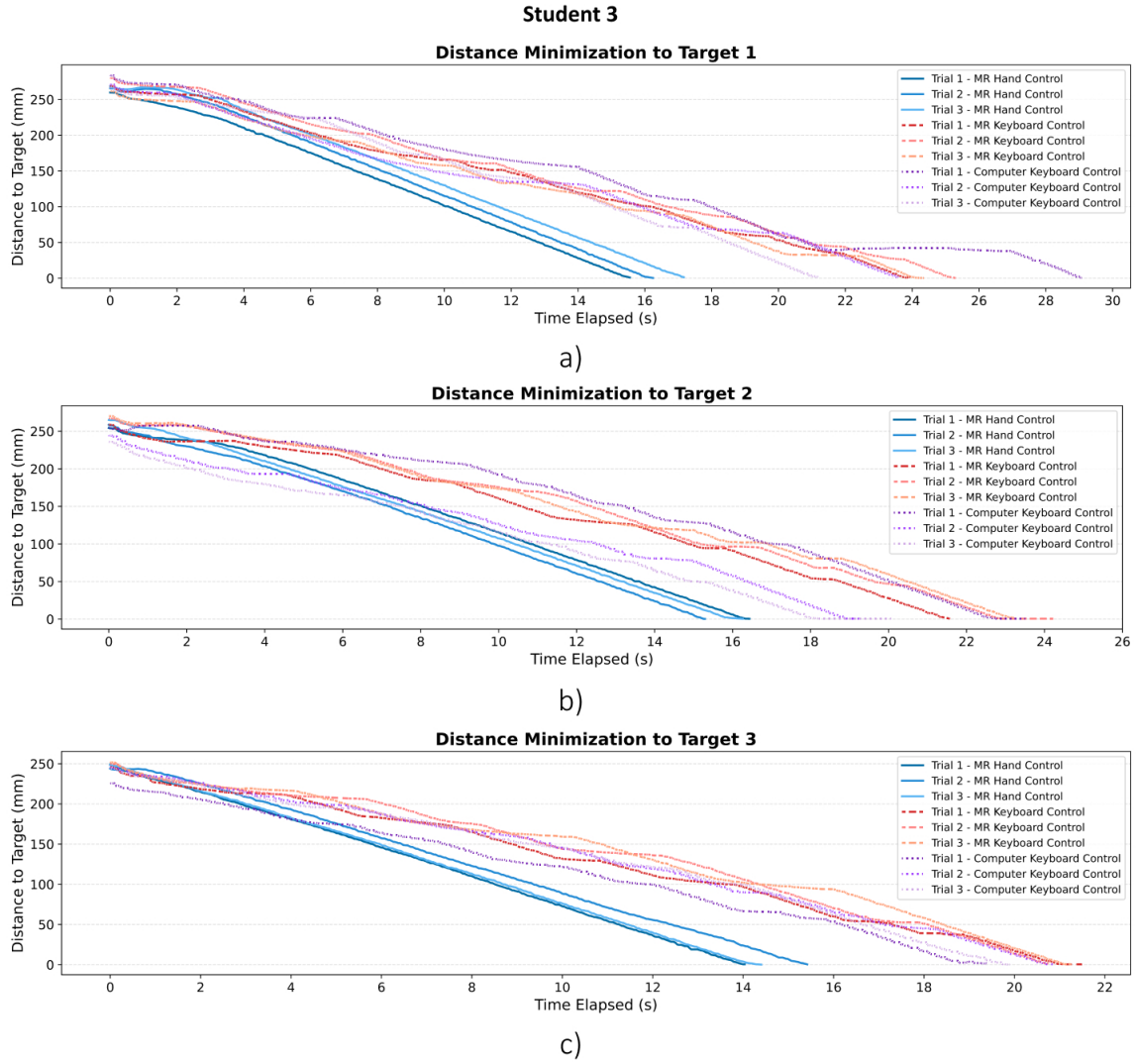


Figure A.13: Student 3 distance minimization plots.

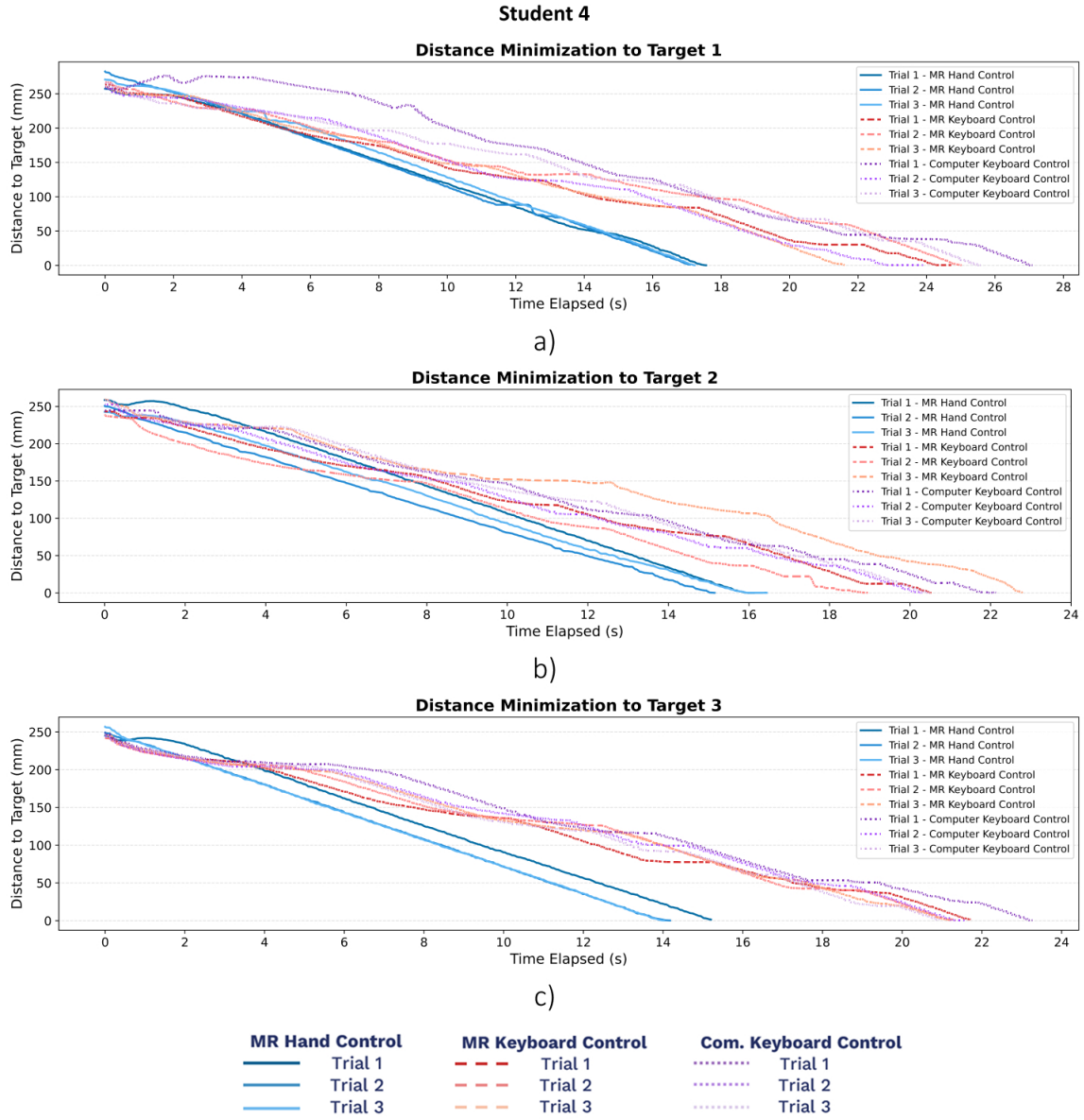


Figure A.14: Student 4 distance minimization plots.

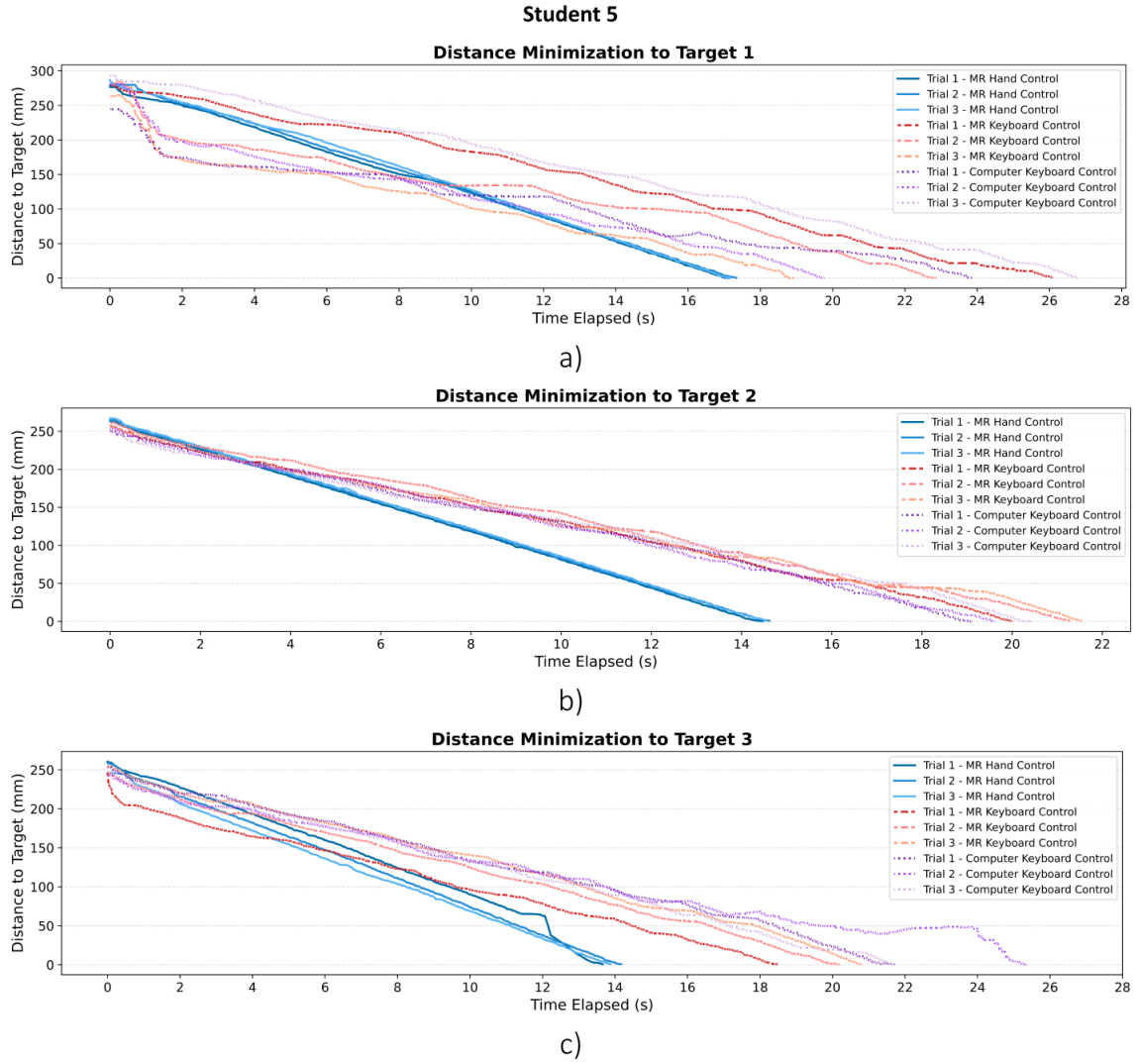


Figure A.15: Student 5 distance minimization plots.

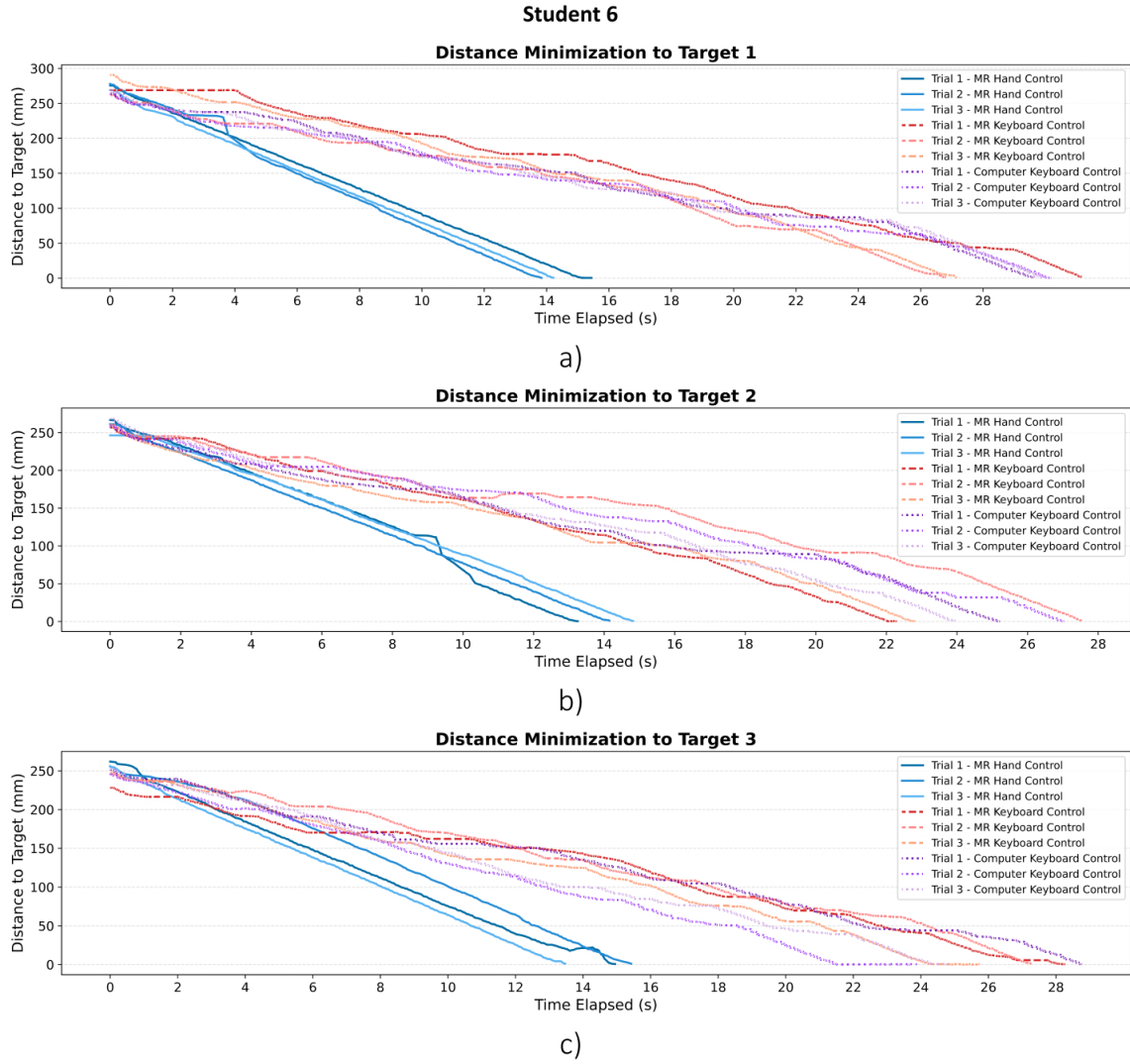


Figure A.16: Student 6 distance minimization plots.

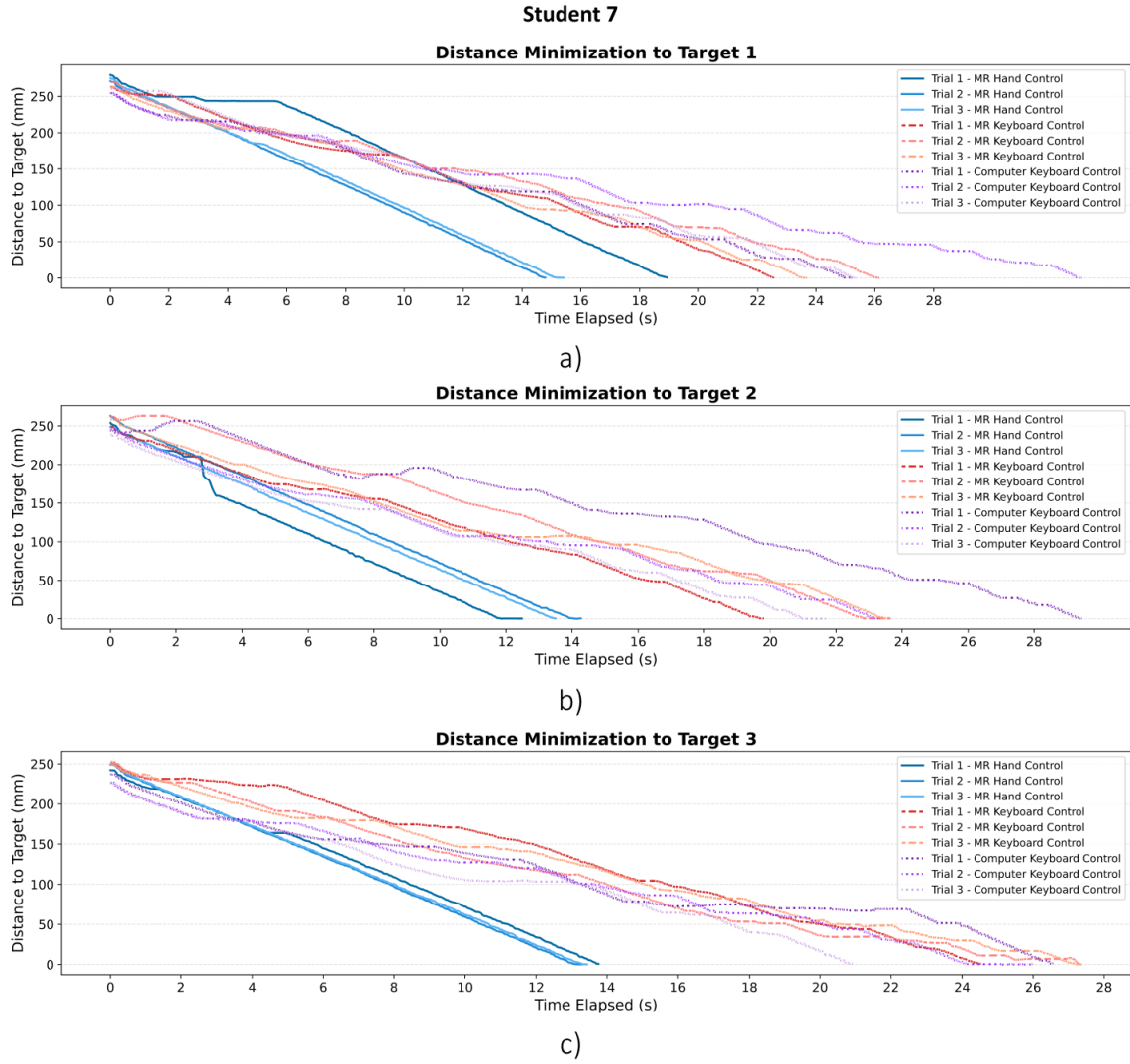


Figure A.17: Student 7 distance minimization plots.

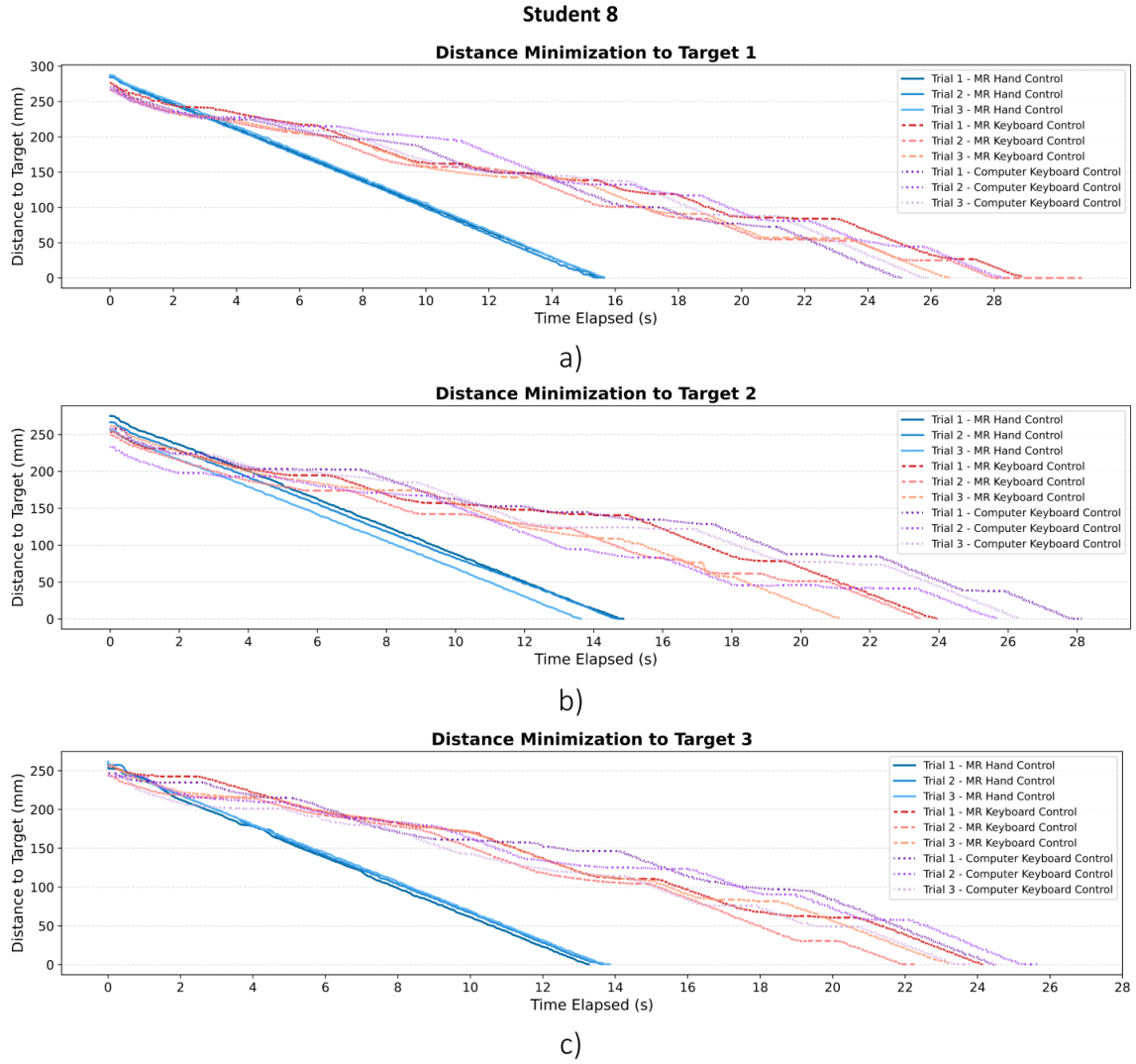


Figure A.18: Student 8 distance minimization plots.

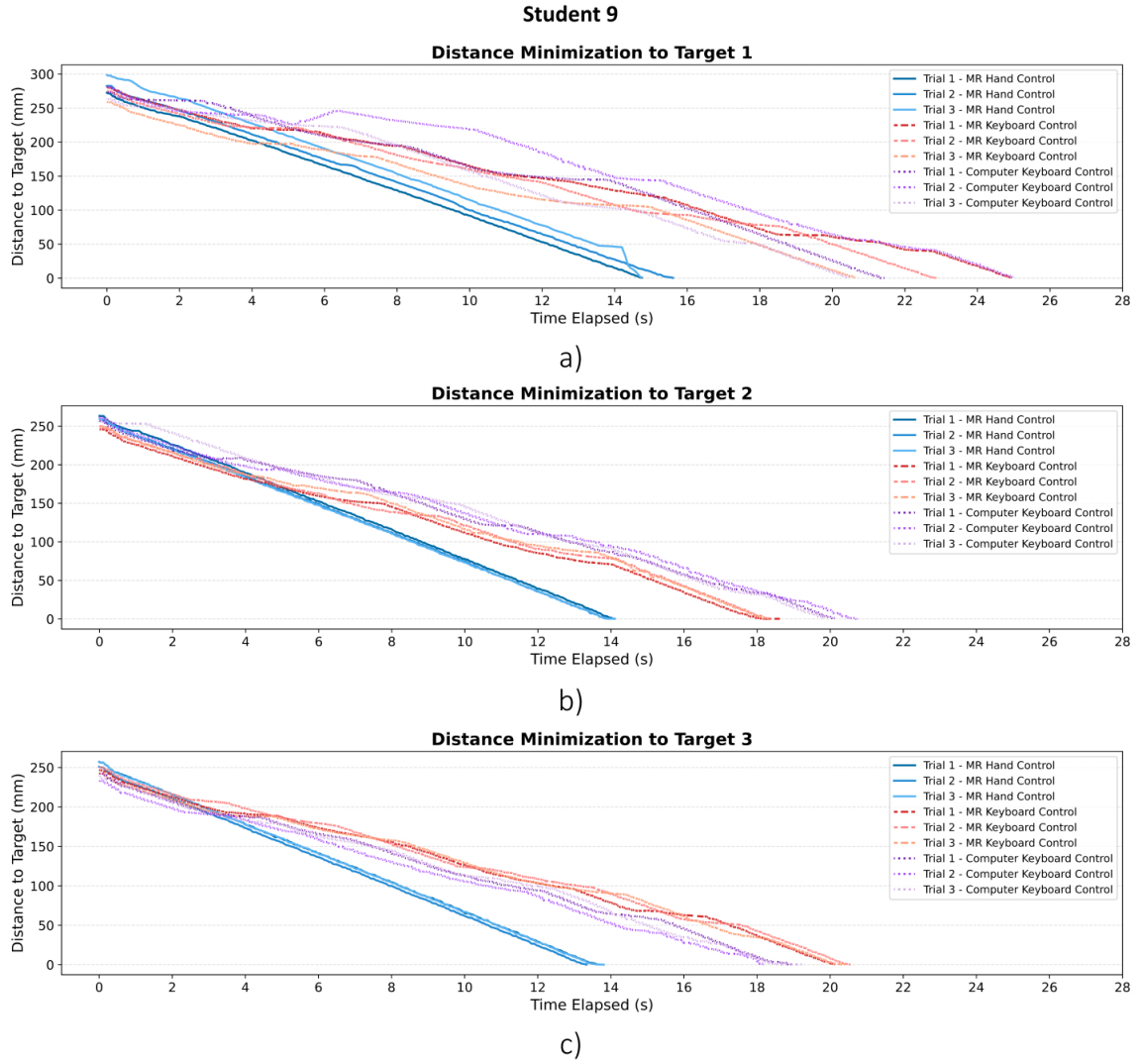


Figure A.19: Student 9 distance minimization plots.

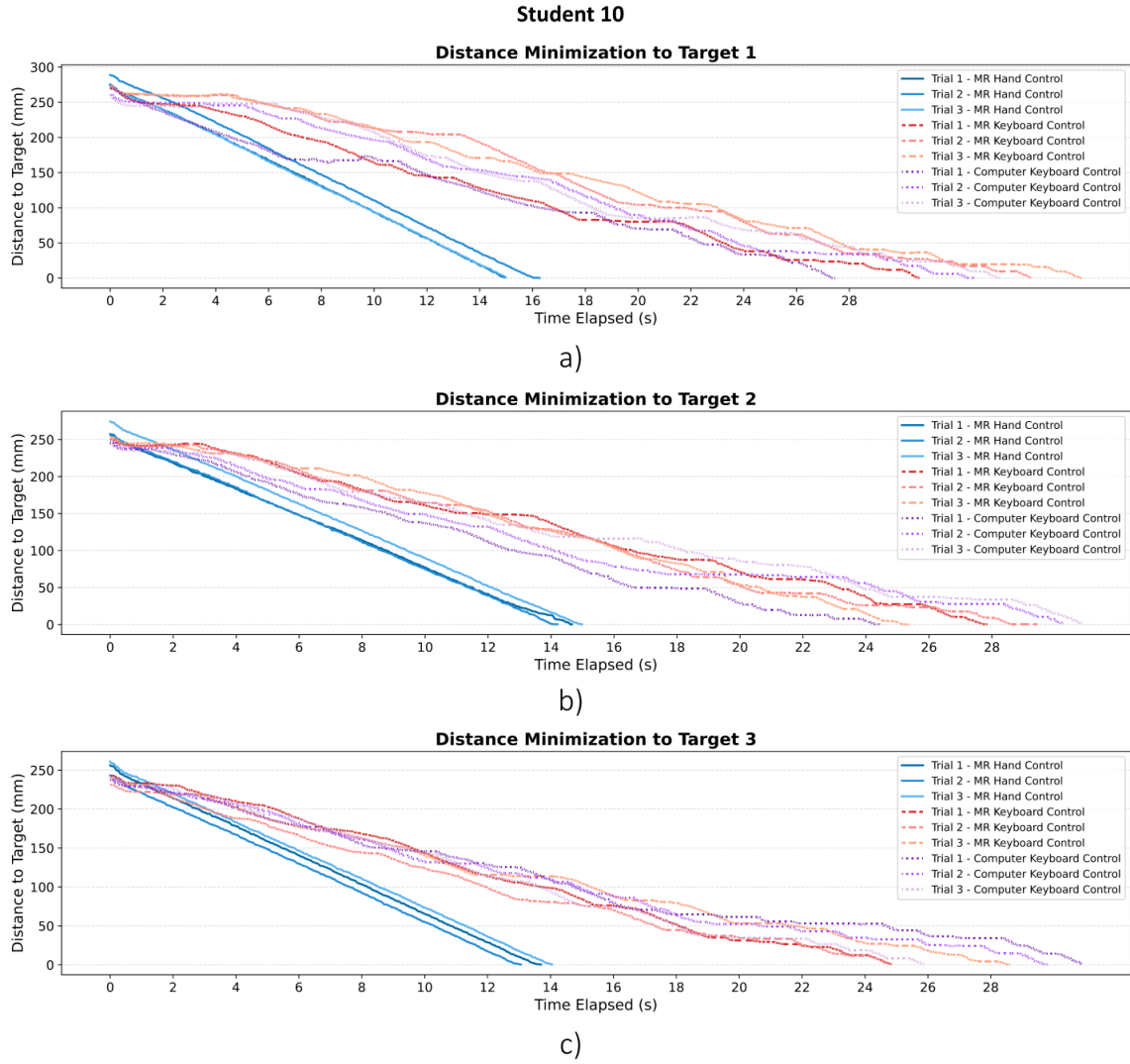


Figure A.20: Student 10 distance minimization plots.

A.3 Jerk Profile Plots

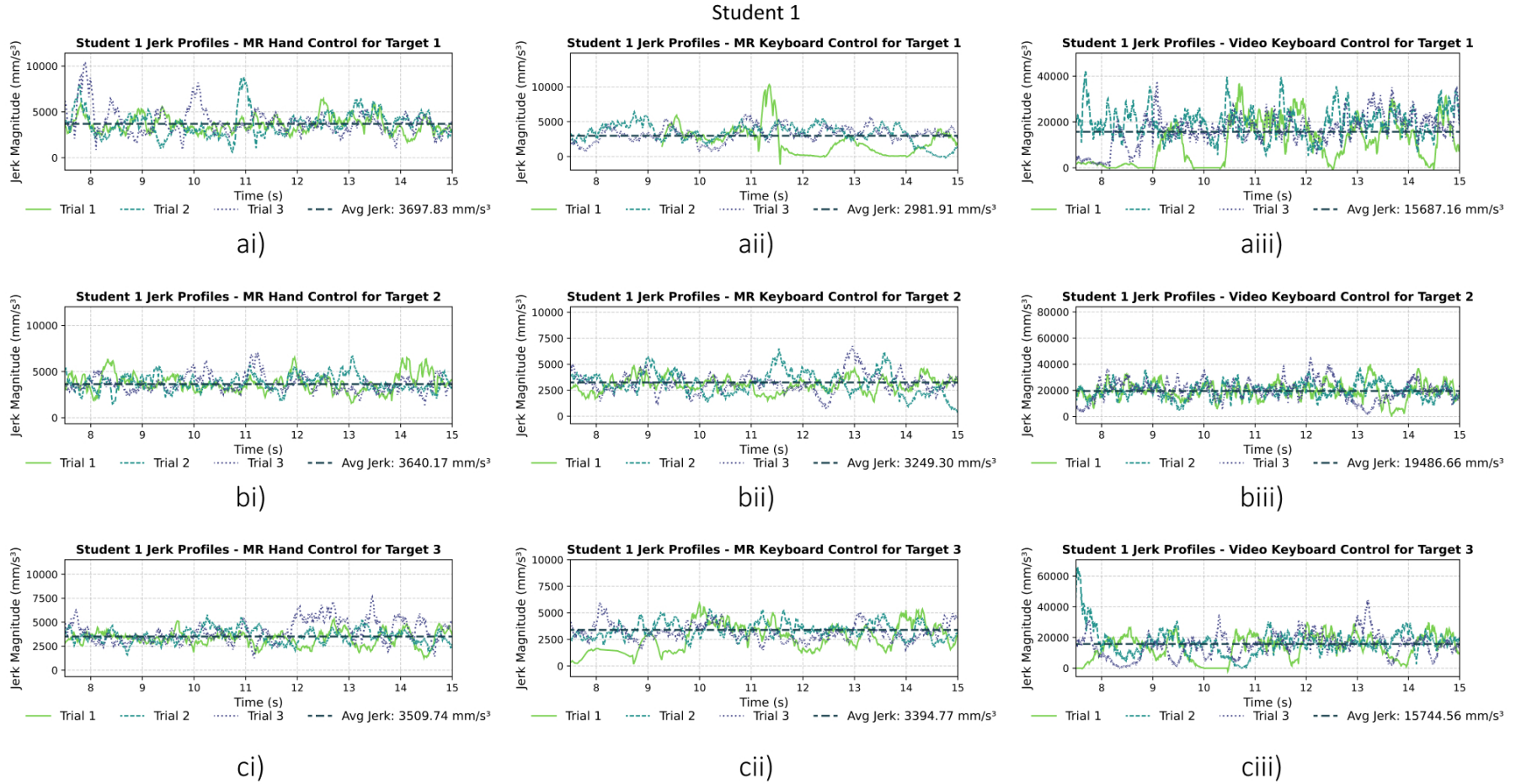
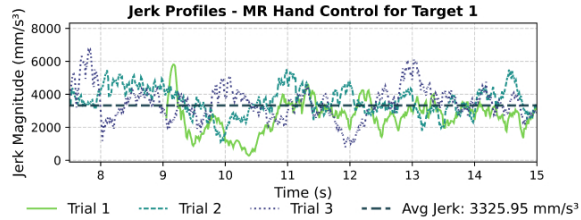
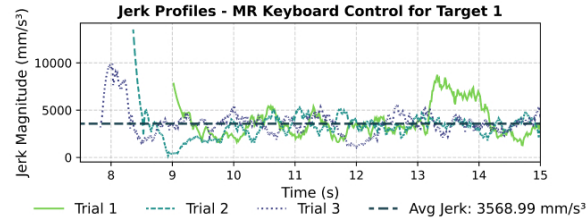


Figure A.21: Jerk profiles for Student 1 across all control types and target cube positions. Each row corresponds to a different target, and each column corresponds to a different control mode. Coloured lines represent filtered jerk magnitude across trials; dashed horizontal lines indicate average jerk.

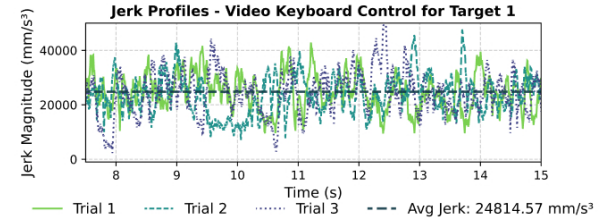
Student 2



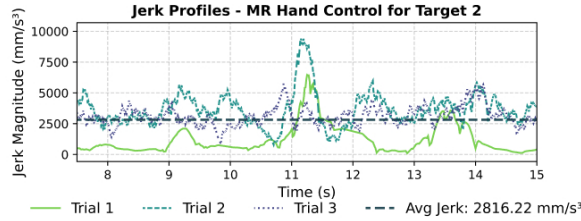
ai)



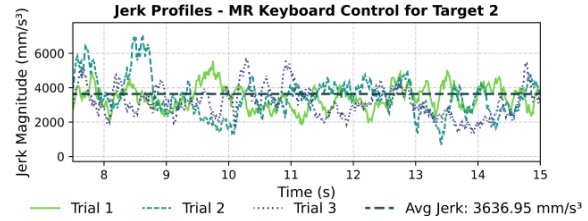
aii)



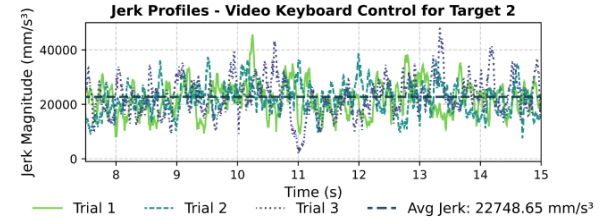
aiii)



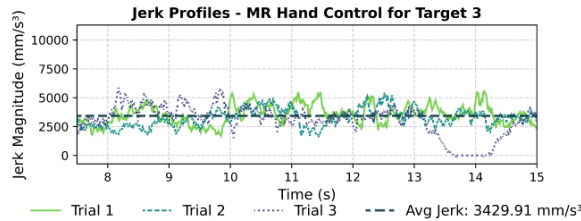
bi)



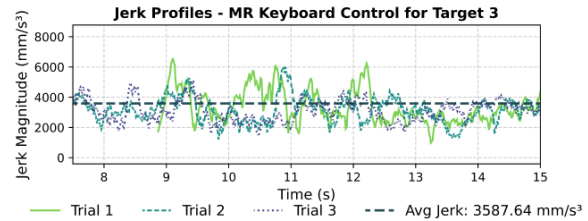
bii)



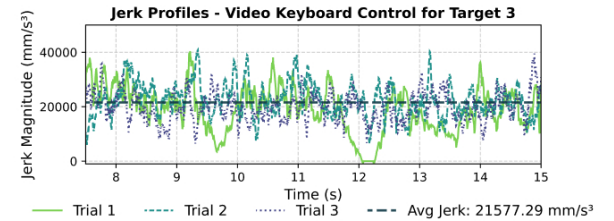
biii)



ci)



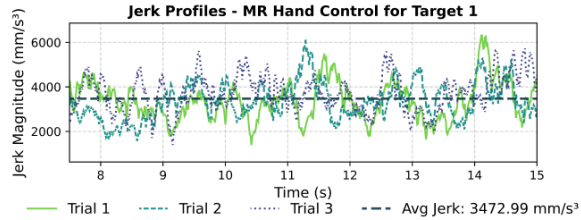
cii)



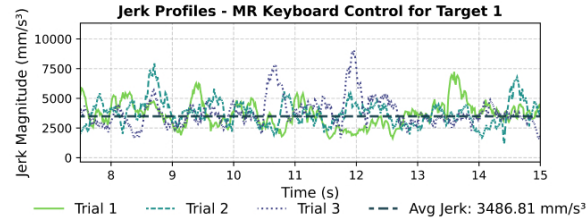
ciii)

Figure A.22: Student 2 jerk profile plots.

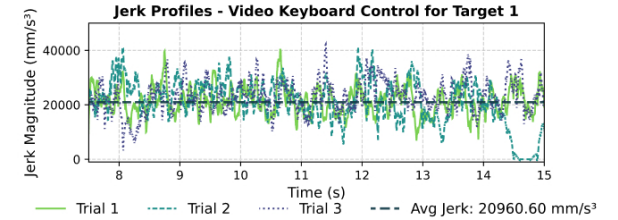
Student 3



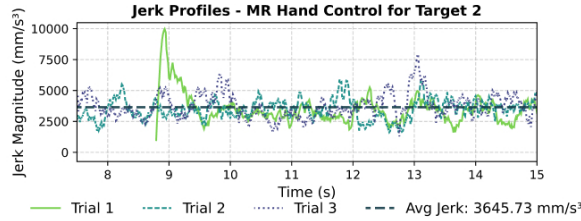
ai)



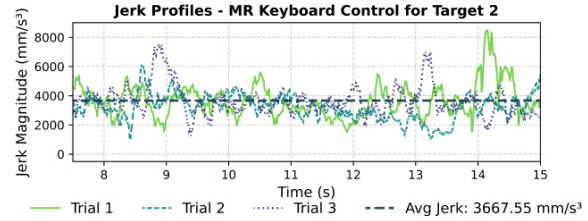
aii)



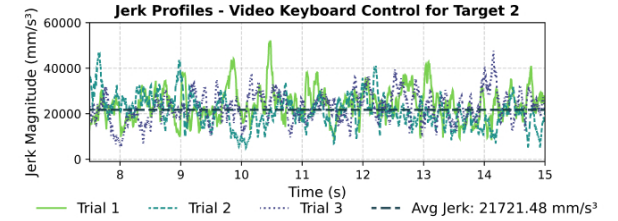
aiii)



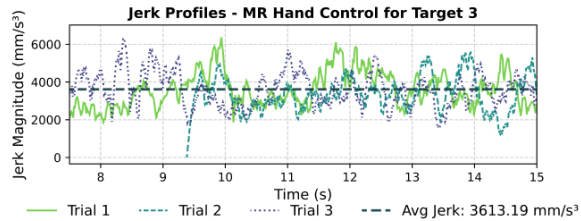
bi)



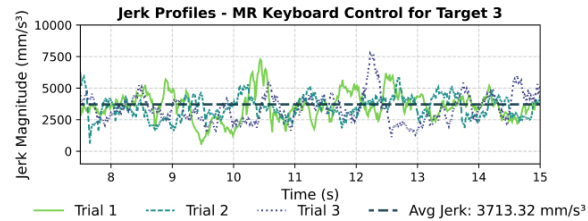
bii)



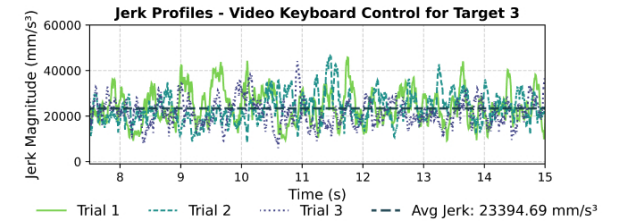
biii)



ci)



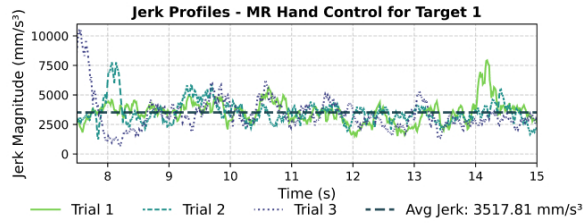
cii)



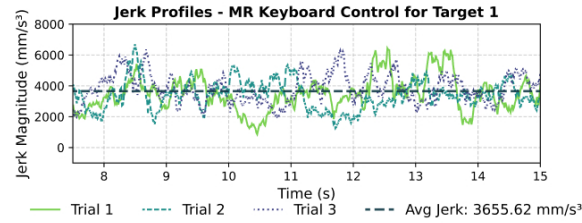
ciii)

Figure A.23: Student 3 jerk profile plots.

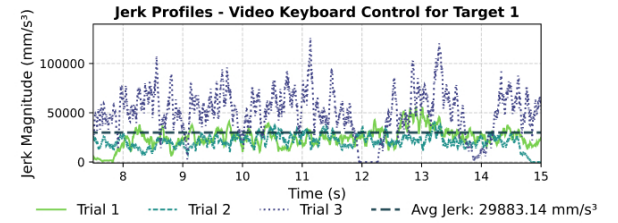
Student 4



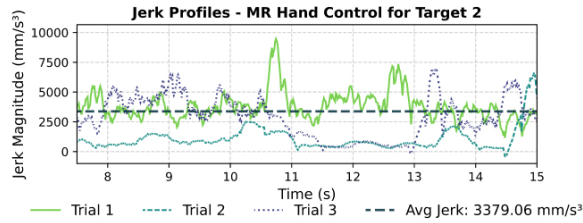
ai)



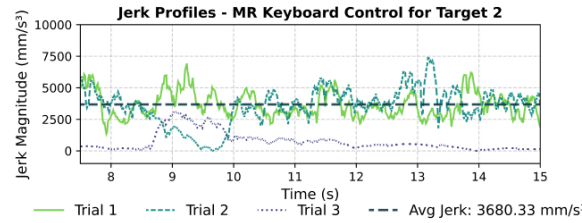
aii)



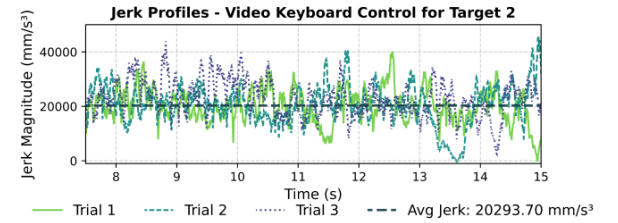
aiii)



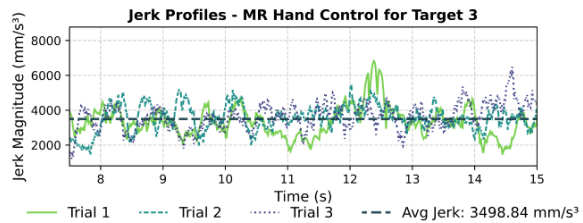
bi)



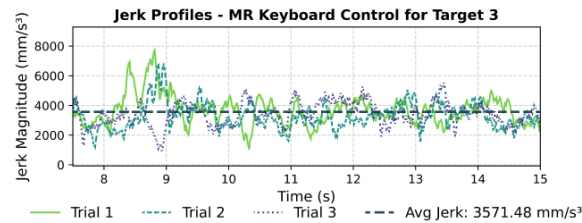
bii)



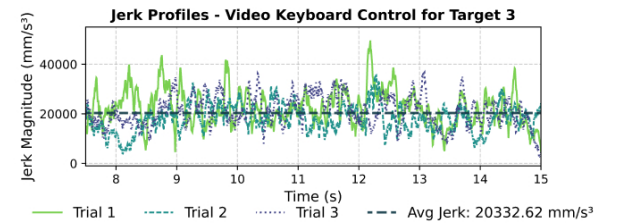
biii)



ci)



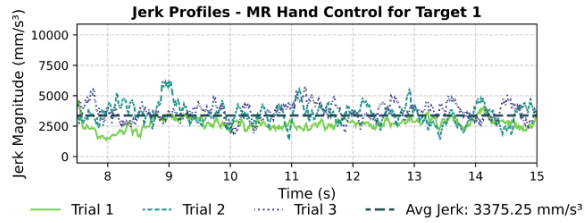
cii)



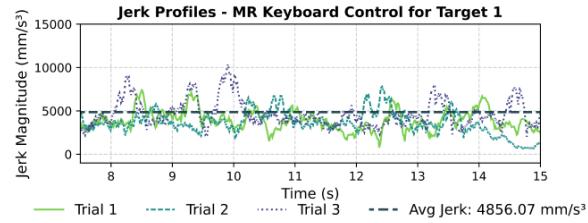
ciii)

Figure A.24: Student 4 jerk profile plots.

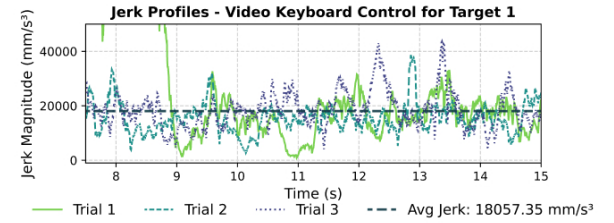
Student 5



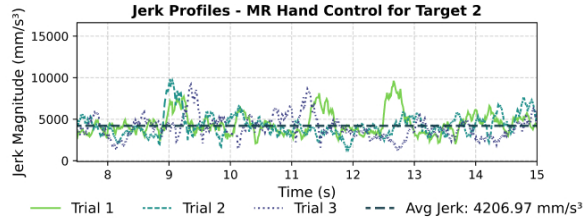
ai)



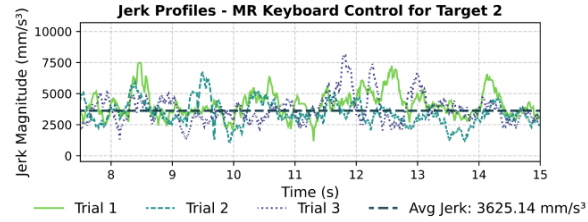
aii)



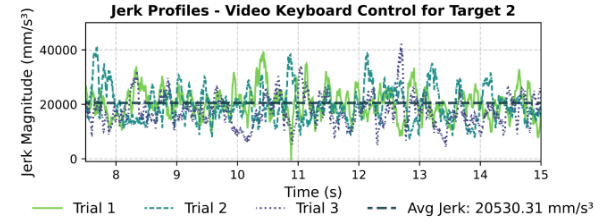
aiii)



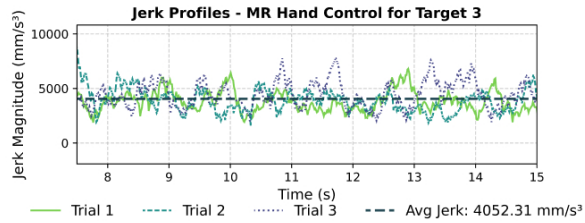
bi)



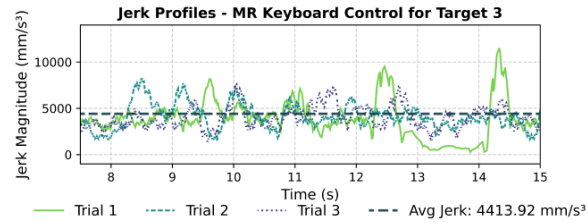
bii)



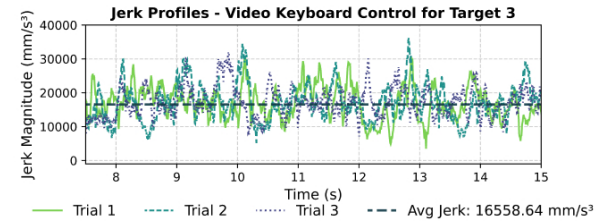
biii)



ci)



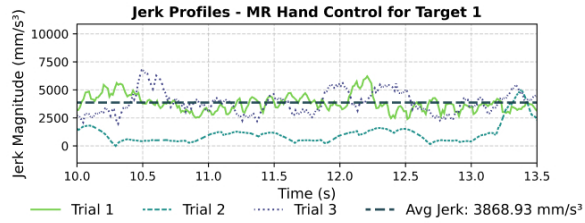
cii)



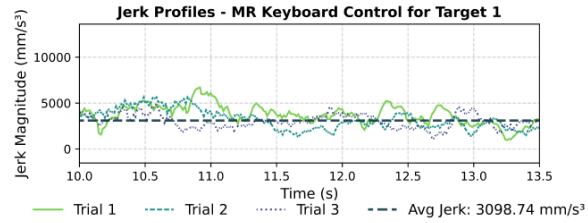
ciii)

Figure A.25: Student 5 jerk profile plots.

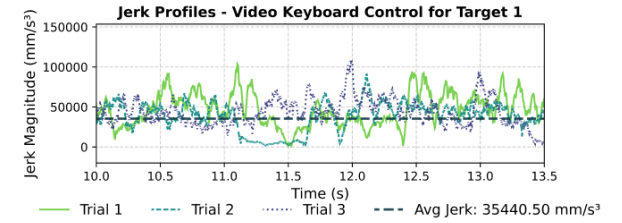
Student 6



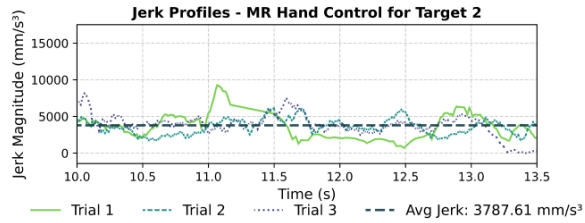
ai)



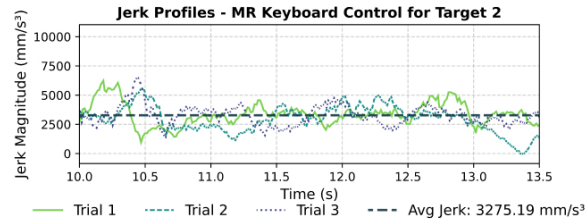
aii)



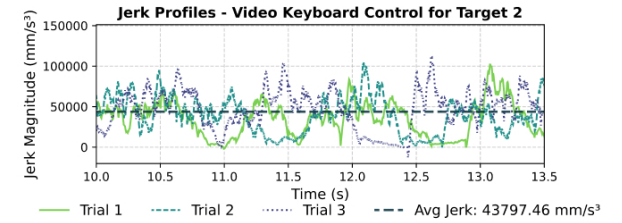
aiii)



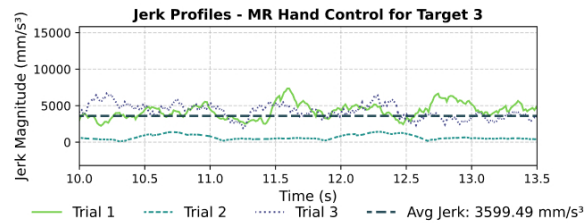
bi)



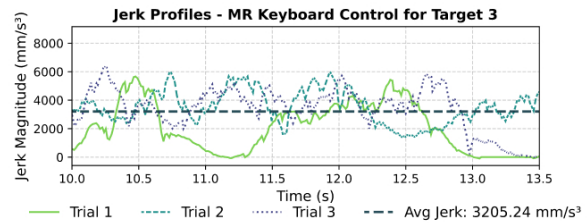
bii)



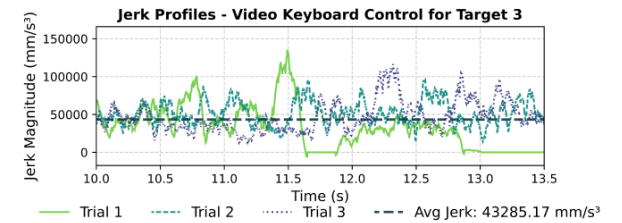
biii)



ci)



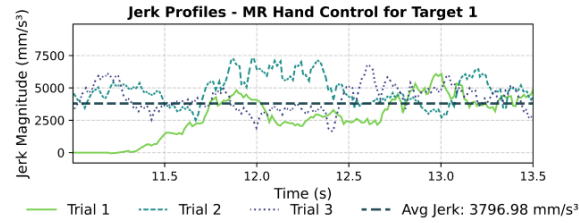
cii)



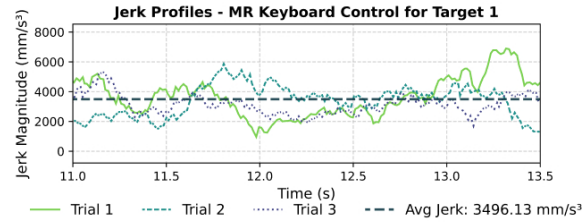
ciii)

Figure A.26: Student 6 jerk profile plots.

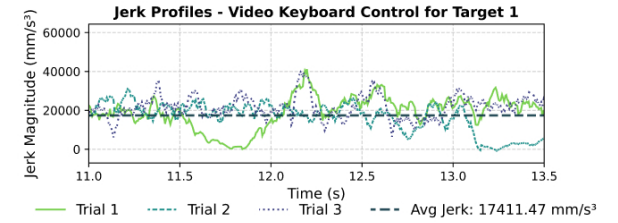
Student 7



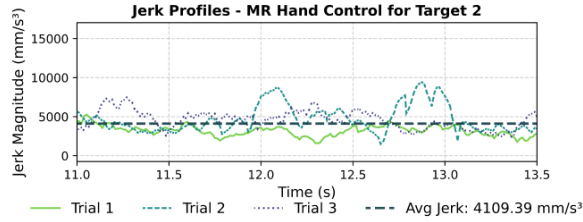
ai)



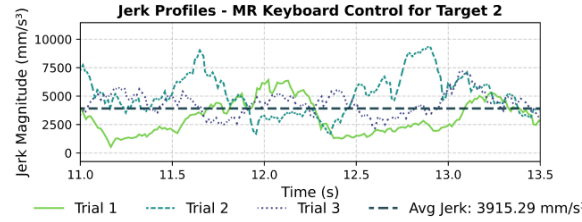
aii)



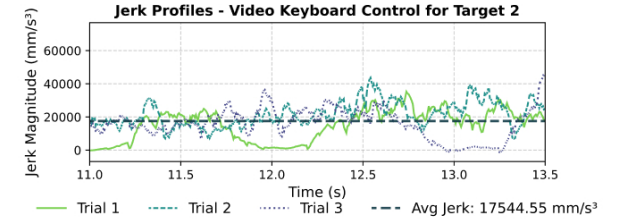
aiii)



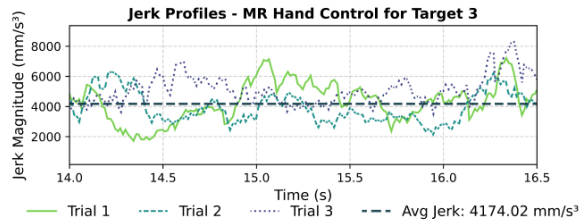
bi)



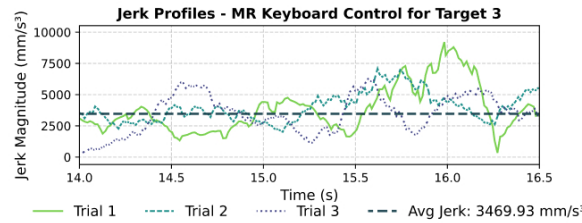
bii)



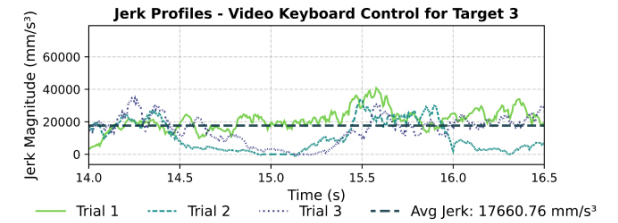
biii)



ci)



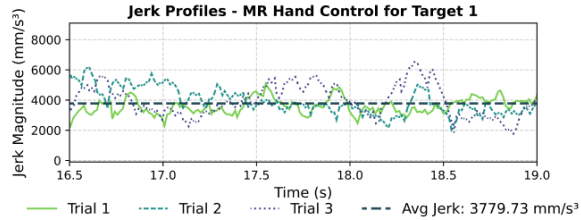
cii)



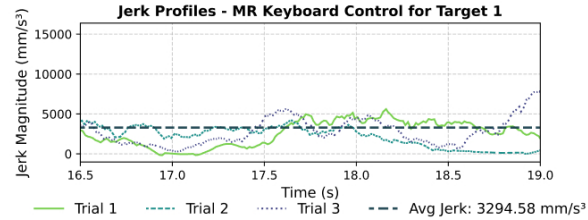
ciii)

Figure A.27: Student 7 jerk profile plots.

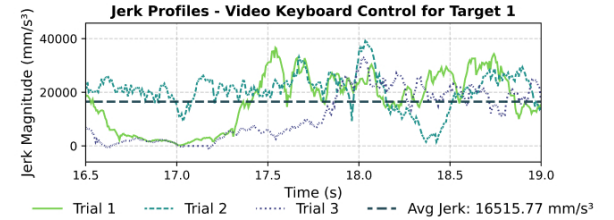
Student 8



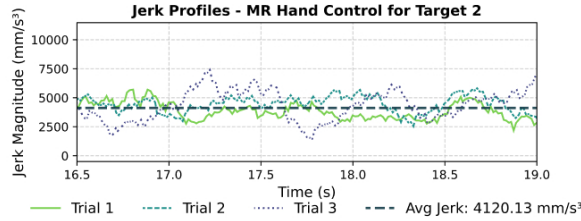
ai)



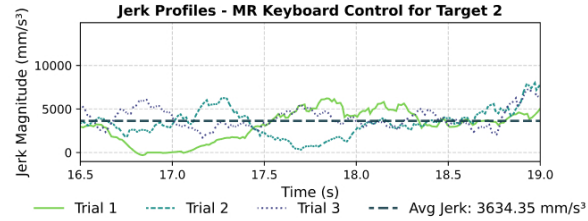
aii)



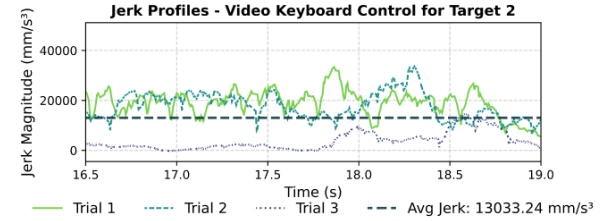
aiii)



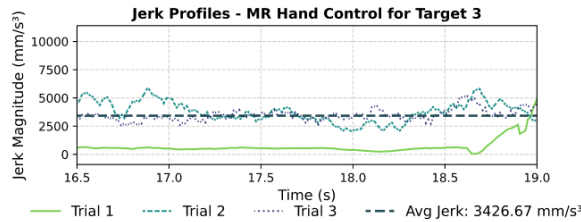
bi)



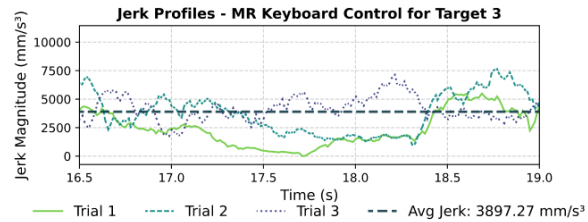
bii)



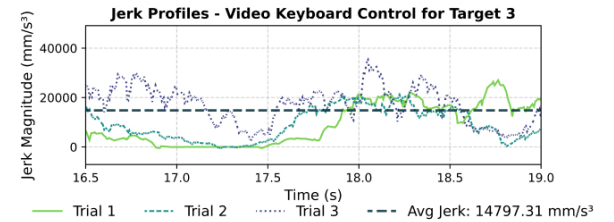
biii)



ci)



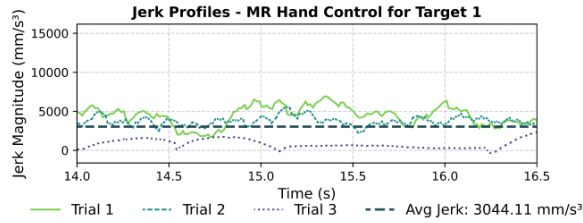
cii)



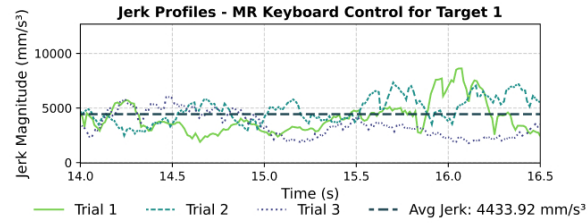
ciii)

Figure A.28: Student 8 jerk profile plots.

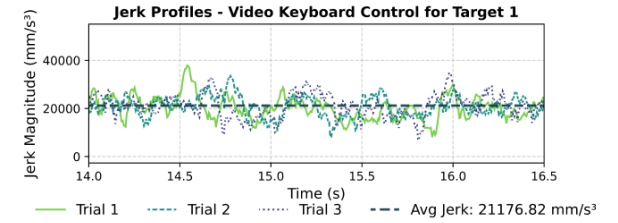
Student 9



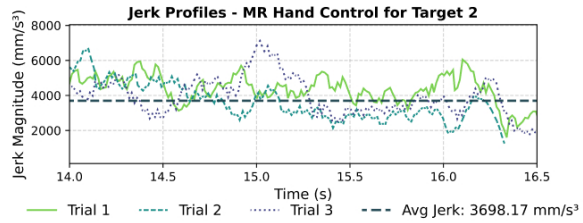
ai)



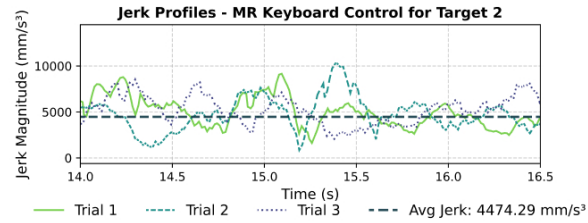
aii)



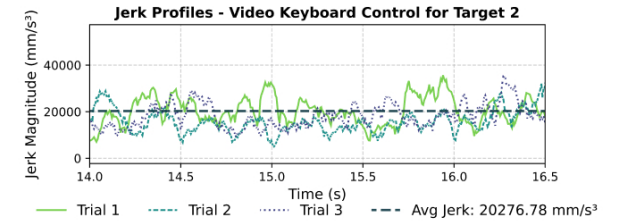
aiii)



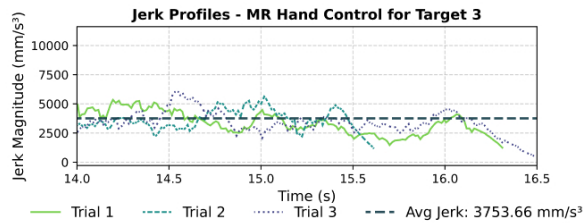
bi)



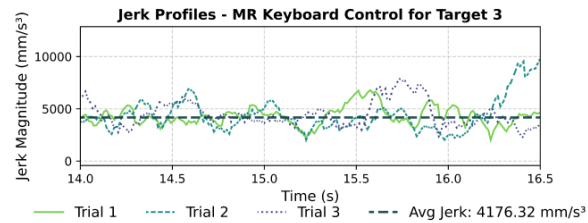
bii)



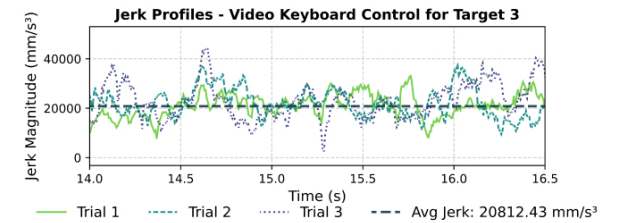
biii)



ci)



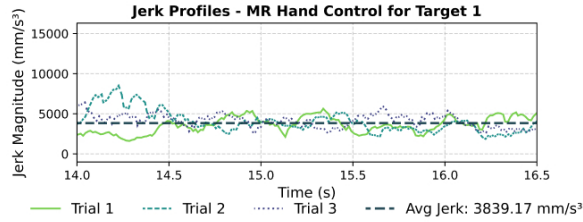
cii)



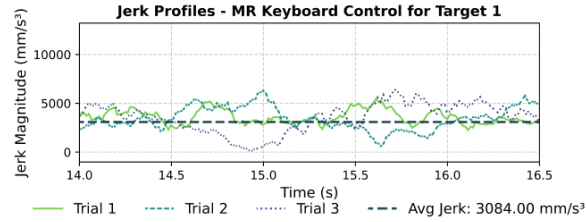
ciii)

Figure A.29: Student 9 jerk profile plots.

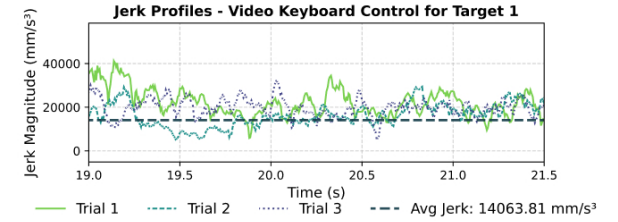
Student 10



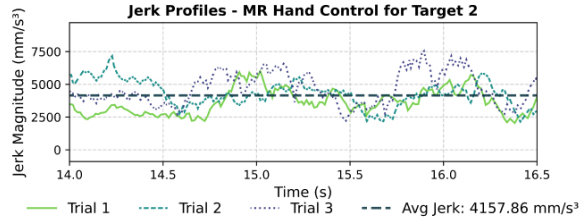
ai)



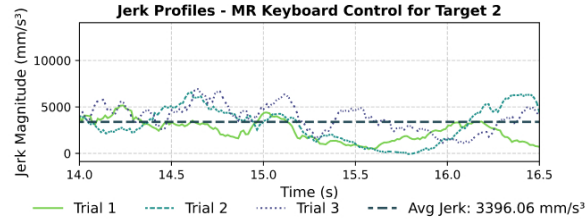
aii)



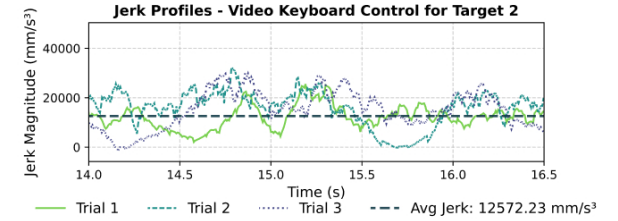
aiii)



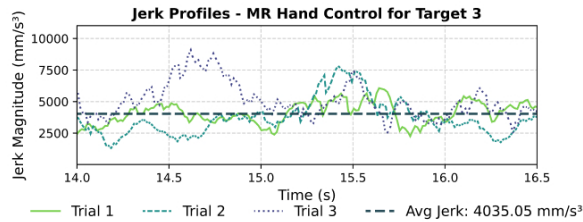
bi)



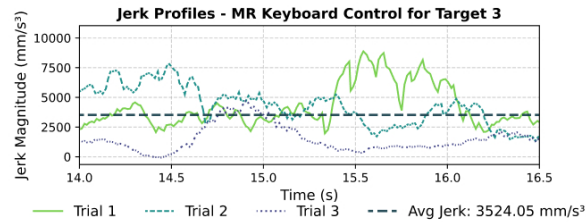
bii)



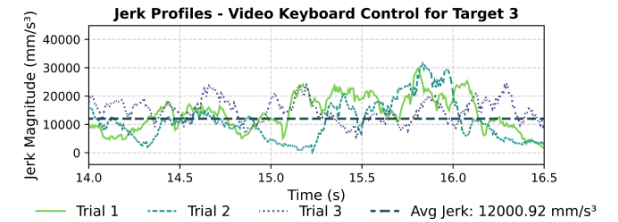
biii)



ci)



cii)



ciii)

Figure A.30: Student 10 jerk profile plots.

Appendix B: User Interface

Segment Code - C#

B.1 User Input Classes

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4 using UnityEngine.SceneManagement;
5
6 public class compute_hand_control_v2 : MonoBehaviour
7 {
8     // Create an instance of the joint_storage class
9     [SerializeField] private joint_storage jointStorage;
10
11     // Store joint-joint distance values
12     [SerializeField] private float dist4_8_rh = 0.0f;
13     [SerializeField] private float dist8_0_rh = 0.0f;
14     [SerializeField] private float dist12_0_rh = 0.0f;
15     [SerializeField] private float dist16_0_rh = 0.0f;
16     [SerializeField] private float dist20_0_rh = 0.0f;
17
18     // Define GameObjects and Transforms
19     [SerializeField] private Transform anchorTransform;
20     [SerializeField] private GameObject motionFlagSphere;
21     [SerializeField] private GameObject gripperFlagSphere;
22
23     // Define materials
24     [SerializeField] private Material greenMat;
25     [SerializeField] private Material redMat;
26     [SerializeField] private Material blueMat;
27     [SerializeField] private Material purpleMat;
28
29     // Define hand gesture thresholds
30     private float dist4_8_rh_pos_thresh = 0.012f;
31     private float dist12_0_rh_pos_thresh = 0.09f;
32     private float dist16_0_rh_pos_thresh = 0.09f;
33     private float dist20_0_rh_pos_thresh = 0.09f;
```



```

34 private float dist8_0_rh_grip_thresh = 0.09f;
35 private float dist12_0_rh_grip_thresh = 0.09f;
36 private float dist16_0_rh_grip_thresh = 0.09f;
37 private float dist20_0_rh_grip_thresh = 0.09f;
38
39 // Define variables for calculations
40 public float[] handControlData = new float[9]; //
    Transmission data array to ROS server
41 private int gripCount = 0; // Number of pinches detected
42 private float gripTimer = 0.0f; // Timer to detect
    consecutive pinches (countdown begins)
43 private float gripWindow = 1.0f; // Fixed time window to
    register multiple pinches
44 private bool isGripping = false; // Flag to check if
    pinching is occurring
45
46 void Start()
47 {
48     // Initialize values of each index to 0f to ensure
    robot arm safety
49     for (int i = 0; i < handControlData.Length; i++)
50     {
51         handControlData[i] = 0.0f;
52     }
53 }
54
55 // Update is called once per frame
56 void Update()
57 {
58     // Compute the distances to joint combinations
59     dist4_8_rh = Vector3.Distance(jointStorage.
        rightHandJoints[4].position, jointStorage.
        rightHandJoints[8].position);
60     dist8_0_rh = Vector3.Distance(jointStorage.
        rightHandJoints[8].position, jointStorage.
        rightHandJoints[0].position);
61     dist12_0_rh = Vector3.Distance(jointStorage.
        rightHandJoints[12].position, jointStorage.
        rightHandJoints[0].position);
62     dist16_0_rh = Vector3.Distance(jointStorage.
        rightHandJoints[16].position, jointStorage.
        rightHandJoints[0].position);
63     dist20_0_rh = Vector3.Distance(jointStorage.
        rightHandJoints[20].position, jointStorage.
        rightHandJoints[0].position);
64
65     // Compute motion gesture detection

```

```

66     if (dist4_8_rh < dist4_8_rh_pos_thresh) // When pinch
67     {
68         motionFlagSphere.GetComponent<MeshRenderer>().
            material = blueMat;
69
70         if ((dist12_0_rh < dist12_0_rh_pos_thresh) && (
            dist16_0_rh < dist16_0_rh_pos_thresh) && (
            dist20_0_rh < dist20_0_rh_pos_thresh)) // When
            grip with pinch
71         {
72             Vector3 directionVector = Vector3.Normalize(
                jointStorage.rightHandJoints[8].position -
                anchorTransform.position); // Compute the
                normalized direction vector
73
74             for (int i = 0; i < 3; i++)
75             {
76                 handControlData[i] = directionVector[i];
                    // Set the direction vector to the
                    global hand control data for
                    transmission (setttting raw data ->
                    Coordinate transformations handled at
                    RDS node)
77             }
78
79             motionFlagSphere.GetComponent<MeshRenderer>().
                material = greenMat;
80         }
81         else // When pinch but no grip
82         {
83             for (int i = 0; i < 3; i++)
84             {
85                 handControlData[i] = 0.0f; // Zero the
                    direction vector
86             }
87         }
88     }
89     else // When no pinch and no grip
90     {
91         motionFlagSphere.GetComponent<MeshRenderer>().
            material = redMat;
92
93         for (int i = 0; i < 3; i++)
94         {
95             handControlData[i] = 0.0f; // Zero the
                direction vector
96         }

```



```

97     }
98
99     // Detect gripper gesture
100    DetectGrip();
101    CountGrips();
102
103    if (Input.GetKey(KeyCode.R))
104    {
105        SceneManager.LoadScene("3rd Person");
106    }
107 }
108
109 // Function: Detects when a gripper gesture has been
invoked
110 void DetectGrip()
111 {
112     // Check whether the thresholds have been satisfied
113     if ((dist8_0_rh < dist8_0_rh_grip_thresh) && (
114         dist12_0_rh < dist12_0_rh_grip_thresh) && (
115         dist16_0_rh < dist16_0_rh_grip_thresh) && (
116         dist20_0_rh < dist20_0_rh_grip_thresh))
117     {
118         if (isGripping == false) // Ensure grip count is
not incremented when gesture is held without
release
119         {
120             isGripping = true;
121             gripCount++;
122             gripTimer = gripWindow; // Reset the timer
when a pinch has been recognized
123         }
124     }
125     else
126     {
127         isGripping = false; // Thresholds were not met
128     }
129 }
130
131 // Function: Count the number of pinches to classify the
gripper state
132 void CountGrips()
133 {
134     // Ensure gripper timer is non-negative
135     if (gripTimer > 0)
136     {
137         gripTimer -= Time.deltaTime; // Count down the
timer

```

```

135
136      // When the timer runs out, process the collected
      counts
137      if (gripTimer <= 0)
138      {
139          if (gripCount == 2) // Open gripper state
140          {
141              handControlData[6] = 1.0f;
142              gripperFlagSphere.GetComponent<
                  MeshRenderer>().material = purpleMat;
143          }
144          else if (gripCount == 3) // Close gripper
              state
145          {
146              handControlData[6] = 0.0f;
147              gripperFlagSphere.GetComponent<
                  MeshRenderer>().material = redMat;
148          }
149
150          gripCount = 0; // Reset the grip count and
              time for next detection window
151      }
152  }
153 }
154 }

```

Listing B.1: Script that utilizes hand tracking data from OpenXR to detect gestures

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class global_robot_ctrl : MonoBehaviour
6  {
7      // Define user parameters
8      [Header("Robot Model Setup")]
9      public Transform EndEffectorTransorm;
10     public GameObject Target;
11     [SerializeField] private Transform RootComponent;
12     public global_hand_ctrl GlobalHandCtrl;
13
14     // Define internal parameters
15     [Header("Target Pose and States")]
16     public Vector3 TargetEndEffectorPos;
17     public Quaternion TargetEndEffectorRot;
18     public Vector3 CurrentEndEffectorPos;
19     public bool IsControlActive = false; // Set to true when
        the this script is executed

```

```

20
21 // Define user input state
22 [Header("Control Type States")]
23 [SerializeField] private bool ComputerBasedControl = false
    ;
24
25 private void Start()
26 {
27     // Initialize components
28     TargetEndEffectorPos = Target.transform.position;
29     TargetEndEffectorRot = Target.transform.rotation;
30
31 }
32
33 void Update()
34 {
35     // Validate components
36     if (!(Target && EndEffectorTransorm && GlobalHandCtrl)
37         )
38     {
39         Debug.LogWarning("[global_robot_ctrl]: One or more
40             objects not been initialized");
41         return;
42     }
43
44     if (!(ComputerBasedControl))
45     {
46         // Compute only position control
47         if (GlobalHandCtrl.RIGHT_PINCH)
48         {
49             TargetEndEffectorPos = Target.transform.
50                 position - RootComponent.position; //
51                 Compute and update the target position (
52                 robot_base_coord)
53             IsControlActive = true; // Set control state
54                 to high
55             return;
56         }
57         else
58         {
59             IsControlActive = false;
60         }
61
62         // Compute only orientation control
63         if (GlobalHandCtrl.RIGHT_PINCH_GRIP)
64         {
65             TargetEndEffectorRot = Target.transform.

```

```

        rotation; // Compute and update the target
        orientation
60     IsControlActive = true;
61     return;
62 }
63 else
64 {
65     IsControlActive = false;
66 }
67
68 // Compute position & orientation control
69 if (GlobalHandCtrl.RIGHT_GRIP)
70 {
71     TargetEndEffectorPos = Target.transform.
        position - RootComponent.position;
72     TargetEndEffectorRot = Target.transform.
        rotation;
73     IsControlActive = true;
74     return;
75 }
76 else
77 {
78     IsControlActive = false;
79 }
80 }
81 else
82 {
83     if (EndEffectorTransorm && Target &&
        GlobalHandCtrl)
84     {
85         // Calculate and set
86         TargetEndEffectorPos = Target.transform.
            position - RootComponent.position; //
            Computes the target position (
            robot_base_coord)
87         CurrentEndEffectorPos.x = EndEffectorTransorm.
            position.x; // Computes the current
            position (robot_base_coord)
88         CurrentEndEffectorPos.y = EndEffectorTransorm.
            position.y;
89         CurrentEndEffectorPos.z = EndEffectorTransorm.
            position.z;
90
91         // Get the quternion rotation of the target
92         TargetEndEffectorRot = Target.transform.
            rotation;
93

```

```

94         IsControlActive = true; // Set control state
           to high
95     }
96     else
97     {
98         // Log a warning and set the control state to
           low
99         Debug.LogWarning("global_robot_ctrl: The end
           effector transform or target object is not
           attached");
100         IsControlActive = false;
101     }
102 }
103
104 }
105 }

```

Listing B.2: Class that computes the end effector pose vectors in the third-person digital twin control mode

B.2 Communication Classes

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  // Required Imports for system
6  using System.Net;
7  using System.Net.Sockets;
8  using System.Text;
9  using System;
10 using System.Threading.Tasks; // Import System.Threading.Tasks
    for async operations
11
12
13 public class udp_client : MonoBehaviour
14 {
15     // Define server IP and socket number
16     public string serverIP = "130.63.230.126";
17     public int serverPort = 2222;
18
19     // Define string object to store message to transmit
20     public string messageTX = "INITIALIZE COMMS";
21
22     // Define a string object to store received message
23     public string messageRX = "INITIALIZE COMMS";

```

```

24
25 // Create UDP client object and IPEndPoint object
26 private UdpClient udpClient;
27 private IPEndPoint remoteEndPoint;
28
29 // Inherit the keyboard_input class and create an instance
    of it
30 public keyboard_input keyboardInput;
31
32 // Construct and instance of the hand_tracking class
33 //public compute_hand_control computeHandControl;
34 public compute_hand_control_v2 computeHandControlV2;
35
36 // Reference data_handler class
37 public data_handler dataHandler;
38
39 // NEW CODE: Reference the global_robot_ctrl class
40 public global_robot_ctrl GlobalRobotCtrl;
41
42 // Define user parameters
43 public float[] robot_pose_data = new float[6]; // [x, y, z
    , phi (rx), theta (ry), psi (rz)] Cartesian robot
    coordinates
44 public string poseDataCode = " POSE,";
45
46 void Start()
47 {
48     // Create an instance of the UDP client
49     udpClient = new UdpClient();
50
51     // Define an IP Address object and attach the server ip
        string object, bind the IP Address object and the
        socket number to create a IP End Point object
52     IPAddress ipAddress = IPAddress.Parse(serverIP);
53     remoteEndPoint = new IPEndPoint(ipAddress, serverPort);
54
55     // Begin the communication and receive loop
        asynchronously
56     _ = CommunicationLoopAsync();
57     _ = ReceiveLoopAsync();
58
59 }
60
61 // Define method to transmit message to UDP server
    asynchronously
62 async Task CommunicationLoopAsync()
63 {

```

```

64     try
65     {
66         while (true)
67         {
68             // Priority input system: if there is no data
coming from the keyboard, prioritize the
hand tracking system as the input system,
else (there is keyboard input), prioritize
the keyboard input system
69             if ((keyboardInput.keydown == "INPUT SYSTEM 0"
70                 ) && (GlobalRobotCtrl) && (GlobalRobotCtrl.
71                 IsControlActive))
72             {
73                 messageTX = dataHandler.
74                     convertGlobalRobotControl(
75                         GlobalRobotCtrl.TargetEndEffectorPos,
76                         GlobalRobotCtrl.TargetEndEffectorRot);
77             }
78             else if (keyboardInput.keydown == "INPUT
79                 SYSTEM 0")
80             {
81                 // Retrieve input data from Hand Tracking
System
82                 messageTX = dataHandler.
83                     convertHandTrackedData(
84                         computeHandControlV2.handControlData);
85             }
86             else
87             {
88                 // Retrieve input data from the Keyboard
System
89                 messageTX = keyboardInput.keydown; //
90                     Already a single string value
91             }
92
93             // Convert transmit message into bytes through
UTF-8 encoding
94             byte[] dataTX = Encoding.UTF8.GetBytes(
95                 messageTX);
96
97             // Transmit bytes asynchronously
98             await udpClient.SendAsync(dataTX, dataTX.
99                 Length, remoteEndPoint);
100
101             // Time between transmission of next message
102             await Task.Delay(TimeSpan.FromSeconds(0.01f));
103         }
104     }

```

```

93     }
94     catch (Exception e)
95     {
96         Debug.LogError("Error in data transmission: " + e.
97             Message);
98     }
99 }
100 // Define method to receive message from UDP server
101 // asynchronously
102 async Task ReceiveLoopAsync()
103 {
104     try
105     {
106         while (true)
107         {
108             // Receive bytes asynchronously
109             UdpReceiveResult bytesRX = await udpClient.
110                 ReceiveAsync();
111
112             // Convert RX bytes into string
113             messageRX = Encoding.UTF8.GetString(bytesRX.
114                 Buffer);
115
116             // Time between reception of next message
117             await Task.Delay(TimeSpan.FromSeconds(0.01f));
118         }
119     }
120     catch (Exception e)
121     {
122         Debug.LogError("Error in data reception: " + e.
123             Message);
124     }
125 }
126 // Define method to close the UDP client system safely
127 // when program stops
128 private void OnDestroy()
129 {
130     // Close UDP client when UdpClient object is destroyed
131     // (when program is finished executing)
132     if (udpClient != null)
133     {
134         udpClient.Close();
135     }
136 }

```


Listing B.3: UDP client script that connects, encodes, transmits, and receives data to/from the robot arm's SBC.

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class data_handler : MonoBehaviour
6 {
7     // Define data codes
8     public string htcDataCode = "HTC";
9     public string grcDataCode = "GRC";
10
11     // Define method to handle the hand tracking data and
12     convert to a transmitting String message
13     // OUT FORMAT: "X.XXX, X.XXX, X,XXX ...."
14     public string convertHandTrackedData(float[] values)
15     {
16         string result = htcDataCode + "," + string.Join(",",
17             values);
18         return result;
19     }
20
21     // Define method to handle the received robot joint angle
22     data into a float array of length 6
23     // OUT FORMAT: [X.Xf, X.Xf, X.Xf, X.Xf, X.Xf, X.Xf]
24     public float[] convertRobotJointAngleData(string messageRX
25         )
26     {
27         // Convert input String into a array of Strings
28         through splitting using a csv format
29         string[] rxMsgStringArray = messageRX.Split(',');
30
31         // Define a float array of length 6 -> implicitly
32         define the length
33         float[] jointAnglesArray = new float[rxMsgStringArray.
34             Length];
35
36         // Convert each string value into a floating point
37         value and populate the float array
38         for (int i = 0; i < rxMsgStringArray.Length; i++)
39         {
40             // Cast string to float
41             jointAnglesArray[i] = float.Parse(rxMsgStringArray
42                 [i].Trim());
43         }
44     }
45 }
```

```

35         // Return the resulting robot joint angles array
36         return jointAnglesArray;
37     }
38
39
40     // FUNCTION: Inputs a vector3 and outputs a string of
41     // format [GRC, X.XXX, X.XXX, X.XXX]
42     public string convertGlobalRobotControl(Vector3
43         TargetPosition, Quaternion TargetRotation)
44     {
45         string result = grcDataCode + "," + TargetPosition[0]
46             + "," + TargetPosition[1] + "," + TargetPosition
47             [2];
48         result = result + "," + TargetRotation.x + "," +
49             TargetRotation.y + "," + TargetRotation.z + "," +
50             TargetRotation.w;
51         return result;
52     }
53
54     // FUNCTION: Converts the received robot pose data into a
55     // list of 6 floating point values
56     public float[] convertRobotPoseData(string
57         RobotPoseDataString)
58     {
59         float[] RobotPoseDataFloatArray = new float[6]; //
60             Define the final array to return
61         string[] RobotPoseDataStrArray = RobotPoseDataString.
62             Split(','); // Seperate the string by commas and
63             create a string array
64         // Iterate through each string, convert each value to
65         // a float, and store in the float array
66         for (int i = 0; i < RobotPoseDataFloatArray.Length; i
67             ++)
68         {
69             RobotPoseDataFloatArray[i] = float.Parse(
70                 RobotPoseDataStrArray[i+1].Trim()); // Trim
71                 each string value and cast into a float value
72         }
73         return RobotPoseDataFloatArray; // Return final list
74             to the user
75     }
76 }

```

Listing B.4: Class designed to encode and decode data based on the communication structures defined for various control methods.

B.3 Data Acquisition and Rendering Classes

```
1 // Default libraries
2 using System.Collections;
3 using System.Collections.Generic;
4 using UnityEngine;
5
6 // Class specific libraries
7 using sl; // StereoLabs
8
9 public class point_extraction : MonoBehaviour
10 {
11     // Define details panel objects
12     [SerializeField] private ZEDManager ZEDManager;
13     private sl.float4 PCValue;
14     [SerializeField] private float MaxDistanceThresh = 1.5f;
15     // Units of meters
16     [SerializeField] private float MinDistanceThresh = 0.0f;
17     // Units of meters
18     public int MaxPoints = 5000; // Maximum number of points
19     // to be plotted
20     public int Stride = 3; // Number of pixels skipped in
21     // search algorithm
22     public List<Vector3> PointPositions = new List<Vector3>();
23     // List to store the positions of the point cloud
24     public List<Color> PointColours = new List<Color>(); //
25     // List to store the corresponding colour values for each
26     // point
27     [SerializeField] private bool Logger = false;
28     private Dictionary<Vector3Int, Color> VoxelMap = new
29     Dictionary<Vector3Int, Color>(); // Define a map to
30     // store the voxel values
31     public float VoxelResolution = 0.01f; // Define the size
32     // of 1 cube in the voxel map - default 1cm spatial
33     // resolution
34     public bool HeatMap = false;
35
36     // Define ZED objects
37     private ZEDCamera ZEDCamera;
38     public sl.ZEDMat PointCloud = new sl.ZEDMat();
39
40     void Start()
41     {
42         // Initialize objects
43         ZEDCamera = ZEDManager.zedCamera;
44         uint mWidth = (uint)ZEDCamera.ImageWidth;
45         uint mHeight = (uint)ZEDCamera.ImageHeight;
```

```

35     PointCloud.Create(mWidth, mHeight, ZEDMat.MAT_TYPE.
        MAT_32F_C4, ZEDMat.MEM.MEM_CPU); // Create the
        point cloud
36     Debug.Log("Init res: " + ZEDCamera.ImageWidth + "x" +
        ZEDCamera.ImageHeight);
37
38 }
39
40 void Update()
41 {
42     // Retrieve the latest point cloud data and store it
        to the point cloud materials
43     ZEDCamera.RetrieveMeasure(PointCloud, MEASURE.XYZRGBA,
        ZEDMat.MEM.MEM_CPU, new sl.Resolution((uint)
        ZEDCamera.ImageWidth, (uint)ZEDCamera.ImageHeight))
        ; // Retrieve the point cloud map for this frame
44
45     // Clear memory of lists to update the point cloud per
        frame
46     PointPositions.Clear();
47     PointColours.Clear();
48
49     // Iterate through pcd material and append valid
        points to list
50     for (int y = 0; y < ZEDCamera.ImageHeight; y += Stride
        )
51     {
52         for (int x = 0; x < ZEDCamera.ImageWidth; x +=
        Stride)
53         {
54             if (PointPositions.Count >= MaxPoints) break;
                // Break out of the inner loop when list is
                full
55
56             // Retrieve the values from the point cloud
                material
57             PointCloud.GetValue(x, y, out PCValue, ZEDMat.
                MEM.MEM_CPU); // Extract the XYZRGBA value
                for this pixel
58             Vector3 Position = new Vector3(PCValue.r,
                PCValue.g, PCValue.b);
59
60             if (!(HeatMap))
61             {
62                 float PackedColour = PCValue.a;
63                 uint packed = System.BitConverter.ToUInt32
                    (System.BitConverter.GetBytes(

```

```

64         PackedColour), 0);
65
66         // Extract individual color bytes using
67         bit masking
68         byte r = (byte)((packed >> 0) & 0xFF); //
69         Red is in the lowest byte
70         byte g = (byte)((packed >> 8) & 0xFF); //
71         Green is next
72         byte b = (byte)((packed >> 16) & 0xFF); //
73         Blue is third
74         byte a = (byte)((packed >> 24) & 0xFF); //
75         Alpha is highest byte
76         Color Colour = new Color(r / 255f, g / 255
77         f, b / 255f, a / 255f); // Convert to
78         Unity Color (0 1 range)
79
80         // Skip to next iteration of the loop the
81         data is not valid
82         float Range = Position.magnitude;
83         if (!(Range < MaxDistanceThresh) && (
84         Range > MinDistanceThresh)))
85         {
86             continue;
87         }
88
89         // Append the values to the lists
90         PointPositions.Add(Position);
91         PointColours.Add(Colour);
92     }
93     else
94     {
95         float Range = Position.magnitude;
96
97         // Skip to next iteration if invalid
98         if (!(Range < MaxDistanceThresh) && (
99         Range > MinDistanceThresh)))
100         {
101             continue;
102         }
103
104         // Normalize the range to [0, 1] for HSV
105         mapping
106         float normalizedDepth = Mathf.InverseLerp(
107         MinDistanceThresh, MaxDistanceThresh,
108         Range);

```

```

96         // Use a color map where 0 = red, 1 = blue
97         (HSV: hue goes from 0 to ~0.66)
98         float hue = Mathf.Lerp(0.0f, 0.66f,
99             normalizedDepth);
100         Color Colour = Color.HSVToRGB(hue, 1f, 1f)
101             ;
102
103         PointPositions.Add(Position);
104         PointColours.Add(Colour);
105     }
106
107     // Convert position vector to a Voxel key in
108     the map
109     Vector3Int VoxelKey = new Vector3Int(
110         Mathf.RoundToInt(Position.x /
111             VoxelResolution),
112         Mathf.RoundToInt(Position.y /
113             VoxelResolution),
114         Mathf.RoundToInt(Position.z /
115             VoxelResolution)
116     );
117     // Add the KEY to the map with its
118     correspdonding Colour VALUE pair
119     if (!(VoxelMap.ContainsKey(VoxelKey)))
120     {
121         VoxelMap[VoxelKey] = Colour;
122     }
123 }
124
125 // FUNCTION: Expose the map to other classes
126 public Dictionary<Vector3Int, Color> GetVoxelMap()
127 {
128     return VoxelMap;
129 }

```

Listing B.5: Interfaces with the ZEDM camera and extracts RGB-D values over a fixed number of pixels and stores them into memory.

```

1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 // Ensure particle system is attached to the same Game Object
as this script

```

```

6 [RequireComponent(typeof(ParticleSystem))]
7
8 public class voxel_renderer : MonoBehaviour
9 {
10     // Define user parameters
11     ParticleSystem System;
12     ParticleSystem.Particle[] Voxels;
13     bool VoxelsUpdated = false;
14     public float VoxelScale = 0.001f;
15     public float Scale = 1f;
16     public point_extraction PointExtraction;
17
18
19     private void Start()
20     {
21         // Initialize components
22         System = GetComponent<ParticleSystem>();
23     }
24
25
26     private void Update()
27     {
28         // Update the particle system
29         if (VoxelsUpdated)
30         {
31             System.SetParticles(Voxels, Voxels.Length);
32             VoxelsUpdated = false;
33         }
34
35         SetVoxels();
36     }
37
38     // FUNCTION: Update the values we want to set for the
39     particle system
40     public void SetVoxels()
41     {
42         if (PointExtraction)
43         {
44             if ((PointExtraction.PointPositions.Count ==
45                 PointExtraction.PointColours.Count))
46             {
47                 // Set the Voxel particle array to the same
48                 length as the Positions list
49                 Voxels = new ParticleSystem.Particle[
50                     PointExtraction.PointPositions.Count];
51                 for (int i = 0; i < PointExtraction.

```

```

49         PointPositions.Count; i++)
50     {
51         // Set each of the values of the Voxel
52         array to the point and colours
53         Voxels[i].position = PointExtraction.
54             PointPositions[i] * Scale;
55         Voxels[i].startColor = PointExtraction.
56             PointColours[i];
57         Voxels[i].startSize = VoxelScale;
58     }
59     VoxelsUpdated = true;
60 }
61
62 Dictionary<Vector3Int, Color> VoxelMap =
63     PointExtraction.GetVoxelMap();
64 if (Voxels == null || Voxels.Length != VoxelMap.
65     Count)
66     Voxels = new ParticleSystem.Particle[VoxelMap.
67         Count];
68
69 int j = 0;
70
71 foreach (var kvp in VoxelMap)
72 {
73     Vector3 Pos = new Vector3(kvp.Key.x, kvp.Key.y
74         , kvp.Key.z) * PointExtraction.
75         VoxelResolution;
76     Color Colour = kvp.Value;
77
78     Voxels[j].position = Pos * Scale;
79     Voxels[j].startColor = Colour;
80     Voxels[j].startSize = VoxelScale;
81     j++;
82 }
83
84 VoxelsUpdated = true;
85 }
86
87 }
88 }

```

Listing B.6: Utilizes the point_extraction class data arrays, in real-time, to generate a point cloud using Unity's particle rendering system

```

1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;

```



```

4
5 // External imports
6 using System.IO;
7 using System.Linq;
8 using System.Text.RegularExpressions;
9
10 // Custom enum to select type of control
11 public enum TrialType
12 {
13     hand,
14     los,
15     rgb
16 }
17
18 public class telemetry : MonoBehaviour
19 {
20     // Define references to external classes
21     [Header("Class & Object References")]
22     [SerializeField] private udp_client UDPCClient;
23     [SerializeField] private data_handler DataHandler;
24
25     // Define user parameters
26     [Header("User Parameters")]
27     [SerializeField] private bool RecordData = false;
28     [SerializeField] private string User = "DefaultUser";
29     [SerializeField] private int TrialNum = 1;
30     [SerializeField] private TrialType Type = TrialType.hand;
31
32     // Define internal parameters
33     private float ElapsedTime;
34     private string TelemetryFilePath;
35     private string CsvFilePath;
36     private string PoseDataCode = "POSE";
37     [SerializeField] private float[] RobotPoseData = new float
38         [6];
39
40 void Start()
41 {
42     // Initialize parameters
43     ElapsedTime = 0.0f;
44     TelemetryFilePath = Path.Combine(Application.dataPath,
45         "Telemetry"); // Define path to dataset folder
46
47     // Create the folder if it does not exist
48     if (!Directory.Exists(TelemetryFilePath))
49     {
50         Directory.CreateDirectory(TelemetryFilePath);
51     }
52 }
53 }

```

```

49     }
50
51     // Define full file path
52     string FileName = $"{User}_{Type}_trial{TrialNum}.csv"
53     ;
54     CsvFilePath = Path.Combine(TelemetryFilePath, FileName
55     );
56
57     // Define header for csv file
58     if (RecordData == true)
59     {
60         string Header = "x,y,z,rx,ry,rz,deltaTime";
61         File.WriteAllText(CsvFilePath, Header + "\n");
62     }
63
64     }
65
66     // Update is called once per frame
67     void Update()
68     {
69         // Convert the raw pose data into a structured array
70         if (UDPCClient.messageRX.StartsWith(PoseDataCode))
71         {
72             RobotPoseData = DataHandler.convertRobotPoseData(
73             UDPCClient.messageRX);
74         }
75
76         // Increment the elapsed time
77         ElapsedTime += Time.deltaTime;
78
79         // Add pose and time entry to the csv file
80         if ((RecordData == true) && (RobotPoseData.Length ==
81         6))
82         {
83             string Line = string.Format("
84             {0},{1},{2},{3},{4},{5},{6}",
85             RobotPoseData[0], RobotPoseData[1],
86             RobotPoseData[2],
87             RobotPoseData[3], RobotPoseData[4],
88             RobotPoseData[5],
89             ElapsedTime.ToString("F4")); // Limit to 4
90             decimal places
91
92             File.AppendAllText(CsvFilePath, Line + "\n");
93         }
94     }
95 }

```

88
89
90

```
}
```

Listing B.7: Interfaces with the data received data from the `udp_client` class to timestamp and store the pose of the end effector to a dynamic CSV file.

Appendix C: Robot Hardware Code - ROS-Python

C.1 Communication Scripts

```
1  #!/usr/bin/env python3
2  import rclpy
3  from rclpy.node import Node
4  from std_msgs.msg import String, Float64MultiArray
5  import socket
6
7  class UdpServer(Node):
8
9      def __init__(self):
10         super().__init__("udp_server")
11
12         # Initialize UDP server parameters
13         self._rx_data = ""
14         self._tx_data = "ROS server"
15         self._buffer_size = 1024
16         self._server_port = 2222
17         self._server_ip = '130.63.213.137'
18         self._client_ip = []
19
20         # Initialize UDP server
21         self._udp_server = socket.socket(socket.AF_INET,
22                                           socket.SOCK_DGRAM)
23         self._udp_server.bind((self._server_ip, self._server_port))
24         self.get_logger().info("UDP server awaiting client
25                                connection...")
26
27         # Construct a publisher object to publish the received
28            data to the topic "raw_input_data"
29         self._publish_data = self.create_publisher(String, '
30             raw_input_data', 10)
```

```

28     # Construct a subscriber object to subscribe to robot
        arm's joint angles data from the topic "
        robot_joint_angles"
29     self._robot_joint_angles_data = self.
        create_subscription(Float64MultiArray, '
        robot_joint_angles', self.parse_joint_angle_data,
        10)
30     self._robot_pose_data = self.create_subscription(
        Float64MultiArray, 'robot_pose', self.
        parse_pose_data, 10)
31
32     # Construct a timer to execute the rx_tx_server
        callback function every millisecond
33     self.create_timer(0.01, self.rx_tx_server) # working
        at 0.001 before
34
35     # Define user parameters
36     self._pose_datacode = "POSE"
37
38     def rx_tx_server(self):
39         # Store the received data and the client ip adress and
        decode the received message
40         self._rx_data, self._client_ip = self._udp_server.
        recvfrom(self._buffer_size)
41         self._rx_data = self._rx_data.decode('utf-8')
42
43         # Debug to the console the received data
44         #self.get_logger().info(f"RX: {self._rx_data}")
45         #self.get_logger().info(f"TX: {self._tx_data}")
46
47         # Transmit a message to the client
48         self._udp_server.sendto(self._tx_data.encode('utf-8'),
        self._client_ip)
49
50         # Publish raw input data
51         msg = String()
52         msg.data = self._rx_data
53         self._publish_data.publish(msg)
54
55     # Define method to convert Float64MultiArray data into a
        string message
56     def parse_joint_angle_data(self, msg):
57         # Convert floating point values into a string value
58         joint_angles_string = [str(value) for value in msg.
        data]
59
60         # Concatenate string array into a single variable

```

```

61         self._tx_data = ','.join(joint_angles_string)
62
63         # Define a method to convert robot pose topic data into a
transmittable string
64     def parse_pose_data(self, msg):
65         pose_string = [str(value) for value in msg.data] #
Convert float values to a stringed array
66         self._tx_data = self._pose_datacode + ',' + ','.join(
            pose_string) # Set the transmit message variable to
the pose data string seperated with a comma
67         self.get_logger().info(f"TX: {self._tx_data}")
68
69
70     def main (args=None):
71         rclpy.init(args=args)
72         node = UdpServer()
73         rclpy.spin(node)
74         rclpy.shutdown()

```

Listing C.1: ROS node for setting the Raspberri Pi 4B as a UDP server to send and receive messages with the Unity instance.

C.2 Control Code

```

1  #!/usr/bin/env python3
2
3  # Objective: Parse through input keyboard data from Unity and
utilize it to move robot arm in coordinate control mode.
4  # Written by: Damith Tennakoon
5
6  # Import ROS2 libraries
7  import rclpy
8  from rclpy.node import Node
9  from std_msgs.msg import String
10 from std_msgs.msg import Float64MultiArray
11
12 # Import myCobot libraries
13 from pymycobot.mycobot import MyCobot
14 from pymycobot.genre import Angle
15 from pymycobot import PI_PORT, PI_BAUD
16
17 # Import signal libraries
18 import time
19
20 # Import computational libraries
21 import math

```

```

22 import numpy as np
23
24 # Function - compute the yaw angle required to align end
    effector with the robot's joint 0 motor
25 def yaw_axis_alignment(current_pose, offset):
26     p1_x = current_pose[0] # Parse the x,y coordinates
27     p1_y = current_pose[1]
28     beta = math.atan(-1*p1_x/p1_y) # Compute the immediate
        angle, radians
29     p2_x = p1_x - offset * math.cos(beta) # Compute the x-
        component due to the offset
30     p2_y = p1_y - offset * math.sin(beta) # Compute the y-
        component due to the offset
31     yaw_angle_degrees = math.degrees(math.atan(-p2_x/p2_y)) +
        90.0 # Compute actual immediate angle and account for
        offset angle, degrees
32     return yaw_angle_degrees
33
34 # Construct Class
35 class KeyEECtrl(Node):
36
37     def __init__(self):
38         super().__init__("key_ee_ctrl")
39         self.get_logger().info("INITIALIZING KEY END EFFECTOR
        CONTROLLER NODE")
40
41         # Initialize connection to robot arm
42         self.get_logger().info("INITIALIZING CONNECTION TO
        ROBOT ARM")
43         self._mc = MyCobot("/dev/ttyACM0", 115200) # Instance
        of the MyCobot class
44         time.sleep(1)
45         self._mc.init_gripper()
46         self._mc.set_gripper_mode(0)
47         time.sleep(1)
48         self.get_logger().info("CONNECTION ESTABLISHED")
49
50         # Initialize robot arm movements
51         self.get_logger().info("INITIALIZING ROBOT ARM JOINTS
        AND WORKSPACE")
52         self._mc.set_color(255, 0, 0)
53         time.sleep(0.5)
54         self._mc.send_coords([93, -120, 280, 180, 7, 95], 10,
        1) # Move to initialize position 2 (start pose)
        using coordinate controller method
55         time.sleep(5)
56         self._mc.set_color(255, 255, 255)

```

```

57     time.sleep(0.5)
58     self._mc.set_gripper_value(20, 50) # Nearly close
        gripper
59     time.sleep(3)
60     self._mc.set_gripper_value(100, 50) # Open Gripper
61     time.sleep(3)
62     self.get_logger().info("ROBOT ARM JOINT INITIALIZATION
        COMPLETE - STATUS [READY]")
63
64     # Create Publisher/Subscriber objects
65     self._publish_robot_pose = self.create_publisher(
        Float64MultiArray, 'robot_pose', 10) # Transmit the
        robot's pose for tracking/plotting
66     self._publish_robot_joint_angles = self.
        create_publisher(Float64MultiArray, '
        robot_joint_angles', 10) # Transmit the robot's
        joints for 3D digital twin
67     self.raw_data_subscriber = self.create_subscription(
        String, 'raw_input_data', self.store_raw_data, 10)
        # Receive the string data from User Input Segment
68
69     # Define variables for local data storage
70     self._input_key = "NONE" # Received string message of
        the keyboard input
71     self._cur_position = self._mc.get_coords() # [x, y, z,
        pitch, roll, yaw]
72     self._incr_pos = 1.0 # Position increment for EE
73     self._command_delay = 0.04 # Delay after transmitting
        motion command
74     self._move_speed = 25 # Arm movement speed in mm/s
75     self._robot_offset = 97 # Offset between the joint 0
        and joint 6 on the xy-plane, in mm.
76     self._gripper_open = 100 # The maximum extension (open
        ) value for the robot gripper
77     self._gripper_close = 20 # The close value for the
        robot gripper
78     self._gripper_speed = 50 # Speed of the gripper, in mm
        /s.
79     self._gripper_delay = 1.0 # Delay after transmitting
        gripper command
80
81     # Create/execute callback functions
82     self._move_robot_timer = self.create_timer(0.01, self.
        move_robot_arm)
83     self._tx_robot_params = self.create_timer(0.1, self.
        tx_robot_arm)
84

```



```

85     # Event Handler method - store the raw User Input Segment's
      s keyboard data, locally
86     def store_raw_data(self, msg):
87         self._input_key = msg.data
88
89     # Callback method - move end effector of robot arm using
      cartesian coordinate control fuction
90     def move_robot_arm(self):
91         # Update position vector based on input key data
92         if (self._input_key == "UpArrow"):
93             self._cur_position[1] -= self._incr_pos #
              Increment on x-axis
94             self._cur_position[5] = yaw_axis_alignment(self.
              _cur_position, self._robot_offset)
95             self._mc.send_coords(self._cur_position, self.
              _move_speed, 1) # Execute coordinate control
              command
96             time.sleep(self._command_delay) # Delay to move
              arm to position
97         elif (self._input_key == "DownArrow"):
98             self._cur_position[1] += self._incr_pos
99             self._cur_position[5] = yaw_axis_alignment(self.
              _cur_position, self._robot_offset)
100            self._mc.send_coords(self._cur_position, self.
              _move_speed, 1)
101            time.sleep(self._command_delay)
102         elif (self._input_key == "RightArrow"):
103             self._cur_position[0] -= self._incr_pos
104             self._cur_position[5] = yaw_axis_alignment(self.
              _cur_position, self._robot_offset)
105             self._mc.send_coords(self._cur_position, self.
              _move_speed, 1)
106             time.sleep(self._command_delay)
107         elif (self._input_key == "LeftArrow"):
108             self._cur_position[0] += self._incr_pos
109             self._cur_position[5] = yaw_axis_alignment(self.
              _cur_position, self._robot_offset)
110             self._mc.send_coords(self._cur_position, self.
              _move_speed, 1)
111             time.sleep(self._command_delay)
112         elif (self._input_key == "N"):
113             self._cur_position[2] += self._incr_pos
114             self._mc.send_coords(self._cur_position, self.
              _move_speed, 1)
115             time.sleep(self._command_delay)
116         elif (self._input_key == "M"):
117             self._cur_position[2] -= self._incr_pos

```

```

118         self._mc.send_coords(self._cur_position, self._
            _move_speed, 1)
119         time.sleep(self._command_delay)
120     elif (self._input_key == "Q"):
121         self._mc.set_gripper_value(self._gripper_open,
            self._gripper_speed) # Open grippper command
122         time.sleep(self._gripper_delay)
123     elif (self._input_key == "W"):
124         self._mc.set_gripper_value(self._gripper_close,
            self._gripper_speed) # Close grippper command
125         time.sleep(self._gripper_delay)
126     else:
127         self._cur_position = self._cur_position
128
129     # Callback method - retrieve the robot arms end effector
        pose and publish them to a topic
130     def tx_robot_arm(self):
131         # Define the data type and msg variable, retrieve the
            pose, and store into publisher variable
132         pose_msg = Float64MultiArray()
133         pose_msg.data = self._mc.get_coords()
134         self._publish_robot_pose.publish(pose_msg)
135
136     # Create main method for looping the ROS node
137     def main(args=None):
138         try:
139             rclpy.init(args=args)
140             node = KeyEECtrl()
141             rclpy.spin(node)
142         except Exception as main_err:
143             print(f"ERROR INITIALIZING NODE: {str(main_err)}")
144         finally:
145             if rclpy.ok():
146                 rclpy.shutdown()
147             print("NODE SHUTDOWN COMPLETE")
148
149     # Execute the main function
150     if __name__ == '__main__':
151         main()

```

Listing C.2: ROS node transmitting control commands over UART using keyboard inputs.

```

1  #!/usr/bin/env python3
2
3  # Objective: Parse through input hand tracking data from Unity
        and utilize it to move robot arm in coordinate control mode
        .

```

```

4  # Written by: Damith Tennakoon
5
6  # Import ROS2 libraries
7  import rclpy
8  from rclpy.node import Node
9  from std_msgs.msg import String
10 from std_msgs.msg import Float64MultiArray
11
12 # Import myCobot libraries
13 from pymycobot.mycobot import MyCobot
14 from pymycobot.genre import Angle
15 from pymycobot import PI_PORT, PI_BAUD
16
17 # Import signal libraries
18 import time
19
20 # Import computational libraries
21 import math
22
23 # Import computational libraries
24 import math
25 import numpy as np
26
27 # Function - compute the yaw angle required to align end
    effector eith the robot's joint 0 motor
28 def yaw_axis_alignment(current_pose, offset):
29     p1_x = current_pose[0] # Parse the x,y coordinates
30     p1_y = current_pose[1]
31     beta = math.atan(-1*p1_x/p1_y) # Compute the immediate
        angle, radians
32     p2_x = p1_x - offset * math.cos(beta) # Compute the x-
        component due to the offset
33     p2_y = p1_y - offset * math.sin(beta) # Compute the y-
        component due to the offset
34     yaw_angle_degrees = math.degrees(math.atan(-p2_x/p2_y)) +
        90.0 # Compute actual immediate angle and account for
        offset angle, degrees
35     return yaw_angle_degrees
36
37 # Construct Class
38 class HandTrackingEECtrl(Node):
39
40     def __init__(self):
41         super().__init__("hand_tracking_ee_ctrl")
42         self.get_logger().info("INITIALIZING HAND TRACKING END
            EFFECTOR CONTROLLER NODE")
43

```

```

44 # Initialize connection to robot arm
45 self.get_logger().info("INITIALIZING CONNECTION TO
    ROBOT ARM")
46 self._mc = MyCobot("/dev/ttyACM0", 115200) # Instance
    of the MyCobot class
47 time.sleep(1)
48 self._mc.init_gripper()
49 self._mc.set_gripper_mode(0)
50 time.sleep(1)
51 self.get_logger().info("CONNECTION ESTABLISHED")
52
53 # Initialize robot arm movements
54 self.get_logger().info("INITIALIZING ROBOT ARM JOINTS
    AND WORKSPACE")
55 self._mc.set_color(255, 0, 0)
56 time.sleep(0.5)
57 self._mc.send_coords([93, -120, 280, 180, 7, 95], 10,
    1) # Move to initialize position 2 (start pose)
    using coordinate controller method
58 time.sleep(5)
59 self._mc.set_color(255, 255, 255)
60 time.sleep(0.5)
61 self._mc.set_gripper_value(20, 50) # Nearly close
    gripper
62 time.sleep(3)
63 self._mc.set_gripper_value(100, 50) # Open Gripper
64 time.sleep(3)
65 self.get_logger().info("ROBOT ARM JOINT INITIALIZATION
    COMPLETE - STATUS [READY]")
66
67 # Create Publisher/Subscriber objects
68 self._publish_robot_pose = self.create_publisher(
    Float64MultiArray, 'robot_pose', 10) # Transmit the
    robot's pose for tracking/plotting
69 self._publish_robot_joint_angles = self.
    create_publisher(Float64MultiArray, '
    robot_joint_angles', 10) # Transmit the robot's
    joints for 3D digital twin
70 self.raw_data_subscriber = self.create_subscription(
    String, 'raw_input_data', self.parse_raw_data, 10)
    # Receive the string data from User Input Segment
71
72 # Define variables for local data storage
73 self._hand_control_data = [0.0, 0.0, 0.0, 0.0, 0.0,
    0.0, 0.0] # Floating point list of hand control
    data
74 self._cur_position = self._mc.get_coords() # [x, y, z,

```

```

    pitch, roll, yaw]
75 self._incr_pos = 1.0 # Position increment for EE
76 self._command_delay = 0.04 # Delay after transmitting
    motion command
77 self._move_speed = 25 # Arm movement speed in mm/s
78 self._robot_offset = 97 # Offset between the joint 0
    and joint 6 on the xy-plane, in mm.
79 self._t_unity_to_robot = np.array([
80     [-1, 0, 0],
81     [0, 0, -1],
82     [0, 1, 0]
83 ]) # Transformation matrix from Unity coordinate
    system to Robot coordinate system
84 self._prev_gripper_state = True # Initialize the
    previous gripper state to be "open"
85 self._cur_gripper_state = self._prev_gripper_state #
    Initialize the current gripper state to the
    previous - stops overlapping signals
86 self._gripper_open = 100 # The maximum extension (open
    ) value for the robot gripper
87 self._gripper_close = 20 # The close value for the
    robot gripper
88 self._gripper_speed = 50 # Speed of the gripper, in mm
    /s.
89 self._gripper_delay = 1.0 # Delay after transmitting
    gripper command
90
91 # Create and excute callback functions
92 self._move_robot_timer = self.create_timer(0.01, self.
    move_robot_arm)
93 self._tx_robot_params = self.create_timer(0.1, self.
    tx_robot_arm)
94
95 # Event Handler method - parse and store the HTC received
    data
96 def parse_raw_data(self, msg):
97     # Parse data if the received message is for HTC
98     if (msg.data[:3] == "HTC"):
99         split_string_list = msg.data.split(',') # Seperate
            the string using csv format
100         # Convert and store the data
101         for i in range(len(self._hand_control_data)):
102             self._hand_control_data[i] = float(
                split_string_list[i+1])
103         # Cast and handle the gripper control values
104         if (self._hand_control_data[6] == 1.0):
105             self._cur_gripper_state = True # Set the

```

```

106         current gripper state to "open"
107     else:
108         self._cur_gripper_state = False # Set the
109         current gripper state to "close"
110     else:
111         for i in range(len(self._hand_control_data)):
112             self._hand_control_data[i] = 0.0 # Zero the
113             data
114
115 # Callback method - move end effector of robot arm using
116 cartesian coordinate control function
117 def move_robot_arm(self):
118     # Update position vector when hand control data is non
119     -zero
120     if ((math.sqrt(self._hand_control_data[0]**2 + self.
121     _hand_control_data[1]**2 + self._hand_control_data
122     [2]**2)) > 0.0):
123         unity_coords = np.array([self._hand_control_data
124         [0], self._hand_control_data[1], self.
125         _hand_control_data[2]]) # Construct numpy array
126         of unity unit vector
127         robot_coords = np.dot(self._t_unity_to_robot,
128         unity_coords) # Perform coordinate
129         transformation - unity frame to robot frame
130         self._cur_position[0] += robot_coords[0] # Move
131         robot along its x-axis
132         self._cur_position[1] += robot_coords[1] # Move
133         robot along its y-axis
134         self._cur_position[2] += robot_coords[2] # Move
135         robot along its z-axis
136         self._cur_position[5] = yaw_axis_alignment(self.
137         _cur_position, self._robot_offset)
138         self._mc.send_coords(self._cur_position, self.
139         _move_speed, 1) # Transmit coordinate control
140         command
141         time.sleep(self._command_delay) # Delay to move
142         arm to position
143     # Update gripper state when robot arm is stationary
144     else:
145         self._cur_position = self._cur_position
146         # Update the gripper state when it's different
147         that the previous state
148         if (self._cur_gripper_state != self.
149         _prev_gripper_state):
150             if (self._cur_gripper_state == True):
151                 self._mc.set_gripper_value(self.
152                 _gripper_open, self._gripper_speed) #

```

```

131         Open gripper command
132         time.sleep(self._gripper_delay)
133     else:
134         self._mc.set_gripper_value(self.
135             _gripper_close, self._gripper_speed) #
136         Close gripper command
137         time.sleep(self._gripper_delay)
138     self._prev_gripper_state = self.
139         _cur_gripper_state # Sync the two states
140
141     # Callback method - retrieve the robot arms end effector
142     pose and publish them to a topic
143
144     def tx_robot_arm(self):
145         # Define the data type and msg variable, retrieve the
146         pose, and store into publisher variable
147         pose_msg = Float64MultiArray()
148         pose_msg.data = self._mc.get_coords()
149         self._publish_robot_pose.publish(pose_msg)
150
151     # Create main method for looping the ROS node
152     def main(args=None):
153         try:
154             rclpy.init(args=args)
155             node = HandTrackingEECtrl()
156             rclpy.spin(node)
157         except Exception as main_err:
158             print(f"ERROR INITIALIZING NODE: {str(main_err)}")
159         finally:
160             if rclpy.ok():
161                 rclpy.shutdown()
162             print("NODE SHUTDOWN COMPLETE")
163
164     # Execute the main function
165     if __name__ == '__main__':
166         main()

```

Listing C.3: ROS node for computing target pose vectors for robot arm through hand gesture control inputs.

```

1  #!/usr/bin/env python3
2  import rclpy
3  from rclpy.node import Node
4  from std_msgs.msg import String
5  from std_msgs.msg import Float64MultiArray
6  from pymycobot.mycobot import MyCobot
7  from pymycobot.genre import Angle
8  from pymycobot import PI_PORT, PI_BAUD
9  import time

```

```

10
11 # Import computational libraries
12 import math
13 import numpy as np
14 from scipy.spatial.transform import Rotation as R
15
16 # Function - compute the yaw angle required to align end
    effector with the robot's joint 0 motor
17 def yaw_axis_alignment(current_pose, offset):
18     p1_x = current_pose[0] # Parse the x,y coordinates
19     p1_y = current_pose[1]
20     beta = math.atan(-1*p1_x/p1_y) # Compute the immediate
        angle, radians
21     p2_x = p1_x - offset * math.cos(beta) # Compute the x-
        component due to the offset
22     p2_y = p1_y - offset * math.sin(beta) # Compute the y-
        component due to the offset
23     yaw_angle_degrees = math.degrees(math.atan(-p2_x/p2_y)) +
        90.0 # Compute actual immediate angle and account for
        offset angle, degrees
24     return yaw_angle_degrees
25
26 # Function - convert all angles between 0-360
27 def to_positive_angle(angle):
28     angle = angle % 360
29     if angle >= 180:
30         angle -= 360
31     return angle
32
33 # Function - position deadband
34 def is_significant_change(new_position, pre_position,
    pos_thresh = 2.0):
35     magnitude = math.sqrt((new_position[0]-pre_position[0])**2
        + (new_position[1]-pre_position[1])**2 + (new_position
        [2]-pre_position[2])**2)
36     if (magnitude >= pos_thresh):
37         return True
38     else:
39         return False
40
41 # Function - orientation deadband
42 def is_significant_rotation(new_rotation, pre_rotation,
    angle_thresh = 2.0):
43     magnitude = math.sqrt((new_rotation[0]-pre_rotation[0])**2
        + (new_rotation[1]-pre_rotation[1])**2 + (new_rotation
        [2]-pre_rotation[2])**2)
44     if (magnitude >= angle_thresh):

```



```

45         return True
46     else:
47         return False
48
49 class GlobalRobotCtrl(Node):
50
51     def __init__(self):
52         super().__init__("global_robot_ctrl")
53         self.get_logger().info("INITIALIZING GLOBAL ROBOT CTRL
54                                NODE...")
55
56         # Initialize connection to robot arm
57         self.get_logger().info("INITIALIZING CONNECTION TO
58                                ROBOT ARM")
59         self._mc = MyCobot("/dev/ttyACM0", 115200) # Instance
60             of the MyCobot class
61         time.sleep(1)
62         self.get_logger().info("CONNECTION ESTABLISHED")
63
64         # Initialize robot arm movements
65         self.get_logger().info("INITIALIZING ROBOT ARM JOINTS
66                                AND WORKSPACE")
67         self._mc.set_color(255, 0, 0)
68         time.sleep(0.5)
69         self._mc.send_coords([93, -120, 280, 180, 7, 95], 10,
70                               1) # Move to initialize position 2 (start pose)
71                                   using coordinate controller method
72         time.sleep(5)
73         self._mc.set_color(255, 255, 255)
74         time.sleep(0.5)
75         self.get_logger().info("ROBOT ARM JOINT INITIALIZATION
76                                COMPLETE - STATUS [READY]")
77
78         # Create Publisher/Subscriber objects
79         self._publish_robot_pose = self.create_publisher(
80             Float64MultiArray, 'robot_pose', 10)
81         self._publish_robot_joint_angles = self.
82             create_publisher(Float64MultiArray, '
83                                robot_joint_angles', 10)
84         self.raw_data_subscriber = self.create_subscription(
85             String, 'raw_input_data', self.store_raw_data, 10)
86
87         # Create/execute callback functions
88         self._move_robot_timer = self.create_timer(0.1, self.
89             move_robot_arm) # DEFAULT: 0.1
90         self._retrieve_joint_angles = self.create_timer(0.01,
91             self.retrieve_joint_angles)

```

```

79
80     # Define variables for local data storage
81     self._input_key = "NONE"
82     self._cur_position = self._mc.get_coords() # [x, y, z,
        pitch, roll, yaw]
83     self._incr_pos = 2.0 # DEF -> 0.5 -> 1 -> 1.3 -> LAST
        CHANGE 0.8
84     self._joint_0_angle = self._cur_position[0]
85     self._prev_joint_6_angle = self._cur_position[5]
86     self._target_position = [0.0, 0.0, 0.0]
87     self._target_rotation = [0.0, 0.0, 0.0, 0.0] # qx, qy,
        qz, qw
88     self._cur_position = self._mc.get_coords() # [x, y, z,
        pitch, roll, yaw]
89     self._robot_offset = 97 # Offset between the joint 0
        and joint 6 on the xy-plane, in mm.
90     self._move_speed = 30 # Arm movement speed in mm/s (
        default 25mm/s)
91     self._command_delay = 0.015 # Delay after transmitting
        motion command (default 0.04s)
92
93     # Define callback function to store topic data into
        internal variables
94     def store_raw_data(self, msg):
95         self._input_key = msg.data
96
97         if (msg.data[:3] == "GRC"):
98             split_string_list = msg.data.split(',') # Seperate
                the string using csv format
99             # Convert and store the data
100             for i in range(len(self._target_position)):
101                 self._target_position[i] = float(
                    split_string_list[i+1])
102             for j in range(len(self._target_rotation)):
103                 self._target_rotation[j] = float(
                    split_string_list[j+4])
104
105
106     # Define a callback function to translate the robot arm's
        end effector in coordinate space
107     def move_robot_arm(self):
108         # Send position data to robot arm
109         if (((self._target_position[0]**2)+(self._
            _target_position[1]**2)+(self._target_position
            [2]**2)) > 0.0):
110             old_position = [self._cur_position[0], self._
                _cur_position[1], self._cur_position[2]]

```

```

111         old_rotation = [self._cur_position[3], self.
112                           _cur_position[4], self._cur_position[5]]
113
114         self._cur_position[0] = self._target_position[2] *
115             1000
116         self._cur_position[1] = self._target_position[0] *
117             -1000
118         self._cur_position[2] = self._target_position[1] *
119             1000
120         self._cur_position[5] = yaw_axis_alignment(self.
121             _cur_position, self._robot_offset)
122
123         new_position = [self._target_position[2] * 1000,
124                         self._target_position[0] * -1000, self.
125                         _target_position[1] * 1000]
126
127         r = R.from_quat(self._target_rotation)
128         euler_angles = r.as_euler('xyz', degrees=True) #
129                 returns roll, pitch, yaw in degrees
130         roll, pitch, yaw = euler_angles
131
132         roll_r = to_positive_angle(roll + self.
133             _cur_position[3])
134         pitch_r = to_positive_angle(yaw + self.
135             _cur_position[4])
136         yaw_r = to_positive_angle(-pitch + self.
137             _cur_position[5])
138
139         new_rotation = [roll_r, pitch_r, yaw_r]
140
141         # Logging
142         #print(f"Target poistion in mm: {self.
143             _cur_position[0:4]}")
144         #print(f"Target rotation: {self._target_rotation
145             }")
146         print(f"Euler Angles: {roll_r, pitch_r, yaw_r}")
147         # Serial communications
148         #self._mc.send_coords(self._cur_position, self.
149             _move_speed, 1) # Execute coordinate control
150             command
151
152         if (is_significant_change(new_position,
153             old_position) or is_significant_rotation(
154             new_rotation, old_rotation)):
155             self._mc.send_coords([self._cur_position[0],
156                 self._cur_position[1], self._cur_position

```

```

140         [2], roll_r, pitch_r, yaw_r], self.
141         _move_speed, 1)
142
143     # Define a callback function to retrieve and store the
144     angle of joint 0
145     def retrieve_joint_angles(self):
146         # Retrieve joint angle data
147         joint_angles_msg = Float64MultiArray()
148         cur_angles = self._mc.get_angles()
149         joint_angles_msg.data = cur_angles
150
151         # Publish the data to the topic
152         self._publish_robot_joint_angles.publish(
153             joint_angles_msg)
154
155     def main(args=None):
156         try:
157             rclpy.init(args=args)
158             node = GlobalRobotCtrl()
159             rclpy.spin(node)
160         except Exception as main_err:
161             print(f"Error initializing node: {str(main_err)}")
162         finally:
163             if rclpy.ok():
164                 rclpy.shutdown()
165             print("Node shutdown complete.")
166
167     if __name__ == '__main__':
168         main()

```

Listing C.4: ROS node computing position and orientation vectors for the robot arm using the third-person digital twin control mode.