



**BRNO UNIVERSITY OF TECHNOLOGY**

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

**FACULTY OF INFORMATION TECHNOLOGY**

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

**DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA**

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

**SHADOW ROBOT:  
MANIPULATION OF A ROBOT MODEL  
IN AUGMENTED REALITY**

STÍNOVÝ ROBOT: MANIPULACE S MODELEM ROBOTA V ROZŠÍŘENÉ REALITĚ

**BACHELOR'S THESIS**

BAKALÁŘSKÁ PRÁCE

**AUTHOR**

AUTOR PRÁCE

**TIMOTEJ HALENÁR**

**SUPERVISOR**

VEDOUCÍ PRÁCE

**Ing. ZDENĚK MATERNA, Ph.D.**

BRNO 2024

# Bachelor's Thesis Assignment



150632

Institut: Department of Computer Graphics and Multimedia (DCGM)  
Student: **Halenár Timotej**  
Programme: Information Technology  
Title: **Shadow robot: manipulation of a robot model in augmented reality**  
Category: User Interfaces  
Academic year: 2023/24

## Assignment:

1. Research existing solutions for controlling robotic arms using augmented reality.
2. Familiarize yourself with AREditor and the existing robotic arm control options.
3. Design a solution allowing to quickly and easily manipulate a model of a robot instead of a real one.
4. Implement the proposed solution.
5. Perform user testing and evaluate whether your solution brings improvements (usability, speed, etc.).
6. Create a video presenting your work.

## Literature:

- Kapinus, Michal, et al. "Spatially situated end-user robot programming in augmented reality." 2019 28th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN). IEEE, 2019.
- De Pace, Francesco, et al. "A systematic review of Augmented Reality interfaces for collaborative industrial robots." *Computers & Industrial Engineering* 149 (2020): 106806.

Detailed formal requirements can be found at <https://www.fit.vut.cz/study/theses/>

Supervisor: **Materna Zdeněk, Ing., Ph.D.**  
Head of Department: Černocký Jan, prof. Dr. Ing.  
Beginning of work: 1.11.2023  
Submission deadline: 9.5.2024  
Approval date: 6.5.2024

## Abstract

This bachelor thesis is concerned with the manipulation of a robot model in augmented reality, using a tablet device. It extends user interaction of the existing application for robot programming – AREditor. When programming in AREditor, the user needs to set robot poses in order to define points in space, which the robot needs to reach during program execution. The existing solution of manually positioning the robot in AREditor only allows incremental stepping of the physical robot, which is, in the majority of use cases insufficient. The main goal is to design and implement an interaction scheme, through which the user can easily position a robot model, displayed on-screen over the real robot. The user first sets the pose of the model and then commands the physical robot to copy the chosen pose. The robot arm tracks the current position of a 3D gizmo, which the user can drag in space by moving the tablet, and its movement can be constrained to one of the axes or planes. The solution also offers movement sensitivity controls, a custom coordinate system, and a separate control for gizmo to camera distance. The proposed workflow streamlines the process of setting the pose of a robot and enables users to speed up the program creation.

## Abstrakt

Bakalárska práca sa zaoberá manipuláciou s modelom robota v rozšírenej realite za využitia aplikácie pre tablet. Riešenie rozširuje možnosti interakcie v existujúcej aplikácii na vizuálne programovanie robotov – AREditor. Užívateľ potrebuje pri programovaní v AREditore nastaviť pózu robota, aby mohol definovať body v priestore, na ktoré sa má robot dostať pri vykonávaní programu. Stávajúce riešenie v aplikácii umožňuje iba krokovat fyzického robota po malých inkrementoch pozdĺž osí, čo je nedostatočné pre väčšinu účelov. Hlavným cieľom práce je navrhnúť a implementovať systém, pomocou ktorého bude užívateľ schopný rýchlo a jednoducho nastaviť pózu modelu robota, ktorý je premietnutý na obrazovke cez skutočného robota. Užívateľ najskôr ľubovoľne nastaví pózu modelu robota, a následne pošle príkaz skutočnému robotovi, aby zvolenú pózu skopíroval. V riešení bolo využité 3D gizmo, ktorého polohu sleduje koncový bod robotického ramena. Gizmo je možné presúvať v priestore za využitia kamery tabletu a rozšírenej reality, a jeho pohyb je možné obmedziť na jednotlivé osi či roviny v priestore. Riešenie poskytuje možnosť ovládať citlivosť, s akou hýbeme bodom v priestore, možnosť zmeniť súradnicový systém, a samostatné ovládanie pre vzdialenosť gizma od tabletu. Vytvorené riešenie ponúka pohodlnejší spôsob, akým sa dá nastaviť poloha robota, a môže zefektívniť proces programovania.

## Keywords

augmented reality, user experience, robot, gizmo, axis, android, tablet, Unity, AREditor, ARCOR2

## Klíčová slova

rozšírená realita, užívateľská skúsenosť, robot, gizmo, os, android, tablet, Unity, AREditor, ARCOR2

## Reference

HALENÁR, Timotej. *Shadow robot: manipulation of a robot model in augmented reality*. Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Zdeněk Materna, Ph.D.

## Rozšířený abstrakt

S nástupom 4. industriálnej revolúcie sa roboty stali neoddeliteľnou súčasťou výrobného procesu vo väčších firmách. Najväčšou výhodou robota je jeho programovateľnosť, vďaka čomu môže byť prispôsobený na rôzne opakujúce sa úlohy. Programovanie robotov je však zdĺhavý proces vyžadujúci expertízu. Pre menšie a stredné firmy, ktoré produkujú široké spektrum produktov v menších kvantitách, to predstavuje bariéru, ktorá im bráni v efektívnom využití robotického ramena. Vývoj nástrojov na vizuálne programovanie sa stal predmetom mnohých výskumných projektov, ktorých cieľom je zjednodušiť a urýchliť proces preprogramovania robotického ramena. Táto práca stavia na tabletovej aplikácii AREditor, ktorá slúži na programovanie robotických pracovísk za využitia rozšírenej reality.

Programovanie v aplikácii AREditor typicky zahŕňa nastavenie robota do určitej pózy a zaznamenanie polohy koncového bodu ramena (tzv. end effectoru) v priestore, čím je vytvorený tzv. akčný bod. Na akčný bod sa môžu viazať rôzne akcie, ktoré má robot vykonať na danom mieste, a prepojením akčných bodov sa vytvára tok programu. Ak je robota možné ručne presunúť do ľubovoľnej polohy, je vytváranie akčných bodov jednoduchou úlohou. Ak však robot neponúka túto možnosť, je potrebné ho nastaviť do pozície v aplikácii. AREditor v súčasnosti ponúka menu na krokovanie ramena o zvolený inkrement pozdĺž jednotlivých osí. Toto riešenie nie je dostačujúce, ak je potrebné vykonať veľký pohyb, pretože by vyžadovalo veľký počet menších pohybov, čo je veľmi zdĺhavé a používateľsky neprívetivé. Táto práca sa zaoberá zlepšením tejto interakcie.

Cieľom tejto práce je navrhnúť a implementovať riešenie, pomocou ktorého bude užívateľ schopný rýchlo a jednoducho nastaviť pózu modelu robota, ktorý na obrazovke prekrýva skutočného robota. Aplikácia využíva rozšírenú realitu, a teda sa na obrazovke zobrazuje živý obraz z kamery, do ktorého sú dosadené objekty navyše. Užívateľ prvotne nastaví pózu modelu, a následne pošle signál skutočnému robotovi, aby zaujal identickú pózu.

Riešenie využíva 3D gizmo, ktoré je možné ťahať v priestore pohybom tabletu. Gizmo sa skladá zo stredobodu, troch šípok reprezentujúce jednotlivé osi, a troch štvorcov, ktoré reprezentujú roviny definované osami. Každý z vymenovaných prvkov je možné uchopiť a tak pohybovať gizmom iba v konkrétnej osi alebo rovine. Taktiež je možné pohybovať voľne v priestore uchopením stredobodu. Riešenie ďalej ponúka možnosť regulovať citlivosť pohybu, čím umožňuje užívateľovi hýbať bodom s vyššou presnosťou. Súčasťou riešenia je navyše možnosť prepnúť na relatívny súradnicový systém, ktorý sa viaže na polohu tabletu v priestore vo vzťahu k robotovi, a tým poskytuje možnosť hýbať bodom pozdĺž užívateľom definovaných osí. V neposlednom rade boli do riešenia zapracované dve tlačidlá na posúvanie bodu dopredu/dozadu.

Riešenie bolo implementované do AREditoru a následne bolo vykonané užívateľské testovanie v robotickom laboratóriu na fakulte informatiky VUT. Testovanie vykonávalo päť subjektov, ktorí jednoznačne zhodnotili, že predstavuje výrazné zlepšenie. Každý testujúci pochopil, ako používať menu do niekoľkých minút a bol schopný vykonávať jednoduché úlohy, ako napríklad nastavenie robota do špecifikovanej polohy či pohyb ramenom pozdĺž zadanej osi, napr. hrany stola.

Implementované riešenie ponúka novú možnosť manipulácie s robotom v rozšírenej realite. Výsledky testovania preukázali, že výsledný spôsob interakcie s robotom je intuitívny, prívetivý, a môže viesť k zvýšeniu efektivity pri vytváraní programov. Riešenie je ideálne na hrubé pohyby, ktoré bolo nepraktické vykonávať za pomoci pôvodného krokovacieho menu, ale pri znížení citlivosti je vhodné aj na precíznejšie doladenie polohy. Menu je možné používať samostatne, alebo v kombinácii s pôvodným krokovacím menu.

# Shadow robot: manipulation of a robot model in augmented reality

## Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Zdeněk Materna. I have listed all the literary sources, publications, and other sources, which were used during the preparation of this thesis.

.....  
Timotej Halenár  
May 6, 2024

## Acknowledgements

I would like to express my gratitude towards my supervisor, Ing. Zdeněk Materna, Ph.D. for his guidance, willingness to answer all of my questions, and valuable advice, through which he greatly supported me in my work.

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Related Work</b>	<b>4</b>
2.1	Wearable devices . . . . .	4
2.2	Handheld devices . . . . .	4
<b>3</b>	<b>ARCOR2</b>	<b>7</b>
3.1	Terminology . . . . .	7
3.2	API . . . . .	8
3.2.1	RPCs . . . . .	8
3.2.2	Events . . . . .	9
3.3	AREditor . . . . .	9
3.3.1	Editor screen . . . . .	10
3.3.2	Architecture . . . . .	12
<b>4</b>	<b>Design</b>	<b>14</b>
4.1	Movement . . . . .	14
4.1.1	Forward/backward buttons. . . . .	17
4.1.2	Sensitivity sliders . . . . .	18
4.1.3	User-relative coordinate system . . . . .	18
4.2	Component layout . . . . .	19
4.2.1	Invalid pose indication . . . . .	20
<b>5</b>	<b>Implementation</b>	<b>23</b>
5.1	Movement . . . . .	23
5.2	Gizmo . . . . .	26
5.3	Clipping shader . . . . .	28
5.4	Invalid pose indication . . . . .	29
5.4.1	Notifications . . . . .	30
5.4.2	Robot display . . . . .	30
<b>6</b>	<b>Usability testing</b>	<b>32</b>
6.1	Evaluation . . . . .	34
<b>7</b>	<b>Conclusion</b>	<b>36</b>
	<b>Bibliography</b>	<b>38</b>
<b>A</b>	<b>CD contents</b>	<b>40</b>

# Chapter 1

## Introduction

Industrial robots have become an inseparable part of the manufacturing process in large enterprises. In small to medium-sized enterprises (SMEs) where more types of products are manufactured in smaller batches, the adoption of robots has been slower. Robots have a high entry cost and are difficult and slow to reprogram. Having the ability to quickly and easily reprogram industrial robots would allow them to be used more flexibly, making them more viable for many SMEs and boosting productivity. The ARCOR2 system [10], along with the Android application, AREditor, aims to streamline the reprogramming process by utilizing augmented reality and a tablet device. However, the options for setting the pose of the robot in the app are limited and hinder its usability.

The ARCOR2 system is a framework for visual programming of robots developed by the Robo@FIT<sup>1</sup> research group. AREditor is a related tablet application, which serves as an interface to the system. When programming robots in AREditor, the user uses the robot arm to define points in space. The robot is moved to the appropriate pose and a point is recorded at the position of its end effector. Actions can then be assigned to these points, which when linked together, create the program flow. When working with collaborative robots, the operator can reposition the robot manually – by grabbing the robot arm and physically dragging it to the desired spot. The task of repositioning the robot arm becomes more tedious when the robot cannot be moved by hand. To move the robot arm, AREditor provides incremental stepping functionality, allowing the user to move the end effector by a selected increment in the direction of a chosen axis. This is sufficient for minor tweaking and fine-tuning but becomes too slow and impractical when a more substantial repositioning is necessary, as it requires multiple steps to achieve the desired pose. The operator may also find it difficult to anticipate where the robot arm will end up due to a lack of visualization. This work aims to improve human-robot interaction (HRI) within AREditor by creating a new solution for robot positioning.

This thesis proposes a novel interaction scheme for rapid positioning of a robot, employing a model robot rendered on-screen over the real robot in the workspace. Through this approach, the user can directly manipulate the model without affecting the robot, which allows for a safe interaction, unconstrained by the movement speed limit of the particular robot. Another benefit of this approach is the ability to visualize different poses before committing to a final one, thus saving time repositioning. The solution takes advantage of the AR nature of the application, utilizing tablet movement to move the end effector in space. The result is a functional component in the AREditor application, that can be used

---

<sup>1</sup><https://www.fit.vut.cz/research/group/robo/.en>

to easily manipulate the robot's pose, and could improve the efficiency of the programming process.

The thesis is structured as follows: In the first chapter, a brief summary of related research is presented. The second chapter contains a rundown of the ARCOR2 system and the related user interface AREditor. In the following chapters, the design and implementation of the final system is described. Finally, the results of the usability testing are presented and evaluated.



## Chapter 2

# Related Work

With the arrival of Industry 4.0 and rising interest in industrial robots, simplification of robot programming has been at the forefront for many researchers. Suzuki et al. [14] compiled 460 papers on AR-enabled human-robot interaction (AR-HRI), categorizing the works into various categories. The survey has shown that most of the research is focused on facilitating programming and supporting real-time control of robot workplaces. However, when it comes to devices used to interface with robots, most researchers favor on-body, wearable devices (especially head-mounted displays) over handhelds, such as tablets.

### 2.1 Wearable devices

Head-mounted displays (HMDs) provide an immersive experience and free up the operator's hands, allowing them to perform work while having crucial information always in their field of vision. However, the lack of a tangible device in the hands of the user raises the question of how they will interact with the workplace. Popular approaches include gestures [6, 12, 2], physical handheld pointers [7, 15], and less commonly voice commands [6]. Park et al. [12] proposed a hands-free system utilizing head gestures and gaze for selection and interaction. Chan et al. [6] created an interface allowing the user to set trajectory points on the work surface using a combination of gaze and voice commands. Ong et al. [11] combined a stereoscopic HMD with a physical handheld pointer, representing the end effector, which can be seen in Figure 2.1. The robot arm follows the position of the pointer in space. In this instance, the display is created by combining a VR headset and two webcams mounted directly onto it. A much more prevalent approach is to use a see-through AR headset, such as the Microsoft HoloLens. The SPEARED framework by Yigitbas et al. [15] uses HoloLens to display the current robot working state and detected objects, but also the current program in the form of code blocks, which can be modified in the AR environment. In the case of SPEARED, there is no direct robot pose manipulation, all interaction is conducted through code. Chan et al. [6] also utilize Microsoft HoloLens, but allow the user to directly set trajectories by using a combination of gaze and voice commands.

### 2.2 Handheld devices

While there have been works focusing on AR-enabled robot programming systems for handheld devices, they certainly represent a minority, especially concerning direct robot pose

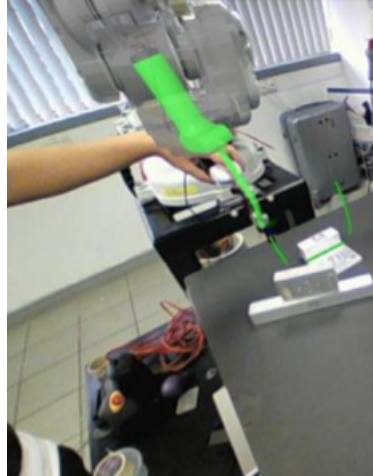


Figure 2.1: Physical handheld pointer used as position target for the end effector used in the work by Ong et al. [11].

manipulation. The interface often only allows the user to program 'pick and place' tasks, or the robot is programmed kinesthetically [5, 9].

Pertaining to controlling the robot pose explicitly, two main approaches can be discerned: individual joint setting and end effector location setting. Setting each joint rotation individually is not suitable for end users lacking experience with robotics, and even skilled operators rarely utilize it. Most implementations, however, provide both options. An earlier work by Frank et al. [8] developed a system to control a planar robot on a tablet device, which provides the user with the ability to control both end effector location and individual angles through tapping and dragging on the screen. Zhang et al. [17] provide the user with a set of arrows to control each joint separately, or to command the end effector directly, which can be seen in Figure 2.2. A similar approach was taken in some older works [1].

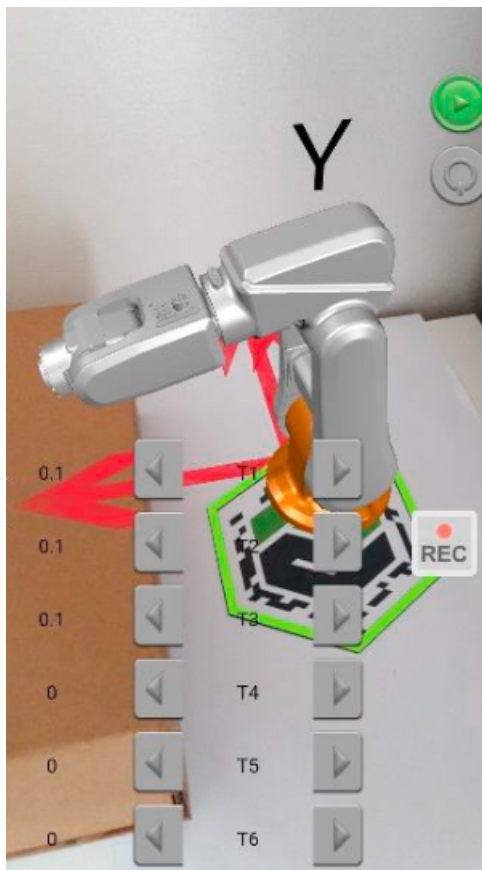


Figure 2.2: Mobile robot manipulation user interface by Zhang et al. [17].

# Chapter 3

## ARCOR2

ARCOR2 (Augmented Reality Collaborative Robot) is a framework for visual programming of industrial robots, that is actively being developed by the research group Robo@FIT. It enables end-users to create programs by defining points in space and program steps that are spatially visualized, allowing for an easy understanding of work processes. The system can generate code from visually defined procedures and facilitates control of its execution. Multiple robots, machines, and services can be included in a single workspace, with multiple concurrent users to be connected. New robots and services can be integrated as plugins by implementing an `Object Type` – a plugin into the system representing an interface between ARCOR2 and a real-world object, e.g. a type of robot [10].

The system is divided into the back end, comprising several independent services, and the front end (in this case, AREditor) [3]. The main service handling all incoming and outgoing communication is ARServer. ARServer is in charge of propagating messages to other services, holding system state, and sending out notifications upon changes. User interface implementations such as AREditor can interact with the system state through RPCs and receive asynchronous events.

### 3.1 Terminology

The following section is derived from [3].

- *Scene* – a scene contains action objects and their spatial relations. It serves as a representation of the workspace and defines its layout.
- *Project* – A project is associated with a scene and consists of action points. It may or may not include logic, which defines the program flow.
- *Execution Package* – A package is an executable snapshot of a project, that is entirely self-contained. When a project needs to be released into a production environment, an execution package is used.
- *Object Type* – A representation of a real-world object within the ARCOR2 environment, providing an interface to the object's functionality. By implementing an `Object Type`, new devices can be easily integrated into ARCOR2.
- *Action Object* – An instance of an `Object Type` within a scene.

- *Action Point* – A spatial point signifying a location for an action to take place. During program execution, a robot reaches defined action points in sequence and performs actions on them.
- *Action* – An action is a method of an `Object` Type, that can be performed by the robot.

## 3.2 API

Communication between ARServer and AREditor takes place through WebSockets. The communication protocol is based on RPCs and events, which are defined in the form of python dataclasses<sup>1</sup> and serialized into JSON when transported.

The dataclass representation of RPCs and events is used to create an OpenAPI<sup>2</sup> representation, which is subsequently used to generate C# models used in AREditor. In AREditor, each RPC is represented by a generated class in the `IO.Swagger.Model` namespace, and contains a method to serialize self to JSON.

### 3.2.1 RPCs

To each RPC (Remote Procedure Call) sent by AREditor, the server elicits a response. The response contains the resulting data. If the request fails, the data field is empty, and a message informing of the error is present. The following segment presents an example of a successful request-response sequence:

RPC request:

```
InverseKinematics.Request(
    id=28,
    request='InverseKinematics',
    args=InverseKinematics.Request.Args(
        robot_id='obj_edc94797032d410289f7cd241b5c6fe8',
        end_effector_id='default',
        pose=Pose(...),
        start_joints=[...],
        avoid_collisions=True,
        arm_id=None))
```

RPC response:

```
InverseKinematics.Response(
    id=28,
    response='InverseKinematics',
    result=True,
    messages=None,
    data=[...])
```

Note that the result field is `True`, and there is no message. Had the request failed, the response would look as follows:

<sup>1</sup><https://docs.python.org/3/library/dataclasses.html>

<sup>2</sup><https://www.openapis.org/>

```
RPC response:
  InverseKinematics.Response(
    id=29,
    response='InverseKinematics',
    result=False,
    messages=['arcor2 (General): Failed to compute IK.'],
    data=None)
```

In AREditor, RPC failure responses are usually handled as exceptions, and notifications are created and displayed.

### 3.2.2 Events

The main purpose of events is to notify connected user interfaces of asynchronous events. Events are an implementation of the observer design pattern<sup>3</sup> and are used to keep all connected interfaces synchronized with the system state. Notifications are usually sent after an RPC has been processed or when an object in the scene has changed, e.g. a robot has moved. The pattern then becomes:

1. RPC request
2. RPC processed; response sent
3. Event broadcasted

A typical use case can be observed in robot stepping. The user sends a request to move the end effector, the end effector is moved, and after the movement has finished, an event is sent to notify AREditor and all other connected user interfaces of the changed robot pose.

## 3.3 AREditor

AREditor is a mobile application used to interface with ARCOR2. Its main goal is to enable users with little or no programming and robotics experience to create programs within a robotic workspace easily. AREditor is currently the only implementation of an ARCOR2 interface, providing access to the full ARCOR2 toolset [3].

From the application, users can create and modify scenes, projects, and execution packages, add action objects to the workspace, manipulate robot poses, define action points and actions, and create program flows by linking actions together. The program flow can be executed and monitored from within the application as well. AREditor can operate in two modes, offline and online. In offline mode, the scene is stopped and robot actions such as setting the pose are not available. In online mode, robot actions become available and interactable, which allows the user to e.g. set the pose of the robot through the robot stepping menu.

After connecting to ARServer, the user is presented with the main screen, from where they can create or open scenes, projects, and packages. When a scene (project, package) is opened, the application switches to the editor where the workspace can be viewed in augmented reality.

Creating an executable program in AREditor involves the following steps:

---

<sup>3</sup>[https://en.wikipedia.org/wiki/Observer\\_pattern](https://en.wikipedia.org/wiki/Observer_pattern)

- *Connect to server* – A user must first connect to the server where ARServer is running by filling out the IP address, port (default is 6789), and username. Multiple users can connect to the same instance of ARServer, provided they choose different usernames.
- *Create a scene* – To set up a workplace, a user must create a scene to represent the given workspace and insert and position action objects in accordance with the real layout of the physical workspace. When adding a robot, the user must fill out the URL of the robot. A robot added to the scene can be moved and rotated using the Object manipulation menu.
- *Create a project* – Each project is tied to a scene. When editing a project, the user adds action points, which are later used as anchors for actions for the robot to perform.
- *Define actions* – User defines actions for the robot to execute at specified action points.
- *Define flow* – Actions are linked together to create program flow.
- *Program execution and monitoring* – The user can begin execution of the program and stop it on demand. The visual representation of the program can be exported into a script.

### 3.3.1 Editor screen

The editor is the AR-enabled application part, where users can interact with the workspace [4]. Available menus are listed on the left side of the screen, organized into groups, and accessible through tabs: Favorites, Add, Utility, Home, and Robot. The right side of the screen is occupied by the selector menu by default, which provides a list of action objects in the workspace. When a different menu is selected from the list, the selector menu is hidden and replaced by the corresponding menu, as pictured in Figure 3.1.

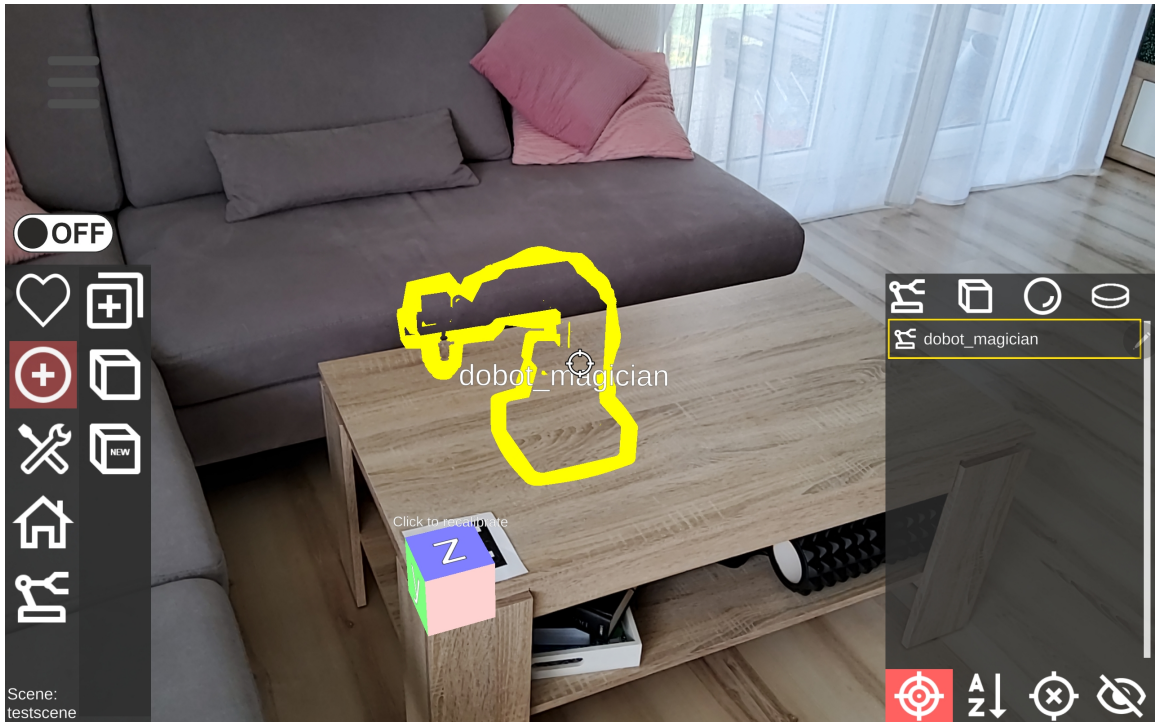
The editor is designed to be used while holding a tablet with two hands. This approach introduces some limitations but helps prevent fatigue, often present when holding a tablet with a single hand for prolonged periods. Widgets are located on either side of the screen to allow for good visibility of the workspace, as well as to make the individual buttons accessible by thumbs.

### 3D sight

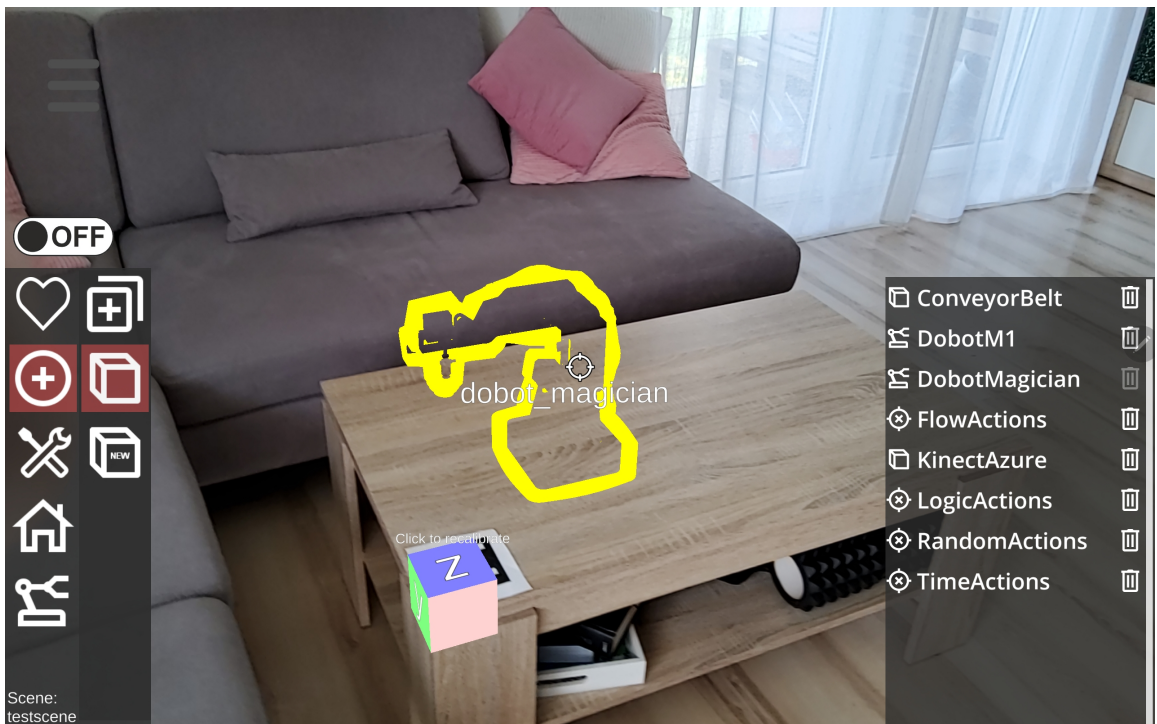
The sight works as a de facto cursor. It is located in the middle of the screen and is the main interaction method between the user and action objects. AREditor perpetually casts rays from the center of the screen forward and checks for collisions with action objects in the scene. By aiming the crosshair at an action object, the object is selected: it is highlighted in the selector menu, and an outline is displayed around it, as seen in Figure 3.1. When using the stepping menu or the object manipulation menu, the crosshair is used to select the axis of movement.

### Object manipulation menu

The object manipulation menu is used when adding action objects to a scene. The menu provides tools for translation, rotation, and scaling of the object. When the menu is activated, a three-axis gizmo appears. The gizmo axes can be selected by aiming at them with



(a) Selector menu, the default.



(b) 'Add object' menu opened.

Figure 3.1: Editor screen.



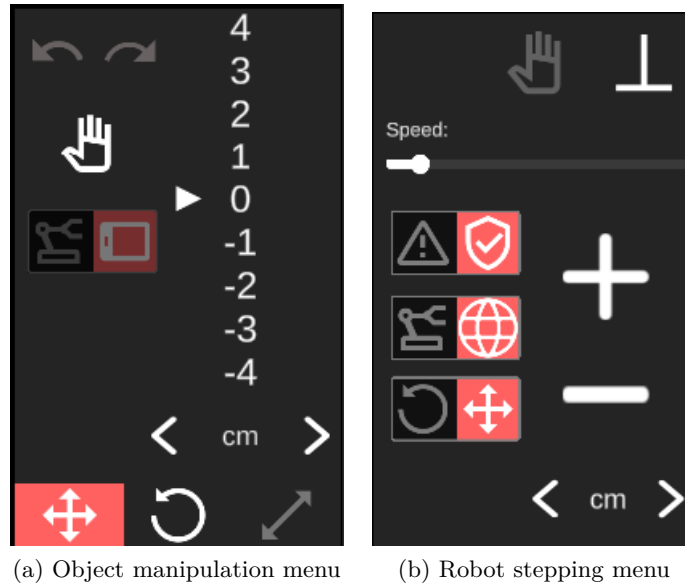


Figure 3.2: Object manipulation and robot stepping menus.

the sight. By pointing the sight at a particular axis, it becomes selected, after which the transform wheel can be used to move (rotate, scale) the object along said axis. The user can also select the unit of the transform wheel, as well as move the object freely by holding the 'hand' icon and moving the tablet. The menu also offers undo and redo capabilities. When creating a scene, the object manipulation menu is used to align action objects with the real-world objects in the workspace.

### Robot stepping menu

When working with non-collaborative robots that do not support physical repositioning, the robot stepping menu is the main way of manually setting the pose of a robot within AREditor. The appearance of the menu is similar to that of the object manipulation menu, but the transform wheel is replaced by plus and minus buttons, which are used to incrementally move the end effector along a specified axis. By stepping the robot, the end effector is moved in the chosen direction by the selected unit.

When a step is initiated, the buttons are disabled until the movement is concluded. While necessary to make the menu safer to use, this presents a limitation in usability. If the robot needs to be moved to a vastly different position, it would require several steps, taking a long time. The menu is best utilized when doing small adjustments to the end effector position, which is also its intended purpose, as reflected by the increments that are available to the user (0.1 mm, 1 mm, 1 cm, 5 cm, 1 dm).

The robot stepping menu also offers two more toggles, a safe/unsafe movement switch and a global/local space switch. Both menus can be seen in Figure 3.2.

### 3.3.2 Architecture

Each individual menu in AREditor is represented by a prefab with a custom controller attached. Aside from the menus, AREditor comprises several singleton classes handling the most important functionality such as communication with ARServer, holding the ap-

plication state, and managing the active scene or project. They handle incoming RPC responses, notifications and provide events, to which the individual screens and menus can dynamically subscribe.

**GameManager** is the main controller of the application. It manages different screens, holds the application state and the connection status, notifies users of changes, and builds and runs execution packages. Connection to ARServer is created here, using **WebSocketManager**.

The **WebSocketManager** is responsible for communication with ARServer. It provides methods for creating a connection, creates and sends RPCs, holds the WebSocket context, processes received data, and broadcasts events, informing subscribed menus of changes on connection status, robot pose, action objects addition/removal, etc.

For outgoing data, each type of RPC request is represented by a class, generated from ARCOR2 dataclasses by the OpenAPI Generator CLI<sup>4</sup>. When an RPC is being sent, the appropriate class is instantiated, and the data is serialized and sent through the WebSocket. The response data are awaited and deserialized using either a corresponding class or an anonymous type<sup>5</sup> and based on the `@event` field value in the response, the corresponding event is dispatched.

---

<sup>4</sup><https://openapi-generator.tech/>

<sup>5</sup><https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/anonymous-types>

# Chapter 4

## Design

In this chapter, I will go over the design requirements for the proposed improvement of the robot interaction in AREditor, and the individual elements of the final design.

When programming robots using the ARCOR2 system, a user typically sets action points at the position of the end effector. An action point is a spatially anchored container, that holds a position and orientation [3]. Actions can then be assigned to action points and program flow is created. The user, therefore, needs to be able to quickly and easily position the robot end effector to the applicable point in space so that they can set an action point there. The design should allow the user to make large movements of the robot arm in a single pass (as opposed to the robot stepping menu, where each movement is made up of several small movements), but also provide a reasonable degree of precision.

In most previous works on mobile devices, the explicit pose setting of a robot was resolved using on-screen arrows. While this approach works and was briefly considered, a better solution exists. In AREditor, selecting (and moving) objects in the object manipulation menu is already done by utilizing rays cast from the camera, aimed at a chosen object within the scene. This principle can be applied to robot positioning, as described in the next section.

The goal of this design was to provide flexibility while not obscuring the screen space. This is a problem often seen in mobile AR interfaces. Another aim was to make the interaction scheme feel natural and easy to learn, making it accessible to end-users of varied levels of experience with tablets or robotics.

### 4.1 Movement

Two approaches to robot pose setting were considered: moving individual joints, and moving the end effector only (utilizing inverse kinematics to calculate the joint values). The first approach was deemed unnecessary, as it is rarely needed. In the vast majority of use cases, setting only the end effector position is sufficient, as the robot is capable of calculating the joint configuration based on a position in space. Secondly, as seen in [18] or [9], giving the user access to manipulating joints individually could result in a high number of buttons, obscuring the screen and complicating usage.

By shifting focus from individual joints to the end effector, the problem can be reduced to a simple point moving in 3D space, and axes can be drawn from the end effector to facilitate movement in all directions. The individual axes serve as handles, which can be

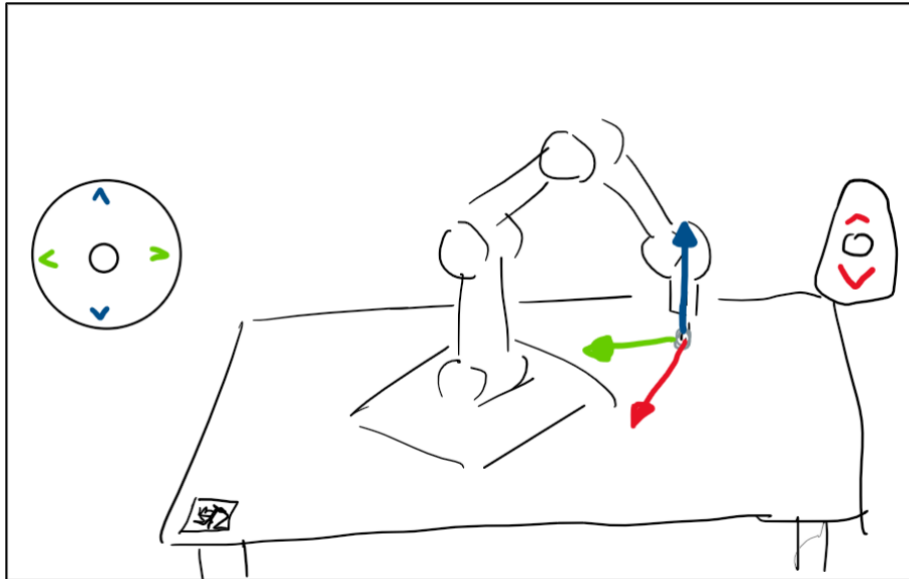
selected with the sight. Early designs featured arrows or joysticks on-screen to be used to move the end effector along the selected axis (see Figure 4.1).

The interface can further be simplified by utilizing tablet movement tracking. After having selected the axis, the point can be simply moved by moving the tablet. The second design iteration featured a single button for axis selection and arrows (joysticks) were abandoned, as shown in Figure 4.2 (a). In this variant, the option to move the point in space freely was added. Instead of selecting one of the axes, the user can select the origin of the gizmo to enable unconstrained movement (Figure 4.2 (b)). Furthermore, inspired by 3D software such as Fusion 360<sup>1</sup> or Godot Engine<sup>2</sup>, the gizmo was modified to include additional handles in the form of planes, which allow movement constrained to two axes at a time. This option could be used when the end effector must be moved to a different location while maintaining its height, for example.

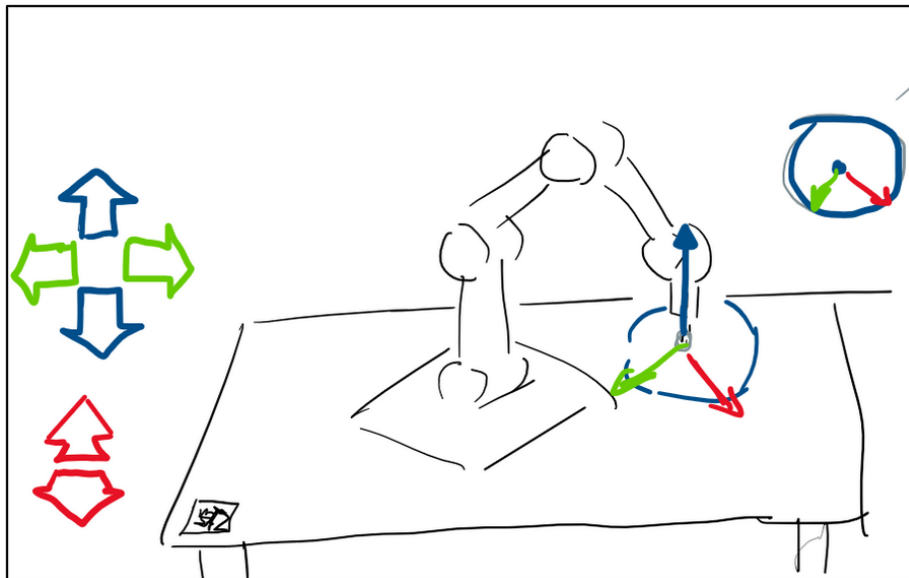
---

<sup>1</sup><https://www.autodesk.com/products/fusion-360/>

<sup>2</sup><https://godotengine.org/>

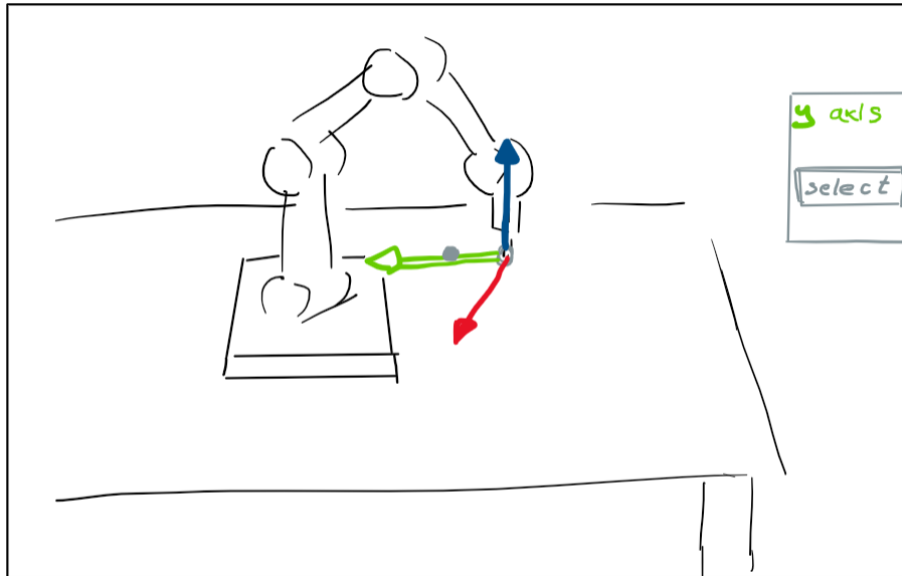


(a) Design with two sets of joysticks, to control all three axes.

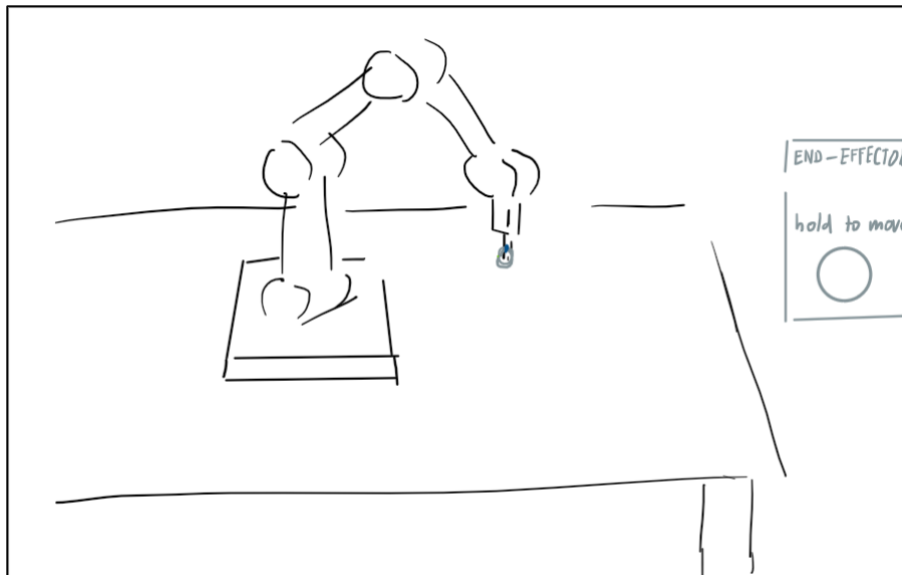


(b) Arrows replace joysticks, and a dial to set the gizmo rotation.

Figure 4.1: Early designs with on-screen touch controls.



(a) Selectable axes



(b) Selectable end effector

Figure 4.2: Design variants employing tablet movement. These were later combined into singular design.

#### 4.1.1 Forward/backward buttons.

To facilitate movement in the direction from the user to the robot, an additional control was added. To move the end effector forward, one would have to move the tablet forward the same amount, which proved to be impractical in some scenarios. In order to allow the operator to keep the same distance from the robot throughout the session, a different way to move the end effector along said axis was necessary. A dual button control was added, featuring slanted arrow icons to illustrate the forward/backward direction of movement (Figure 4.3).

The original intended functionality of the buttons was to move the point along a line defined by the tablet position and the direction the camera is facing. Through testing, it was discovered to be frustrating to use, as the point would move in a slight downward/upward direction, because there was always a height difference between the tablet and the point position. During the implementation phase, the buttons' effect was modified to only affect the point's position in the horizontal plane, ignoring the vertical axis. This change reflected the usual intended use of the buttons better.



Figure 4.3: Forward/backward buttons.

#### 4.1.2 Sensitivity sliders

When dragging the point, the end effector position moves to the exact position of the raycast at a specified distance. This is the most intuitive outcome, as the user can always predict, where the end effector will end up, but positioning with higher precision can be cumbersome. By introducing variable sensitivity, the user can modify the ratio of tablet movement to end effector movement, and thus make more precise adjustments easier.

Another issue is distance. When standing farther away, the same tablet movement results in a larger robot movement. For example, when standing 1.5 meters away and rotating the tablet by  $15^\circ$ , while having the X axis selected, the end effector will move by 40 centimeters. If the operator stood a meter farther, the end effector would move by 66 centimeters, which is illustrated in Figure 4.4. Having adjustable sensitivity can therefore be useful in scenarios, where the operator has to stand far away. The slider value acts as a distance multiplier and is by default set to 1, which moves the end effector to the exact final raycast hit and can be lowered to shorten the distance traveled by the robot arm.

Similarly, a slider for the forward/backward buttons was added, to control the speed of the movement.

#### 4.1.3 User-relative coordinate system

Often when positioning the robot, the user could benefit from being able to move the robot in a straight line along an arbitrary axis in space, e.g. to align it with a different object in the scene. The user could also want to move the robot arm perpendicular to their own current position in space.

To achieve this functionality, two approaches were examined. Initially, a dial for rotating the gizmo around its vertical axis was considered (Figure 4.1(b)), which would be used by manually dragging using a thumb. While this could work, it would require the user to focus on setting the rotation of the gizmo first, before being able to move along desired axis. It

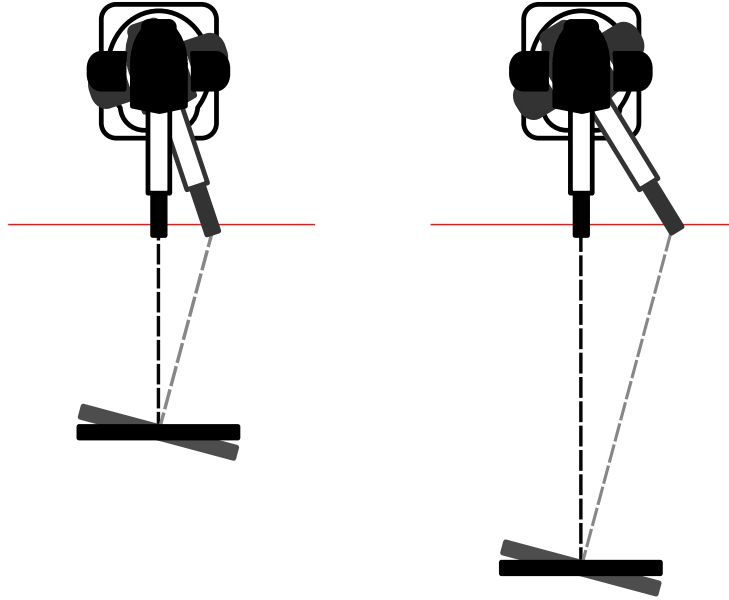


Figure 4.4: The same tablet rotation from different distances.

would also add another control to the screen. A more intuitive approach is to make the gizmo rotate relative to the tablet position. In this scenario, the gizmo rotates as the user moves around the workplace, with one axis staying perpendicular to the camera position and the second one oriented towards the camera position, which is illustrated in Figure 4.5. Another addition is a lock button, which allows the operator to lock the gizmo in its current rotation. This solution enables movement along arbitrary axes, which can be aligned with other objects in the workplace by simply walking toward them while maintaining the repeatability of movements provided by the lock button.

## 4.2 Component layout

The user interface was designed as a set of several separate submenus: the main menu, forward/backward buttons, and a confirmation dialog. An 'invalid pose' notification was also included in the design.

The main menu's central component is the select button, but it also contains a label indicating the current selection, a sensitivity slider, and the coordinate system switch. The select button has a label describing the functionality, which is hidden when dragging to indicate that the movement is active. The coordinate system switch has two switchable states, *world* and *user*, through which the user-relative coordinate system can be enabled. When the *user* state is selected, a lock button appears, which locks the gizmo's current rotation.

The layout follows previously established conventions in AREditor. The main menu is located on the right side of the screen, with the confirmation dialog beneath. The main



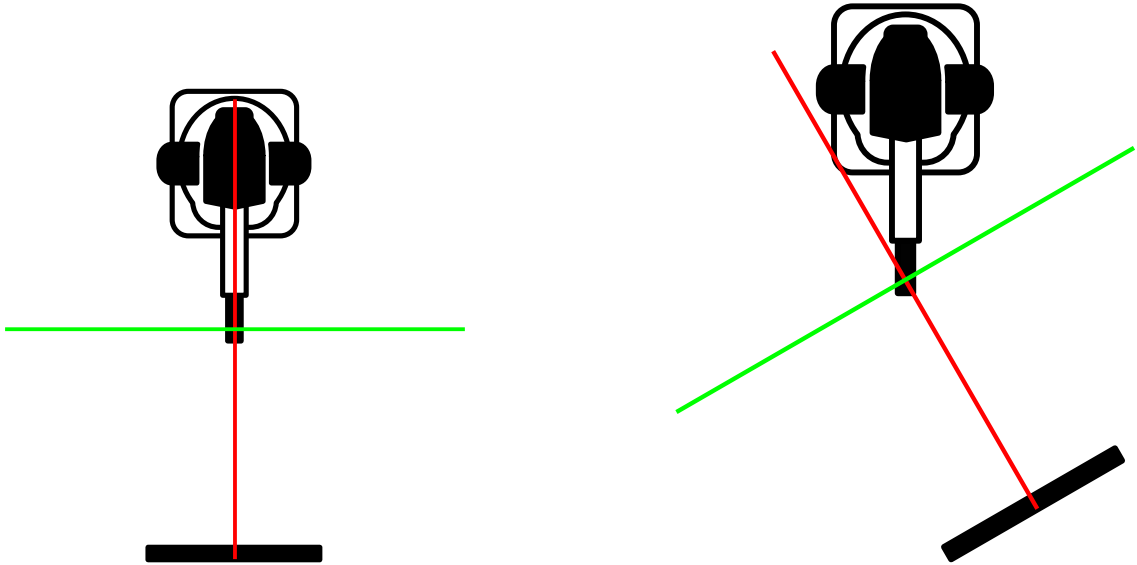


Figure 4.5: The user-relative coordinate system demonstration.

menu is always visible, the other menus are turned on and off as required. If the pose of the robot model is the same as the real robot, the confirmation dialog is not visible. It only appears, when a movement robot model is executed, and is only enabled, when the user concludes the movement. The confirmation dialog contains two buttons, *apply* and *revert*, through which the operator can send the pose to be applied to the real robot, or revert the model to the default state (copy the pose of the real robot). Additionally, while dragging, the left menu along with the online/offline switch are hidden. The layout of the menus can be seen in Figure 4.6.

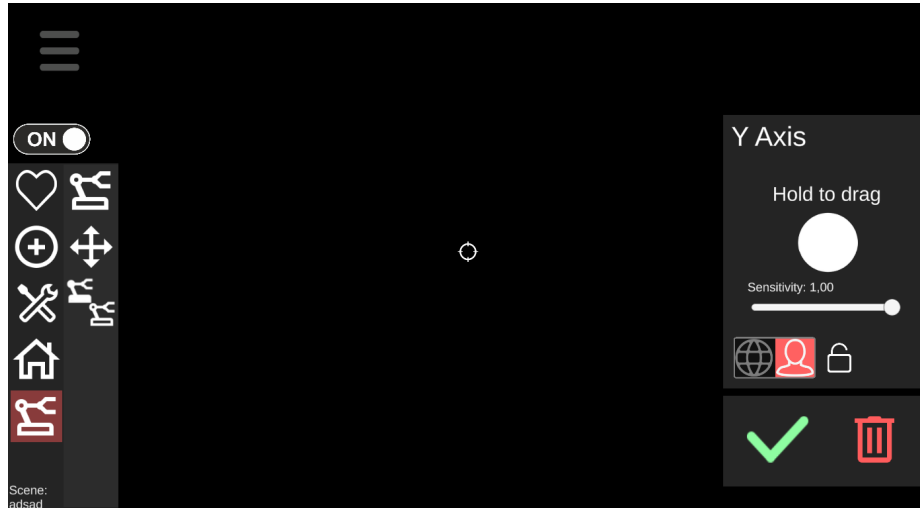
#### 4.2.1 Invalid pose indication

To inform the user that the pose is out of bounds, visual indicators were included in the design. The original notification system was insufficient for the purposes of this solution for two reasons: readability and performance, the first of which will be characterized in this section. Performance is discussed in the *Implementation* chapter.

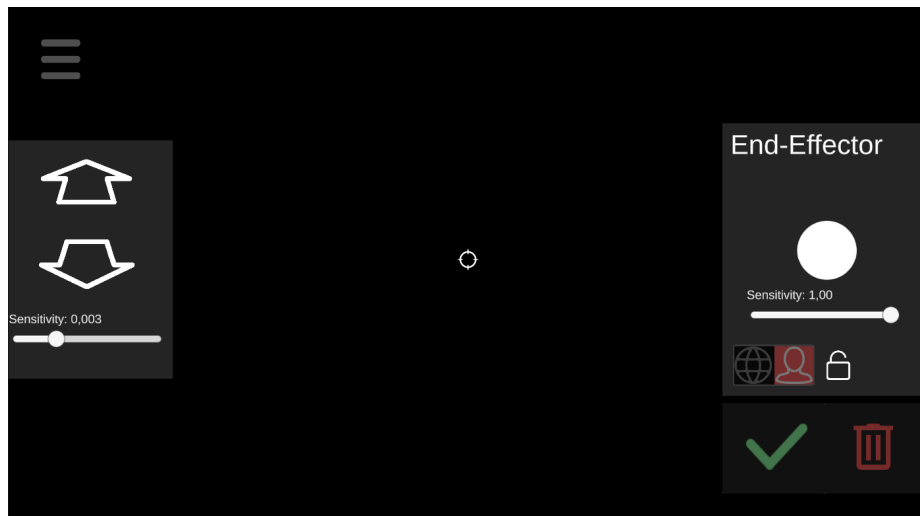
Notifications in the original notification system are implemented as temporary messages in the top right corner of the screen. A notification is presented in the form of a semi-transparent grey box, which contains text. If the notification informs of a failed RPC request, it usually contains a title and the error message from ARServer. The notification fades in, is displayed for a brief period, and then fades out.

This work entails a different form of notification. The user has to be notified the moment an impossible pose is reached, and vice versa, when the point is back in a valid position, for quicker understanding of the current state. The message should also be simpler, as there is no need to display the full RPC response message.

The new notification is a red box, containing the text 'Invalid pose' written in large, white text. The RPC response message was removed. It appears immediately after the user moves the end effector beyond the reach of the robot, and is hidden once the end effector is back in the work envelope of the robot arm. A comparison between the old and the new notifications can be seen in Figure 4.7.

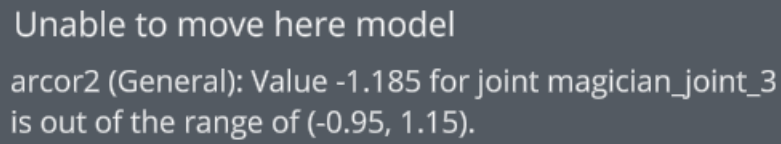


(a) The layout when not moving.



(b) The layout when moving the end effector.

Figure 4.6: The layout of menus.



Unable to move here model  
arcor2 (General): Value -1.185 for joint magician\_joint\_3  
is out of the range of (-0.95, 1.15).

(a) The old notification.



Impossible pose

(b) The new notification.

Figure 4.7: A comparison of the old notification and the new notification.

An additional visual cue was added to further indicate, that the robot is in an unreachable pose. Upon leaving the work envelope, the robot model is displayed in red. A more elaborate, albeit similar approach was taken in [11], where the overlaid robot model arm is displayed in colors from green to red, based on the manipulability of the robot. The system calculates manipulability based on a scale proposed by Yoshikawa [16]. In the ARCOR2 system, no such manipulability scale is implemented and is out of the scope of this work, therefore a simpler, yet still effective, approach was chosen.

# Chapter 5

## Implementation

The system was integrated into AREditor as a new menu in the 'Robot' category titled Model positioning menu. The menu is a single prefab, with the main part of the solution being the attached `ModelPositioningMenu` controller. In order to use the menu, the scene must be in online mode, and a robot and end effector must be selected. By opening the menu, the robot model is displayed in full opacity, and the `SceneManager` is temporarily unsubscribed from the events `RobotEefUpdated` and `RobotJointsUpdated`. By doing so, the robot model is disconnected from the real-world robot and its pose can be manipulated independently. When the menu is closed, the connection is restored.

### 5.1 Movement

Upon opening the menu, a 3D gizmo is instantiated in the position of the end effector. Each of the gizmo's axes and planes, as well as the center point of the gizmo, serve as a handle, which can be selected and dragged by moving the tablet. Raycasting is utilized to track the tablet's movement in space:

1. When the user starts dragging, a ray is cast from the screen's center and a point along the ray is recorded at a specified distance (`originalPosition`).
2. During every frame while dragging, a new ray is cast and the `targetPosition` is recorded.
3. The difference between `originalPosition` and `targetPosition` serves as the translation vector.

Originally, the collision point between the ray and the handle was used as the `originalPosition`. While this may seem like the more appropriate approach, it would require the user to be pointing directly at a handle when initiating movement, requiring additional precision. This made starting the movement unreliable and frustrating. An attempted solution to this problem was to use the last recorded collision between the handle and the ray as the starting position, but that created a problem where the gizmo would snap into position when the movement was initiated, as the `originalPosition` and `targetPosition` were already in two different locations at the beginning of the movement.

By defining the original position as a point along a ray in the current frame, the starting point of the translation is in the exact spot the camera is presently facing, eliminating initial snapping to position, as well as the need to aim directly at the handle.

Raycast collisions are still used to select the handle. When a collision is reported, the length of the ray from the camera to the handle is recorded, which later serves as the distance of the `originalPosition` and `targetPosition` along the ray. In Figure 5.1, a visual description of how a movement is projected onto an axis can be found.

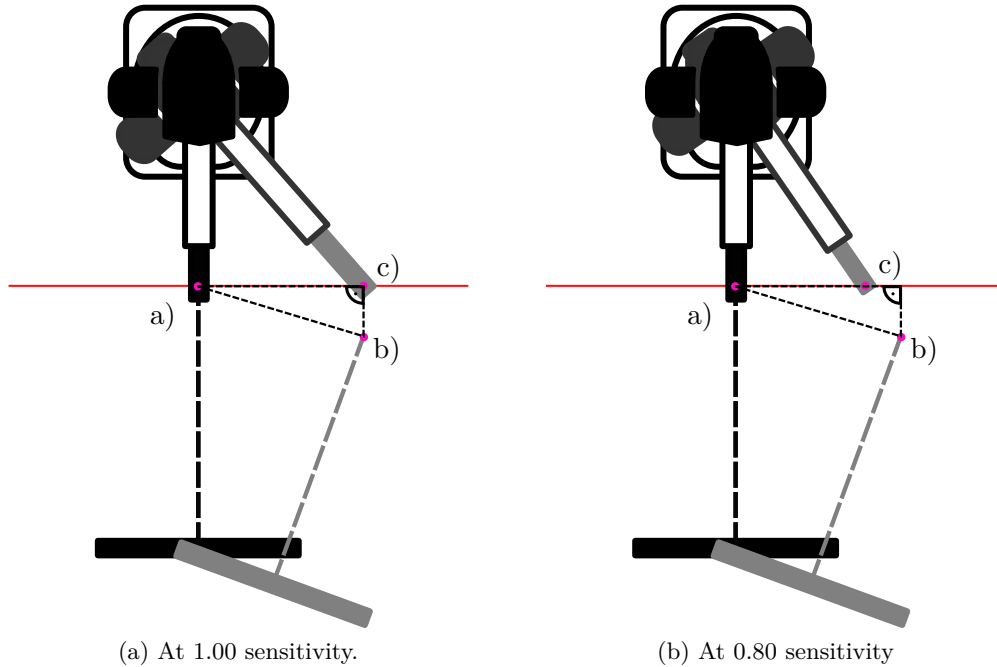


Figure 5.1: An illustration of movement along the X-axis. a) original point position, b) target point position, c) final end effector position.

The translation vector is transformed into the local space of the gizmo using the Unity method `Transform.InverseTransformVector()` and is first applied to a dummy `GameObject`. In order to gain access to an empty `Transform`, creating an empty `GameObject` first is necessary because Unity does not allow `Transforms` to be instantiated if they're not connected to a `GameObject`. The dummy's `Transform` is reset at the beginning of every frame; the position is set to the original point position pre-translation, and the rotation is set to the gizmo's rotation. Using the Unity method `Transform.Translate()`, the translation is applied, either unconstrained or only on selected axis/axes, depending on the selected handle. By using the `Translate()` method, the translation is applied relative to the object's local axes, taking the current rotation of the gizmo into account, which enables the user-relative coordinate system to function by simply rotating the gizmo to face the user.

Forward/backward buttons' functionality is facilitated by moving the target point position in world space. During every frame, if either the forward or backward button is held, the direction vector from the camera to the target point is obtained and normalized, multiplied by the current sensitivity multiplier, and added to a `Vector3` variable, `forwardBackwardAdjust`. Originally, the entire `forwardBackwardAdjust` was then added to the target point, but it was later changed to only add the X and Z coordinates. This constrains the effect of the forward/backward buttons to the horizontal plane, which was discovered to be a more user-friendly approach. The variable is reset when the movement is completed.

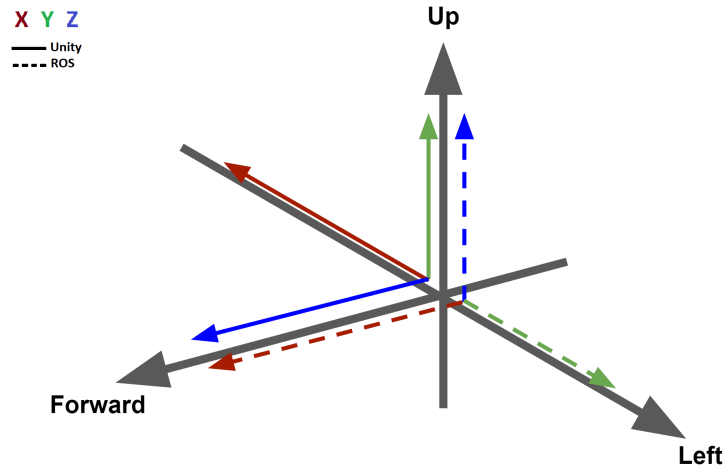


Figure 5.2: Unity vs ROS world coordinate systems [13].

### Calculating pose

The `MoveHereModel()` method is responsible for sending inverse kinematics requests. While a movement is active, the `MoveHereModel()` method is continually called from the `Update()` function. The method is asynchronous and is only called if the previous call has finished, which is achieved by using a flag. This also prevents overlapping requests from being sent to `ARServer`. The target position, represented by a `Vector3`, is processed in three steps:

1. Because the Unity world space is not equivalent to the scene space, the point is transformed from world space to the local space of the scene using `Transform.InverseTransformPoint()`.
2. The point is converted from the Unity coordinate system to the ROS<sup>1</sup> coordinate system. Unity uses a left-handed, with a y-up world coordinate system. Ros is slightly different, as it uses a right-handed, with a z-up world coordinate system [13]. In Figure 5.2, the difference between Unity and ROS coordinate systems is displayed.
3. A `Position` is created from the `Vector3` representation, which casts the float values to decimal. Decimals are more appropriate when dealing with small values, as the system handles them more accurately<sup>2</sup>.

### Maintaining unconfirmed pose

When the menu is closed before a pose is applied, the pose is logged and reapplied on the next open. When the menu is first opened, the controller checks, whether the variable for holding the last confirmed position has been set. If it has not been set yet, it means, that the menu is opened for the first time, and the pose of the model should be set to the pose of the real-world robot. If it has been set, it compares the value to the current position of the end effector of the real-world robot. If the values are not equivalent, the model pose

<sup>1</sup><https://www.ros.org/>

<sup>2</sup><https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types>

was not applied before closing the menu, and should therefore be restored. The model is moved to the unapplied position and the confirmation dialog is enabled. When a scene is closed, the unapplied poses are reset.

## Multiple robot support

The menu provides the illusion of a separate session for each robot in the workspace. This is done by maintaining a dictionary of last point positions, where the keys are robot IDs. Upon opening the menu, a value retrieval is attempted using the current selected robot's ID as the key. If the key does not exist in the dictionary, it indicates that the menu has not been opened before for the selected robot. When the menu is closed, the dictionary entry is created or updated. The method `Dictionary.TryGetValue()` was used to avoid having to catch exceptions, as that would often be the case<sup>3</sup>.

## 5.2 Gizmo

The 3D gizmo used in this system is a variant<sup>4</sup> of the gizmo prefab used in other menus, such as `RobotSteppingMenu`. The gizmo was modified by removing distance labels and adding planes, as well as a custom controller, `GizmoVariant`, which provides helper methods for highlighting planes and axes, as well as flipping the gizmo around the X, Y, and Z axes. Upon opening the menu, the gizmo is instantiated as a child of `DraggablePoint`, which is located in the gizmo origin and is used as a handle for dragging in all three axes.

When an axis (plane, point) is selected, the movement can be initiated by pressing and holding the main button. To signify which handle has been selected, all unselected axes and planes are gradually hidden by manipulating their scale, and the selected axis or axes are lengthened. The effect is displayed in Figure 5.3. If the `DraggablePoint` is the selected handle, all axes are hidden. A coroutine was used to gradually change the scale of a chosen axis across several frames<sup>5</sup>. The advantage of using a coroutine is that they can be started and stopped, instead of perpetually checking the state every frame. This makes them better suited for infrequent tasks, such as scaling the axes upon initiating movement. Another benefit is cleaner code in the `Update()` method.

```
//the parameter "scale" is the target scale of the axis
private IEnumerator AxisScale(GameObject axis, Vector3 scale) {

    //coroutine is concluded once the scale is within a~threshold
    while (Vector3.Distance(axis.transform.localScale, scale) > 0.01f) {

        //linear interpolation is used to smooth the animation
        axis.transform.localScale = Vector3.Lerp(axis.transform.localScale,
            scale, 0.25f);

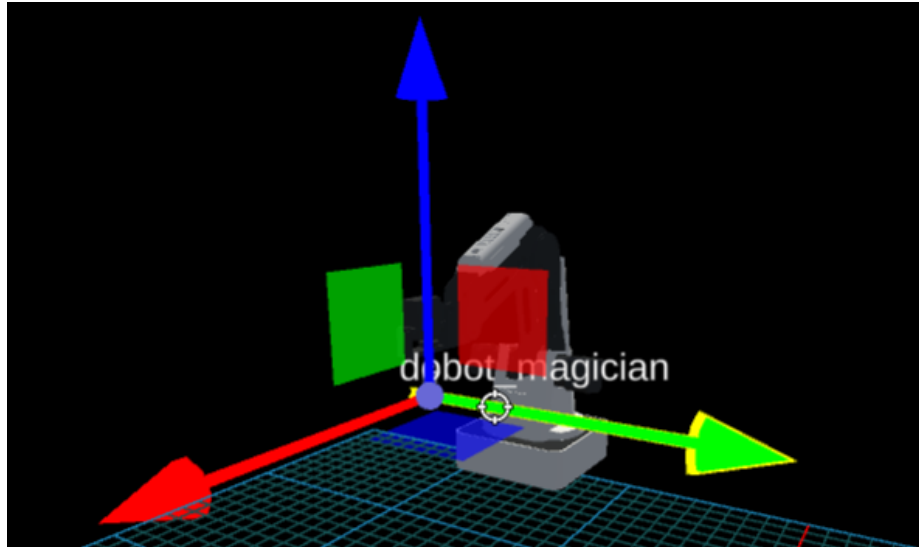
        //return control to Unity, and continue in the next frame
        yield return null;
    }
}
```

<sup>3</sup><https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2.trygetvalue?view=net-8.0>

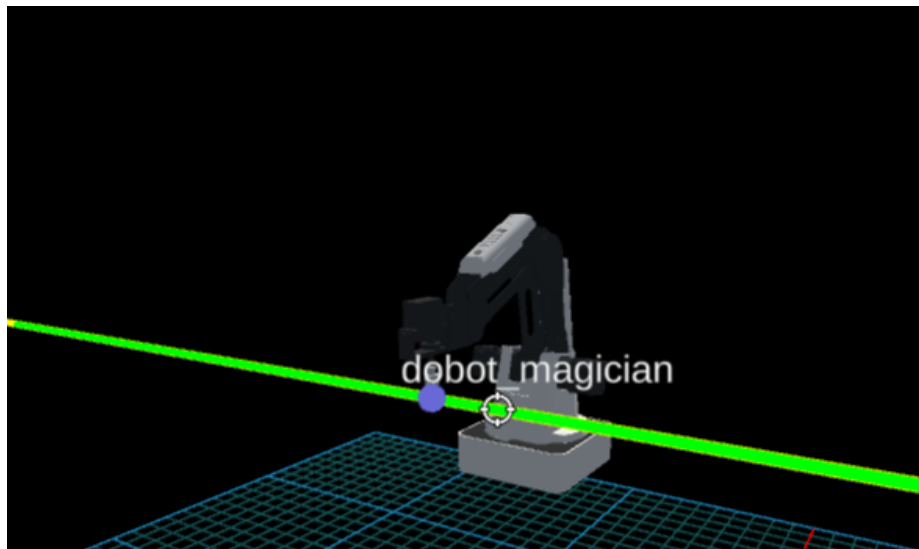
<sup>4</sup><https://docs.unity3d.com/Manual/PrefabVariants.html>

<sup>5</sup><https://docs.unity3d.com/Manual/Coroutines.html>

```
}  
  
//after the threshold is crossed, the final scale is applied directly  
axis.transform.localScale = scale;  
}
```



(a) Gizmo while not moving, with the Y-axis selected.



(b) Gizmo while moving along the Y-axis.

Figure 5.3: The gizmo.

The typical use case is that of a user walking freely in the workspace, looking at the robot from different sides. To allow easy access from all sides, the gizmo is flipped on the corresponding axis, if the camera moves across it. This is done by transforming the position of the camera from world space into local space of the `DraggablePoint` using the Unity method `Transform.InverseTransformPoint()` and checking the coordinates



for positive/negative values. The flip itself is achieved by simply multiplying the chosen coordinate value by -1.

When the user-relative coordinate system is enabled, the gizmo's X-axis always faces the camera. The Unity method `Transform.LookAt()` was used, and the rotation was limited by modifying the target's Y-coordinate:<sup>6</sup>

```
//Y coordinate set to self to constrain rotation
Vector3 targetPosition = new Vector3(
    Camera.main.transform.position.x,
    pointInstance.transform.position.y,
    Camera.main.transform.position.z
);

pointInstance.transform.LookAt(targetPosition);
```

### 5.3 Clipping shader

When a plane is selected as a form of movement, it needs to be visually indicated. An approach that was initially tested was stretching the plane, analogous to the way axes are elongated when selected. The idea was, that the plane would cut through the middle of the robot, visualizing its position in relation to the robot in space. Due to how rendering works, however, this is not easily achievable. Either the robot or the plane will always be rendered on top. The solution to this problem is a custom shader.

When enabled, the clipping shader is assigned to the material of the robot renderer. A `float4` property `_Plane` represents the clipping plane in the shader. The first three values are the normal of the plane, and the fourth is the distance to the origin, along the plane's normal. In the surface shader, the dot product of the surface point's world position and the plane position is calculated. The fourth value is added to account for the distance of the plane from the origin. If the result is positive, the point is above the plane, if the result is zero, the point lies directly on the plane, and if the result is negative, the point is below the plane. All the points below the plane have an alternate color added to their base color.

```
void surf (Input i, inout SurfaceOutputStandard o) {
    //dot product is evaluated
    float distance = dot(i.worldPos, _Plane.xyz);

    //distance from origin is added
    distance = distance + _Plane.w;

    //point is above the plane, render normally
    if (distance > 0) {
        fixed4 col = tex2D(_MainTex, i.uv_MainTex);
        col *= _Color;
        o.Albedo = col.rgb;

    //point is below the plane, add tint
    } else {
```

<sup>6</sup><https://discussions.unity.com/t/lookat-to-only-rotate-on-y-axis-how/10895/3>

```

        fixed4 col = tex2D(_MainTex, i.uv_MainTex);
        col *= _Color;
        o.Albedo = col.rgb + _AlternateColor.rgb;
    }
}

```

The clipping plane property is set from the `ClippingPlane.cs` script, which is attached to an empty `GameObject` and inherits the gizmo's `Transform`. The clipping plane holds a list of materials with the clipping shader assigned. Materials are added and removed from the list to enable and disable the shader during runtime. In each frame, the script<sup>7</sup> creates a new plane, loops through the material list, and sets the `_Plane` property in the shader.

```

void Update() {
    //create plane
    Plane plane;

    plane = new Plane(transform.up, transform.position);

    //transfer values from plane to vector4
    Vector4 planeRepresentation = new Vector4(plane.normal.x, plane.normal.
        y, plane.normal.z, plane.distance);

    //pass vector to shaders
    foreach (Material material in Materials) {
        material.SetVector("_Plane", planeRepresentation);
    }
}

```

This creates a horizontal plane at the position of the gizmo. There are, however, three planes that can be selected. In order to properly split the model when a different plane is used, the `ClippingPlane` object is rotated accordingly. Another aspect to consider is the camera's location in reference to the gizmo; if not accounted for, it would result in the colors being switched (red in front of the plane, regular behind it).

This approach to visualizing the plane was abandoned in later development. While the effect worked well, the stretched plane covering much of the workspace was deemed too irritating. A more suitable approach was used to indicate, which plane the movement is constrained to: instead of stretching the plane itself, the two axes that define the plane are elongated. This method is more elegant, as it uses visual elements the user is already familiar with, instead of introducing new ones.

## 5.4 Invalid pose indication

The pose validity is indicated by a custom notification and by tinting the robot itself. In this section, the performance issues caused by the original notification system are described, and the new replacement notification is showcased. The section also provides information on how the red tint effect was achieved.

<sup>7</sup>Adapted method from <https://www.ronja-tutorials.com/post/021-plane-clipping/> and <https://www.youtube.com/watch?v=shOH7pJSxk8>.

### 5.4.1 Notifications

Severe performance issues were noted during development when dragging the point out of bounds. The notifications were identified as the root cause. Each notification is an instance of `NotificationEntry`, created by the singleton `Notifications`. The lifespan of each instance is tied to a timer. When dragging the robot arm, an inverse kinematics request is sent to `ARServer` nearly every frame. If the operator moves the target point outside the work envelope of the robot, the RPC fails, and an RPC response informing of failure is sent. `AREditor` creates a notification for each of these responses. This essentially floods the app with notifications and results in severe lag, sometimes causing the tablet to freeze for tens of seconds at a time.

The new notification is a single `GameObject` already instantiated within the `ModelPositioningMenu` and its state is switched between active and inactive based on the current target point position and therefore pose validity.

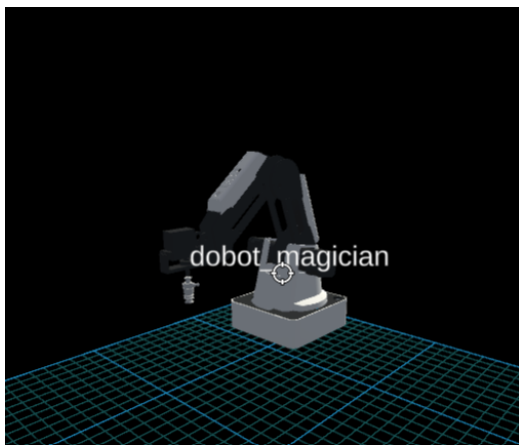
### 5.4.2 Robot display

When the target point is unreachable, the robot turns red. This was implemented with a surface shader in the file `InvalidPose.shader`. The red color is added to the base color, which tints the robot red while preserving the original texture:

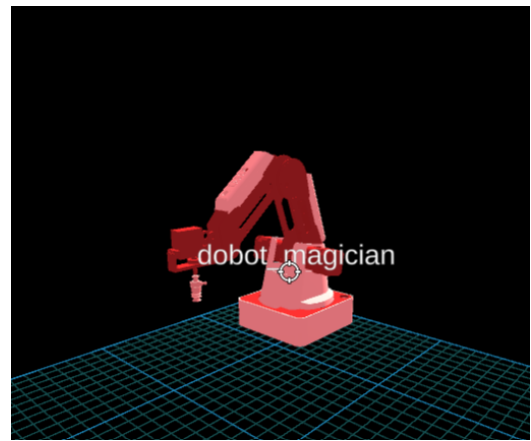
```
void surf (Input IN, inout SurfaceOutputStandard o)
{
    //main color is obtained from the main texture
    fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
    //red color is added
    o.Albedo = c.rgb + _RedColor.rgb;
}
```

The shader is applied to the materials of the robot renderers at runtime. A robot renderer contains three materials when the outline is enabled: mask, object material, and outline. If the outline is disabled, only the object material is present. The following code snippet illustrates how the red shader is enabled (adapted from a function for setting visibility in `RobotActionObject`). Disabling the shader is done by applying the 'Standard' shader, which is done correspondingly. The effect is pictured in Figure 5.4.

```
private void EnableInvalidShader() {
    materialEnabled = MaterialEnabled.invalid;
    foreach (Renderer i in robot.robotRenderers) {
        if (i.materials.Length == 3) {
            i.materials[1].shader = Shader.Find("InvalidPose");
        } else {
            i.material.shader = Shader.Find("InvalidPose");
        }
    }
}
```



(a) Default robot materials.



(b) 'Invalid pose' shader.

Figure 5.4: Dobot Magician displayed with the default materials, and with the red tint applied.

## Chapter 6

# Usability testing

Testing took place in the robotics laboratory at FIT. Due to scheduling conflicts, the testers were split into two groups of 3 and 2, and the testing was done in two separate sessions. The test subjects were students of the third year at BUT FIT.

The test subjects were first asked questions to assess their level of experience with tablets, AR, and robotics. All subjects shared a moderate level of experience with tablets and a low to moderate level of experience with AR, but none have come in contact with robot programming. As the solution is targeted towards end-users with no experience with robotics, this makes the subject group ideal for usability testing. The general outline of the testing consisted of several points:

- Ask questions about experience, briefly explain, what AREditor is for
- Present the subject with the application, observe initial instincts
- Explain, how the model positioning menu generally works, notice how quickly the subject picks it up
- Gradually explain all the features of the menu
- Present simple tasks to the subject
- Show the subject the robot stepping menu
- Ask them to compare the two menus
- Conclude testing, ask for any additional feedback

Subjects were also encouraged to ask questions and give feedback throughout the entire testing.

### Subject 1

The first subject initially assumed, that the select button worked like a joystick and attempted to use it as such (pulling to the sides when holding it down). However, after explaining how the menu is supposed to be used, they picked it up within seconds.

The subject was able to accomplish simple tasks such as positioning the robot to the given coordination and moving the end effector along a global axis. When tasked with

moving the end effector along the edge of the table which is not aligned with the global coordinate system, the subject enabled the user-relative coordinate system, aligned themselves with the table, and moved the robot using the correct axis. The subject correctly assumed the purpose of the 'Apply pose' and 'Revert pose' buttons.

After reviewing the robot stepping menu, the subject concluded, that the new solution is a significant improvement. They especially liked the ability to simply drag the model into a pose and visualize it, in contrast with the robot stepper, where they could not easily tell where the robot arm would end up after a button press.

The subject struggled when they stopped moving the point while out of reach of the robot. They did not know, how to navigate back into the robot work envelope. The point's position in space was difficult to understand to the subject, as the robot arm was used as the main reference point. The subject also suggested labeling the sensitivity slider, and assessed, that before using a plane for movement, they couldn't tell, what it was for, although that was apparent once they selected it for the first time. When it comes to the clipping shader, the subject failed to notice it, and when brought up, they were unsure of its purpose.

## Subject 2

When tasked with moving the robot arm to a designated location, without any prior explanation of how the system works, the second subject successfully did so. Given the task of moving the robot arm along an arbitrary axis, the subject correctly assessed, that the user-relative coordinate system should be enabled, but then they selected the horizontal plane instead of the Y-Axis accidentally. The subject also correctly assumed the purpose of the forward/backward buttons, but the sensitivity range was too high, making it impractical to use. This was amended before further testing.

When using the robot stepping menu, the subject also found it hard to predict, where the end effector would end up, and preferred the model positioning menu due to its transparency.

The second subject, just like the first subject, suggested labeling the sensitivity sliders. They also found it frustrating when going out of bounds and having to find the way back, partly due to having no sense of depth. The clipping shader served no purpose other than a source of confusion in this testing too.

## Subject 3

The third subject initially attempted to drag the gizmo arrows with their finger on-screen, but quickly caught on. When tasked with moving the point along the axis of the table, the subject correctly utilized the user-relative coordinate system, aligned themselves with the edge of the table, and proceeded to move the point along the right axis. They also correctly understood the functionality of the planes. In comparison with the robot stepping menu, the model positioning menu was deemed far more user-friendly.

Various times during the testing, when attempting to move the point across the robot itself, the subject stopped and backtracked, if the robot turned red. This suggests, that the subject also struggles to interpret, where the point is in relation to the robot while out of bounds. As with the previous subjects, the clipping shader also went unnoticed and misunderstood.

## Subjects 4 and 5

On the second day of testing, the tasks were slightly modified to better reflect real-life scenarios. Colored cubes were used as targets, and the subjects’ job was to move the end effector towards them.

The fourth and fifth subjects came in at the same time and tested alternately. The fourth subject attempted to move the gizmo using their finger, analogous to subject 3, but very quickly realized how it worked. As the fifth subject was present during this time, their first instinct could not be observed. They both understood the purpose of the user-relative coordinate system, and they utilized the forward/backward buttons during various tasks.

One thing that subjects 4 and 5 did differently was utilize the robot stepping menu. While they agreed that it is tedious to use for large movements, it can be useful for precise adjustments. When tasked with aligning the end effector with a cube, they often opted for a workflow that included both the model positioning menu and the robot stepping menu.

### 6.1 Evaluation

In total, 5 people over 2 days tested the application. They all concluded, that the model positioning menu is an improvement over the robot stepping menu in terms of usability, speed, and transparency. Table 6.1 below shows the results of the tests.

	1	2	3	4	5
<b>Principle grasp</b>	fast	immediate	fast	very fast	very fast
<b>Movement choice</b>	EE, axes	EE, planes	EE, axes	Axes, EE	EE, axes
<b>Movement issues</b>	no issues	misselected axis	no issues	misselected axis	no issues
<b>User-relative coord. system usage</b>	yes	yes	yes	yes	yes
<b>FW/BW buttons usage</b>	no	no	occasionally	often	occasionally
<b>Sensitivity slider usage</b>	no	yes	yes	yes	no
<b>Robot stepping menu usage</b>	no	no	no	yes	yes
<b>Clipping shader understanding</b>	no	no	no	no	no

Table 6.1: Usability testing evaluation

This testing yields several key insights. Firstly, most subjects naturally gravitated towards the end effector as their initial choice of movement mode. Axes saw more use as each session progressed, showing that they are better suited for tasks requiring a level of precision, but they are not as intuitive to use at first. Axes were most often used in combination with the user-relative coordinate system. The planes were used sparsely throughout the testing, and occasionally they were selected accidentally. Consequently, the

planes were made smaller and were moved further away from the axes to prevent any future incidents of similar nature.

The forward/backward buttons saw moderate use, the problem was the sensitivity range, which was lowered afterward. Because the robot used in the testing is small, simply moving the tablet forward in place of using the buttons was the easier option in most circumstances.

The clipping shader as a visual indicator provided no value to any subjects when testing, even confusing some, and was subsequently removed altogether.

The most common problem among the testers was understanding, where the point was located if they accidentally went outside the robot's work envelope. This stems from the lack of depth perception when working with augmented reality on a tablet. While moving the point within range, the robot arm pose functions as the main source of visual information, which the user can use to identify where the point is located. If the point is dragged outside of the range of the robot, the robot pose is no longer updated and the user loses a visual reference point. To address this issue, a slight modification to the way movement works was made. After the modification, if the user releases the button and concludes the movement while out of bounds, the point snaps back into the last valid position. Originally, the point would stay in the invalid position, leaving the user to manually drag it back.

Overall, the outcome of the testing proves the implemented solution's usability. The test subjects were able to learn to use it quickly and complete the tasks presented without major complications. The menu presents an improvement over the robot stepping menu, but the two menus can be used in tandem, as demonstrated by subjects 4 and 5. Because the overlaid model is never calibrated with perfect precision, it can be difficult to use when making smaller adjustments, which is where the robot stepping menu is better utilized.



## Chapter 7

# Conclusion

The goal of this thesis was to design and develop an interaction method for positioning a robot model in augmented reality, that can be used while programming robots in AREditor. The implemented system should allow operators to efficiently set poses of robots that do not support manual repositioning, and thus enable them to visually program more efficiently.

Current research in controlling robot arms using augmented reality was assessed and the gained knowledge was used to inform the design process. The existing options for positioning robot arms in AREditor were also evaluated and built upon.

A new system for manipulating the robot was designed, which takes advantage of the AR nature of the application and utilizes tablet movement tracking to move the end effector. The new system provides users with a flexible toolset for performing complex and precise robot movements while consisting of a low number of on-screen controls. The system also provides visual feedback for indicating movement constraints and invalid pose reports.

The system was implemented into AREditor as a new menu, alongside the robot stepping menu. Raycasts were used as a basis for tablet movement tracking, and a variety of techniques were employed to visualize the selected handle, the validity of the robot pose, and the dimension of movement. To signify a plane's position relative to the robot, a custom shader was written, but was later abandoned, as it provided limited clarity of vision.

Usability testing was realized in the robotics laboratory at FIT over two days. The subjects unanimously concluded that the new system is a significant improvement over the original solution both in terms of speed and ease of use. Each subject was able to learn to control the system within minutes and was able to accomplish simple tasks consisting of moving the robot arm to given positions in the workspace and moving the arm in the direction of an arbitrary axis. Given the positive results of the testing, the goal to improve the robot positioning process in AREditor can be considered accomplished.

In future works, the system can be built upon in a variety of ways. The lack of depth perception is currently the biggest limitation when working with augmented reality on tablet devices. Future research aiming to alleviate this issue could focus on projecting the point on the table, employing shadows, or displaying guiding lines of some form. A welcome feature in the menu would also be the option to rotate the end effector, which is currently only available from the robot stepping menu.

Another aspect that can be improved in this work is the approach to invalid poses. A better solution would be to only allow the point to move within the robot's work envelope, making every pose set by the user valid. This is not possible using the current workflow of

sending inverse kinematics requests and awaiting results, and would require a predefined area, where the robot can reach.

# Bibliography

- [1] ABBAS, S. M., HASSAN, S. and YUN, J. Augmented reality based teaching pendant for industrial robot. In: *2012 12th International Conference on Control, Automation and Systems*. 2012, p. 2210–2213.
- [2] ARAIZA ILLAN, D., DE SAN BERNABE, A., HONGCHAO, F. and SHIN, L. Y. Augmented Reality for Quick and Intuitive Robotic Packing Re-Programming. In: *2019 14th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*. 2019, p. 664–664. DOI: 10.1109/HRI.2019.8673327.
- [3] MATERNA, Z. *ARCOR2* [online]. 2022 [cit. 2024-04-06]. Available at: <https://github.com/robofit/arcor2>.
- [4] MATERNA, Z. *AREditor* [online]. 2022 [cit. 2024-04-06]. Available at: [https://github.com/robofit/arcor2\\_areditor](https://github.com/robofit/arcor2_areditor).
- [5] CHACKO, S. M. and KAPILA, V. An Augmented Reality Interface for Human-Robot Interaction in Unconstrained Environments. In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2019, p. 3222–3228. DOI: 10.1109/IROS40897.2019.8967973.
- [6] CHAN, W. P., HANKS, G., SAKR, M., ZUO, T., LOOS, H. Machiel Van der et al. An Augmented Reality Human-Robot Physical Collaboration Interface Design for Shared, Large-Scale, Labour-Intensive Manufacturing Tasks. In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020, p. 11308–11313. DOI: 10.1109/IROS45743.2020.9341119.
- [7] FANG, H., ONG, S. K. and NEE, A. Y.-C. Robot Programming Using Augmented Reality. In: *2009 International Conference on CyberWorlds*. 2009, p. 13–20. DOI: 10.1109/CW.2009.14.
- [8] FRANK, J. A. and KAPILA, V. Towards teleoperation-based interactive learning of robot kinematics using a mobile augmented reality interface on a tablet. In: *2016 Indian Control Conference (ICC)*. 2016, p. 385–392. DOI: 10.1109/INDIANCC.2016.7441163.
- [9] GRADMANN, M., ORENDT, E. M., SCHMIDT, E., SCHWEIZER, S. and HENRICH, D. Augmented Reality Robot Operation Interface with Google Tango. In: *ISR 2018; 50th International Symposium on Robotics*. 2018, p. 1–8.
- [10] KAPINUS, M., MATERNA, Z., BAMBUŠEK, D., BERAN, V. and SMRŽ, P. ARCOR2: Framework for Collaborative End-User Management of Industrial Robotic Workplaces using Augmented Reality. *ArXiv preprint arXiv:2306.08464*. 2023.

- [11] ONG, S., YEW, A., THANIGAIVEL, N. and NEE, A. Augmented reality-assisted robot programming system for industrial applications. *Robotics and Computer-Integrated Manufacturing*. 2020, vol. 61, p. 101820. DOI: <https://doi.org/10.1016/j.rcim.2019.101820>. ISSN 0736-5845. Available at: <https://www.sciencedirect.com/science/article/pii/S0736584519300250>.
- [12] PARK, K.-B., CHOI, S. H., LEE, J. Y., GHASEMI, Y., MOHAMMED, M. et al. Hands-Free Human–Robot Interaction Using Multimodal Gestures and Deep Learning in Wearable Mixed Reality. *IEEE Access*. 2021, vol. 9, p. 55448–55464. DOI: 10.1109/ACCESS.2021.3071364.
- [13] CAKAL, B. A. *ROS - Unity Coordinate System Conversions* [online]. Siemens, 2019 [cit. 2024-04-27]. Available at: [https://github.com/siemens/ros-sharp/wiki/Dev\\_ROSUnityCoordinateSystemConversion](https://github.com/siemens/ros-sharp/wiki/Dev_ROSUnityCoordinateSystemConversion).
- [14] SUZUKI, R., KARIM, A., XIA, T., HEDAYATI, H. and MARQUARDT, N. Augmented Reality and Robotics: A Survey and Taxonomy for AR-enhanced Human-Robot Interaction and Robotic Interfaces. In: *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 2022, p. 1–32. Available at: <https://doi.org/10.1145/1122445.1122456>.
- [15] YIGITBAS, E., JOVANOVIKJ, I. and ENGELS, G. Simplifying Robot Programming using Augmented Reality and End-User Development. june 2021.
- [16] YOSHIKAWA, T. Manipulability of Robotic Mechanisms. *The International Journal of Robotics Research*. 1985, vol. 4, no. 2, p. 3–9. DOI: 10.1177/027836498500400201. Available at: <https://doi.org/10.1177/027836498500400201>.
- [17] ZHANG, F., LAI, C. Y., SIMIC, M. and DING, S. Augmented reality in robot programming. *Procedia Computer Science*. 2020, vol. 176, p. 1221–1230. DOI: <https://doi.org/10.1016/j.procs.2020.09.119>. ISSN 1877-0509. Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 24th International Conference KES2020. Available at: <https://www.sciencedirect.com/science/article/pii/S1877050920320196>.
- [18] ZHANG, F., LAI, C. Y., SIMIC, M. and DING, S. Augmented reality in robot programming. In: *International Conference on Knowledge-Based Intelligent Information & Engineering Systems*. 2020. Available at: <https://api.semanticscholar.org/CorpusID:226255892>.

# Appendix A

## CD contents

The CD included with this work contains the following items:

- A PDF version of this thesis, along with the  $\text{\LaTeX}$  source files.
- Source code: as AREditor is a large project, only the relevant files have been included.
- `git.diff`: a file listing all changes made to the original repository.
- A short video showcasing the work.
- The exported version of AREditor in APK format.
- `link.txt`: A text file containing a link to the AREditor GitHub, where installation instructions can be found.