POLITECNICO
MILANO 1863

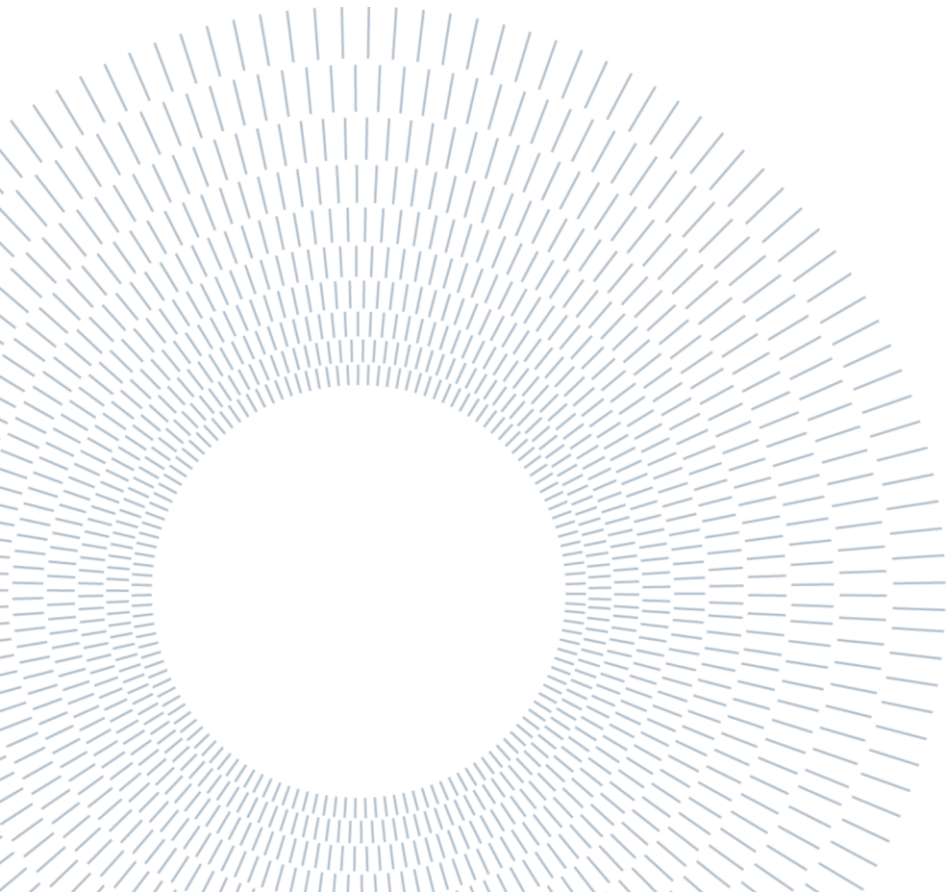
SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Human-Robot Collaboration Simulation in Virtual Reality with Multiple Configurations

TESI DI LAUREA MAGISTRALE IN
MECHANICAL ENGINEERING-INGEGNERIA
MECCANICA

Author: **Kaan Özkaya**

Student ID:	10721817
Advisor:	Monica Bordegoni
Academic Year:	2021-22



Acknowledgement

I would like to thank my supervisors Professor Monica Bordegoni and Professor Marco Rossoni for giving me the opportunity to work with them and their invaluable support during this thesis project.

Abstract

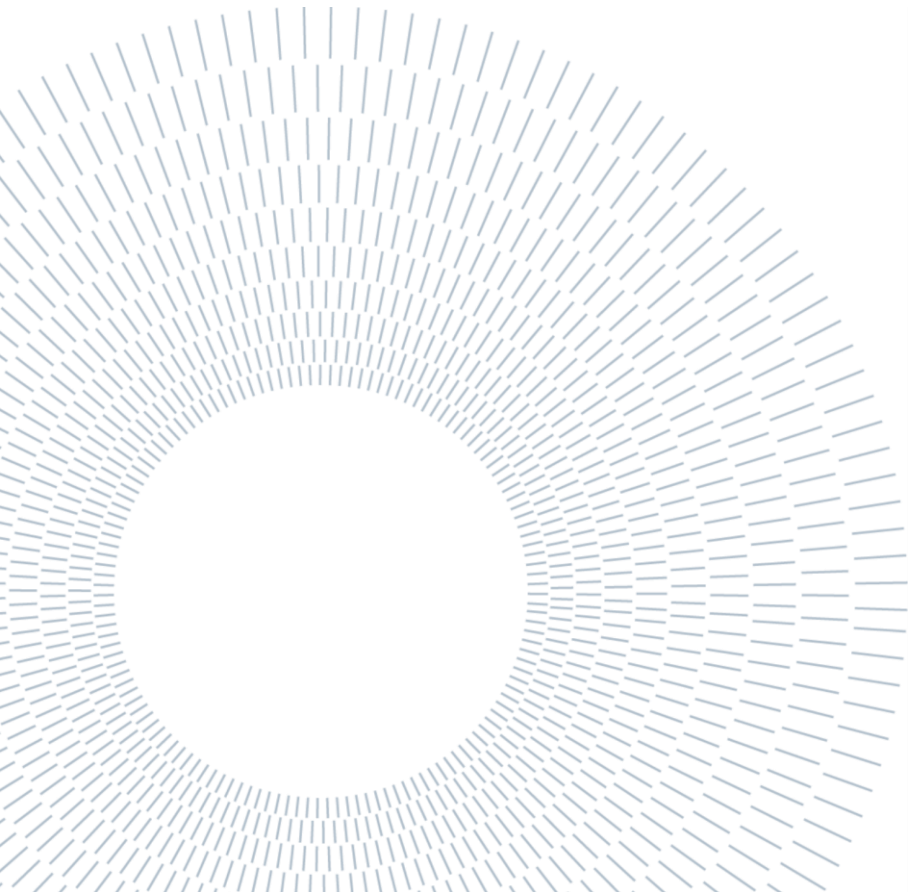
Human-Robot Collaborative (HRC) applications are becoming increasingly more popular in the industry to address the new demand profile. Therefore, the design of these kinds of systems are becoming a primary concern. Because of the new challenges in terms of safety and optimization introduced by these systems, a way to efficiently simulate them is required to conduct tests on them. While virtual simulations are a solution, the examples in literature require a considerable amount of work to configure and usually only work for the specific setup they were configured for. The aim of the project is to address this issue by creating a virtual environment which can simulate multiple HRC applications with varying robot arms and end effectors. To ensure that the system is able to simulate different types of HRC applications, two different robot models and three different end effector models are used. Both of the robots are connected with all of the available end effectors resulting with six different configurations. These configurations vary in geometry and complexity. They also differ in terms of application type because one of the end effectors tested is a gripper which performed a pick and place operation while others are welding torches performing welding operations. The system is configurated using ROS and Unity software together by connecting them with available packages. Moveit motion planner is the main reason ROS is used as this package is used to calculate the trajectory of the robot movement. Unity was preferred because it has a vast library of packages and integrated tools which can be used to create virtual simulations. The system is also integrated with virtual reality (VR) equipment to make the simulation of the human operator more immersive. Steam VR package was preferred during this integration because it makes development simpler and works with multiple different VR headsets. The resulting system can simulate the six default configurations and the report also contains a guide explaining how to add new robots or to end effectors to the system. This system can be used to create virtual simulations with minimal effort, allowing the simulators to allocate more time for testing.

Key-words: human-robot collaboration, simulation, virtual reality, robotics, Unity, ROS, Steam VR.

Abstract in lingua Italiana

Le applicazioni di collaborazione uomo-robot (HRC) stanno diventando sempre più popolari nel settore per soddisfare il nuovo profilo della domanda. Pertanto, la progettazione di questo tipo di sistemi sta diventando una preoccupazione primaria. A causa delle nuove sfide in termini di sicurezza e ottimizzazione introdotte da questi sistemi, è necessario un modo per simularli in modo efficiente per condurre test su di essi. Sebbene le simulazioni virtuali siano una soluzione, gli esempi in letteratura richiedono una notevole quantità di lavoro per la configurazione e di solito funzionano solo per la configurazione specifica per cui sono stati configurati. Lo scopo del progetto è affrontare questo problema creando un ambiente virtuale in grado di simulare più applicazioni HRC con diversi bracci robotici ed effettori finali. Per garantire che il sistema sia in grado di simulare diversi tipi di applicazioni HRC, vengono utilizzati due diversi modelli di robot e tre diversi modelli di effettori finali. Entrambi i robot sono collegati con tutti gli effettori finali disponibili risultando con sei diverse configurazioni. Queste configurazioni variano in geometria e complessità. Differiscono anche in termini di tipo di applicazione perché uno degli effettori finali testati è una pinza che ha eseguito un'operazione di presa e posizionamento mentre altri sono torce di saldatura che eseguono operazioni di saldatura. Il sistema viene configurato utilizzando il software ROS e Unity insieme collegandoli con i pacchetti disponibili. Moveit motion planner è il motivo principale per cui viene utilizzato ROS poiché questo pacchetto viene utilizzato per calcolare la traiettoria del movimento del robot. Unity è stato preferito perché ha una vasta libreria di pacchetti e strumenti integrati che possono essere utilizzati per creare simulazioni virtuali. Il sistema è inoltre integrato con apparecchiature di realtà virtuale (VR) per rendere più immersiva la simulazione dell'operatore umano. Il pacchetto Steam VR è stato preferito durante questa integrazione perché semplifica lo sviluppo e funziona con più visori VR diversi. Il sistema risultante può simulare le sei configurazioni predefinite e il report contiene anche una guida che spiega come aggiungere nuovi robot o come effettori finali al sistema. Questo sistema può essere utilizzato per creare simulazioni virtuali con il minimo sforzo, consentendo ai simulatori di allocare più tempo per i test.

Parole chiave: collaborazione uomo-robot, simulazione, realtà virtuale, robotica, Unity, ROS, Steam VR.



Contents

Introduction

As the demand shifts from high quantity, low customization goods to high mix, low volume products, the established manufacturing methods fail to satisfy the expectations. This situation can't be solved even utilizing state of the art robust robotic systems [2]. Since using flexible robots are not efficient in term of cost, the need for dexterity should be satisfied by human operators. Human operators' cognitive skills also make them advantageous compared to robots in solving complex tasks and resolving unexpected situations. However, human performance diminishes as cognitive fatigue sets in [2]. To solve this issue, human robot collaborative systems (HRC) are designed which combine the dexterity of a human operator with robots' ability to solve repeatable tasks with ease. In this configuration, robot helps to reduce the workload of the human operator and human operator solves the problems which are too advanced for the robot.

This new type of working configuration offers new challenges to make the system safe and optimized. While optimizing the system, a shared workspace should be taken into consideration for both the robot and the human operator. In other words, as both will work on the same product, their actions will inevitably affect each other. In the study "Integration of the human-robot system in the learning factory assembly process", it is found that in sequential configurations it is more efficient to start with the robot. [3] However the best result is achieved when some tasks individually performed by the robot and the human operator are done at the same time. Furthermore, sharing the workspace also increases the work efficiency meaningfully. This can be seen clearly from the work of Andronas et al. [4] which shows that shared workspace increases efficiency in both cases which involve lightweight robots or heavyweight robots. Therefore, from the optimization perspective, the human operator and the robot should be able to support each other and work on the same workpiece, in the same workspace, at the same time. This supportive type of collaboration is also a trend in the industry as majority of the research done in the field is performed on these type of configurations [2].

While these configurations increase efficiency, they offer additional challenges in terms of safety. In the case of heavyweight robots, human operator can be harmed

significantly if something unexpected happens. Therefore, shared workspace and working at the same time of a robot and human operator can only be possible with many precautions. Traditional methods usually consider sequential configurations for robot and human operator actions on the same workpiece. This means that methods such as designating safety areas for human operators or prohibiting robot movement when operator is present are considered to be sufficient. However, these methods directly interfere with the aim of the supportive configuration. To achieve safety in an optimized HRC configuration, the robot should be able to read human state and act accordingly [1]. This can only be possible with the use of many sensors to have the necessary information about the operator's actions. With the help of sensors, rather than inhibiting robot movement, monitoring of speed and limitations of power and force can be utilized to ensure the safety of the operator [3].

While the challenges of HRC are apparent, there aren't enough collective information to address these issues in an optimal manner since the technology is relatively new. Despite design methods such as the FMEA analysis can be used to find the most critical areas of improvement, the results are not reliable as the occurrence parameter is given based on assumption [5]. Therefore, a method to simulate these kinds of systems is needed to improve them without putting operators at risk. In this regard, digital twin which aims to create a virtual copy of the physical robotic system and to simulate data on the virtual model is a suitable method to improve the performance of the real system [6]. Using the virtual model, the issues of cost and safety can be simulated and understood without creating and operating a real system. After the prototyping phase, the virtual model doesn't become obsolete because it can still be used in various ways such as process control, operator training, predictive maintenance and real-time analytics [6]. The main challenge of this method is the creation of the virtual copy of the system and making the simulation accurate enough to give results close to the on-field experience.

Virtual robot models were developed in the past with similar goals however these models had limited functions as most of the workload was given to creating the model rather than adding functions to it. They also had different formats depending on the manufacturer which made simulations with multiple robots impossible [7]. Because of these challenges, Robot Operating System (ROS) was created. It is an open-source software which is used to create robot models and perform calculations on them. This software reasonably cuts the amount of time needed to create a virtual robot model and increases the flexibility of the models as it can be used for robots from any manufacturer.

Next step to create an accurate simulation is to create an immersive virtual human operator. This became possible with the improvements in the Virtual Reality (VR) and

Augmented Reality (AR) fields. New VR equipment allows the user to take on the role of the operator in the simulation in an immersive way which creates results closer to reality. This configuration offers better performance in terms of cost, versatility and accuracy compared to other simulation methods such as the ones provided by the manufacturers or other commercial tools [7]. Interacting with ROS using VR however is not an easy task because while ROS is very competent at simulating robots, it is not adequate when it comes to simulating human operators. While handling of the physics can be coded manually, nowadays different engines such as Unity are present which have built in kinematics [9]. While originally made to create computer games, the Unity engine provides many valuable tools to model industrial applications. It also has many packages which can be imported to be used in projects which diminishes the workload. The Unity environments can be made realistic with the utilization of CAD models and the kinematic solver can be adjusted to make it more realistic [10]. Additionally, software to connect Unity with ROS is available because Unity is used in robotics. Unity and VR connection software is also available in multiple forms because both platforms are made for computer gaming. The preferred connector for Unity and VR headset was Steam VR as it has a comprehensive library and offers easier troubleshooting.

All these software being open source helps with the availability of them and they reduce the workload required to create a virtual simulation. Because of these reasons, they are frequently used together in scientific research. One example is the use of these software during the optimization of factory ergonomics for HRC layouts [11]. In that article, VR is used with Unity to capture the posture of the worker to make ergonomic analysis. However, creating a model using these software together takes a considerable amount of time and effort. Furthermore, Unity and ROS using different coding languages also adds to the complexity. Even if a model is created, these models usually work only for the modeled robot and activity, so additional effort has to be put in place for the creation of another simulation. Hence, while previous research defines the benefits of using virtual reality equipment in virtual simulation of HRC systems, they all offer systems which can only perform specific tasks. This limits the informative capabilities of the systems as they only offer results for singular or similar configurations.

In this report, a system which is customizable and can perform multiple tasks is created. The robot and end effector in the simulation can be configured and new models can be introduced to the system which increase the testing capabilities. This is the novelty of the system. The system will reduce the workload of creating such virtual models even further, allowing the modelers to focus on the quality of the simulation rather than creating the environment. The configurations have two different robot models which are the UR3 and UR5 of Universal Robots. The end effectors are two

different welding torches and the Robotiq two finger gripper. These configurations were made to test if the system works with different robots, end effectors and processes. As ROS requires Linux operating system and some Unity packages need Windows operating system, multiple computers or a virtual box are needed to run the system. VR equipment used in testing is Oculus Quest 2 however the system can also be launched with other VR headsets if they are compatible with Steam VR.

The paper consists of 5 parts which are:

Chapter 1 explains the state of the art by focusing on similar projects in the industry. Thus, many other projects simulating HRC applications with the help of virtual reality equipment is investigated. This chapter also explains the mathematical theory behind the solver of Moveit package, which is the ROS package used to calculate robot trajectory. Information about knowledge required to understand inverse kinematics problems and how to solve them are also provided.

Chapter 2 is about the architecture of the project which defines why and how the system was built.

Chapter 3 consists of the work done during the project which includes connecting the systems, coding the movement scripts and configuring the VR environment. It also includes a guide to add new robots and end effectors to the system to reduce the workload of a potential user.

Chapter 4 mentions the capabilities of the end product and the results gathered through operating the system.

Chapter 5 is the conclusion which comments on the results of the project work and explores potential areas of improvement. Some shortcomings of the used software are also commented on in this part which should be accounted for in a future work using the same software.

1. State of the Art

Because of the reasons discussed in the introduction part, there are many articles about using virtual reality in human-robot collaboration simulations. Recent scientific articles in this field were investigated to understand the current developments and study where the direction of research is going. One of these articles is the "Human-machine collaboration in virtual reality for adaptive production engineering" [10]. In this article the three key elements to achieve immersion in these kinds of simulations were defined as the creation of a realistic virtual space, the participation of the user in the system and the user seeing the results of their actions. The first can be achieved by using CAD models of the industrial machines. Still, some preprocessing is required as most CAD models are too complex in geometry for collision calculations. Therefore, using real CAD models for visual meshes and using less complex counterparts for collision meshing is advised. The article also addresses the benefits of using a gaming engine such as Unity during development by saying the existing packages make the process of creating a simulation easier. The application in the article is a robot arm simulation which is controlled by a human using virtual reality equipment. The goal of the system is to perform a simple collaborative task with this setup. Inverse kinematics calculations are made on the controller held by the human arm to find the corresponding joint positions of the robot. The degrees of freedom (DOF) of the robot were limited to increase the smoothness of the robot motion in the simulation. The final conclusion is that these kinds of simulations are informative about robotics and virtual reality increases user experience by making the process more immersive.

In the second article named "Virtual reality in manufacturing: immersive and collaborative artificial-reality in design of human-robot workspace" [16], VR integrated HRC simulation is used to model HRC workspaces. The process of designing a HRC workspace is split into three parts. First part is about the design of the end product, the process and the requirements. In this part the robot arm, the end effector, workspace tools, materials and the human operator are considered. As a second step, the validation of the prototype design by using virtual simulations is

performed. Human operator is simulated by using camera capture of a real human. Last step is the use of the existing simulation to develop the system. This step utilizes virtual reality assets to immerse the user into the simulation which allows them to analyze the system thoroughly. The robot, workspace and human operator is modeled for the simulation. This simulation is used to optimize the system and increase safety of HRC systems. As a result of the study, the process is found to be faster and simpler compared to existing methods.

Similarly, in the article "Using virtual manufacturing to design human-centric factories: an industrial case"; virtual reality techniques were again used to model HRC workspaces [11]. In this research, the factory ergonomics is the area of focus. HTC VIVE headset is integrated with Unity to create the virtual simulation with additional assistive tools for more accurate motion capture. In previous works, the ergonomics assessment was done using virtual mannequins in a virtual simulation controlled with keyboard and mouse. The testable number of postures were limited, and accuracy of the results were highly dependent on mannequin placement. Creator of the simulation had to have a comprehensive understanding of the system because of these reasons. In the article referenced, VR integration offers a chance to test the simulation with final users and hence to achieve more precise results. The virtual simulation is created integrating the hardware and software mentioned. After the simulation is run, most critical postures are isolated and assessed to complete the posture analysis. The three metrics considered for posture assessment are the Ovako Working Posture Analyzing System, the Rapid Entire Body Assessment and the European Assembly Work-Sheet. Two use cases were investigated using this system. First case is a Selective Catalytic Reduction assembly. This case was found to be unergonomic as the operator was constantly in an undesirably curved position during the virtual simulation. Therefore, the assembly process had to be redesigned. The second case involves cables and brake pipes of a tractor which makes the operator work under the cabin. The cabin lifter was modified with the results of the analysis to make the process more ergonomic. The results of these two cases were compared to the results of the previous approaches on the same cases. Despite the virtual reality approach has a longer preparation phase compared to the existing simulations, it is found to be advantageous in the long term as it offers faster and more precise results. The increase in precision was assessed comparing the results of both methods with on-field assessments.

In another article called "Digital Twin and Virtual Reality Based Methodology for Multi-Robot Manufacturing Cell Commissioning" [7], virtual reality is again used to test and analyze robotic systems. The digital twin concept is combined with virtual reality interface to better designate potential mistakes using virtual simulations to reduce development costs. The roadmap is to first design the system, and then to design the virtual reality integrated simulation. If the simulation is successful, the real

system is commissioned, and the digital twin is integrated into this system for the various benefits it offers which was discussed during the introduction. Importance of user immersion and detailed digital twin modelling is also given. Use case contains an explanation on how the roadmap is applied on an assembly process with human-robot collaboration. The system contains two robot arms, a conveyor belt and a human operator. Safety system with sensors and correction of human actions by the robot is present in the system. In addition, with the use of virtual reality, different task allocations between the human and the robot are investigated to optimize the process. When the virtual model is validated, a physical model of the system is created, and the digital twin is used with the virtual reality controller to inspect the real process. As a result, virtual reality simulations are found to be an effective model to design digital twin applications.

Another article named "Human-Robot Collaboration: Task Sharing Through Virtual Reality" applies the basic theory of HRC applications on an assembly process using virtual reality integrated simulations [8]. A simple operation of putting a nut on a screw and turning the screw is investigated. The repetitive part of the operation is designated as the turning operation while the part requires precision in placing the nut and the tasks are distributed to the human operator and the robot accordingly. First, the human operates on the system and then the robot is activated with voice commands. As the system has the human operator and the robot working in the same workspace at the same time, safety precautions are taken. A GUI is also made to guide the operator during the operation. The result of the virtual model was used to create the real version of the system similar to the previous case. With this research, it is further proven that VR integrated simulations can be used to design HRC systems with high level of collaboration.

As explained before, safety is one of the most important aspects of HRC applications. In the article called "Exploration of two safety strategies in human-robot collaborative manufacturing using Virtual Reality", a simulation is used to assess the effectiveness of two different safety methods [30]. The simulation is designed as a Windows application using the Unity engine. The designed virtual environment contains the models of the human operator, the robot and the surroundings. During the simulation, the user experience is in first person using virtual reality equipment and collision-based interaction methods are adopted. Kinematics calculations explained in the later parts of this chapter are used to calculate robot motion. The simulation represents a panel and shell manufacturing operation for the aerospace industry and involves different collaboration levels between the human operator and the robot. The collision-based system is made using mesh colliders rather than shape colliders to increase accuracy. The user is guided using cognitive aids during the operation in the simulation. The two different investigated safety mechanisms are the speed reduction

and move back strategies. One of them limits the speed of robot joints and the other moves the robot links away from the operator. For these systems to activate, various checks such as speed, distance and directory of the end effector are also considered. The test group are considerably experienced but most of them failed to complete the process because of motion sickness caused by the virtual reality simulation. This highlights the importance of setting the VR integration of such simulations properly. The move back strategy was found inferior because it reduced the system efficiency more compared to the other configuration while both were sufficiently safe. During testing, collision safety was assessed using maximum acceptable robot speeds depending on the body part collided and all collisions were in the safe zone. This example can also be used to assess different safety strategies however, it is not possible to consider the strategies on a different robot or end effector.

Virtual reality can also be used in HRC applications without a robot arm. In the article "Virtual Reality for Virtual Commissioning of Automated Guided Vehicles", these kinds of virtual simulations were used to assess an application with an automated guided vehicle (AGV) [31]. An interaction device was used by the human operator to communicate with the robot. The methodology steps included the creation of the simulation environment, testing and analysis of the results. The surroundings during the operation are parallel racks. Human operator was simulated with a virtual model and the actions were captured from reality using virtual reality equipment. Optical and inertial systems are investigated for motion capture and it is concluded that while optical systems offer better precision, inertial systems were preferred because of their comparatively lower cost. The lack of precision present in the motion capture was compensated by using a headset which offers absolute tracking. HTC VIVE headset and Unity software are again used in this system while creating the simulation. A smartwatch was also connected to the system as the interaction equipment between the robot and the human operator. This setup was used to test and validate the operation before the real application was built similar to the other cases. Original control software of the robot provided by the manufacturer was used during the simulation which increases the accuracy which is needed in these kinds of prototyping activities. This research proves that virtual simulations can be used to assess all kinds of robotic operations rather than just assembly operations which were the examples in the previous research cases.

Because there are lots of academic research going on about this subject there are also articles which aims to create a framework for such applications. In "A Framework for Realizing Industrial Human-Robot Collaboration through Virtual Simulation", such undertaking is done prioritizing safety, intuitive interfaces and design methods [32]. The framework consists of five different steps which are supposed to be used in an iterative manner rather than linear. The first step is the definition of scope which can

be done in top-down or bottom-up manners. The feasibility of making a HRC application at this stage should be assessed by considering its pros and cons. For example, ergonomics and safety of the human operator; maintenance and cleanliness of the robot should be considered. In addition, work partition should be made according to HRC theory. The second step is about describing the current state which is achieved by collecting all of the existing data relevant to the system. Accordingly, all information about the operations in the system, components of the system, flows and layouts in the system should be investigated. Third phase is the definition of objectives which introduces Key Performance Indicators (KPI) to the system to optimize it. Which KPIs to prioritize should be defined according to the goals. Fourth part is about creating conceptual solutions which are created with educated estimates. These solutions can be verified with the simulation at a later stage. Final phase is about creating the simulation with virtual reality and virtual environment integration. The model creation phase can also be implemented as three steps with first being a static model, second being a model with motion and third being an interactive stage. In the case study part, the five steps were followed to create a virtual simulation. After defining the scope with a bottom-up approach and creating a table for recognizing the current state, KPIs were found. Labor cost, cycle time and job satisfaction are among the KPIs considered. Two conceptual solutions are found and compared using these indicators to find the appropriate solution. However, it was noted in the conclusion that the amount of work required to create a new simulation is high and the model reusability is low.

The last article examined uses machine learning to train a collaborative robot using a virtual reality simulation. This article is named "Machine Learning Concepts for Dual-Arm Robots within Virtual Reality" [33]. Unity was used in this application too because of its comprehensive libraries. The YuMi- IRB 14000 robot is used in the application which is a dual armed robot. This robot can better emulate human operator's actions because it is closer in geometry to a human body. Real characteristics of the robot is captured from the reference model to increase the accuracy of the system. The task used in the research is an assembly task consisting of pin-back buttons which are assembled with a mechanical press. The task is separated in time domain considering robot idle time, robot alone operation time and collaboration time. Safety precautions were included during collaboration as usual. Thirty different configurations were run at once with categorized models to reduce the iterations required. A reward and penalty method was utilized first. Parameters such as end effector location, goal location and gripper dimensions are used. Aim was to reduce the distance between container and end effector at each step however because of instinctive problems of Unity engine the results didn't converge as expected. The first correction done to prevent this issue was modifying the reward and penalty

mechanisms. New penalties are introduced to the system and reward distribution is modified to make the convergence faster. This methodology worked as the convergence occurred however, the movement of the robot was not feasible as it ignored collisions with the environment. To solve this issue, a collision detection system is integrated which cancels the iterations that have active collisions. The collision-based approach is found to be better than the firstly tried vector-based approaches as it gave proper results. It is mentioned that the system can fail if multiple working colliders are present in the range and the data transfer limit prohibits the code from working on a real machine the same way it works on a simulation. This example shows how theories from different fields can be tested using these kinds of simulations.

Following parts of this chapter explains the theoretical math behind the forward and inverse kinematics operations which are vital operations in robotics systems and also includes required previous knowledge to understand such operations. In this thesis project, these calculations are done by the Moveit motion planner which is a package for ROS. The methodology used by this planner and an example algorithm is also included in this part of the report. It is an informative part to review the knowledge and it will also be referenced in the other chapters to explain the reasoning behind some outcomes.

1.1 Trigonometry

Basic trigonometric knowledge is required to solve forward and inverse kinematics problems. These formulas are added here as a quick review.

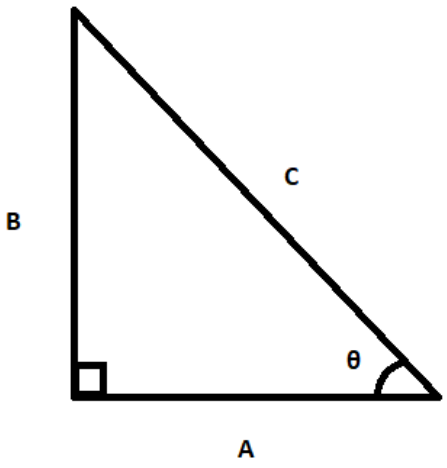


Figure 1: A right triangle.

Pythagorean Theorem: Explains the relationship between the length of the hypothenuse of a right triangle and the length of the other two sides.

$$A^2 + B^2 = C^2 \tag{1.1a}$$

Definition of Basic Trigonometric Functions: Explains the definitions of mostly used trigonometric functions which are sines, cosines and tangent functions.

(1.2a)

$$\begin{cases} \sin(\theta) = B/C & \text{(1.2b)} \\ \cos(\theta) = A/C & \\ \tan(\theta) = B/A & \text{(1.2c)} \\ \text{atan}(\theta) = A/B & \text{(1.2d)} \end{cases}$$

Law of Cosines: Explains the relationship between the length of one side of any triangle, the length of the other sides and the angle between the other sides.

$$\sqrt{A^2 + B^2 - 2AB * cos\theta} = C \tag{1.3a}$$

1.2 Rotation and Transformation Matrices

A pose is defined with a position and orientation. To make mathematical calculations using various poses, the relationship between them must be understood. This section gives information about rotation and transformation matrices in 3D space. The methodology of explaining the transformation matrices and basics of forward and inverse kinematics is taken from the informatory videos prepared by Woofrey [29].

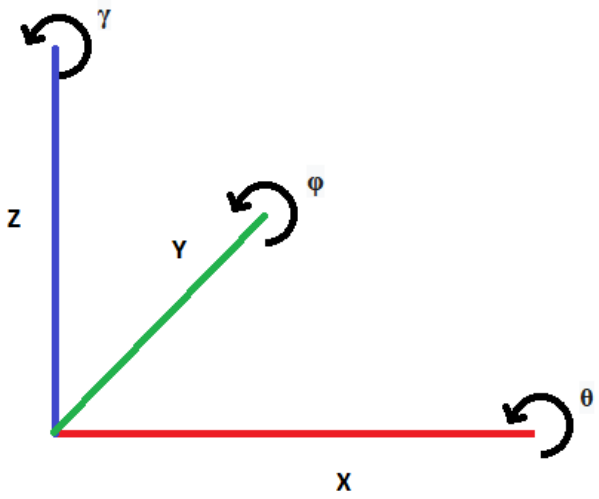


Figure 2: Axes on a 3D plane.

Translation in 3D space: Basic translation operation for two points in the same reference frame. Here you can see a translation from point 2 to point 1.

$$p_2^1 = \begin{bmatrix} x_1 - x_2 \\ y_1 - y_2 \\ z_1 - z_2 \end{bmatrix} \tag{1.4a}$$

1.2.1 Rotation Matrices

Rotation in 2D space: Rotation from orientation 2 to orientation 1 has the following formation in the 2D space.

$$R_2^1(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \tag{1.5a}$$

Rotation in 3D space: When we translate the same matrix into 3D space, we get the following results. In these formulations θ shows the rotation around x axis, φ shows the rotation around y axis and γ shows the rotation around z axis.

$$\begin{cases} R_2^1(\theta, \varphi, \gamma) = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} & (1.6a) \\ R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix} & (1.6b) \\ R_y(\varphi) = \begin{bmatrix} \cos(\varphi) & 0 & \sin(\varphi) \\ 0 & 1 & 0 \\ -\sin(\varphi) & 0 & \cos(\varphi) \end{bmatrix} & (1.6c) \\ R_z(\gamma) = \begin{bmatrix} \cos(\gamma) & -\sin(\gamma) & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 \\ 0 & 0 & 1 \end{bmatrix} & (1.6d) \end{cases}$$

From the rotation matrix in 3D space the roll, pitch and yaw values can be calculated using the following formula. It should be noted that atan2 function is similar to atan function however it gives results in the range of pi to -pi.

$$\begin{cases} \theta = atan2(r_{32}, r_{33}) & (1.7a) \\ \varphi = \text{atan2}(\text{r}_{21}, \text{r}_{11}) & (1.7b) \\ \gamma = \text{atan2}\left(-\text{r}_{31}, \dfrac{r_{21}}{\sin(\varphi)}\right) or\ \text{atan2}\left(-\text{r}_{31}, \dfrac{r_{11}}{\cos(\varphi)}\right) & (1.7c) \end{cases}$$

These rotation matrices are used to describe the relative rotation between the reference frames during robotics calculations.

1.2.2 Transformation Matrices

As the reference frames in our calculations usually have different orientation and location, the translations and rotations should be both taken into account. The resulting transformation matrix is as follows.

$$T_2^1 = \begin{bmatrix} r_{11} & r_{12} & r_{13} & x_1 - x_2 \\ r_{21} & r_{22} & r_{23} & y_1 - y_2 \\ r_{31} & r_{32} & r_{33} & z_1 - z_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.8a)$$

It can be noted that the upper left 3x3 part of the matrix is the rotation matrix between the two reference frames and the upper right 3x1 part is the translation matrix between the two reference frames. If transformation between multiple frames is required, these transformation matrices can be used as follows.

$$T_D^A = T_D^C T_C^B T_B^A \quad (1.9a)$$

To use a transformation matrix to translate the location of a point it is impossible to multiply the 4x4 translation matrix with a 3x1 point. So, the point vector is modified to become a 4x1 matrix to allow the calculation.

$$p_1' = \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} \quad (1.10a)$$

1.3 Forward Kinematics

The forward kinematics problem is a common case in robotics where the joint positions **q** is known and the location of the end effector of the robot is desired. As the length and geometry of the links of the robot is constant, the location of the end effector can be found using the joint positions. In simple cases, the location of the end effector can be calculated using basic trigonometric relations (1.2). Below is an example for a robot with two links and two joints in a 2D space.

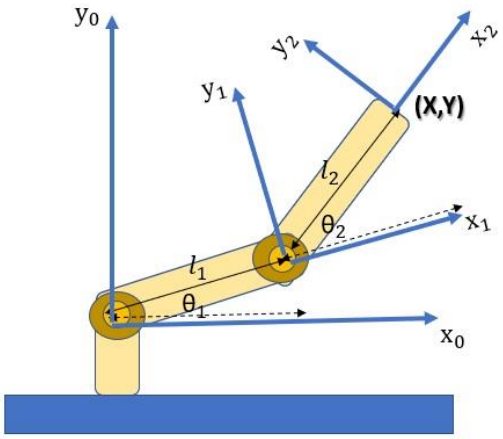


Figure 3: 2 joint, 2 link robot in 2D space [17].

$$\begin{bmatrix} x \\ y \\ \theta \end{bmatrix} = \begin{bmatrix} l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2) \\ l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2) \\ \theta_1 + \theta_2 \end{bmatrix} \tag{1.11a}$$

In this example the joints are revolute joints. In the case of 3D space, the revolute joint has 3 types. The ball joint can rotate among all 3 axes, the universal joint can rotate in only 2 axes and the single revolute joint is limited to 1 axis of rotation. Another common type of joint is the prismatic joint which offers a translative motion rather than rotation. The robots used in the thesis only consists of single revolute joints so joint position information contains only one angle for each joint.

While this example is rather basic, in a normal robotics calculation more joints are present, and 3D space should also be considered. This makes the calculation rather difficult with just using the basic trigonometric functions. To solve this issue, the transformation matrix (1.8a) between each joint can be written and multiplied (1.9a) to achieve the pose of the end effector. The transformation matrices between the links are written with four parameters. The parameters designating the third axes is not needed as the cross product of the other axes already cover all possible motion in that axis. These four parameters needed is called Denavit-Hartenberg parameters. While the order of the parameters can change based on nomenclature, the sequence should be the same throughout the calculations because the matrices are not commutative. The calculations needed to get these parameters are given below.

$$T_{i-1}^{i} = T_{Rz}(\gamma_i) T_z(c_i) T_{Rx}(\theta_i) T_x(a_i) \tag{1.12a}$$

In this case the first matrix symbolizes a rotation by γ around the z axis. The second matrix is a translation by c in the z axis. The third matrix is a rotation by θ around the

x axis. The last matrix is a translation in the x axis by a. As the cross product of x and z axes cover the y axis, the parameters of that axis were omitted. This helps with the computational speed when many joints are present as having more parameters for each joint significantly effects calculation time. The four matrices explained above have the following structures.

$$T_{Rz}(\gamma_i) = \begin{bmatrix} \cos(\gamma_i) & -\sin(\gamma_i) & 0 & 0 \\ \sin(\gamma_i) & \cos(\gamma_i) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.13a}$$

$$T_z(c_i) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & c_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.13b}$$

$$T_{Rx}(\theta_i) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta_i) & -\sin(\theta_i) & 0 \\ 0 & \sin(\theta_i) & \cos(\theta_i) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.13c}$$

$$T_x(a_i) = \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.13d}$$

It should be noted that the above matrices come from the previous matrices noted in this chapter. The upper right 3x3 part of the first matrix is clearly the rotation matrix in the z axis (1.6d) and the upper right part of the third matrix is the rotation matrix in the x axis (1.6b). To calculate the transformation matrix of the end effector now we need to start from the base and include all of the transformation matrices in between. For an end effector which is the nth reference frame in the calculation, the formula is given below.

$$T_0^n = \prod_{i=1}^{n} T_{i-1}^i \tag{1.14a}$$

Most joint types can't have motion in all four DH parameters. In the single revolute type of joint for example, only one rotation parameter can be used with the rest of the matrices being reduced to identity matrices by aligning the local axes accordingly.

Similarly, when using a prismatic joint only one translation matrix can be used if the motion provided by the joint aligns with one of the axes taken.

1.4 Inverse Kinematics

Inverse kinematics problem is the inverse of the forward kinematic problem which searches for the joint values when the position of the end effector is given. This information is useful in motion planning because usually the area of interest is the location of the end effector and to move the end effector to the desired location, an inverse kinematics calculation is required to get the required joint values. As a basic example we can again consider the robot arm in the Figure 3 and this time try to find the joint values θ_1 and θ_2 using the end effector location (x,y). As with the notation in 1.11a we have 2 unknowns and 2 given variables.

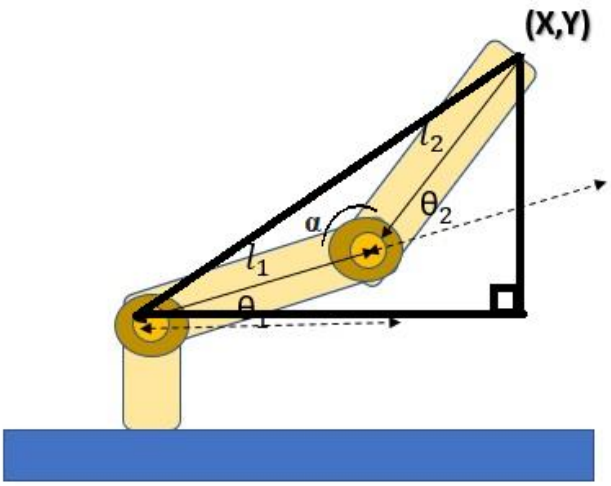


Figure 4: 2 joint, 2 link robot in 2D. Modified to find θ_2 [18].

The angle between the robot links is designated as α and a right triangle is drawn so the trigonometric relations can be used. According to the Pythagoras theorem (1.1a) the length of the hypotenuse of this triangle can be calculated as $\sqrt{x^2 + y^2}$. Then the law of cosines (1.3a) can be applied to the triangle consisting of this hypotenuse side and the two links of the robot.

$$l_1{}^2 + l_2{}^2 - 2l_1l_2 * cos\alpha = x^2 + y^2 \tag{1.15a}$$

Modifying this equation, we can find α which can be used to find θ_2. It can be noted that there are two possible solutions for θ_2.

$$\alpha = \text{acos}\left(\frac{l_1^2 + l_2^2 - x^2 - y^2}{2l_1 l_2}\right) \tag{1.16a}$$

$$\theta_2 = \pi - \alpha = \pi - \text{acos}\left(\frac{l_1^2 + l_2^2 - x^2 - y^2}{2l_1 l_2}\right) \tag{1.16b}$$

$$\theta_2 = -\alpha = \text{acos}\left(\frac{x^2 + y^2 - l_1^2 - l_2^2}{2l_1 l_2}\right) \tag{1.26c}$$

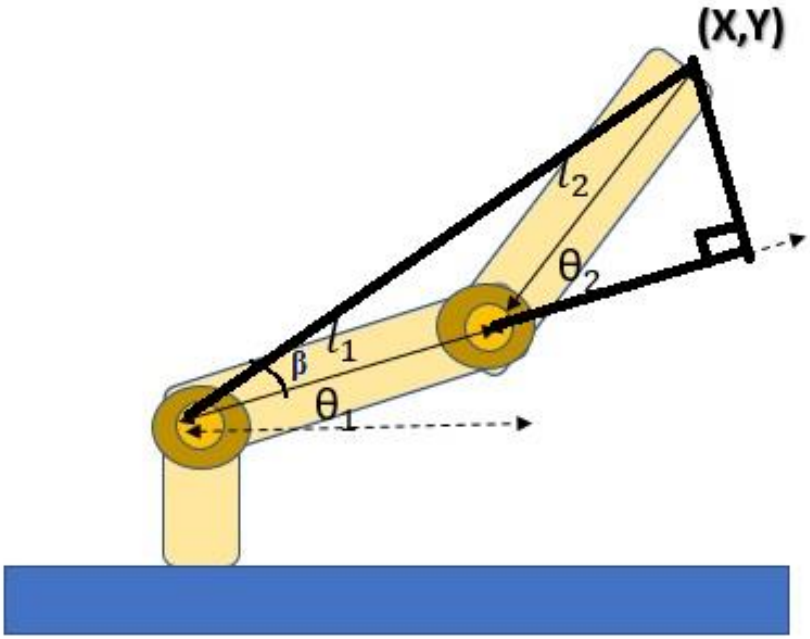


Figure 5: 2 joint, 2 link robot in 2D. Modified to find θ_1 [18].

To find θ_1, the creation of another right triangle is needed. The length of the additional sides is found using trigonometric functions. Like the previous case, two solutions are present because θ_2 has two possible solutions. This means that two distinct joint configurations can result with the same end effector location. These two configurations are shown in Figure 6.

$$\theta_1 + \beta = \tan(\frac{y}{x}) \tag{1.17a}$$

$$\beta = \operatorname{atan}(\frac{l_2 \sin(\theta_2)}{l_1 + l_2 \cos(\theta_2)}) \tag{1.17b}$$

$$\theta_1 = \operatorname{atan}\left(\frac{y}{x}\right) - \operatorname{atan}(\frac{l_2 \sin(\theta_2)}{l_1 + l_2 \cos(\theta_2)}) \tag{1.17c}$$

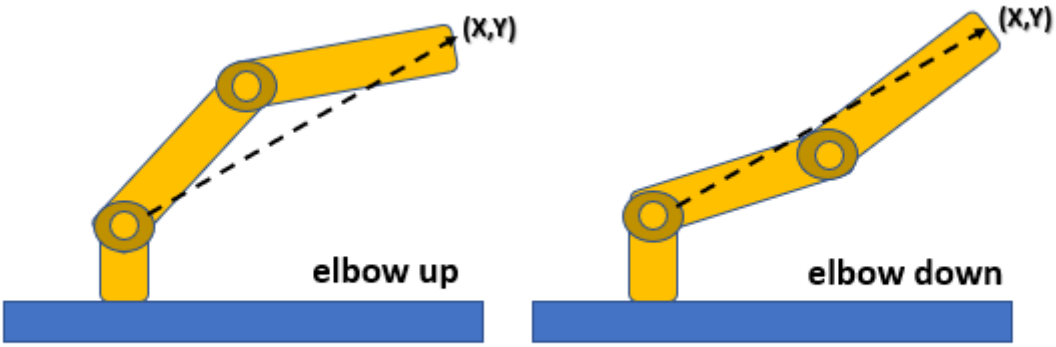


Figure 6: Two distinct joint configurations which satisfy the equation [18].

In the cases where there are more joints than two, the problem becomes more complex as the unknown number surpasses the number of given values. This creates an infinite number of possible solutions. Of course, it is still possible to have unique solutions in special cases or no solutions in the cases which the robot can't reach the desired end effector location. Reachability problems is usually caused by the desired end effector locations being too close or too far compared to the base of the robot.

Like the forward kinematics problem, inverse kinematics problem also increases in complexity with an increase in the number of joints. However, unlike forward kinematics, inverse kinematics problems don't have unique solutions, so some type of assumptions are required to solve these problems. There are many methods for solving inverse kinematics problems with different types of approaches to the problem. In the following part of this subchapter some of the popular methods for solving this problem is explained. Which one to choose between these methods usually depends on the parameters of the given problem.

1.4.1 Analytic Solutions

One way to approach the inverse kinematic problem is to solve it analytically. In the previous part it is shown that in the cases more complex than the one studied, there are usually infinite solutions. This requires some kind of assumptions to sort out these solutions and to find the most suitable one. The most suitable solution can be addressed as the most stable one considering the motion of the robot [12]. The analytical solution that will be considered is made for 6R robotic manipulators. This means that it is made for a robot with a single arm with 6 different joints. The robots used in the next chapters of the thesis falls in this category. The transformation matrix used in the inverse kinematics calculations are given below [13].

$$A_i = \begin{bmatrix} cos\theta_i & -sin\theta_i cos\alpha_i & sin\theta_i sin\alpha_i & a_i cos\theta_i \\ sin\theta_i & cos\theta_i cos\alpha_i & -cos\theta_i sin\alpha_i & a_i sin\theta_i \\ 0 & sin\alpha_i & cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.18a}$$

The θ parameter in this matrix symbolizes the joint rotation angle of the ith joint. α is the twist angle of the joint and a is the length of the next link. The last parameter d is the offset distance at joint i [13]. The result expected is the transformation matrix for the end effector from base. This transformation matrix is found using the formulation at 1.9a as show below [13].

$$A_{ee} = A_1 A_2 A_3 A_4 A_5 A_6 \tag{1.19a}$$

$$A_{ee} = \begin{bmatrix} l_x & m_x & n_x & q_x \\ l_y & m_y & n_y & q_y \\ l_z & m_z & n_z & q_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.19b}$$

Because the A_{ee} matrix is orthonormal, only 6 unique equations are present in the calculation [13]. If we consider that there are only 6 joints in a 6R manipulator and all of them are single revolute joints, the problem has 6 equations and 6 unknowns.

In the Raghavan and Roth solution, the system is reduced to a 16-variable polynomial from which the angle ϑ_3 can be calculated [13]. The other variables are later found substituting this angle in their respective equations. First the equation at 1.19a is modified so that three of the transformation matrices are in the same side with the transformation matrix for the end effector [13].

$$A_{ee} A_1^{-1} A_2^{-1} A_6^{-1} = A_3 A_4 A_5 \tag{1.20a}$$

Then the equations are represented in a different form consisting of two matrices. These are the Q matrix which is a 14x8 matrix and P matrix which is 14x9. The entries of the Q matrix are the parameters of the manipulator and the pose of the end effector while the entries of the P matrix include functions of $sin\theta_3$ and $cos\theta_3$ [13].

$$(Q)\begin{pmatrix} sin\theta_1 sin\theta_2 \\ sin\theta_1 cos\theta_2 \\ cos\theta_1 sin\theta_2 \\ cos\theta_1 cos\theta_2 \\ sin\theta_1 \\ cos\theta_1 \\ sin\theta_2 \\ cos\theta_2 \end{pmatrix} = (P)\begin{pmatrix} sin\theta_4 sin\theta_5 \\ sin\theta_4 cos\theta_5 \\ cos\theta_4 sin\theta_5 \\ cos\theta_4 cos\theta_5 \\ sin\theta_4 \\ cos\theta_4 \\ sin\theta_5 \\ cos\theta_5 \\ 1 \end{pmatrix} \tag{1.21a}$$

These relationships expressed are then used to eliminate some of the variables in the function. Left-side terms are written with their equivalents from the right-side terms to eliminate θ_1 and θ_2 from the equation [13]. Then using some trigonometric substitutions, the equation is written as follows [13].

$$sin\theta_i = \frac{2x_i}{1-x_i^2} \tag{1.22a}$$

$$cos\theta_i = \frac{1-x_i^2}{1-x_i^2} \tag{1.22b}$$

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} & o \\ A_{21} & A_{22} & A_{23} & o \\ o & A_{11} & A_{12} & A_{13} \\ o & A_{21} & A_{22} & A_{23} \end{pmatrix}\begin{pmatrix} x_4^3 x_5^2 \\ x_4^3 x_5 \\ x_4^3 \\ x_4^2 x_5^2 \\ x_4^2 x_5 \\ x_4^2 \\ x_4 x_5^2 \\ x_4 x_5 \\ x_4 \\ x_5^2 \\ x_5 \\ 1 \end{pmatrix} = 0 \tag{1.22c}$$

The x_i notation present in the formulas represents $tan\frac{\theta_i}{2}$. A_{ij}terms are 3x3 matrices and o terms are 3x3 null matrices [13]. The determinant of the left-side of the matrix is a polynomial of the 24th degree in x_3 which can be divided by $(1 + x_3^2)^4$ to find 16 roots for the solutions of the inverse kinematic problem [13]. After finding θ_3, the rest of the unknowns can be calculated using the relations created by the transformation matrices to find the joint values required. There are other analytical methods which uses pre-processing, eigenvectors and eigenvalues [13].

1.4.2 Numeric Solutions

Instead of finding a general solution for the problem, the numerical methods aim to solve the problem using iteration. A cost function is created and is tried to be minimized to achieve the end result [12]. There are many numeric methods to approach the inverse kinematic problem in the literature. The method explained below is a Jacobian based method which offers a linear approximation to the inverse kinematic problem [12].

The end effector's location can be shown as a function of joint values as seen in the forwards kinematics section. In this method, a Jacobian matrix is created to approximate the joint values using the end effector position function s_i [14].

$$J(\theta) = \left(\frac{\partial s_i}{\partial \theta_i}\right)_{i,j} \tag{1.23a}$$

Then the joint values can be calculated using the following formula of forward dynamics [14].

$$\dot{s} = J(\theta)\dot{\theta} \tag{1.24a}$$

The variables for the joint values and the target location are known at the start of the calculation. The s_i function can be calculated using these with the 1.14a formula. With all these known, the Jacobian matrix can also be calculated. The method aims to increase joint angles (θ) incrementally to bring the end effector location closer and closer to the desired target location [12]. It is done using the approximation given below in which the Jacobian matrix is calculated with the unchanged values of s and θ.

$$\Delta s \approx \vec{e} = J\Delta\theta \tag{1.25a}$$

The iterative update continues until the end effector location is sufficiently close to the desired target location [14]. There are different methods to choose how to update the joint angles and some of them is explained below.

Jacobian transpose method: Uses the transpose of the Jacobian matrix during the calculations rather than the inverse [14]. The equation for $\Delta\theta$ becomes as follows where α is a scalar parameter.

$$\Delta\theta = \alpha J^T \vec{e} \tag{1.26a}$$

It can be deduced that the transpose of a matrix is not always equal to the inverse of the same matrix. In this situation however the transpose of the Jacobian matrix can be used as a substitute of the inverse of the Jacobian matrix in terms of virtual forces [14]. As explained in the study [14], by choosing the α value small enough the error vector $\vec{e}$ can be minimized. The α value recommended is given below.

$$\alpha = \frac{\vec{e} \cdot JJ^T \vec{e}}{JJ^T \vec{e} \cdot JJ^T \vec{e}} \tag{1.27a}$$

Jacobian pseudoinverse method: This method uses the pseudoinverse if the Jacobian matrix which is defined for all Jacobian matrices [14]. The main equation used in this method is the one below where J^γ symbolizes the pseudoinverse of the Jacobian matrix.

$$\Delta\theta = J^\gamma \vec{e} \tag{1.28a}$$

Starting from the equation 1.25a we can derive the equation for the pseudoinverse of the Jacobian matrix [14]. The derivation is given below. First both sides are multiplied with the transpose of the Jacobian matrix. Then the $\Delta\theta$ parameter is left alone in one side of the equation.

$$J^T \vec{e} = J^T J \Delta\theta \tag{1.29a}$$

$$\Delta\theta = J^T (JJ^T)^{-1} \vec{e} \tag{1.29b}$$

While this method gives the best solution for the $\vec{e}$ equation, it usually performs poorly because of instability near singularities [14].

Damped least squares method: This method aims to stabilize the previous method. It changes the cost function to be minimized from the 1.25a function to the following [14]. The λ in this equation is a scalar damping constant.

$$\|J\Delta\theta - \vec{e}\|^2 + \lambda^2 \|\Delta\theta\|^2 \tag{1.30a}$$

This equation can be rewritten as the following after some matrix manipulations. Then the position of the J^T term can be moved to the beginning of the right side because of the equality of the functions [14].

$$\Delta\theta = (J^T J + \lambda^2 I)^{-1} J^T \vec{e} \tag{1.31a}$$

$$\Delta\theta = J^T (J^T J + \lambda^2 I)^{-1} \vec{e} \tag{1.31b}$$

There are other numeric methods in the literature such as the selectively damped least squares method which utilizes singular value decomposition to get better results however, this method is a stable method to solve for $\Delta\theta$ in any range.

1.4.3 Sampling Based Methods

In the case of Analytic or Numeric methods it can be seen that the computational power required, and complexity increases exponentially as the number of joints in the robot increases. To solve this issue, randomized and probabilistic methods were investigated. While they don't offer a complete solution, they are less heavy in the computational requirements [15]. Sampling based methods are among these types of motion planning methods. The Open Motion Planner Library (OMPL) contains implementations of many algorithms in this category when calculating trajectory and it is the main solver of Moveit. As Moveit is the motion planner used in the project, these calculations are relevant to the thesis. Below the Probabilistic Roadmap Methods (PRM) will be explained which is one of the algorithms present in the OMPL.

The PRM method aims to create a roadmap in the pre-processing phase which will then be used to connect the start and end location in the current task. This reduces the calculation time greatly and many queries can be answered in the manner of fractions of seconds [15].

The pre-processing phase of the PRM method uses two parts to create a roadmap. The first part is the node generation which uses node generation algorithms to create robot configurations which will be used as nodes in the roadmap [15]. The configurations exhibiting collisions are eliminated to get the meaningful nodes. This step is simplified in Moveit by creating a self-collision matrix for the robot which reduces the computational power required for these checks. A uniform sampling strategy can be used to create the nodes however, this will most likely create insufficient samples in the areas which the robot has difficulty accessing. Because of this issue, most sampling algorithms uses information about the environment and robot geometry to modify the generation of nodes in order to sufficiently create nodes in these critical regions [15]. This collision checks that are done by Moveit only considers the environment present in the ROS workspace so any obstacles present in other workspaces should also be modeled in ROS environment for the motion planner to work correctly.

The second phase of pre-processing is the connection of these nodes to create the roadmap. Local planning methods are used to connect these nodes starting from the less computation heavy connections moving on to the more complex ones later on [15]. This part is where the majority of the computation is done in the pre-processing phase. The easiest node connections are made utilizing thousands of the fastest local planning

methods. Then more powerful connecting methods are used to connect intermediate node connections. The third stage involves addition of new nodes to try to connect the parts of the previous roadmap [15]. This strategy has multiple phases so it can use lightweight connection techniques to create the majority of the roadmap and utilize more computationally heavy ones to create the critical regions. This reduces the overall computational power needed while keeping the coverage of the code constant. This roadmap created is then used to plan the movement of the robot in the designated domain.

2. Architecture of the Project

The goal of the thesis project is to create a system in which multiple different robots with different end effectors and an immersive and realistic simulation of a human operator can be simulated in a digital environment. As the main targeted use of the system is educational, it is designed to familiarize the user with how robot simulating works and how different applications can be simulated with open-source tools. In order to achieve this objective, multiple robot arms are included in the project and multiple end effectors for the arms are selectable. The motion in the scene is also modified based on the nature of the end effector selected. Configuration of new robots and end effectors are also possible for the system and a detailed guide on how to add a new robot or end effector is included in the part 3.5 of the report.

As these collaborative robots are usually used in HRC applications, an immersive and realistic simulation of the human operator is required. Similar to many applications in the industry, a VR headset was used to connect the user to the simulation in the role of the operator. This allows the system to simulate not only the robot but the whole HRC application which increases its educational capabilities. The system can be used to give basic information about how these systems work however, as the robot parameters are given arbitrarily, the simulation can only be accurately used for operator training if robot parameters are correctly input.

The system can be divided into two different environments. The first environment is the ROS part of the system which runs in Linux software. As stated before, ROS is an open-source robot operation software which is the go-to tool when creating these kinds of simulations. It can also operate real world robots with the same code, provided that the files created give the necessary information required by the system to make the right calculations. This part of the system is where the calculations of the robot movement happen. ROS is chosen frequently for these calculations because it has the Moveit package which is an advanced motion planner. This package reduces the work required to create such simulations immensely by offering a premade planner with up-to-date inverse kinematics calculators, controllers, collisions checks and many more functions which are required in such calculations. Configuration is also somewhat straightforward as most ROS packages work with Unified Robotics Description Format (URDF) files which are simple to create from a CAD drawing. The only downside of this software is the need for a Linux operating system which is not a common operating system used by users however, a virtual box which allows Linux to operate in other operating systems can be utilized to counteract this issue.

The second environment of the project is the Unity environment. As explained before Unity is a platform to create computer games. However, it has many tools such as physics calculators which can be beneficial for an industrial simulation. Also, because it is a popular platform, it has many packages which have animators, UI functions and props which can be used in any application. The main use of the Unity software in the project is the simulation of the surroundings of the robot including the human operator. While ROS is very adept at simulating the robot movement accurately, it is hard to simulate humans with it as the system was not designed for this purpose. Unity offers an easier solution as it contains various tools to simulate humans and environments which are both essential in computer games. In other words, Unity oversees the environment and basic physics calculations in it. Unity also offers many connection packages which makes it the ideal tool to use ROS with VR as it has multiple packages to connect both of these software to Unity. Unity can be used in various different operating systems; however, it was used with Windows operating system in the project to make the VR connection easier. The schematic for the system is given below.

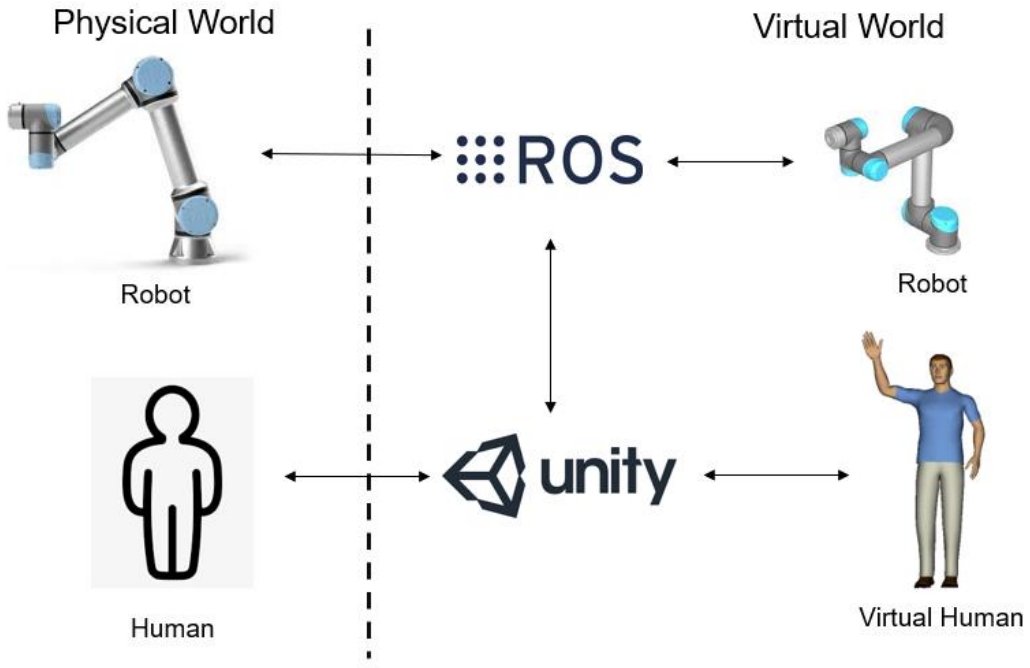


Figure 7: The basic schematic of the system in the right and what they simulate in the left. The software used is given in the middle [19].

2.1 ROS Environment

The ROS environment consists of the part of the project which is run in the Linux operating system. As two different operating systems are required in total, this

operating system was run on a virtual box. To create the virtual box, Oracle VM Virtual Box software was used on a Windows machine and Ubuntu package was set up. ROS Noetic package was built in the virtual box as the software is designed for a Linux operating system. ROS Noetic was chosen because it was the latest ROS package at the time and many tutorials exists to connect it with Unity and configure Moveit with it.

Unlike Unity and most other software, ROS doesn't have a user interface so most of the information here is contained in the packages created. These packages are located in the designated workspace for ROS and are run with the use of terminal. To run a ROS package, the creation of a workspace is mandatory as the software needs to have a location to search for the packages. The steps to create a workspace is explained in many ROS guides and also in ROS documentation. Updating the workspace is done with the terminal commands "catkin build" or "catkin_make" in a folder called "src" in the workspace. This procedure is required to be repeated after configuring a new robot for the system. The packages created and imported can be seen in the screenshot of the ROS workspace below.

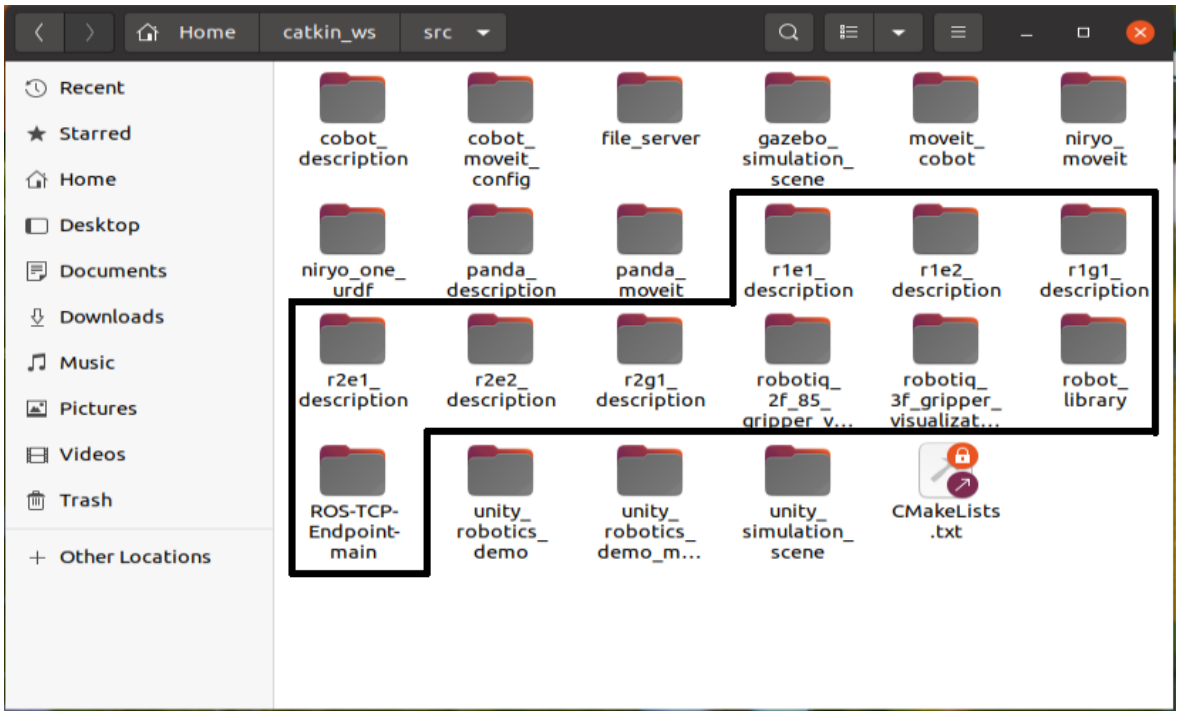


Figure 8: ROS workspace. The marked documents are related to the project.

Most of these files contain information required by Moveit to calculate robot trajectories. The URDF files which contain the information about the characteristics of the robots and the Moveit configurations made from these URDF files create most of the files here. The last marked file however, as its title states is a connector. This is the

ROS side of the ROS TCP Connector software which is the software used to connect ROS to Unity.

2.2 Unity Environment

Unity environment is the part of the project which runs in the Windows operating system. It contains five packages which are imported using Unity's package manager. The first package is a generic package which contains various prefabs which are used in the creation of the environment. A similar package or other strategies could be used in a similar manner. Therefore, this package doesn't contain anything strictly required for the system. TextMeshPro is another package used in a similar manner, which contains some helpful tools when creating the UI in Unity. In other words, Custom created UI or other similar packages can be used for the same result. The three crucial packages are ROS Unity TCP Connector, SteamVR Plugin and URDF importer. ROS Unity TCP connector is the Unity side of the connector package which establishes connection between the two environments. This is done via network as the operating system on the virtual box can't communicate with the home operating system in a meaningful manner. The URDF importer is also a tool to help with the ROS integration. It creates Unity models from URDF files which are the main description files for ROS. However, this package has some limitations which are discussed in the implementation part. The last package used is the SteamVR plugin which contains the necessary software to connect VR hardware with Unity. Details on why this package was chosen and how it operates are also given in later sections.

Other than the packages, Unity also contains many scenes and assets for the scenes. The scenes contain the game objects which are connected to the necessary information in the assets such as scripts, models and materials. Each scene can represent a level or a menu in a computer game. In this application one represents the UI while others are for the simulations. As six different configurations exist in the current state of the project, six different simulation scenes are present. The rest of the scenes seen in figure 9 are the ones used before VR implementation and the ones used to make the implementation easier. For clarification, the unmarked scenes are not used in the final version of the project.

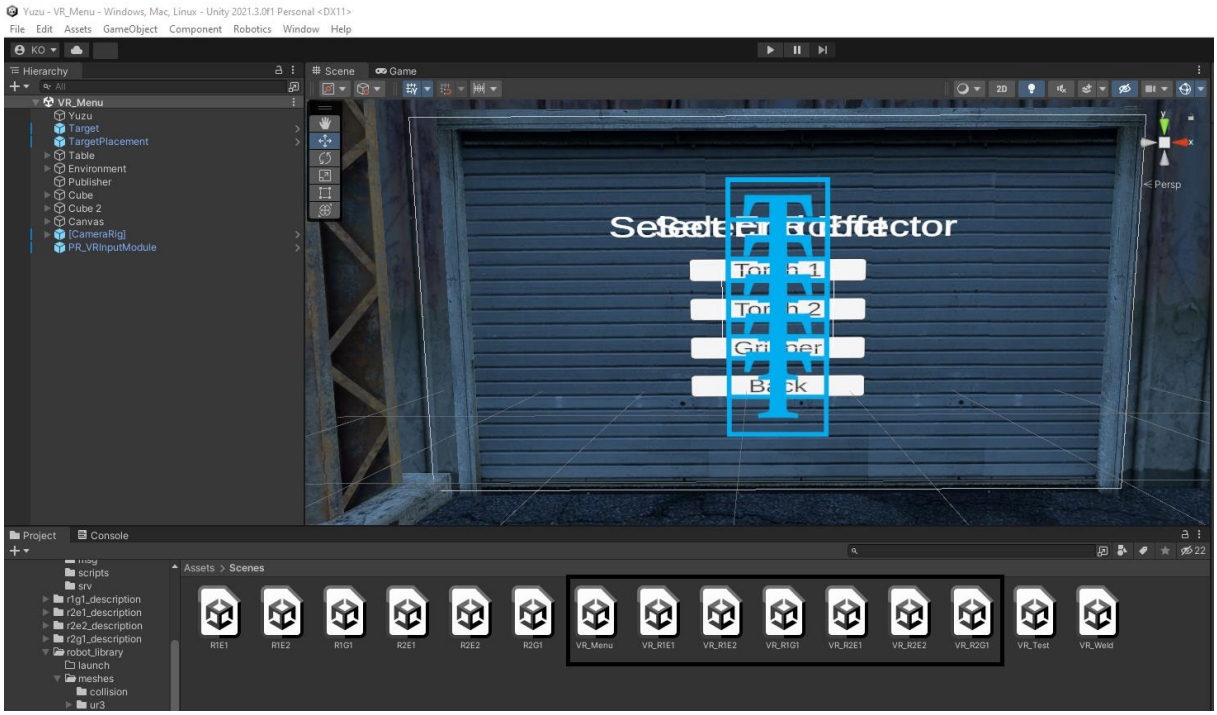


Figure 9: Unity project. The marked scenes are used in the final version of the project.

Unity contains many assets from the ROS environment too as this is a requirement for ROS TCP connector. Most of the files that are used in the communication also exist in the Unity side as the Windows operation system has no access to the original files in the virtual machine. URDF files are also present here to be imported in Unity scenes by the URDF importer package.

2.3 Other Requirements

Other than Unity and ROS, some other software are also required to run the project in its current state. For the SteamVR Plugin to work, the Steam software should be downloaded, and Steam VR should be run from the tools section in the library. Both Steam and Steam VR is available for free however they require an account to be used. This is another software made for gaming repurposed in the context of the thesis for simplification of many tasks which would take more time otherwise.

The other required software is the Oculus software to connect the VR headset to the computer. This is the software required by the specific Oculus Quest 2 headset used for testing. Likewise, other headsets might require their own software. Downloading and running these software are enough to launch the system, hence no configuration needs to be made on them.

While Oculus Quest 2 headset is not required specifically, one VR headset with controllers is required to run the simulation. Controllers are needed to navigate in the created environment and interact with the objects. A cable is also required to connect the headset to the computer. As Steam VR is an application for the computers operating system, the system can't run on the headset alone. This however doesn't really create an extra limitation as ROS also can't run in the headsets operating system, so the use of a computer is mandatory in any case.

3. Implementation

The implementation of the project has four different parts. The first part is the creation of the ROS environment, the second part is the creation of the Unity environment, the third part is the configuration of the movement script, and fourth part is the addition of the VR integration to the Unity environment. VR integration and movement script are parts of the ROS and Unity environments, but they have their own subchapters because of their complexity. A guide to help with the addition of new configurations into the system is also provided in this chapter.

3.1 ROS Implementation

The first thing done in the Linux side of the project was the importation of the robot and end effector URDF files. The URDF files for UR3 and UR5 robots of Universal Robots were taken from GitHub [20]. The depository has the URDF file for UR10 as well, but this robot was not configured in the project. The Unity models of the robots connected with the first welding torch is shown in figure 10. Models present in Rviz is also the same with the imported models in Unity as both are imported from the same URDF files. Rviz is the 3D visualization tool used by ROS as ROS itself has no UI. The robots are very similar in geometry, but their size is vastly different which means they still require different configuration.

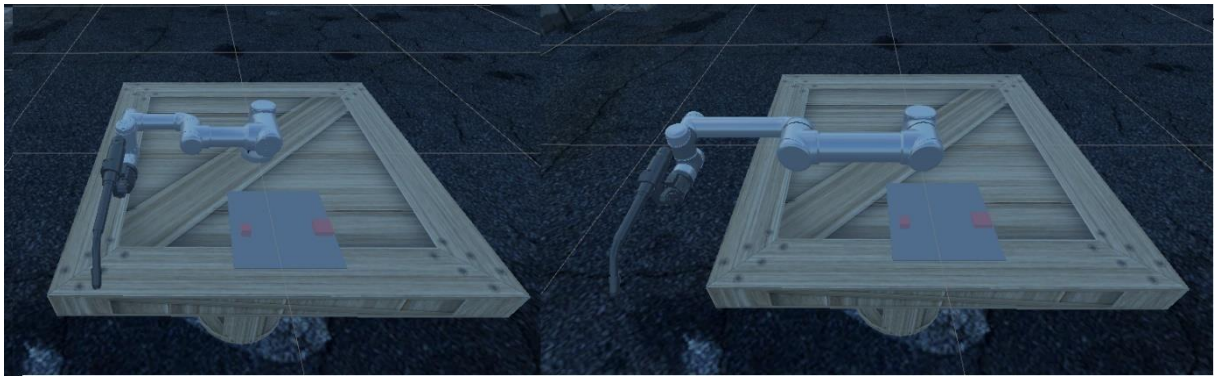


Figure 10: Unity models of the robots with the same end effector.

The end effectors implemented are one gripper and two welding torches. The gripper is the 2f-85 gripper of Robotiq and the URDF was taken from a GitHub depository [21]. One of the torches is a detailed drawing from GrabCad [22] and a free to use asset. The other is a simple model taken from GitHub [20]. A variety of end effectors were

downloaded to show the versatility of the system. The second welder was implemented to test how a new end effector could be implemented in case another with similar purpose already exists in the system. The gripper was implemented to test the new configurations required to add a new action to the system. Welding and pick-place operations are different enough to warrant different integration which might also be needed for another possible end effector addition. The models of the end effectors in Unity are shown in figure 11. It should be noted that because of the parent structure of Unity, the gripper has less virtual connections than the actual produced gripper. The Robotiq 3f gripper was also tried to be implemented however the URDF's internal collisions created perpetual movement in Unity environment. These limitations are further explained in the following sections.

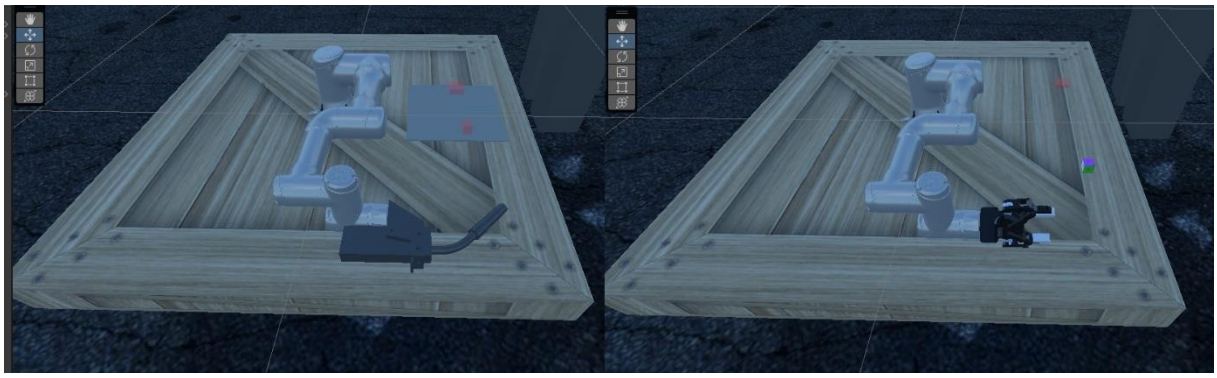


Figure 11: Unity models of the first robot with the remaining end effectors.

Most of these URDF files are put in the "robot_library" folder in ROS. This folder has no launch file, and its sole purpose is to be a library for the robot models. The creation of combined URDF files of robots and end effectors were also performed here. This connection was done with the use of xacro files. Xacro is an XML based macro language which is frequently used during the creation of URDF files. The need for it comes from the fact that URDF files can't contain macros by themselves. Methodology of a xacro file is given in algorithm 1 below.

Algorithm 1 Connecting URDF files with Xacro

1: Designate xml version.
2: Call xacro from ROS depository. Name the xacro file.
3: Import robot and end effector URDF models.
4: Connect the last link of the robot to the end effector.
5: Create a fixed world joint between world and base link.

After the URDF files are combined using xacro files, these files are converted back to URDF format using ROS's conversion command in the terminal. This conversation is necessary as Moveit Setup Assistant application which is used to create Moveit plugins accepts files in URDF format. The basic schematic of this operation is given below.

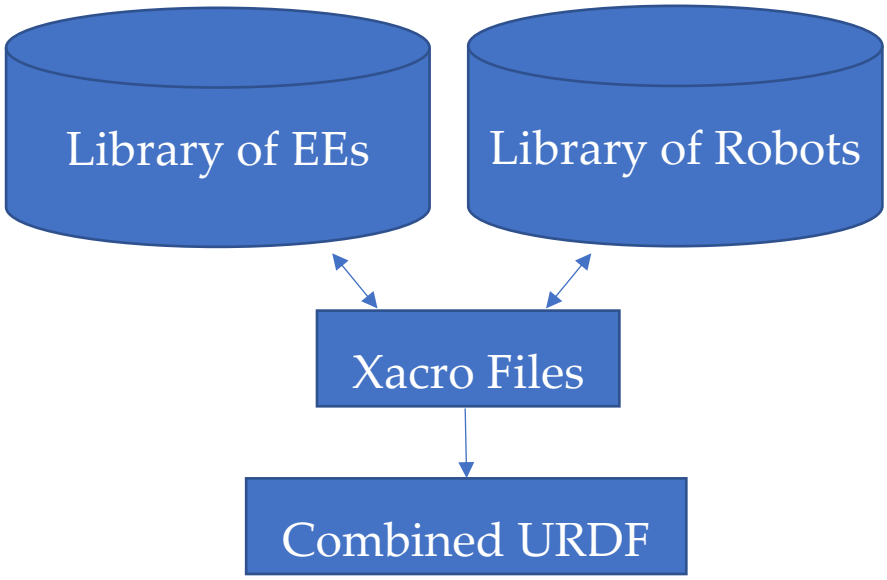


Figure 12: Schematic of the creation of combined URDFs of robots and end effectors.

The next step in ROS domain is the creation of Moveit plugins. Moveit has a tool named Moveit Setup Assistant which simplifies this step. As all ROS plugins, it is run directly from the terminal. Most of the configuration is similar to the one in the Moveit Setup Assistant tutorial [23] with a few changes. The changes are mostly parameter names which should match the ones in the movement code which will be discussed later in the report.

First the self-collision matrix is generated. This step tries possible joint configurations of robot and checks for the links which never come in contact with each other. While calculating the trajectory, this information helps reducing the self-collision checks required hence reducing computation time. Then a virtual joint is added to the robot. This is a fixed joint between the base link of the robot and world node which ultimately keeps the base of the robot in place. Most URDFs have this feature so it may not be required to make this addition. After this stage, planning groups are defined. These are groups of links or joints which are going to be used together during planning. The first group created is named "arm" and it contains all of the joints of the robot arm. KDLKinematicsPlugin is chosen as this group's kinematics solver. This group will be the group that is going to be used during the planning as trajectory calculation will

only be done for the robot arm. The next group created is the hand group which is the group for the end effector. The links of the current end effector is added in this group and no solver is chosen. The last step is creating a controller. It is named "arm_position_controller" in the sample cases but the name can be changed in future configurations as this name is never referenced in the movement code. "JointTrajectoryController" under the "position_controllers" group is chosen as the type of the controller, and it is added to the arm group as expected. Then, the configuration is generated in the workspace. This procedure should be repeated for every robot and end effector combination added to the system.

Moveit Setup Assistant returns a configuration which can be used by Moveit to calculate trajectories in the ROS environment. However, to use this configuration with Unity, additional steps need to be taken. This includes adding additional files to enable the connection with Unity, modifying the launch file for the new configuration and lastly writing the ROS part of the movement script. The first two parts of this process is explained in this subchapter while the movement code is explained in its own section as it contains elements in both ROS and Unity sides.

Three files are created for the connector to use during the data transfer. These files consist of two message files and one service file. A "msg" named folder is created for the message files and a "srv" named folder is created for the service files as these are the directories ROS searches for these files. First message file is named MoveitJoints.msg and it contains the information that is going to be transferred from Unity to ROS during the trajectory calculation. The first line is an array which contains the joint positions of the robot. As seen in the theoretical review part of the report, this information can be used to make forward kinematics calculations to get the position of all links of the robot and end effector. Next lines contain two pose data for the beginning and the end of the current operation. For pick and place operation, these are the pick pose and the place pose. Similarly for the welding application, one of the poses is where the weld starts and the other is where the weld ends. This means only linear welds are possible with the current application. These pose data are used in the inverse kinematics calculations to get the trajectory. The second message contains the trajectory output which will be transferred from ROS to Unity to visualize the movement in the virtual world. Service file contains the same information in both of the message files. This schematic for file transfer was taken from the tutorial of the ROS TCP Connector package which is the package used for the connection [24], and it was modified to work on multiple robots as the code in the tutorial only worked for Niryo One.

The launch file should be modified as this is the file that ROS runs for the operation. As expected, it should contain the ROS TCP Connector package or else it wouldn't be

able to communicate with Unity side. Therefore, a new node named server endpoint is added to the launcher and set as a default server endpoint from the connector package. This node needs two inputs which are the IP address of the Linux machine and the TCP port which is going to be used during the transfer. The default port is 10000 but if another port is preferred for security reasons, port opening procedure should be performed on the router. The IP can be found in the terminal in Linux machines easily with the right commands however, as our project operates on a virtual box, the IP address of the virtual box should be set accordingly. Virtual machines by default don't have unique IP addresses so the network settings of the virtual box should be changed to "bridge adapter" for this connection to work. Then the IP can be checked in the terminal and changed accordingly. The IP of the device can change for various reasons, thus in each launch this value should be checked and corrected if necessary. Movement script on ROS side is also run with the launcher.

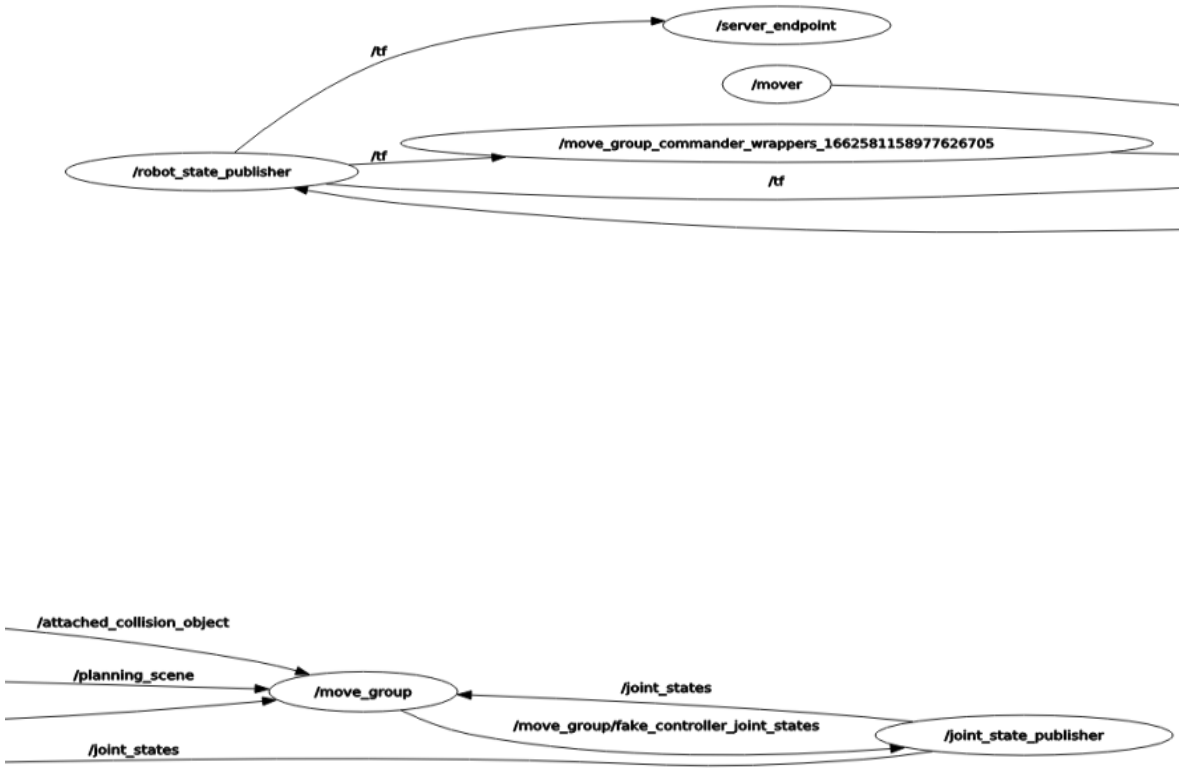


Figure 13: ROS node graph showing the new node added by the connector. It is shown in two parts to increase clarity.

Following these the CMakeLists.txt and package.xml files were edited for ROS to know that these newly added features exist in the project. The scripts are also included in this part. There are two scripts files in the “scripts” folder of the project. These files are written in Python but if desired they can also be written as C++ scripts as ROS accepts both languages.

When the modified launch file is launched, Moveit readies to make trajectory calculations on the robot. Rviz is also launched to visualize the system in ROS side. As seen in figure 14, this tool contains limited information about the environment as only the table and the robot are present. These are the only objects in the environment which are relevant to the trajectory calculation as the robot can’t realistically collide with anything else other than the human operator. Collision with the human operator is not in the scope of the project and adding it would require many additions such as meshes for the human operator and additional virtual sensors to detect the operator’s position compared to the robot.

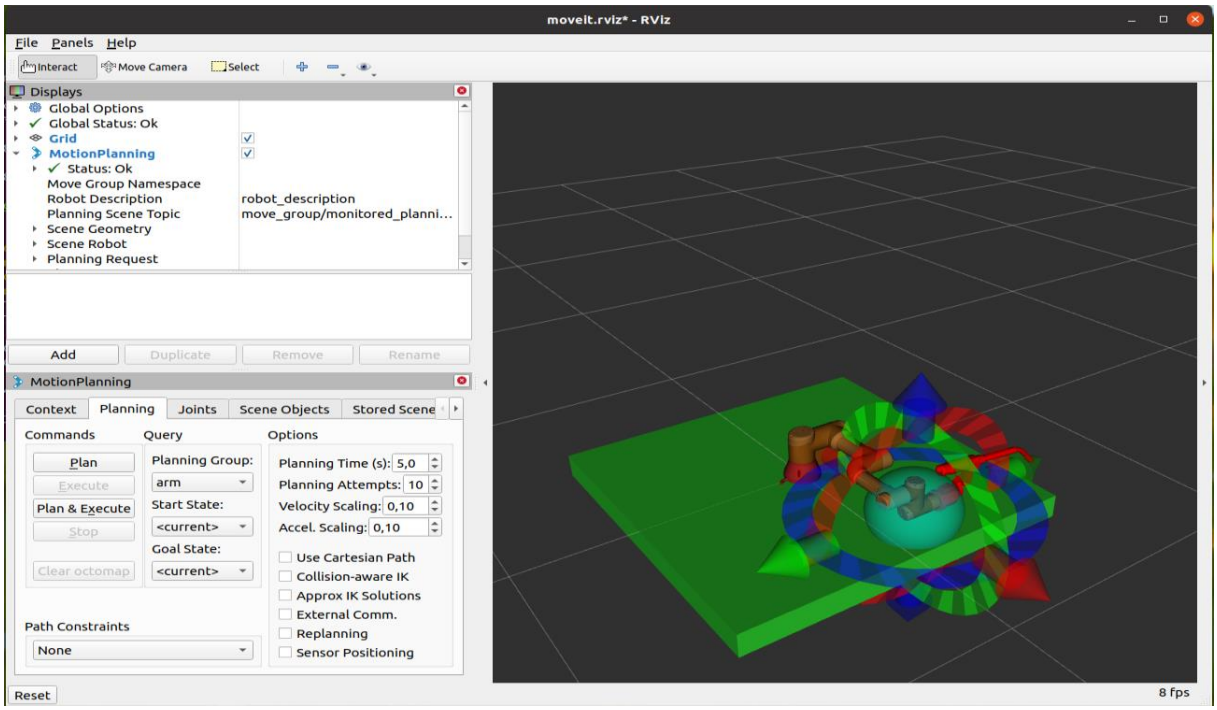


Figure 14: Rviz screen for UR3 robot with torch 1 end effector.

3.2 Unity Implementation

Unity project is started with the creation of the environment. A warehouse model was taken from a Unity asset package to create surroundings [25]. The floor is also taken from the same asset package. These are put into the environment game object in order not to clutter the left bar of Unity. Two game objects were taken from the pick and

place tutorial of Unity Robotics Hub [24]. These are the target which is a simple box and the target placement which is used as the goal during pick and place. It is a transparent area which turns green when the target game object is inside of it. Other than these, three game objects were created with Unity 3D objects. One of them is a table which hosts the robot during the operation which was created by combining a cube with a cylinder. The other two are buttons which the human operator is supposed to press to control the operation. More information about how these were created are given in the VR integration section as they were added during that phase of the project. For the welders, an additional plate is added to the scene and the target object is modified to be transparent and have no collisions for it to just work as a goal location. The robot arm still moves from "target" to "targetplacement" however, the motion is different because of the difference in the movement code which will be explained later.

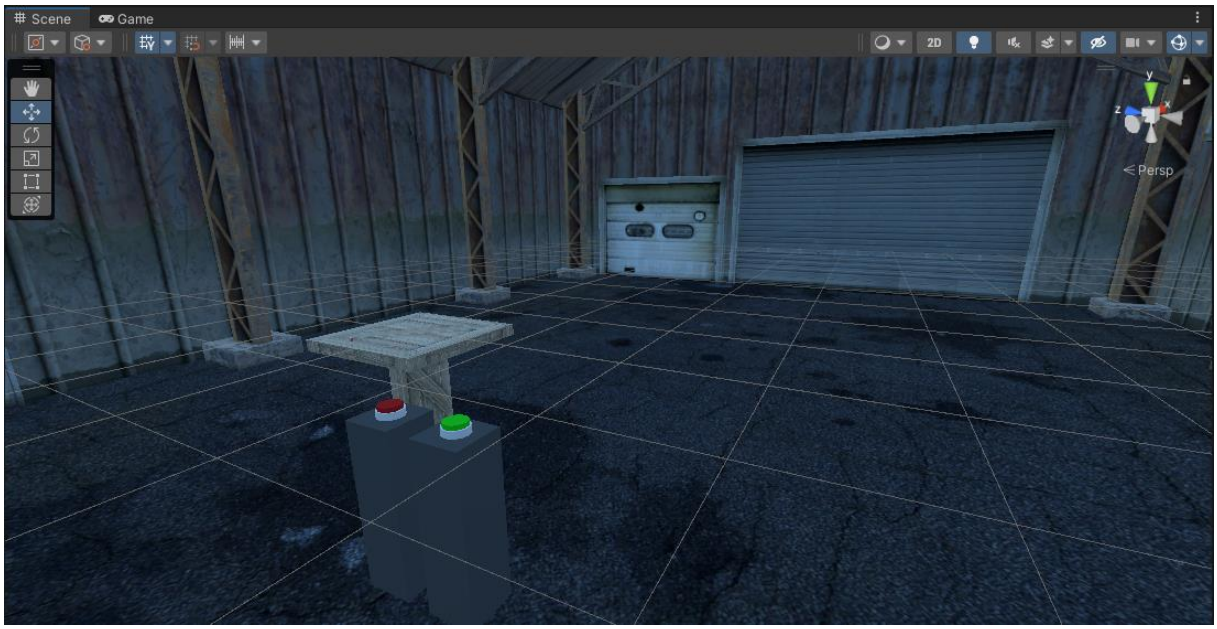


Figure 15: Unity environment before importing the robot.

To add the robot model to Unity side of the project, the URDF Importer package was used [24]. This package imports the robot geometry to Unity projects by using the URDF files. As the ROS is built in the virtual machine, all of the files in ROS workspace are inaccessible by Unity which runs in the Windows operating system. To solve this issue, all files in the ROS workspace were copied into the assets folder of Unity. As each robot requires a different scene, the previous scene was copied six times as there are six configurations. During the importation, most of the URDFs were imported without any issues. Torch1 however, was not imported correctly as most of the mesh was missing. This is caused by the naming structure of the importer. To create the mesh, the importer creates a torch1_0 file which contains the other meshes which start

from torch1_1 to torch1_n. However, if the model has too many meshes, the torch1_0 file can't hold all of them. So, the system creates other similar files and continue to name them from torch1_1. As this name was already used in the depository before, this creates an error and all of the meshes which are not included in the torch1_0 file disappear. This problem was fixed manually after the importation and the importer itself should be updated for this issue to not be repeated again. Other robot and end effector meshes didn't have this problem as they are relatively simpler geometries.

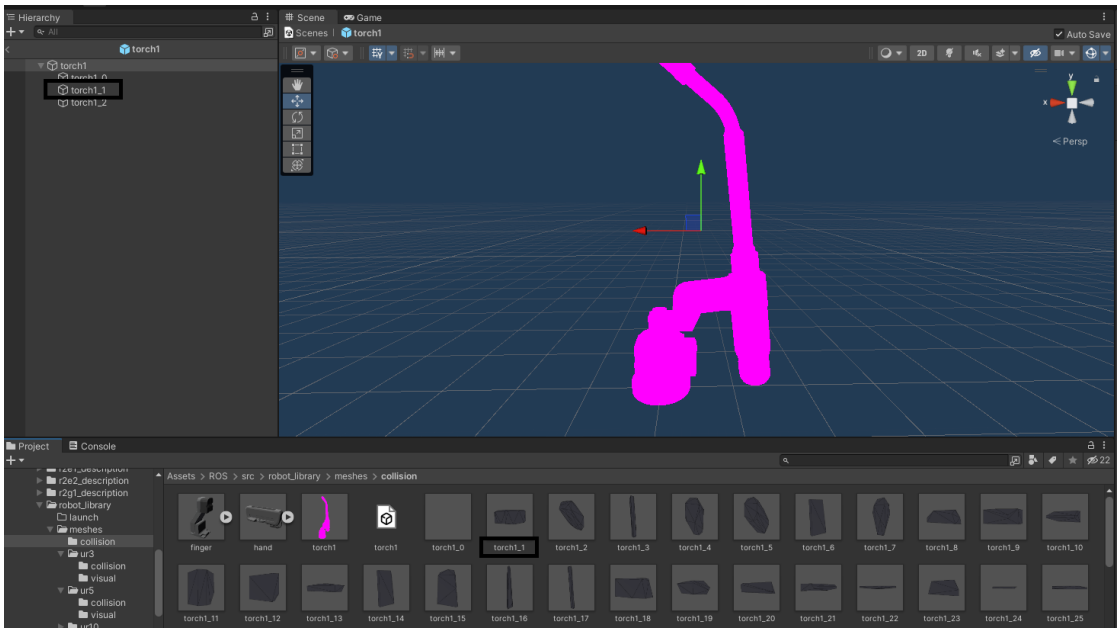


Figure 16: Torch1 prefab showing the naming structure which causes the error.

The next step of the Unity implementation is setting up the Unity side of the ROS TCP Connector. First the package is downloaded using the package manager of Unity. Then the IP address is setup from the ROS Settings bar near the Unity inspector. This IP address and port number here should match the ones in ROS for the connection to work properly. Local IP is used because both operating systems are running on the same machine and thus are connected to the same router. Then ROS messages are built. This is done from the top left menu called Robotics which is added by this package. The message and service files created in ROS and copied to Unity are selected here. RobotTrajectory.msg under "moveit_msgs" is also built as it is required by the movement code. After that, a game object to communicate with ROS side is created. This object is called Publisher and contains the Unity side of the scripts. These are two C Sharp scripts which are used during the trajectory calculation. More details about these scripts are given in part 3.3 of the report.

If the scene is started at this point, it can be seen that the robot movement is unstable, and the base is not connected to the table. These issues can be fixed by configuring the

imported robot object's parameters. The first thing done is selecting the base link and making it immovable. This will create the impression that the robot is connected to the table by restricting the movement of this link. The next part is giving parameters to the controller script of the robot game object. The parameters given were 10000 for stiffness, 100 for damping, 1000 for force limit, 30 for speed and 10 for acceleration which are taken from the pick and place tutorial of Unity Robotics Hub [24]. These are artificial numbers to simulate a realistic movement however if precision is required, they should be reevaluated.

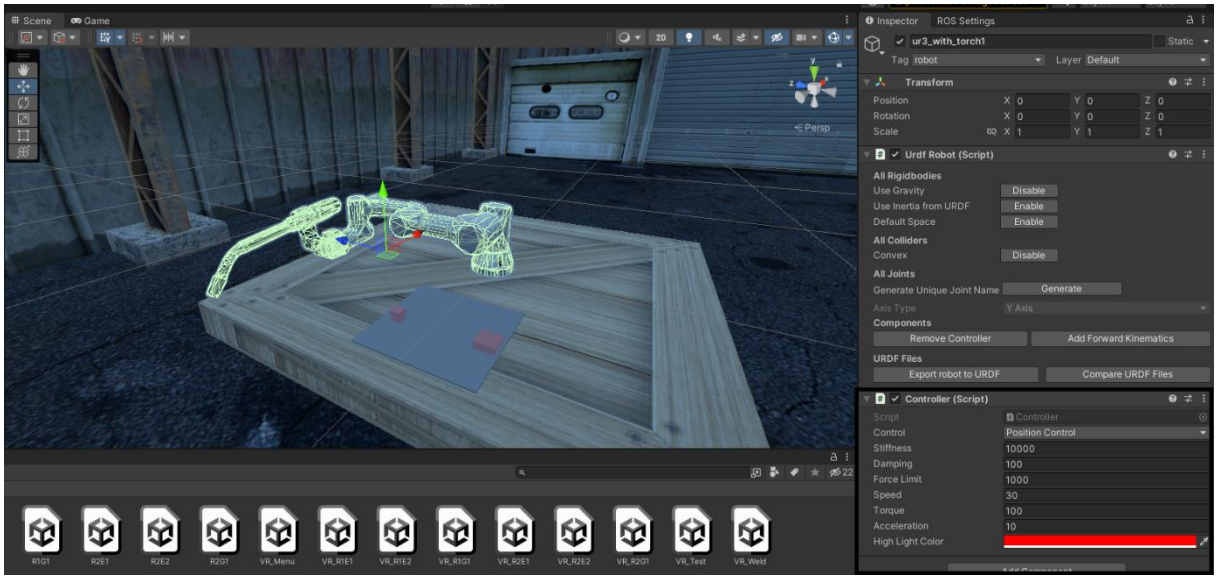


Figure 17: Controller script of a robot game object.

The physics solver of Unity is also changed as the normal solver creates instability in joint actions. The solver selected is Temporal Gauss Seidel because this solver creates more realistic movement. This solution was taken from the pick and place tutorial as well.

3.3 Configuring the Movement Script

With both Unity and ROS sides setup, the next step is to configure the movement scripts. The goal is to use the data from the Unity side to plan a trajectory on the ROS side and then send it back to Unity to visualize the motion. The transfer of data is done with the connection provided by the ROS TCP Connector package which has been built on both sides as explained before. For this transfer to work, there needs to be scripts working on both sides. ROS accepting scripts written in C++ or Python and Unity only accepting C Sharp scripts means knowledge in multiple coding languages

is necessary. The general scheme of the movement scripts on both sides accomplishing the movement action is given in figure 18.

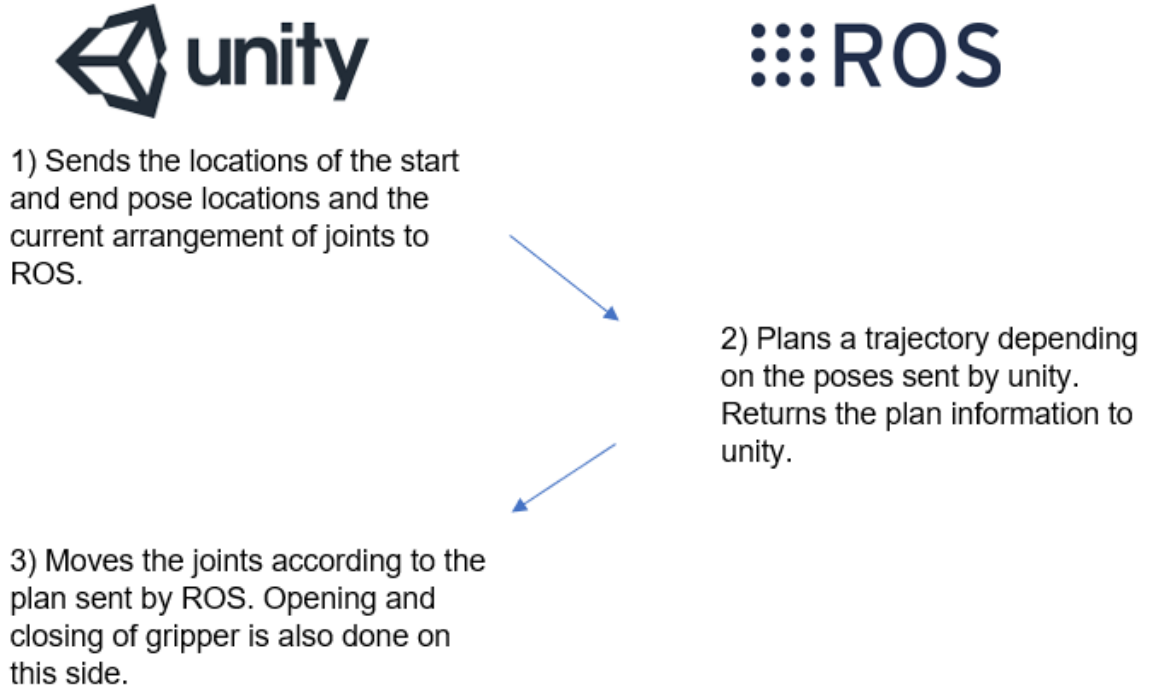


Figure 18: General scheme of the movement code.

First the movement code for the welder end effector is explained. The goal is to make a linear weld between two points. These points are the two game objects' locations, and no collision with these objects happens because the objects are only visual. To calculate this trajectory, ROS needs the information transferred with the message file which contains the poses of the goal objects and the joint positions of the robot arm. For this reason, the movement code starts on the Unity side as it contains this information.

There are two Unity scripts which are named Source Destination Publisher (SDP) and Trajectory Planner (TP). These were taken from the pick and place tutorial of Unity Robotics Hub [24] and was modified to fit the requirements of the system. For each robot-end effector combination, there is one SDP and one TP script because these scripts must be customized for the specific robot and end effector. The welder script is explained first because changing the end effector type changes the nature of the motion so the reasoning behind the movement script changes drastically. Comparatively changing the robot or picking a similar end effector are less impactful on the overall state of the script. The SDP script is a testing script which checks if the initial message

is given to the ROS side and is only still present because TP script imports values from it. TP is the main script which runs on the Unity side.

In the first part of the code, variables and inputs are defined. There are four inputs in total which are the three relevant game objects and the ROS TCP Connector service. The three game objects are the target, targetplacement and the robot which are designated in the Unity inspector after this script is added to a game object. Many constants are also defined such as the number of robot joints, wait timers and customizations for the poses. The pose customization ensures that the end effector is perpendicular to the table and is high enough to not collide with it. This is done by giving a quaternion orientation and a vector shift in the upwards direction. Of course, package imports and callings are also done in this part.

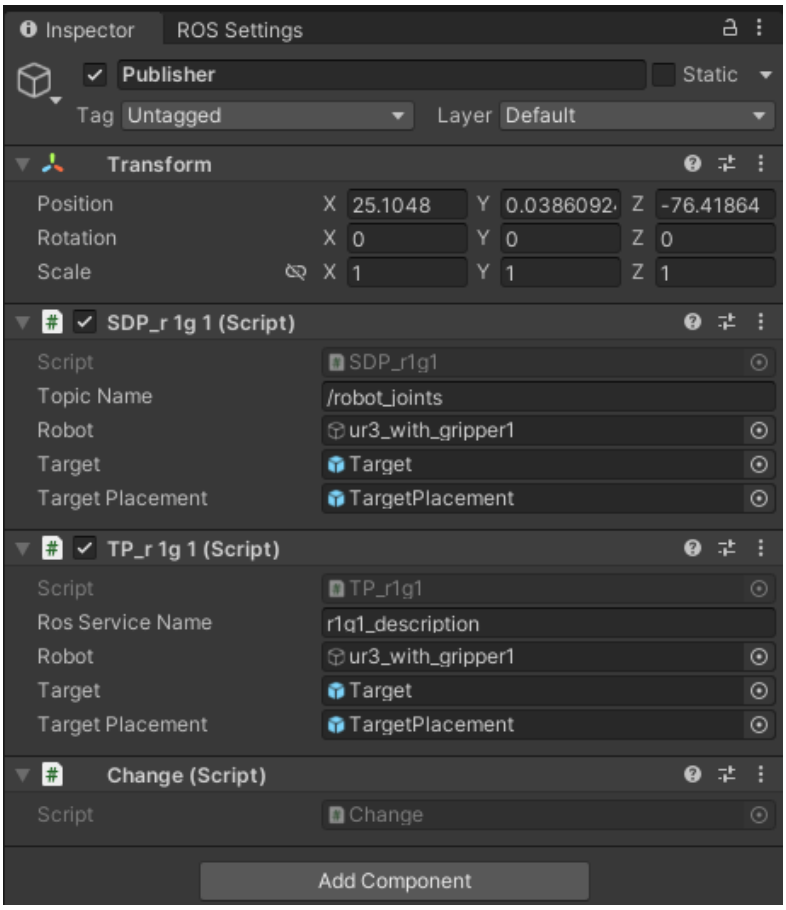


Figure 19: Game objects added to the script in the Unity inspector.

Following that the joint positions is calculated using the robot game object. While filling the articulation bodies array, the link names are taken from the SDP script. The link names in the SDP script should match the ones in the URDF for this part to work properly. Message name, class names and joint number should also be configurated

on SDP script however, customization of the remaining code is not necessary as it is not used in the final version. This articulation body array is then used to get the joint positions using the “jointPosition” command with a for loop.

After the joint positions are calculated, the next step is to calculate the goal poses. These are calculated in the publisher as they are simple calculations. The poses of the two game objects are taken. Then the offset and orientation modification are done to make the end effector placement proper. As all the data required by ROS is calculated now, it is published to the ROS side using the message files as a connection. For the connection to work the service name defined in both sides should match.

Then the calculations continue on the ROS side of the project. The main script of this side is the mover.py script which is a Python script in the scripts file of Moveit configurations. Each configuration has its own mover.py script as it is also customized for the robot-end effector combination. It starts with importing various packages including the service file created during the ROS side setup. Then, joint names are defined with their description in the URDF here as well which should also match with the current robot’s ones. This part also contains trajectory calculation code which calculates trajectory when joint states and destination pose is given. There is also a simple code which makes the script compatible with both ROS noetic and melodic packages. ROS packages are named alphabetically so melodic is the previous package compared to ROS but even between them there are a lot of changes in the nomenclature which is why this kind of corrections exist.

Following part of the code is the motion planning part. In this part, first the group name set during the Moveit Setup Assistant configuration is called. The “arm” group is called as this is the actuated part of the robot. The joint positions are taken from the data sent by Unity and the poses are used to create goals. The welding motion is a relatively simple one however, because of how inverse kinematics calculations work, the robot arm rarely makes the correct motion when only the start and the end positions are given. Because of this, two goals in the middle were calculated using 2D geometry calculations and the motion was thus divided into three parts. First the welder starts from the start pose and moves to intermediate pose one. Then the welder moves to intermediate pose two and finally the end goal. All these goals are in the same line so linear movement is preserved. This process made the movement more stable however, in most cases the inverse kinematics trajectory calculation still resulted with the welder being moved in undesirable configurations. Therefore, a more comprehensive method was needed to solve this problem.

To create a linear movement in which the welder stays perpendicular to the sheet and on similar heights, a constraint had to be introduced. This was done with the position constraint function of the Moveit package. Position constraint is a type of constraint

when if defined, the designated link can't leave the designated area. Because the end effector is not part of the planning group, the last link of the robot was used as the designated link. An invisible solid primitive box was used as the constraint area, which is small enough to limit the movement of the end effector on a line but large enough to not interfere with the inverse kinematics calculations. Because of the way sampling methods work as explained in 1.4.3 part, giving a small area or a linear line eliminates all possible configurations in the roadmap. To test the position of the box, additional commented code is added to the constraint code, to make the limit area visible in Rviz. This however interferes with the trajectory calculation as the visual of the box has a collision, and therefore makes planning in it impossible, so it should be commented out in actual uses. While this constraint limited the robot trajectory, the end effector orientation still varied in some end results. Because of this issue, another constraint was added to the system. This constraint is an orientation constraint which prevents the end effector orientation from deviating compared to the perpendicular orientation of the first pose.

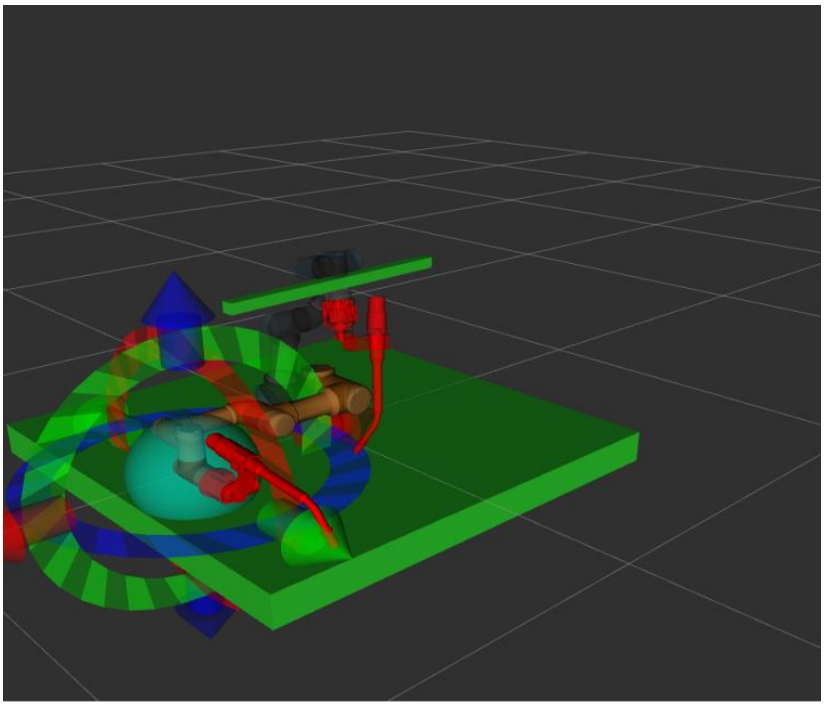


Figure 20: Visualizing the position constraint in Rviz. The rectangle above is the constraint area in this welding operation.

With the addition of both constraints, the trajectory is calculated as intended and returned as an input to Unity if the calculations are successful. This is done by collecting the trajectories and turning them into a response. Then, the two trajectories are sent from ROS to Unity. The first trajectory is used to move the robot from the

initial condition to the start goal and the second creates the linear welding between the start and end goals. The constraints are only active during the second trajectory and is cleared afterwards to allow following welding motions.

As seen in figure 20, the ROS side of the system also has the table in it. As explained before, only table is included because other elements are too far for interaction. The inclusion of the table also happens in the mover script. It is added with the shape of a box because the robot can't realistically reach the leg of the table. After returning the trajectories to Unity side, the ROS side of the movement calculation scripts rests until further motion is demanded from Unity side. This is similar to the initial state of the ROS side, as the script is run before connecting the system with Unity. It sits idle until a message from the Unity side arrives demanding the calculation of trajectories.

Trajectory Planner script continues to operate with the given response from the ROS side. The movements following the trajectories sent by the ROS side is visualized in the Unity environment. Ideally the motions on both sides should be the same, however because of different collision and physics calculations of both systems, sometimes the movement varies. Unity has a much stricter internal and external collision check system compared to ROS which causes issues in some plans. These errors are sometimes unnoticeable but sometimes they crash the Unity project by making the robot model disappear. This problem can't be solved unless these software are modified more to work with each other seamlessly. If the script runs successfully, it can be used again without restarting the Unity side. The positions of the goals can also be changed at will however, moving the robot arm manually is not recommended as this usually breaks the synchronization with the ROS side.

The grasp operation has the same steps with the welding operation in the first phase of Unity. Again, two poses and joint positions are sent to Unity in a similar manner. The difference starts with the calculation of trajectory in ROS side as the approach to the two poses are different in this case. This time four goals are created instead of two. These are the pre grasp pose, grasp pose, pick up pose and place pose. Pre grasp pose positions the end effector above the start goal with a vector displacement and perpendicular orientation similar to the welding example. This is an approach pose. Then the end effector is moved closer to the goal, in a state where it can grab the end effector if the fingers are closed. The grab doesn't happen on the ROS side as the calculations for it is simple enough for Unity to run it smoothly and accuracy in the Rviz simulation is not a goal of the project. The next goal is again above the first goal which is to make the robot arm pick up the target. Last goal is above the second goal which is the desired location of the target at the end of the process. As the grasp doesn't happen in ROS side, there is no need to code release in this part too.

Because there is no need for a linear motion in this arrangement, the constraints don't exist in this script. However, it has the model of the cube in its initial state to avoid collisions between the robot and the cube before the grasp. Similar to the welding application, the trajectories are returned to Unity side however this time four trajectories are returned because there are four different motions.

Unity side has many differences compared to the previous one because of the grasping motion. Left and right fingers of the gripper are designated as articulation bodies in addition to joints. Their IDs are found using the link destination on the Unity side. This is done by taking the last robot link name from the SDP and adding all the connections between that and the fingers of the end effector. Two functions are made to enable the motion of the gripper. One of them moves the fingers in a closing motion for a grasp and the other opens them for a release. These functions are called in their respective places during the operation when the Unity is applying the trajectories sent by the ROS side. Close gripper is called after the grasp pose is achieved and open gripper is called after the place pose is achieved.

While the grasp operation is done successfully, the motion of the grabber is somewhat unrealistic as the model has a flaw. As seen in figure 21 the gripper has a unique design. The base link is connected to four links which are designated as the right inner knuckle, right outer knuckle, left inner knuckle and left outer knuckle. Both of the right knuckles are connected to the right outer finger and both of the left knuckles are connected to the left outer finger in the real end effector. This is not the case in the Unity model as game objects can only have one parent. So, the inner knuckles are not connected to their respective outer fingers which compromises the simulation. This is a software-based problem and while the grasp operation happens as expected, it reduces the immersivity of the project.

Figure 21: Unity model of the Robotic 2f 85 gripper.

3.4 VR Integration in Unity

As the ROS side and the computational side of Unity are completed, the next step is the VR integration of the project. It should be noted that without the VR integration, the robot simulation works as expected. The VR tools doesn't directly interfere with the robot part of the simulation other than giving the virtual operator means of interacting with the environment. Its goal is to enable the simulation of a human operator in a virtual environment.

Because both Unity and VR headsets are both originally designed for gaming, they have a lot of packages to work together. The two packages considered for the project was Oculus' own package and Steam VR package. Oculus package works only for their own headsets, but it enables the Unity side of the system to work without connection to the computer. This is done with launching the project on the operating system of Oculus Quest 2. While this is surely beneficial, because the ROS side can't be run on this operating system, a computer is still required. Moreover, development takes a lot of time in this setup as the project has to be loaded on the VR headset each time something is changed during testing. Compared to Oculus, Steam VR doesn't work wirelessly however, it is not restricted to Oculus headsets, and it runs on the computer, so no additional data transfer is required. Steam VR also has a lot of assets for creating computer games which can be repurposed to be used in an industrial simulation. Its drawback is that it should be connected to the computer and as the wireless option of Oculus Quest 2, Airlink, is unreliable this should be made with a cable. Still, Steam VR was chosen as the package because the benefits outweigh the cons.

The Steam VR package was installed from the Unity package manager. First thing done was making the scene VR compatible. This was done mostly following Valem's Ultimate VR Developer Guide [26]. It is started with deleting the camera game object in Unity. The camera of Unity is designed for computer-based projects so a new camera compatible with VR is required. Therefore, the player prefab of Steam VR is added as a game object. This prefab is designed to be the player for VR games, so it has most of the functionalities required for such applications like arranging camera according to the movement of the headset. This prefab is customized further by changing the look of the controllers to human hands to increase immersion. With Oculus Quest 2's advanced haptics system, this model can even move some individual fingers.

Figure 22: Perspective of the user using the player prefab.

Next step is enabling player movement. This is done as a controller-based movement in which the left controller can be used to move, and right controller can be used to turn. While this is not strictly necessary as walking or turning while using VR creates similar effects, the area around the user might not permit such actions. Turning manually using VR also creates motion sickness in inexperienced users so it should be avoided if possible. To differentiate left controller from the right one, the Steam VR input manager is used. A new touchpad action is created to be used for the movement and it is bound to only the left controller. Right controller still has the default turning input for the touchpad. To configure the movement, a "PlayerController" script is written in Unity which moves the player in the direction they move the joystick. The movement is made perpendicular to the ground and collision checks are added with functions of Steam VR package. Gravity effect is also added by creating a downward acceleration on the player model constantly.

Other than being able to move, the user should also be able to interact with the scene. This is done by creating a "SimpleAttach" script. This script uses the hover actions defined by Steam VR package which makes coding it much simpler. Hover actions allows coding events to happen when the hands of the user start hovering something, ends hovering it and also can have a continuous function during hovering. The hover beginning and ending triggers are used to give the user the button information to grab the object. This and the highlight of the objects also show the user that the object is grabbable. An update trigger is also written to attach the object to the players hand when pressing the designated grab button and releasing the object if the button is

released. Then this script is added to the object. For it to work properly however, the interactable script should also be added to the object which is a part of the package. The “target” and “targetplacement” objects from the previous parts are made grabbable with this configuration. For welding applications, a plate is added, and it is also configured similarly for the user to be able to move it.

Figure 23: Highlight and hint during the hover event for the cube.

With the controllers used for player movement, another trigger is needed to run the movement script for the robot. This is done with a button object which is pressed physically by the user, using the virtual hands. The button physics and meshes were done following the guide of Justin P. Barnett in his video about VR buttons [27]. It is a cylindric button on a rectangle pedestal. The base of the button is made immovable and an additional colored part which is slightly smaller is added on top of it to act as the pressable part. This part is given a collider and is given a joint which acts like a spring to reset button position when it is not pressed. By also making the button interactable, a button pressable by the user is created. Two of these buttons are created and scripts are added to them using Interactable Hover Events script provided by Steam VR. One of them starts the movement script when released and the other closes the application. They are copied into all of the six scenes with robots.

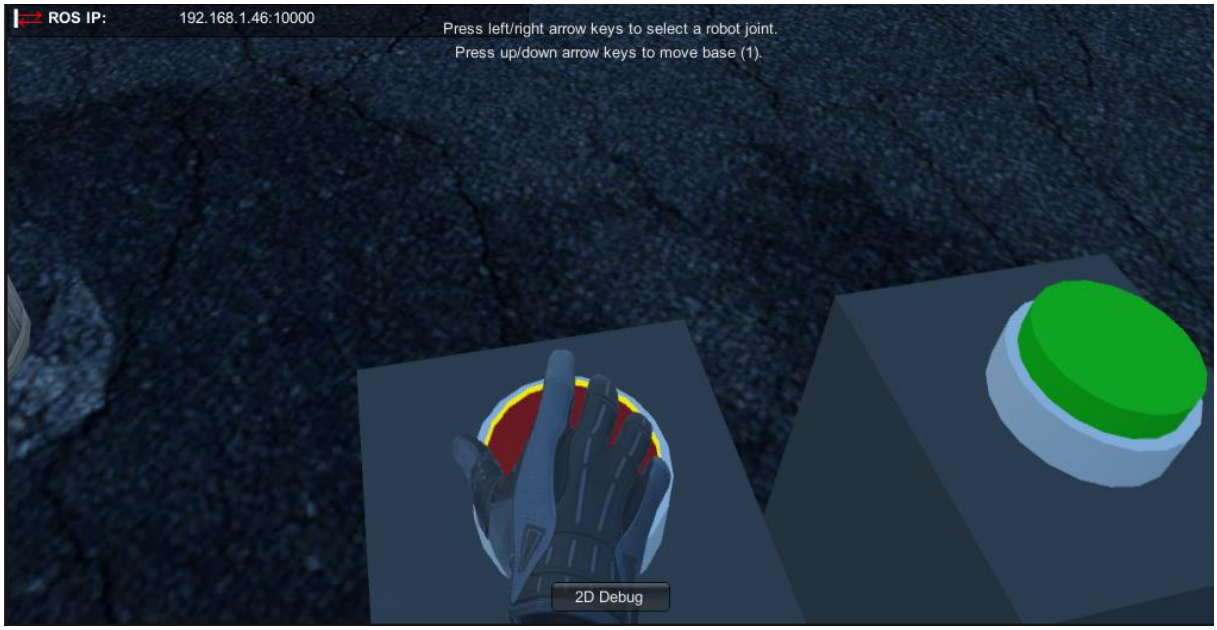


Figure 24: Virtual operator interacting with a button.

At this point, all of the scenes are working with VR integration individually. The last step of the VR integration is adding a connection between these scenes. This is done in the form of a User Interface (UI) which is a menu scene. This scene is launched at the start of the project and other scenes are accessible through it by selecting the desired robot and end effector from the menu. It is also interactable with the VR controllers which is the most complex part of this configuration. Because VR doesn't have a mouse feature, a pointer is made to enable the interaction with the menu. This pointer is present only in the VR menu scene which is the starting scene of the project. It is basically a ray that extents from the top end of the controller and used like a laser pointer to select menu elements. The interaction button is designated as an upward motion on the touchpad. As the pointer is on the right controller, the input should also be given with the right controller.

Configuration of UI is started with the creation of the canvas pointer. This whole process is done following the methodology of the series Steam VR canvas pointer of VR with Andrew channel [28]. Instead of a player prefab, a camera rig is imported from the Steam VR package. This is done because player movement and environment interaction are not necessary in the menu scene. Then a game object named PR pointer is created and put under the right controller game object under the camera rig. This game object contains a camera and a line renderer. The camera is not an active camera, it is configured as a linear beam, which will be used with a graphics ray cast at the center of it, to interact with the menu. It is also set to not render anything to avoid a

performance drop in the system by rendering it two times. The line renderer is used to create the line of the pointer and configure its color. A sphere game object is added below the pointer game object which will be used as the dot at the end of the pointer, which will show the user the pointer has collided with the menu. The collider is deleted as this would make the pointer collide with other objects in the scene.

For the pointer to work it needs two scripts. One of them is the pointer script which configures the game objects created to be used with the pointer. The other is a new input module because the basic input module of Unity is not sufficient for this application. The pointer script has two inputs which are the sphere created as the end point and the new input module game object which will be created later. It is attached to the PR pointer game object and the line renderer is defined in the awake function. Pointer first gets data from the input module and then creates a ray cast. The sphere object is put in the end of the ray cast and line renderer is put between the controller and the sphere. Length of the pointer is updated if it hits something which sets the end point of the ray cast to the position of the hit.

Figure 25: Pointer coming out of the right controller.

The new input module is started by deleting the default input module from the event systems game object. Then an input module script is written with Valve VR package imported. Valve is the company which owns the Steam platform, so this is a Steam VR package. Three inputs are taken in this new script which are the camera from the pointer game object, target source and click action. Click action configures the button used for interaction and target source is designated as right hand as this is the controller that is going to be used. The button is set as Teleport which means it is

activated by moving the joystick of the controller upwards. Pointer event data function of Steam VR is defined in the awake function. This function is used with a ray cast coming from the middle of the camera created in the pointer game object. If the ray cast hovers a game object, two functions can be used which are the press and release functions. The press function uses the ray cast with a down handler or a click handler. The object pressed is set as the current object. Release function works similarly with an up handler and a click handler. With both functions, an event set to activate during the press or release can be used with this input module. For click handler events to work, the same object must be targeted during the press and release actions. This new input module is just used in the menu scene of the project as other scenes don't have UI interaction.

After the pointer is configured, the menu configuration is started. A world space canvas is created as this is the type of canvas which is interactable with a VR pointer. This canvas has three panels which can be navigated by using the buttons. The first panel asks for the desired robot and the other panels asks for the desired end effector for the robots. The second panel is for UR3 and the third panel is for UR5. These panels contain buttons which can be interacted with the pointer. They change to light blue when hovered and dark blue when pressed. After the robot and the end effector are selected, the respective scene for the desired combination is launched.

Figure 26: Menu with the pointer coming from the controller. As seen, the buttons change color when in contact with the pointer.

Three scripts are written to configure the menu aspect of the project. First is a basic script named “Change” which handles the scene changes and contains the quit command for the project. The second script is the menu manager script which hides the inactive panels at the start of the scene. When a button is pressed, this script hides the current panel and makes the desired panel visible. This script is attached to the canvas game object. The camera of the PR pointer is also configured in this object and a graphic ray caster is defined to make the panels interactable with the pointer. The last script is a panel script which defines the children of the canvas game object as panels. This is a short script which contains the show and hide functions used in the previous script. Panel script and a graphics ray caster is attached to all the panels. With this step completed, the project has a VR interactable menu as its UI.

3.5 Configuration of New Robots and End Effectors

While the system contains six robot-end effector configurations by default, new configurations can be added with minimal effort. This can be in the form of a new end effector or a new robot. For completeness, it is recommended to match new robots with all existing end effectors and vice versa. The configuration is divided into three parts unlike the integration process, as movement script changes are divided into the ROS and Unity parts. All configurations of the process which is unique to the current combination will be listed and how they should be modified are explained below.

It is assumed that the URDF file of the robot or the end effector is present to begin the configuration. This file contains information which is not present in CAD files so this is the minimum required file type. First combined URDF’s are created in the robot library using the same process explained in part 3.1 by matching robots with end effectors. Each combined URDF should be configured separately in the following steps. Then, the Moveit Setup Assistant is run. All the steps explained in the part 3.1 of the report is done sequentially. With this configuration to work, the robot needs to be a single arm with a single end effector. This is a defining quality of the system and robots not falling in this category can’t be simulated by it. The package is generated in the catkin workspace as before with the new configuration’s name. Files added later for the ROS TCP connector can be copied from another configuration. However, the package names in CMakeLists.txt, package.xml and the launch file should be corrected. Mover.py needs to be modified substantially or lightly depending on the difference to the previous configuration. Class names, joint names and service names should be adjusted, and movement code needs to be tweaked or rewritten depending on the new application. In the welder-to-welder transition only a small tweak to the offset was feasible however comprehensive change was needed between a welder to gripper transition. The joint array in the MoveitJoints.msg file should also be adjusted

to the joint number of the new robot if robot arm is changed. Standard procedure of running catkin build command and sourcing the workspace should be done afterwards which is always done when changing the files in the ROS workspace. The new robot's ROS side then can be run by launching the designated launch file using "roslaunch" command in the terminal.

In Unity side, the VR scene closest to the new configuration can be copied to have a head start. As done in part 3.2, all ROS files are imported to Unity assets folder and the robot model is imported to Unity using the URDF importer. New messages are then built using the generate ROS messages function. SDP script is modified with the new class name, joint number and link names. TP script should be modified comprehensively similar to the mover.py script on the ROS side. Class name, joint number, SDP script name and movement code should be adjusted. When both of the scripts are done, they are added to the publisher game object and their respective inputs are dragged in the Unity inspector similarly to the process in 3.2 part of the report. The modifications on the robot model are done which are making the base link immovable and giving parameters to the controller script.

Changes on VR integrations depend on if the process simulated being changed or not. Defining a new operation requires setting up new game objects and changing interactions. In a similar operation however, only a few things are changed. The "PublishJoints" function of the new configuration is put in the on hand hover end trigger of the red button. This starts the movement code when the button is released by the virtual operator. The menu should also be edited by adding new configurations to the existing panels. Edition of a new robot means an additional panel should be added too. The configuration can be copied from the previous panels. Change script should be edited to contain new scene transitions.

These changes should be repeated for all of the new URDF configurations. In the end, the new robot or end effector can be used with all of the existing counterparts. This process is way less time consuming compared to doing everything from the beginning and provides a simulation with enough immersivity for educational purposes on HRC systems.

4. Results

As explained in the previous chapters, the system is capable of working with new robots. How it behaves when a new robot or end effector was configured was tested with the configurations present in the system. The six total configurations cover how the system needs to be reconfigured when a new robot or end effector is introduced and when a completely new operation is introduced. This is done by matching the two different sized robots with the three available end effectors. Two torches are used to find the possible adjustment required for a similar end effector while the gripper introduces a new operation. The robot sizes are different enough to warrant different movement codes while their geometry is mostly similar. In the menu, first the robot and then the end effector is selected to get to the scene of the desired configuration.

The system can simulate four welding and two pick and place operations with the current configuration. These simulations and what they imply are given in the following subchapters. To launch the system Oculus app, Steam VR, Unity project and ROS launcher should all be working simultaneously. This creates a workload on the computer so a PC with an up-to-date processor and graphics card is required. This workload can be partitioned between computers if the ROS side is run on a different machine. The process should still work as long as the machines are connected to the same local network. During each launch, the IP addresses should be corrected if the IP address of the ROS environment is not set as static. Launch files in ROS side and ROS Settings in Unity side are the location where the IP addresses should be updated.

4.1.1 Welding Simulations

The target locations and the plate are interactable by hand in these simulations. They can be grabbed by the virtual operator. The plate is affected by gravity, but the target locations are not, as they are not physical entities in the simulation. They are moved by grabbing because the controller buttons are already being used for move and turn actions by the virtual operator. In the simulation, after the red button is pressed, the welding torch is positioned above the first target. Then it is linearly moved while keeping its orientation, to the end goal during successful welds. After a weld is completed, the scene doesn't have to be reset. The user can move the plates and the targets if desired and press the red button again for an additional welding operation. It should be noted that the operation can fail because of the operation range or the angle limitations of the robot.

The first welding simulation is the combination of UR3 with torch 1. Torch 1 is the complex torch model which is bigger in size compared to the other one and UR3 is the smaller one of the robots. This is a problematic configuration as the end effector is too large for the robot to operate properly. The operation range is severely limited. This happens because the movement code tries to place the end effector perpendicularly above the targets and the robot can only hold the end effector in that position if it is very close to it. A finished weld can be seen in figure 27 which shows that the robot needs to stand upright even in very close applications.

Figure 27: Finished weld in UR3 and torch 1 configuration. The press buttons indicator at the top is not visible while using the VR headset.

Next configuration investigated is the combination of U3 with torch 2. The size of this welding torch is more compatible with the size of this robot, so operation range is higher. This results in more successful operations compared to the previous example even in the effective range. Welding operations sometimes fail with the current movement code because of the sampling algorithm of the planner and mismatch of physics engines of ROS and Unity. Even with the given constraints and angle restrictions in the poses, the robot sometimes exhibits erratic movement either caused by the planner selecting an unsuitable route or the path being unstable for Unity's physics engine. To achieve consistency in welding operations it is advised to move the joints manually in the movement code or write a more comprehensive movement code which eliminates ROS's underlying issues. A synchronization error between Unity and ROS is apparent because the robot fails to obey the constraints given in the ROS side in the Unity simulation. In the Rviz simulation however, these constraints are obeyed,

and the movement is more stable. This is caused by the collision calculation difference of the software. In some extreme cases, this situation causes the disappearance of the robot model in the Unity simulation after exhibiting unstable behavior. A similar issue was also observed in the model of the Robotiq 3f gripper that was tried during the development process. The model was static in ROS side while it exhibited perpetual motion in the Unity side.

Figure 28: Welding torch readied over the start position in UR3 and torch 2 combination.

The third configuration is the combination of the bigger robot UR5 and torch 1. Because the robot is bigger compared to the previous example, it can handle the torch better. The operation range is bigger and operation success rate is higher because of this situation. This situation can clearly be seen in figure 29 as the robot doesn't have to stand upright to make a weld even when the plate is further away from the robot compared to the previous case. Comparatively, at this range the configuration with UR3 and torch 1 fails to complete the weld because of the range limitation. This is a prime example of how this simulation can be used to select the suitable robot for an available end effector. The considered collaborative robots can be configurated into the system with the desired end effector and their performance can be tested as explained in this example.

Figure 29: Completed weld with UR5 and torch 1 configuration.

UR5 and torch 2 combination is the most versatile of the welding configurations as it contains the bigger robot with the smaller welding torch. It has the best success rate and the biggest range among the welding configurations. The only limiting factor for this configuration is the minimum range. As the robot arm is bigger compared to UR3, it can fail welding operations if the welding line passes through the minimum range area of the robot arm.

Figure 30: Snapshot taken mid-weld in UR5 and torch 2 combination.

4.1.2 Pick and Place Simulations

Pick and place simulations contain a target and a target placement which are interactable by the user. The target placement turns green when the operation is completed as seen in figure 32. The target is affected by gravity because it is a physical object in the simulation. Pick and place operations are much more stable than the welding operations in terms of synchronization of ROS and Unity sides. This situation is caused by the lack of constraints. Constraints are not needed in pick and place operations as the orientation of the end effector during the trajectory is not a concern. Without the constraints, the system can assess more pathing options and thus the suitable path found has more chances of being stable.

The first configuration contains the Robotiq 2f 85 gripper and UR3 robot. The gripper is small enough to allow the robot to move it without any problems like the ones seen in the welding torches. Operation rarely fails and the operation range is higher compared to first welding configuration. Failures are caused by the collision of the cube and end effector before pickup or the end effector dropping the cube during the motion. The first can't happen according to ROS because the cube is shown as a constraint however, because of the synchronization problems, it rarely happens. The second one happens because of the friction calculation of Unity and gripper model being not entirely accurate. Most similar simulations attach the object to the gripper during the motion however, this was not added to not compromise the realism of the simulation.

Figure 31: UR3 grabbing the cube at the start of the pick and place operation.

UR5 robot and gripper configuration behaves similarly to the UR3 one. One difference is that the maximum range and minimum range of the robot is higher. This is expected because of the kinematics theory. A flaw of this configuration is that because there are many suitable paths, sometimes strange pathing is exhibited. As explained before, the planning system doesn't always result with the shortest path. If the path chosen involves a lot of motion, the cube sometimes drops during the movement. This reduces the success rate of the system somewhat compared to the previous configuration. Still, the size of UR5 can cover almost the entire table so much more freedom is present when placing the targets.

Figure 32: UR5 robot after a successful operation.

4.1.3 Remarks

The final product is a system which can simulate combinations of various robot arms and end effectors. While much can be learned from the initial configurations, the main benefit of the system is the ease of adding new configurations in it. This can circumvent one of the most common problems mentioned about virtual reality integrated HRC simulations which is the amount of work needed to create the simulation [32]. Many of the previous research was done by only using one robot configuration while differentiating other factors such as safety mechanisms and workspace. With this setup, these kinds of tests can be done also considering different robot configurations with ease which will increase the comprehensiveness of the tests. Also, the template can be used by people with limited knowledge in robotics to test theories from other fields, like machine learning algorithms, in a

virtual simulation. Some theory used in the project can also be used to solve the problems observed in previous research. The mentioned DOF limitation of Unity can be solved by integrating ROS as the robotics planner [10]. Unity's engine is designed to simulate humans and humans have lower number of joints in each limb compared to robots, which causes this issue. Additionally, Steam VR making the simulation compatible with various headsets is also a benefit to the system because as seen in the State of the Art chapter, HTC VIVE is also a popular headset being used for these kinds of simulations [11, 31].

Problems caused by the limitations of the Unity engine was also seen in the project. In addition to the memory leakage problem mentioned in other works, there are also physics engine limitations which interfere with robotics simulations. However, usage of separate models for collision and visuals of the objects somewhat reduces the impact of this issue [10].

Some other aspects of other projects can also be integrated in this project with minimal effort. For instance, including a new workspace to test workspace optimization is as simple as modelling the environment in ROS and Unity [16, 11]. Modelling in ROS is less straightforward compared to Unity however, only the objects reachable by the robot must be modeled in ROS and model visuals are of no concern as they are not shown to the user. The system can also be used in prototyping similar to the other projects and if the operation runs smoothly, the real model can be commissioned. This however requires using the real parameters of the robot during the calculations which was not the case in default configurations. The robot parameters can be taken from the real-life model of the robot similar to the other projects.

5. Conclusion and further development

The project can simulate various different HRC systems in a virtual environment. The virtual reality aspect increases realism of the system and makes it more interactive for the user. Educational aspect of the system can also be seen even with the default configurations as many issues that affect robotic operations can be detected by comparing different setups. For instance, with the comparison of the systems with UR3 and UR5 the effects of robot dimensions are investigated. Similarly, configurations with different torches can be used to investigate the effects of end effector dimensions and complexity. The addition of the pick and place operation gives insights about how to model different operations in a virtual simulation. Because of the simplified process of adding new robots and end effectors to the system, other combinations can also be investigated with relatively little effort. However, using this amount of independent software together presents some drawbacks. As seen in the thesis project, in their current state Unity and ROS don't work with each other seamlessly. While they start from the same URDF file, there are many different parameters and calculations in the two systems which results with errors. Some examples of these are the physics solvers being different, controller parameters being given arbitrarily on both sides and collision detection being much more forgiving in ROS side. These issues can't be solved unless these platforms are modified to work together easily. ROS also sometimes gives impossible to achieve trajectories in which the whole orientation of the robot is changed in a single frame which means Moveit trajectory planner needs to be updated to completely solve these issues. Another lacking part of the simulation is the lack of interaction between the location of the virtual operator and the system. To make it more in line with the real-life applications, a safety system can be implemented with the use of virtual sensors which limits robot movement or changes the trajectory when the virtual operator is in range. These kinds of safety systems are mandatory in real life applications to ensure that the operator is not harmed during the HRC application. Additionally, the new robot introduction system can be automated more using a tool similar to the Moveit Setup Assistant which makes the process more streamlined. This however would require coding multiple plugins as the system contain files that needs to be changed in two different operating systems.

Bibliography

[1] Aaltonen, Iina, et al. “Refining Levels of Collaboration to Support the Design and Evaluation of Human-Robot Interaction in the Manufacturing Industry.” Procedia CIRP, vol. 72, 2018, pp. 93–98.

[2] Segura, Pablo, et al. “Human-Robot Collaborative Systems: Structural Components for Current Manufacturing Applications.” Advances in Industrial and Manufacturing Engineering, vol. 3, 2021, p. 100060.

[3] Aljinovic, Amanda, et al. “Integration of the Human-Robot System in the Learning Factory Assembly Process.” Procedia Manufacturing, vol. 45, 2020, pp. 158–163.

[4] Andronas, Dionisis, et al. “Design of Human Robot Collaboration Workstations – Two Automotive Case Studies.” Procedia Manufacturing, vol. 52, 2020, pp. 283–288.

[5] Antonelli, D., and D. Stadnicka. “Predicting and Preventing Mistakes in Human-Robot Collaborative Assembly.” IFAC-PapersOnLine, vol. 52, no. 13, 2019, pp. 743–748.

[6] Pires, Flavia, et al. “Digital Twin in Industry 4.0: Technologies, Applications and Challenges.” 2019 IEEE 17th International Conference on Industrial Informatics (INDIN), 2019.

[7] Pérez, Luis, et al. “Digital Twin and Virtual Reality Based Methodology for Multi-Robot Manufacturing Cell Commissioning.” Applied Sciences, vol. 10, no. 10, 2020, p. 3633.

[8] Shu, Beibei, et al. “Human-Robot Collaboration: Task Sharing Through Virtual Reality.” IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society, 2018.

[9] Ottogalli, Kiara, et al. “Flexible Framework to Model Industry 4.0 Processes for Virtual Simulators.” Applied Sciences, vol. 9, no. 23, 2019, p. 4983.

[10] De Giorgio, Andrea, et al. “Human-Machine Collaboration in Virtual Reality for Adaptive Production Engineering.” Procedia Manufacturing, vol. 11, 2017, pp. 1279–1287.

[11] Peruzzini, Margherita, et al. “Using Virtual Manufacturing to Design Human-Centric Factories: An Industrial Case.” The International Journal of Advanced Manufacturing Technology, vol. 115, no. 3, 2020, pp. 873–887.

[12] Aristidou, A., et al. “Inverse Kinematics Techniques in Computer Graphics: A Survey.” Computer Graphics Forum, vol. 37, no. 6, 2017, pp. 35–58.

[13] Manocha, D., and J.F. Canny. “Efficient Inverse Kinematics for General 6r Manipulators.” IEEE Transactions on Robotics and Automation, vol. 10, no. 5, 1994, pp. 648–657.

[14] Buss, Samuel. “Introduction to Inverse Kinematics with Jacobian Transpose, Pseudoinverse and Damped Least Squares methods.” 2009

[15] Amato, Nancy, et al. “OBPRM: An Obstacle-Based PRM for 3D Workspaces.” Robotics: The Algorithmic Perspective, 1998, pp. 165–178.

[16] Malik, Ali Ahmad, et al. “Virtual Reality in Manufacturing: Immersive and Collaborative Artificial-Reality in Design of Human-Robot Workspace.” International Journal of Computer Integrated Manufacturing, vol. 33, no. 1, 2019, pp. 22–37.

[17] Webmaster, “Forward Kinematics”, Available at https://www.meccanismocomplesso.org/en/forward-kinematics/

[18] Webmaster, “Inverse Kinematics”, Available at https://www.meccanismocomplesso.org/en/inverse-kinematics/

[19] URL https://www.universal-robots.com/

[20] Peng, Rui, “Welding Robot”, Available at https://github.com/professor1996/welding-robot

[21] Price, Andrew, “robotiq arg85 description”, Available at https://github.com/a-price/robotiq_arg85_description

[22] Prayoga, Budhi, “Abicor Binzel A300 CAT2 & Wear Parts”, Available at https://grabcad.com/library/abicor-binzel-a300-cat2-wear-parts-1

[23] Saini, Neeraj, “Moveit Setup Assistant”, Available at https://ros-planning.github.io/moveit_tutorials/doc/setup_assistant/setup_assistant_tutorial.html

[24] Younesy, Hamid, “Unity Robotics Hub”, Available at https://github.com/Unity-Technologies/Unity-Robotics-Hub

[25] Kutsenko, Dimitrii, “RPG/FPS Game Assets for PC/Mobile (Industrial Set v3.0)”, Available at https://assetstore.unity.com/packages/3d/environments/industrial/rpg-fps-game-assets-for-pc-mobile-industrial-set-v3-0-101429

[26] Valem, “STEAM VR - The Ultimate VR developer guide - PART 1”, Available at https://www.youtube.com/watch?v=5C6zr4Q5AlA&t=699s

[27] Justin P Barnett, “A Beginner's Guide to Making VR Buttons”, Available at https://www.youtube.com/watch?v=HFNzVMi5MSQ

[28] VR with Andrew, “[01] [Unity] Menu System for VR”, Available at https://www.youtube.com/watch?v=__iTtJHZg6k&t=62s

[29] Woofrey, “41013 Robotics” Available at https://www.youtube.com/user/Woolfrey

[30] Vosniakos, George-Christopher, et al. “Exploration of Two Safety Strategies in Human-Robot Collaborative Manufacturing Using Virtual Reality.” Procedia Manufacturing, vol. 38, 2019, pp. 524–531.

[31] Allmacher, Christoph, et al. “Virtual Reality for Virtual Commissioning of Automated Guided Vehicles.” 2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR), 2019.

[32] Land, Niklas, et al. “A Framework for Realizing Industrial Human-Robot Collaboration through Virtual Simulation.” Procedia CIRP, vol. 93, 2020, pp. 1194–1199.

[33] Arntz, Alexander, et al. “Machine Learning Concepts for Dual-Arm Robots within Virtual Reality.” 2021 IEEE International Conference on Artificial Intelligence and Virtual Reality (AIVR), 2021

List of Figures