ECE 445

SENIOR DESIGN LABORATORY

DESIGN DOCUMENT

Robotic Arm Integrated into Wheelchair with MR Interface

Team #12

XINGRU LU (xingrul2@illinois.edu)
YILIN WANG (yilin14@illinois.edu)
YUNYI LIN (yunyil3@illinois.edu)
YINUO YANG (yinuoy4@illinois.edu)

TA: Yun Long

April 14, 2025

Contents

1 Introduction

1.1 Objective and Background

1.1.1 Goals

Wheelchair users often face significant challenges when interacting with objects beyond their immediate reach, particularly behind them. Without external assistance, tasks such as pressing buttons or navigating through environments with complicated surroundings can become difficult. These difficulties are compounded when operating independently, highlighting the need for supplementary support to simplify routine activities. Additionally, wheelchair users may struggle with limited situational awareness, as their field of view is primarily forward-facing. As a result, there is a pressing need for innovative solutions that enhance both accessibility and autonomy, enabling wheelchair users to interact more conveniently with their surroundings.

1.1.2 Functions

Our solution integrates a rear-facing camera that streams real-time visuals to a Mixed Reality (MR) interface, allowing wheelchair users to gain visual awareness of their surroundings, including blind spots behind them. Additionally, a robotic arm mounted at the back of the wheelchair can be controlled through MR, enabling users to perform assistive actions such as pressing buttons and interacting with objects beyond their physical reach. This system enhances both situational awareness and independent mobility, providing a more intuitive and convenient way for users to navigate and interact with their environment.

1.1.3 Benefits

Our solution helps people with wheelchairs by enhancing **situational awareness** and **independent mobility**. It provides real-time imagery with the Mixed Reality (MR) interface via the rearview camera, allowing users to see blind spots and navigate more securely. Additionally, Users can use the robotic arm, controlled by MR, to push buttons, for example, to make it easier to perform everyday tasks independently. It gives users an easier and more intuitive way to interact with the environment, ultimately resulting in greater autonomy and accessibility.

1.1.4 Features

- **Mixed Reality Integration:** Combines real-time MR with assistive robotics, offering an intuitive control experience.
- **Enhanced Situational Awareness:** A rear-facing camera provides live visuals of blind spots, improving navigation.
- **Extended Reach with Robotic Arm:** The Open Manipulator-P allows users to interact with objects beyond their grasp, such as pressing buttons behind them.

- **Apple Vision Pro for Control & Feedback:** Tracks hand movements for precise robotic control and provides interactive feedback.
- **Low Latency & Safety Focus:** Ensures smooth operation without compromising wheelchair stability.

1.2 High-Level Requirements List

- **Precision:** The robotic arm should reliably press buttons with a diameter of at least **35mm**, which is a common size of elevator buttons. The force applied must be sufficient to activate buttons without excessive pressure that could cause damage or failure.
- **Safety and Stability:** Users should be able to see both the front and rear environments through Vision Pro, while also adjusting the robotic arm's perspective to gain a broader field of view.
- **Reach:** The robotic arm should be able to reach a height from 110cm to 160cm.

2 Design

2.1 Block Diagrams

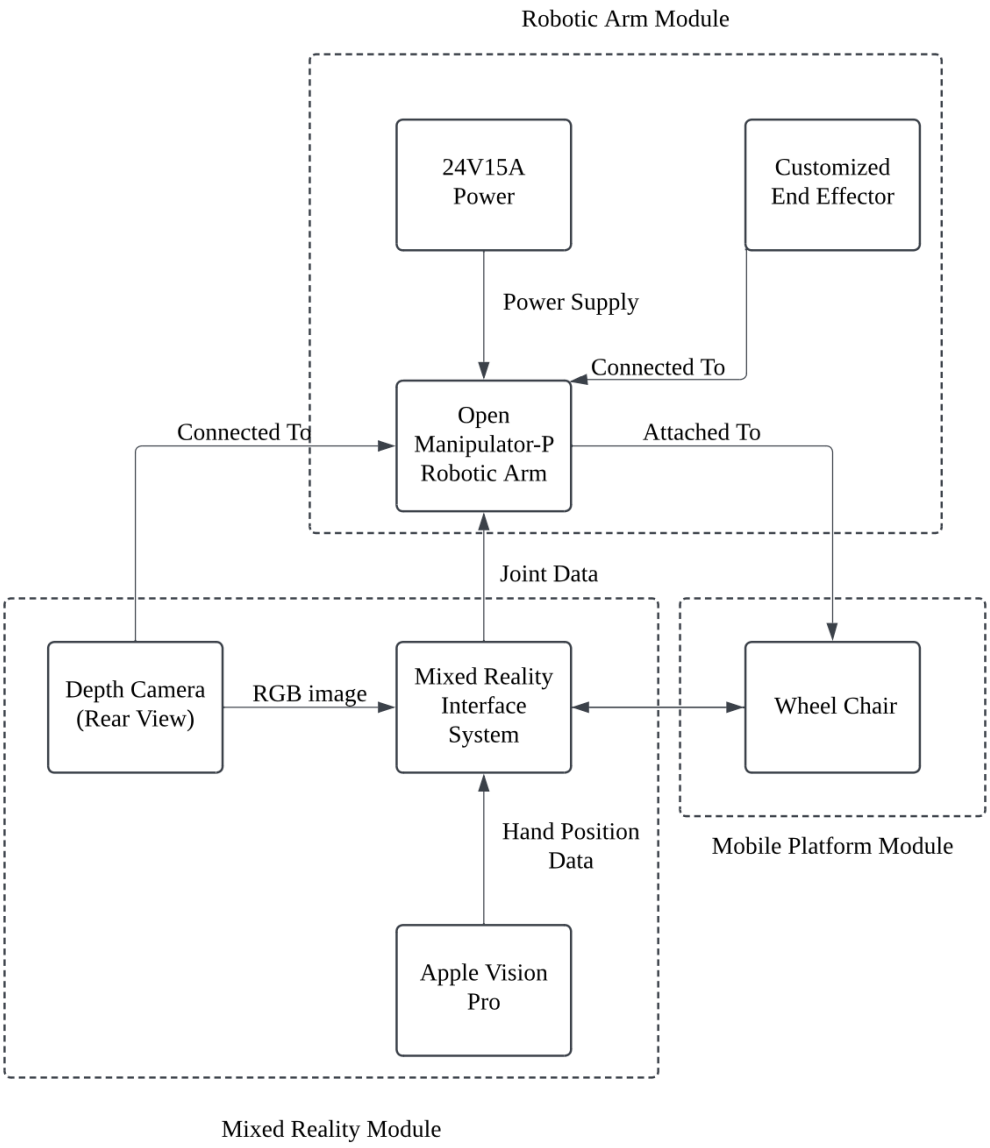

Figure 1: Block Diagram

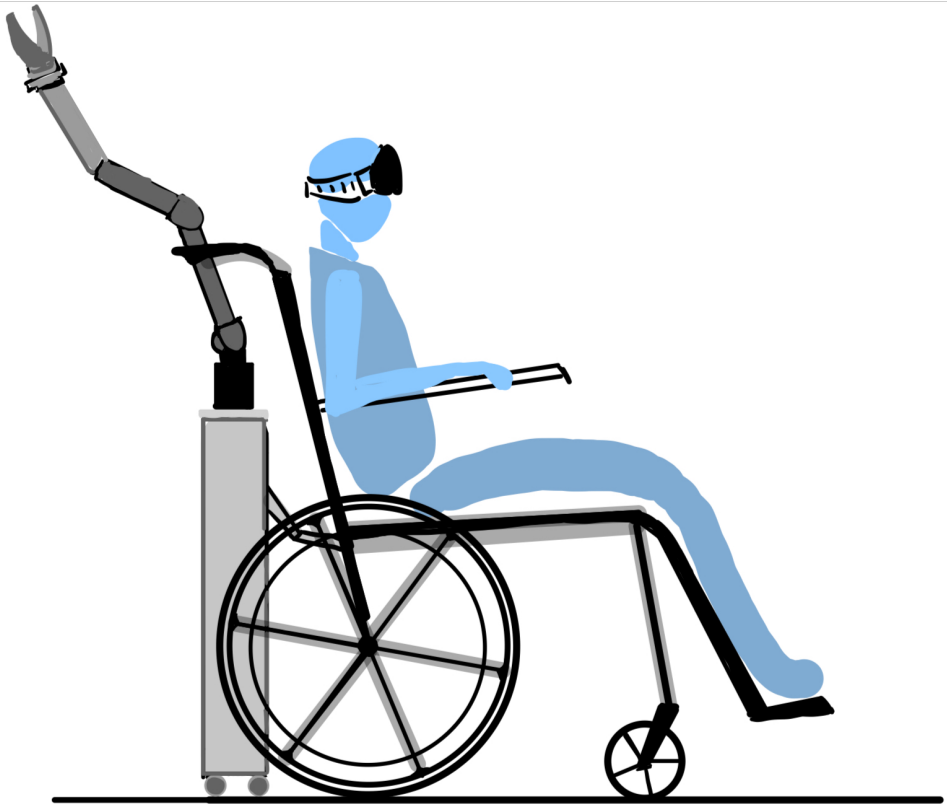

Figure 2: Physical Overview

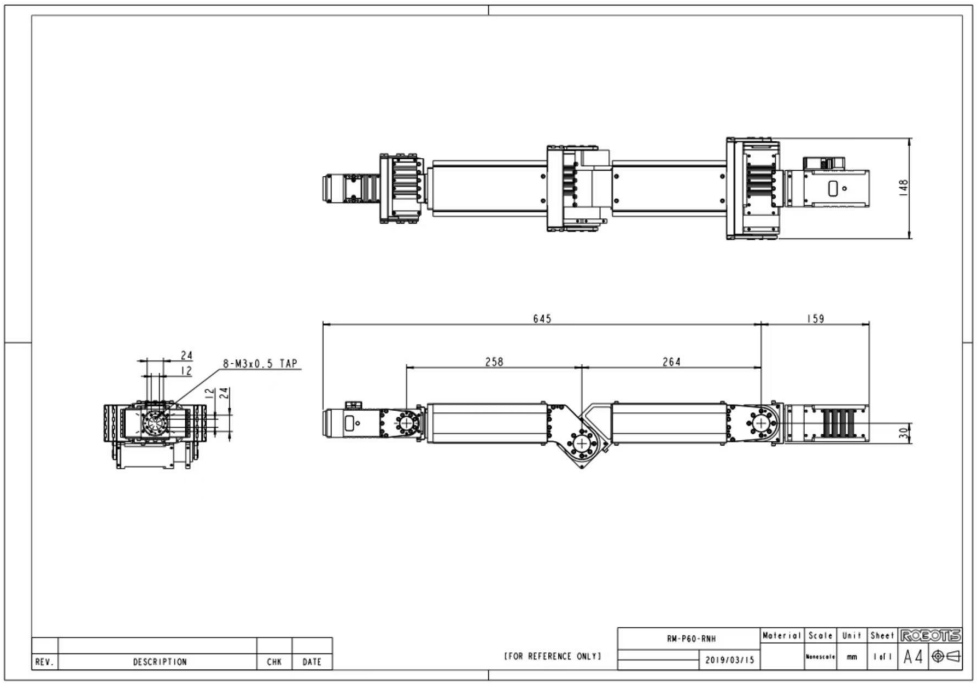

Figure 3: Dimensions of OpenManipulator-P [1]

2.2 Block Descriptions

2.2.1 Customized End Effector

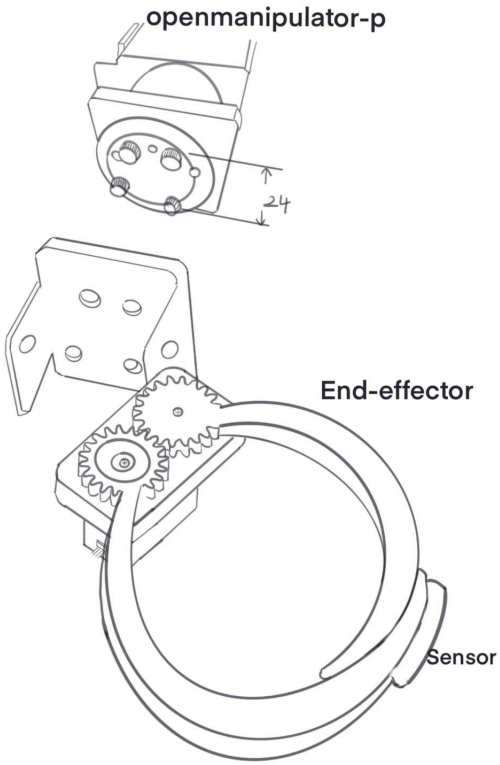

Figure 4: Sketch of Our End-effector Design

The Customized End Effector is a gripper-style clamp designed for basic interaction tasks, such as pressing buttons, supporting tasks that require moderate accuracy but not extreme precision. It is mounted mechanically on the Open Manipulator-P Robotic Arm and operates based on the signals provided by the Mixed Reality Interface System.

A thin-film force-sensitive resistor (FSR) is mounted on the robotic end-effector to detect contact force during interactions such as elevator button presses. Designed for 5 - 20 N range, it ensures reliable actuation without excessive force that could cause damage to objects.

A development board will act as the control unit for the end-effector and force sensor. It processes the data from the sensor and controls the motor actuators to adjust the gripper's force. The system also provides real-time display and monitoring of the applied pressure.

2.2.2 Open Manipulator-P Robotic Arm

Open Manipulator-P Robotic Arm is powered by a 24V15A Power Supply and equipped with a Customized End Effector to interact with the environment, such as to press but-

tons. It is physically mounted on the Mobile Platform Module (Wheelchair) and connected to the Mixed Reality Module for feedback and control. The OpenMANIPULATOR-P system relies on several key ROS packages to bridge Unity-based control interfaces with low-level robotic execution. These packages collectively handle launch configuration, message definition, motion planning, hardware control, and communication bridging.

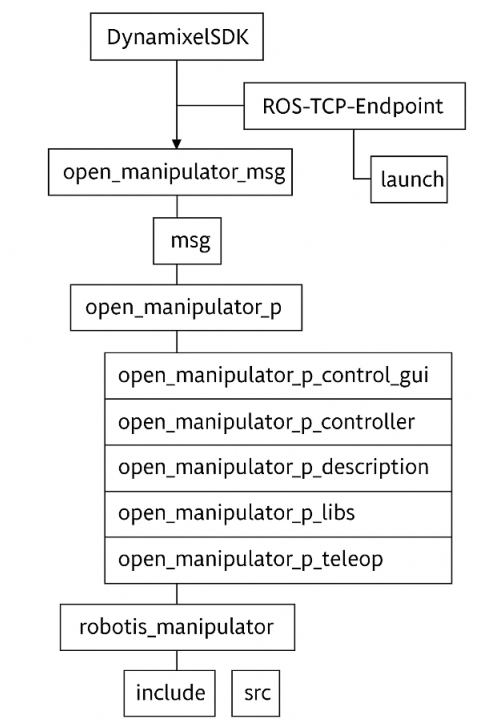

Figure 5: ROS Package Dependencies for OpenMANIPULATOR-P System

i. open_manipulator_p The **open_manipulator_p** package serves as the main ROS launch and configuration interface for the OpenMANIPULATOR-P. It provides high-level launch files to initialize and manage the robot.

- **Launch Files:** Unified scripts to initialize the robot controller, state publisher, and joint trajectory control services.
- **Parameter Configuration:** Loads URDF and control parameters into the ROS parameter server.

Role in the System: Acts as the central hub for bootstrapping the robot; Unity indirectly uses this via `roslaunch`.

ii. open_manipulator_msgs The **open_manipulator_msgs** package defines custom ROS message and service types tailored to the manipulator's operation. These are essential for sending and receiving structured commands from Unity.

- **SetJointPosition.srv:** Service definition for commanding joint angles with path timing.
- **JointPosition.msg:** Structured joint data including joint names, positions, and control scaling.
- **TaskSpacePath.msg:** Used for controlling the end-effector in Cartesian space.

Role in the System: Provides the message format for Unity $\rightarrow$ ROS communication and is required to compile any Unity–ROS interface.

iii. robotis_manipulator The **robotis_manipulator** package implements a lightweight motion planning and control framework, used internally by the OpenMANIPULATOR controller. It contains classes and algorithms for kinematics, dynamics, trajectory generation, and actuator interfacing.

- **Forward/Inverse Kinematics:** Calculates pose from joint angles and vice versa.
- **Trajectory Planning:** Supports linear, cubic, and custom joint motion interpolation.
- **Modular Actuator Interface:** Abstract layer for hardware-agnostic actuator control.

Role in the System: Functions as the mathematical and logical backend of the manipulator. It is not directly called from Unity, but it's essential for executing Unity commands properly.

iv. ROS-TCP-Endpoint The **ROS-TCP-Endpoint** is a ROS-side communication bridge provided by Unity's Robotics Hub. It enables bi-directional TCP/IP communication between Unity and ROS. Unity sends service requests or publishes messages to ROS via this bridge.

- **Service Server Interface:** ROS receives Unity's service calls (e.g., `SetJointPosition`).
- **Message Publishing/Subscription:** Allows Unity to publish to or subscribe from ROS topics.
- **Flexible Transport Layer:** Uses TCP protocol for reliable, real-time data transfer.

Role in the System: Enables real-time command and feedback exchange between Unity and ROS nodes—without it, Unity cannot communicate with the robot.

v. DynamixelSDK The **DynamixelSDK** is a low-level hardware communication library used by Robotis devices. It provides the necessary interface to send and receive commands from Dynamixel servo motors, which power each joint of the OpenMANIPULATOR-P.

- **Packet Protocol (1.0 / 2.0):** Reliable serial communication over USB/UART.
- **Synchronous Write/Read:** Efficient multiple motor control and feedback.
- **Cross-platform C/C++ API:** Used by higher-level controller packages.

Role in the System: Acts as the hardware abstraction layer. The robot's joints won't move without this library—even if the Unity command is received correctly.

2.2.3 Depth Camera

The **Intel RealSense Depth Camera D435i** is a high-performance depth-sensing camera designed for applications requiring precise 3D vision, depth mapping, and motion tracking. It integrates an RGB sensor, stereo depth sensors, and an inertial measurement unit (IMU) to provide synchronized depth, color, and motion data.

i. Stereo Depth Sensors

- Two high-resolution global shutter infrared (IR) sensors for accurate depth perception.
- Active IR projector for improved depth accuracy in low-light conditions.
- Depth resolution up to 1280×720 @ 90 fps (VGA @ 30 fps recommended for optimal performance).

ii. RGB Sensor

- Full HD (1920×1080) color camera with a rolling shutter for high-quality 2D imaging.

iii. Inertial Measurement Unit (IMU)

- Built-in 6-DOF IMU (accelerometer + gyroscope) for motion tracking.
- Enables sensor fusion applications by aligning depth data with motion data.

iv. Processing Unit

- Onboard depth processing with Intel's RealSense Vision Processor (D4 ASIC).
- Reduces host CPU load by handling depth calculations internally.

Output Streams

- **Depth Stream:** Depth data in millimeters (up to 10 meters range, adjustable).
- **RGB Stream:** Color video feed (1920×1080 @ 30 fps).
- **IR Stream:** Left and right IR images for stereo matching.
- **IMU Data:** Accelerometer and gyroscope readings for motion tracking.
- **Point Cloud:** 3D spatial data (via SDK post-processing).

Key Features & Specifications

- **Depth FOV:** 85° × 58° (H × V)
- **RGB FOV:** 69° × 42° (H × V)
- **Depth Accuracy:** ±2% at 2m distance
- **Minimum Depth Distance:** ∼0.2m (with near-field mode enabled)
- **Connectivity:** USB 3.1 Type-C
- **Software Support:**
  - Intel RealSense SDK 2.0 (Windows, Linux, macOS)
  - ROS (Robot Operating System) integration

The camera will be mounted on the robotic arm and will move with the arm, providing real-time RGB video to track the end effector's motion and actions.

2.2.4 Apple Vision Pro

The **Apple Vision Pro** aims to capture and process environmental data to enable robust spatial computing and natural user interaction. It operates by gathering high-resolution camera images and depth sensor inputs, which are essential for understanding the surrounding environment. The cameras capture high-resolution RGB images for image processing and object detection, while the depth sensor generates precise depth maps by measuring distances to objects, directly contributing to accurate coordinate computation. This encapsulated image and depth data is then processed through the XR hand subsystem toolkit in Unity. The toolkit interprets the visual and spatial inputs to generate hand coordinate data, effectively mapping the user's hand movements into a skeletal framework for natural interaction within a mixed reality environment.

Input:

- **Camera Image:** This input is captured using multiple cameras, which can capture high-resolution RGB images. These images enable image processing and object detection.
- **Depth Sensor Input:** The depth sensor generates depth maps by measuring the distance to objects. This input is crucial for spatial computing since it directly contributes to coordinate computation.

Output:

- **Hand Coordinate Data:** This output is generated using the XR hand subsystem toolkit in Unity. It reads the encapsulated image data from the Apple Vision Pro and processes this data to generate the hand's skeleton coordinates.

2.2.5 Mixed Reality Interface System

The Mixed Reality Interface Module processes sensor inputs, it combines RGB images from the depth camera with hand position data from the Apple Vision Pro. It seamlessly fuses virtual elements with the physical environment to construct an interactive display, while simultaneously generating real-time control commands (joint data for robotic arm movement) that enable coordinated system-wide operations. Additionally, it provides a user-friendly GUI for controlling the robotic arm and delivers feedback on system performance.

Input:

- **RGB image from rear view:** This input is generated from the depth camera attached to the end effector of the robotic arm. It captures the environment from the rear view of the user and thus provides feedback to the user.
- **Hand Position Data from Apple Vision Pro:** This input is generated from Apple Vision Pro and procced by the XR interaction toolkit in Unity. It allows users to interact with the virtual scene in the mixed reality interface.

Output:

- **Joint Data for Open Manipulator-P robotic arm:** This output is generated using the inverse kinematic toolkit provided by Unity Ros package. It convert the hand coordinate data to 6 DOF joint data for robotic arm.

2.2.6 Mobile Platform

The Mobile Platform Module is the foundation for the mobility in the system, allowing users to navigate the space while maintaining control of the Robotic Arm Module. It's mainly a wheelchair, which is controlled by the Mixed Reality Interface System to make the navigation either speech-based or gesture-based. When integrated with the Robotic Arm Module, the platform allows users to work with objects even when moving, enhancing independence and accessibility.

2.3 Software Flowchart

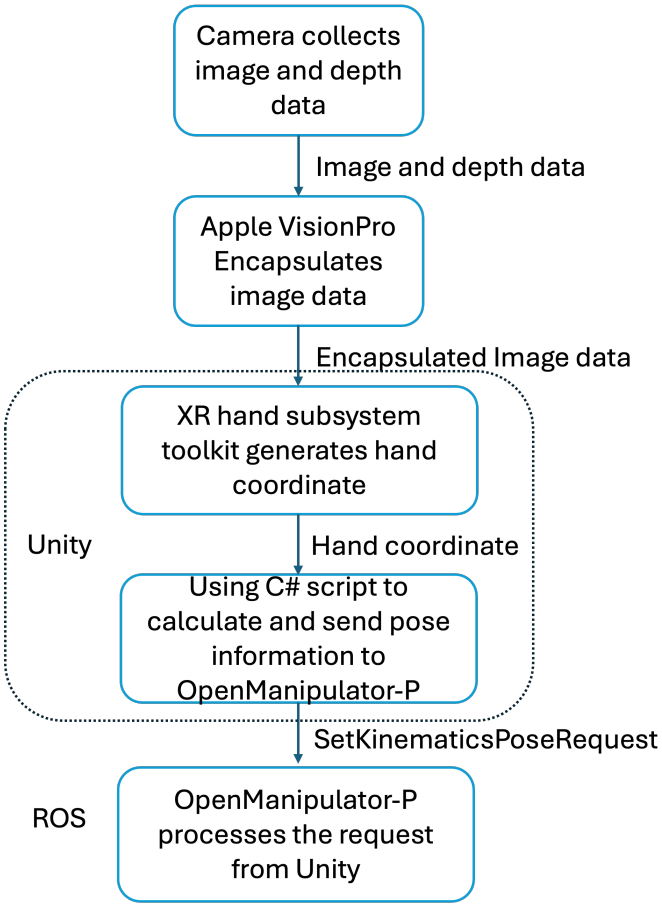

Figure 6: Diagram illustrating the process flow of controlling the OpenManipulator-P robotic arm using the Apple VisionPro system, Unity, and ROS, where image and depth data from the camera are processed to generate hand coordinates, which are then used to send pose information to the robotic arm for real-time control.

2.4 ROS-Unity Integration

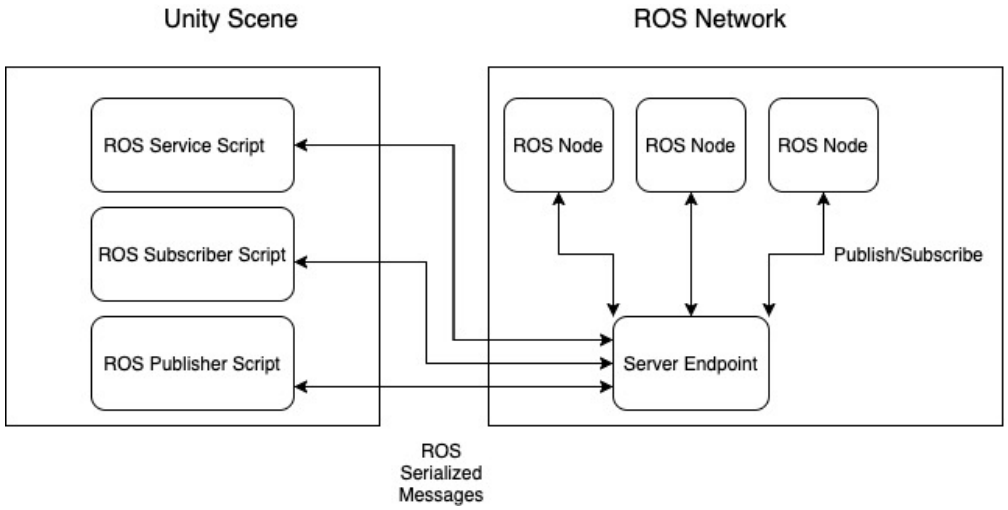

Figure 7: ROS-Unity Communication Structure [2]

The ROS–Unity communication bridge plays a critical role in our system because the pose information used to control the robotic arm is generated within the Unity environment. However, the services that execute these pose commands and control the physical robotic arm reside on the ROS side, running on an Ubuntu system. Therefore, real-time communication is necessary to ensure that Unity can request and receive robotic actions from the ROS backen.

Communication between ROS nodes follows the traditional ROS **Publish/Subscribe** model, allowing for decoupled and scalable message passing. Unity interacts with this system in real time, sending commands and receiving feedback, making it possible to simulate, visualize, and control physical robotic systems directly from a Unity-based application.

This architecture enables effective and synchronized bidirectional communication between Unity and ROS, essential for applications involving real-time robotic control, simulation, and user interaction through immersive interfaces like VisionPro. The communication between Unity and ROS is facilitated through a structured interface composed of publisher, subscriber, and service scripts on the Unity side, and corresponding nodes and a server endpoint on the ROS side.

Within the **Unity Scene**, three types of scripts are responsible for exchanging data with the ROS system:

- **ROS Publisher Script:** Sends messages from Unity to ROS, such as robot control commands or object positions.
- **ROS Subscriber Script:** Receives messages from ROS, including sensor data and robot status, and makes them available to the Unity environment for visualization or feedback control.

- **ROS Service Script:** Initiates service requests (e.g., `SetKinematicsPoseRequest`) and processes responses from ROS, enabling more structured and request-response-type communication.

These scripts communicate with a centralized **Server Endpoint** in the **ROS Network**, which acts as a gateway, handling the serialization and deserialization of ROS messages. The server endpoint further interfaces with various **ROS Nodes**, each responsible for specific robotic tasks such as inverse kinematics, motion planning, or sensor integration.

2.5 Calculations: Inverse Kinematics

In robotic arm control, the calculation of the joint path from the end effector pose is a fundamental process known as **inverse kinematics**. Given the desired position and orientation of the end effector, the goal is to compute the corresponding joint angles or positions that will achieve this pose.

End Effector Pose: The end effector pose is represented as a homogeneous transformation matrix T_{ee}, which combines both the position and orientation of the end effector. It is typically written as:

$$T_{\text{ee}} = \begin{bmatrix} R_{\text{ee}} & p_{\text{ee}} \\ 0 & 1 \end{bmatrix}$$

where: - R_{ee} is the 3x3 rotation matrix representing the orientation of the end effector, - p_{ee} is the 3x1 position vector of the end effector.

The inverse kinematics solution involves determining the joint angles $\theta_1, \theta_2, \ldots, \theta_n$ that produce a specific T_{ee}. These joint angles correspond to the values in the robot's joint space, which will generate the desired pose.

Inverse Kinematics: The inverse kinematics problem can be described mathematically by a set of equations that relate the joint variables θ_i to the end effector pose. For a robotic arm with n joints, the transformation from joint space to task space is expressed as a function of the joint variables:

$$T_{\text{ee}} = f(\theta_1, \theta_2, \ldots, \theta_n)$$

This function $f(\cdot)$ typically consists of a series of transformations that account for the arm's kinematic structure, such as link lengths and joint types. The IK solution seeks to find $\theta_1, \theta_2, \ldots, \theta_n$ such that:

$$T_{\text{ee}} = T_{\text{target}}$$

where T_{target} is the desired target pose. In practice, solving the inverse kinematics problem can involve algebraic methods for simpler arms or numerical methods (such as gradient descent or Newton-Raphson) for more complex arms.

Jacobian Matrix: To compute the joint velocities or the joint path from the end effector velocity, the Jacobian matrix J is used. The Jacobian relates the joint velocities $\dot{\theta}$ to the end effector velocity $\dot{p}$ in the task space:

$$\dot{p} = J(\theta)\dot{\theta}$$

where $J(\theta)$ is the Jacobian matrix, which is a function of the joint angles θ. The Jacobian matrix provides the linear mapping between the joint space and the task space. Inverse kinematics algorithms often use the Jacobian to iteratively adjust the joint angles to minimize the difference between the current and target end effector pose.

3 Requirements and Verifications

3.1 Customized End Effector

The Customized End Effector is a gripper-style clamp designed for basic interaction tasks, such as pressing buttons, supporting tasks that require moderate accuracy but not extreme precision. It is mounted mechanically on the Open Manipulator-P Robotic Arm and operates based on the signals provided by the Mixed Reality Interface System.

Requirements	**Verification**
1) Measure 5–50 N with ±20% error on arc-shaped surface.	1) Use a force gauge to apply 5/10/20/50 N loads. Compare sensor output (voltage→force conversion) against theoretical values (F=mg).
2) Complete successful button presses on elevator with force less than 30N.	2) Test on more than 5 elevator buttons (of different height and position), verify that buttons are successfully pressed under the pre-programmed threshold.

3.2 Open Manipulator-P Robotic Arm

Requirements	**Verification**
1) The robotic arm shall maintain an end-effector positioning accuracy within 10 mm during standard trajectory execution.	1) Execute programmed joint-space and task-space movements with known targets. Measure final end-effector position using a calibrated ruler or laser measurement device. Repeat over 10 trials.
2) The robotic arm shall apply force within the range of 0.5–5N to activate buttons or interact with objects, ensuring successful actuation without damage.	2) Use a calibrated force sensor integrated into the robot's end-effector to measure the applied force during button pressing.
3) The system shall respond to command inputs within 300 ms, including Unity-to-ROS transmission, processing, and actuator movement initiation.	3) Use timestamp logging in Unity and ROS to measure time from service request to motor actuation. Measure multiple trials and compute average latency.
4) The manipulator shall move at a speed not exceeding 0.2 m/s during human interaction tasks, ensuring safety and control.	4) Record video of end-effector motion with timestamp overlay. Use frame-by-frame analysis or motion capture to calculate peak velocity.

3.3 Depth Camera

Requirements	Verification
1) The Unity application shall display the RealSense RGB stream in a viewport that can be zoomed in and out. When zoomed in to occupy full view, video should have 1920×1080 resolution at 30 fps, maintaining native quality.	1) Capture Unity's screen output during runtime and analyze frame rate (Unity Profiler) and resolution (pixel measurement tool). Verify no downscaling or frame drops.
2) End-to-end latency (camera capture to Unity display) shall be 150ms for RGB stream.	2) Use a high-speed timer (e.g., Stopwatch in C#) to log timestamps from frame capture (RealSense SDK) to Unity rendering. Average 10 samples.
3) The camera shall maintain an unobstructed view of the robotic arm's end-effector while moving synchronously with it, ensuring continuous tracking.	3) Test robotic arm full-range motion and verify that no blind-spot exists.

3.4 Mixed Reality Interface System

Requirement	Verification
1) The interface shall display a 3D model of the user's hands with end-to-end latency no greater than 50 ms.	1) Mount high-speed cameras to monitor both real-world hand motion and the virtual hand representation. Compare timestamp logs or frame-based recordings to calculate the total system latency. Confirm 50 ms.
2) The interface shall interpret basic hand gestures (e.g., pinch, grab) with an accuracy of at least 80% under normal operating conditions.	2) Define a test set of gestures. Record the system's recognition results against ground-truth labels. Evaluate the result to confirm the gesture accuracy is 80%.
3) The interface shall provide a user-friendly GUI for robotic arm control, displaying real-time feedback on both status and errors.	3) Conduct user tests to ensure the GUI reacts to user inputs. Monitor system logs to verify error handling.

3.5 Apple Vision Pro

Requirement	Verification
1) The device shall provide hand-tracking coordinates with an error margin 5 mm relative to actual hand positions.	1) Use a motion capture system as a ground-truth reference. In multiple trials, record Apple Vision Pro's hand tracking data and compare coordinate sets with the motion capture baseline. Confirm the error 5 mm.

3.6 Mobile Platform

Requirements	Verifications
1) The Robotic Arm must be securely attached to the wheelchair using a flange column and other physical mounting devices to ensure stability and safety during operation.	1) Perform a mechanical stress test where the platform (wheelchair with robotic arm) is subjected to various forces to ensure the arm remains securely attached without detaching or loosening.
2) The system must ensure the safety and stability of the user while controlling the robotic arm and navigating the wheelchair. The wheelchair must be stable and not prone to tipping over when interacting with the robotic arm.	2) Conduct tipping tests by simulating various load conditions and interactions with the robotic arm, ensuring the wheelchair remains stable and does not tip over under normal use.

4 Tolerance Analysis

4.1 End-Effector Position Error

The end-effector position error is determined by the angular resolution of each motor joint and the arm length. The calculations below outline the process of calculating the error for each joint and summing the errors from all joints.

Step 1: Angular Resolution for Each Joint The angular resolution for each motor is calculated using the encoder resolution. The formula for angular resolution is:

$$\text{Angular Resolution} = \frac{360^\circ}{\text{Encoder Resolution (pulses/rev)}}$$

For Joints 1, 2, 3, and 4 (with 501,923 pulses/rev):

$$\text{Angular Resolution} = \frac{360^\circ}{501,923} = 0.000718^\circ \text{ per pulse}$$

For Joints 5 and 6 (with 303,750 pulses/rev):

$$\text{Angular Resolution} = \frac{360^\circ}{303,750} = 0.001186^\circ \text{ per pulse}$$

Step 2: End-Effector Position Error To calculate the position error at the end-effector, we use the following equation:

$$\text{End-Effector Position Error} \approx \text{Joint Error} \times \text{Arm Length} \times \sin(\theta)$$

Where: - Joint Error is the angular resolution of the motor (in radians), - Arm Length is the maximum reach of the robotic arm (645 mm).

For Joints 1 to 4 (with 501,923 pulses/rev):

$$\text{Angular Resolution} = 0.000718^\circ = 0.000718 \times \frac{\pi}{180} \text{ radians} = 1.254 \times 10^{-5} \text{ radians}$$

For each joint:

$$\text{Position Error (Single Joint)} = 645 \times 1.254 \times 10^{-5} = 0.0081 \,\text{mm}$$

Since there are 4 joints contributing to the total error:

$$\text{Total Position Error (Joints 1-4)} = 4 \times 0.0081 = 0.0324\,\text{mm}$$

For Joints 5 and 6 (with 303,750 pulses/rev):

$$\text{Angular Resolution} = 0.001186^\circ = 0.001186 \times \frac{\pi}{180}\ \text{radians} = 2.071 \times 10^{-5}\ \text{radians}$$

For each joint:

$$\text{Position Error (Single Joint)} = 645 \times 2.071 \times 10^{-5} = 0.0134\,\text{mm}$$

Since there are 2 joints contributing to the total error:

$$\text{Total Position Error (Joints 5-6)} = 2 \times 0.0134 = 0.0268\,\text{mm}$$

Finally, summing the errors from all joints:

$$\text{Total End-Effector Position Error} = 0.0324 + 0.0268 = 0.0592\,\text{mm}$$

To make the analysis more realistic and reflect real-world conditions, it is important to consider additional error sources that can significantly affect the accuracy of hand tracking and robotic control. VisionPro, for instance, has its own intrinsic tracking error. When considering the robotic arm itself, additional mechanical issues like mounting instability or slight movement of the arm or end-effector, especially under varying loads, can introduce further deviations. These errors can lead to a final error in hand position that is much larger than the theoretical motor error alone.

4.2 Head Movement Impact on Hand Tracking Accuracy

When using Vision Pro for hand tracking, the movement of the head-mounted device (HMD) introduces errors in the perceived hand position. Since the tracking system is based on the relative position of the hands with respect to the headset, any movement of the headset alters the reference frame, potentially causing discrepancies between the actual and detected hand positions.

Let:

- H be the headset coordinate frame,
- H' be the hand coordinate frame (relative to the headset),
- ΔH be the headset movement relative to the world coordinate system,
- Δh be the hand movement relative to the headset,

- $\Delta h'$ be the actual hand movement in the world coordinate system.

The actual hand displacement in the world frame should be calculated as:

$$\Delta h' = \Delta H + \Delta h$$

However, if the system does not compensate for head movement, it assumes:

$$\Delta h' = \Delta h$$

This leads to errors in hand position estimation due to the head movement altering the reference frame.

4.3 Latency in the System Pipeline

Latency is a critical factor in the real-time control of a robotic arm using Vision Pro hand tracking. The total system latency consists of four main components:

- L_{total} - Vision Pro Hand Tracking Latency: The time delay between the user's actual hand movement and the system's recognition of the new hand position. According to Road to VR, Vision Pro's measured hand tracking latency is approximately 128 ms.
- $L_{\text{processing}}$ - Program Processing Latency: The time required for the computing system to process the received hand position data and generate corresponding control commands for the robotic arm. This depends on computational complexity, data transmission speed, and software optimizations.
- L_{usb} - USB Communication Latency (DYNAMIXEL U2D2): As noted in the DYNAMIXEL U2D2 e-Manual, when connecting the robotic arm to the PC via USB, the default USB latency time is 16 ms.
- L_{robot} - Robotic Arm Reaction Latency: The delay between receiving the control command and the robotic arm physically executing the movement. This includes actuation delay, motor inertia, and mechanical constraints.

Thus, the total system latency can be expressed as:

$$L_{\text{total}} = L_{\text{tracking}} + L_{\text{processing}} + L_{\text{usb}} + L_{\text{robot}}$$

Substituting known values:

$$L_{\text{total}} = 128\,\text{ms} + L_{\text{processing}} + L_{\text{usb}} + L_{\text{robot}}$$

For real-time interaction, it is essential that L_{total} remains below the human perceptual threshold for smooth interactions, typically around 200-250 ms for motion feedback applications. If L_{total} exceeds this limit, users may experience noticeable lag, affecting the precision and responsiveness of robotic control.

Assuming:

- $L_{\text{processing}} = 10\,\text{ms}$ to $30\,\text{ms}$,
- $L_{\text{robot}} = 50\,\text{ms}$ to $150\,\text{ms}$,
- $L_{\text{usb}} = 1\,\text{ms}$ to $16\,\text{ms}$,

Then the total system latency falls within:

$$L_{\text{total}} = 128\,\text{ms} + (10\,\text{ms} - 30\,\text{ms}) + (1\,\text{ms} - 16\,\text{ms}) + (50\,\text{ms} - 150\,\text{ms}) = 189\,\text{ms to } 324\,\text{ms}$$

5 Cost & Schedule

5.1 Cost

Item Description	Rate/Price (RMB)	Hours/Qty	Subtotal (RMB)
Labor Cost			
Yinuo Yang	50 (per hour)	250	12,500
Yunyi Lin	50 (per hour)	250	12,500
Xingru Lu	50 (per hour)	250	12,500
Yilin Wang	50 (per hour)	250	12,500
Total Labor Cost			**50,000**
Parts Cost			
Wheel Chair	250	1	250
24V15A Power Supply	73	1	73
Thin Film Pressure Sensor	300	1	300
Single-Channel Module	300	1	300
Robotic Arm Bracket	300	1	300
End Effector	200	1	200
USB 3.2 Data Cable	30	1	30
Total Parts Cost			**1,453**
		Grand Total Cost:	**51,453**

Table 1: Cost

5.2 Schedule

Week	Task
3.15 - 3.21	Write a program to detect hand gestures and drag virtual objects in Unity. Install ROS and OpenManipulator package, achieve basic control over the robotic arm.
3.22 - 3.28	Connect Unity with Isaac Sim using ROS2 and simulate a virtual environment in Isaac Sim. Establish bi-directional communication between Unity and ROS (Ubuntu).
3.29 - 4.04	Successfully control the arm in Unity and use VisionPro to control the robotic arm.
4.05 - 4.11	Start designing the mounting system to attach the robotic arm to the wheelchair.
4.12 - 4.18	3D print and install a gripper that meets the requirements. Install a pressure sensor on the gripper.
4.19 - 4.25	Write code on the development board to control the opening and closing of the gripper and display pressure readings, while establishing a connection with Unity so that VisionPro's hand gesture recognition can control the gripper's operation.
4.26 - 5.02	Integrate the full control system, including VisionPro and pressure sensor feedback. Complete mounting with the wheelchair.
5.03 - 5.09	Start to improve the system based on tasks and feedback.
5.10 - 5.12	Project wrap-up, documentation, and presentation.

Table 2: Project Schedule

6 Ethics and Safety

6.1 Safety

Robotic Arm Operation Safety: Our system includes a robotic arm extending from the rear of the wheelchair, which introduces potential risks if not properly designed. To avoid these risks, we implement the following safeguards:

Hardware/Software Safety: The arm will remain folded when inactive, ensuring it does not occupy additional space beyond the wheelchair and cause potential collision.

Speed Constraints: Arm motion speed will be limited to prevent high-impact collisions.

Safe Operation Limits: The Open Manipulator-P arm will be programmed to operate within predefined safety thresholds for users and bystanders. Specifically, it will avoid the space that the user occupies. Furthermore, Apple Vision Pro's depth and spatial awareness capabilities will be utilized to enhance situational awareness and prevent unintended interactions.

Privacy and Data Protection: User privacy is a critical consideration in our system, particularly given the use of real-time cameras and Mixed Reality (MR) technology. Our system does not store or transmit user data to any external servers. All video processing and interaction tracking occur locally. The rear-facing camera feed is processed in real time solely for user awareness and robotic arm control. Similarly, Apple Vision Pro's hand-tracking data is processed locally, without transmitting biometric or movement data beyond the device[3].

6.2 Ethics

6.2.1 Privacy and Data Protection

Aligning with IEEE/ACM principles, our system prioritizes user privacy:

Local Processing: No user data (camera feeds, hand-tracking biometrics) is stored or transmitted externally.

Real-Time Use Only: Rear-facing camera data is processed locally solely for arm control and user awareness [3].

6.2.2 User Autonomy and Accessibility

Inclusive Design: The MR interface offers intuitive controls, respecting human dignity[4], [5].

Safety in Design: The MR interface ensures frontal visibility is never obstructed during use.

Transparency: Users will be informed of system capabilities/limitations to manage expectations.

References

[1] ROBOTIS. "Openmanipulator-p - specification," ROBOTIS e-Manual. (Accessed: Mar. 14, 2025), [Online]. Available: https://emanual.robotis.com/docs/en/platform/openmanipulator_p/specification/#specifications (visited on 03/14/2025).

[2] U. Technologies. ""ROS-Unity Integration Tutorial"." (2023), [Online]. Available: https://github.com/Unity-Technologies/Unity-Robotics-Hub/blob/main/tutorials/ros_unity_integration/README.md (visited on 04/14/2023).

[3] Apple. ""Apple Vision Pro Privacy"." (2023), [Online]. Available: https://www.apple.com/privacy/ (visited on 04/12/2023).

[4] IEEE. ""IEEE Code of Ethics"." (2020), [Online]. Available: https://www.ieee.org/about/corporate/governance/p7-8.html (visited on 02/08/2020).

[5] ACM. ""ACM Code of Ethics and Professional Conduct"." (2018), [Online]. Available: https://www.acm.org/code-of-ethics (visited on 02/08/2020).