Assignment of master's thesis

Title:	Simulation and Visualization of Motion Plans for a Desktop Robotic Arm with the ROS and Unity Platforms
Student:	Bc. Ján Chudý
Supervisor:	prof. RNDr. Pavel Surynek, Ph.D.
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2023/2024

Instructions

This thesis aims to propose and test the possibilities of visualization and simulation of motion plans for a desktop 3D-printed robotic arm, such as the faculty-developed RR1. We expect that the proposed solution will be realized in the context of the standard environment for robot control ROS, so the tested motion plans are easily transferable to real robots in the future. With the potential utilization of the simulation in research of multi-robot motion planning, where we need to simulate many robots in real-time, we would like to experiment with the use of the scalable graphical and physical platform Unity. The tasks for the student are as follows:

1. Familiarize yourself with the problem of motion planning for robotic arms and the simulation of motion plans.
2. Based on the findings, propose an approach of simulation and visualization of motion plans for a specific desktop robotic arm, such as RR1.
3. Implement the proposed simulation as a prototype using the Unity platform and integrate it in the context of the ROS environment.
4. Perform relevant performance experiments that compare the proposed simulation and visualization with existing systems, such as Gazebo and RViz.

[1] Steven M. LaValle: Planning Algorithms. Cambridge University Press 2006.
[2] Yoonseok Pyo, Hancheol Cho, Leon Jung, Darby Lim: ROS Robot Programming, ROBOTIS,

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 2 February 2023 in Prague.

2017.
[3] Paris Buttfield-Addison, Jon Manning, Tim Nugent: Unity Game Development Cookbook: Essentials for Every Game, O'Reilly Media 2019.
[4] Pavel Surynek: RR1 – Real Robot One, github repository, https://github.com/surynek/RR1, 2022 [accessed: January 2023].

Master's thesis

SIMULATION AND VISUALIZATION OF MOTION PLANS FOR A DESKTOP ROBOTIC ARM WITH THE ROS AND UNITY PLATFORMS

Bc. Ján Chudý

Faculty of Information Technology
Department of Applied Mathematics
Supervisor: prof. RNDr. Pavel Surynek, Ph.D.
May 4, 2023

Citation of this thesis: Chudý Ján. *Simulation and Visualization of Motion Plans for a Desktop Robotic Arm with the ROS and Unity Platforms.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

Acknowledgments **ix**

Declaration **x**

Abstract **xi**

Acronyms **xii**

Introduction **1**

Aim of the thesis 1
Expected outcome 2
Structure of the thesis 2

1 Theoretical background **3**

1.1 Robotic arms 3
1.1.1 Joints 4
1.1.2 End-effectors 4
1.1.3 2R planar manipulator 5
1.2 Simulation in 3D 5
1.2.1 Scene representation 5
1.2.2 Physics simulation 7
1.3 Motion planning 8
1.3.1 Workspace 8
1.3.2 Configuration space 8
1.3.3 Forward kinematics 11
1.3.4 Inverse kinematics 11
1.3.5 Motion planning problem 12
1.4 Digital twin 12
1.4.1 Motivation for digital twins 12
1.4.2 Core characteristics 13
1.4.3 Twinning 13
1.4.4 Life cycle 14
1.5 Physics engines 15
1.5.1 Coordinate representation 15
1.5.2 Precision-speed trade-off 16

2 Technological background **17**

2.1 Robot operating system 17
2.1.1 Motivation for robotic frameworks 18
2.1.2 ROS 2 overview 18
2.1.3 Concepts 19
2.1.4 Unified robotics description format 23
2.1.5 ros2_control 25
2.1.6 Visualization tools 27

2.1.7 MoveIt . . . 28
2.2 Gazebo . . . 29
2.2.1 Components . . . 30
2.2.2 SDF . . . 32
2.2.3 Alternatives . . . 32
2.3 Unity game engine . . . 34
2.3.1 Main modules . . . 35
2.3.2 Game loop . . . 37
2.3.3 Plans and licensing . . . 38
2.3.4 Unity for robotics . . . 39
2.3.5 Alternatives . . . 43

3 Robotic arm RR1 47
3.1 Motivation . . . 47
3.2 RR1 in detail . . . 48
3.2.1 Overview . . . 48
3.2.2 Joints and actuators . . . 49
3.2.3 End-effector . . . 50
3.2.4 Control computer RB1 . . . 50
3.3 Second prototype . . . 51

4 Prototype rationale 53
4.1 Prototype motivation . . . 53
4.2 Choice of technologies . . . 53
4.3 Expectations . . . 54
4.4 Related work . . . 54
4.4.1 Simulator comparison for agricultural robotics . . . 54
4.4.2 Gazebo, V-REP, and Unity quantitative study . . . 55
4.4.3 Gazebo and Unity physics comparison . . . 56

5 Prototype realization 57
5.1 Overview . . . 57
5.1.1 Development steps . . . 57
5.1.2 Gripper simulation . . . 58
5.2 ROS package organization . . . 58
5.2.1 rr1_bringup . . . 58
5.2.2 rr1_rviz . . . 59
5.2.3 rr1_gazebo . . . 59
5.2.4 rr1_control . . . 59
5.2.5 rr1_interfaces . . . 60
5.2.6 rr1_moveit_config . . . 60
5.2.7 rr1_experiments . . . 60
5.3 Robot description . . . 60
5.3.1 Model preparation . . . 60
5.3.2 RR1 description . . . 62
5.4 Robot control . . . 66
5.4.1 Overview . . . 67
5.4.2 Extending the URDF . . . 68
5.4.3 Controller configuration . . . 69
5.4.4 Sending control trajectories . . . 70
5.5 Gazebo simulation . . . 71
5.5.1 Starting Gazebo . . . 71
5.5.2 Spawning RR1 . . . 72

5.5.3 Multi-robot simulation 73
5.6 Unity simulation 74
5.6.1 Unity-ROS integration 74
5.6.2 Importing robot into Unity 76
5.6.3 Unity robot controllers 77
5.6.4 Other components 80
5.6.5 Multi-robot simulation 80
5.6.6 VR simulation 81

6 Experiments **83**
6.1 Overview 83
6.1.1 Performance metrics 83
6.1.2 Workstation specifications 84
6.1.3 Experiment and measurement methods 84
6.2 Scenarios 86
6.2.1 TCP endpoint latency 86
6.2.2 Static scene 87
6.2.3 Dynamic scene 88
6.2.4 Multi-robot simulation 89
6.3 Simulator comparison 90
6.3.1 Scene manipulation 93
6.3.2 Adding custom models 93
6.3.3 VR capabilities 94
6.3.4 Application issues 94
6.3.5 Community support 94
6.4 Performance evaluation 94
6.4.1 TCP endpoint latency 94
6.4.2 General performance 95
6.4.3 Multi-robot simulation 100
6.5 Summary of results 102

Conclusion **103**
Review of the thesis aims 103
Future work 104
Transfer into practice 104

A Large figures **105**

B Experiment plots **107**

Contents of enclosed SD card **127**

List of Figures

1.1 Standford Arm designed by Victor Scheinman in 1969 3
1.2 Illustration of three mechanical joints 4
1.3 Simple two-joint plannar robotic arm with gripper end effector 5
1.4 Transformation of a rigid body in a scene and a scene graph example 6
1.5 Illustration of the workspace and configuration space of the example robotic arm 10
1.6 Visualization of robots workspace and configuration space with obstacles and trajectories 10
1.7 KUKA industrial robotic manipulator with its digital twin 12
1.8 The *twinning* process between physical and virtual entities 14
1.9 Example of a digital twin life cycle relative to the life cycle of a robotic arm . . . 15

2.1 Client library API stack in ROS 2 19
2.2 ROS 2 nodes and communication patterns with their interfaces 20
2.3 Visualization of nodes and topics running in ROS 21
2.4 Simple robot example described in URDF visualized in RViz and Gazebo 23
2.5 An illustrated example of controllers and hardware interfaces managed by the controller manager 26
2.6 Multiple RQt plugins docked in RQt layout 28
2.7 RViz visualization of a robotic arm with MoveIt plugins 29
2.8 Screenshot from Gazebo 29
2.9 Illustration of Gazebo using plugins to interact with ROS 31
2.10 Screenshot from CoppeliaSim simulator 33
2.11 The default layout of the Unity Editor 34
2.12 Script lifecycle flowchart in the Unity game engine 37
2.13 MonoBehavior and coroutine example in Unity 39
2.14 Unity-ROS integration using *ROS Sharp* and *RosBridgeSuite* 41
2.15 Unity-ROS integration using the *ROS TCP Endpoint* provided by *Unity Robotics Hub* 42
2.16 Screenshot of the *Unreal Engine* user interface 44
2.17 Screenshot of the *Godot Engine* user interface 45

3.1 First functional prototype of RR1 robotic arm 48
3.2 Model of split-ring planetary gearset used in RR1 49
3.3 Control computer Real Box One (RB1) 50
3.4 Render of the RR1 “rev. 2” in orange color 51
3.5 Upper arm link with two main joints for the second prototype 52

5.1 The RR1 ROS package structure 59
5.2 RR1 meshes used for the robot description 61
5.3 Links and joints of RR1 and their transform origins visualized in RViz 62
5.4 File structure of the robot description 63
5.5 RR1 base link visualized in RViz with visual mesh, collision mesh, and inertial . 65
5.6 Overview of the RR1 control setup with ros2_control 67

5.7 RR1 Gazebo simulation landscape 72
5.8 Multi-robot simulation scenario in Gazebo 74
5.9 RR1 Unity simulation landscape 75
5.10 RR1 simulation prototype in Unity 75
5.11 Unity-ROS integration on Windows 76
5.12 Shader issue encountered when using Unity's Universal Render Pipeline instead of the Built-in Render Pipeline 77
5.13 Screenshots from multi-robot simulation in virtual reality 82

6.1 Illustration of the TCP endpoint latency experiment 86
6.2 Model of 3DBenchy boat used in experiments 87
6.3 Static scene experiment in Gazebo and Unity 88
6.4 Dynamic scene experiment in Gazebo and Unity 89
6.5 Multi-robot simulation experiment with twelve robots 90
6.6 Results of the TCP endpoint latency experiments 95
6.7 Effects of static and dynamic scene complexity on the real-time factor in Gazebo 96
6.8 Static and dynamic scenes - FPS 97
6.9 Static and dynamic scenes - CPU cores 98
6.10 Static and dynamic scenes - RAM, VRAM 99
6.11 Multi-robot simulation experiment with eighty robots 100
6.12 Multi-robot simulation - FPS 101
6.13 Multi-robot simulation - RAM 102

B.1 Results of the TCP endpoint latency experiments 107
B.2 Effects of static and dynamic scene complexity on the real-time factor in Gazebo 107
B.3 Static and dynamic scenes - CPU cores 108
B.4 Static and dynamic scenes - CPU, GPU 109
B.5 Static and dynamic scenes - RAM, VRAM, GPU temperature 110
B.6 Static and dynamic scenes - FPS 111
B.7 Dynamic scene with scaled time - CPU, GPU 112
B.8 Dynamic scene with scaled time - RAM, VRAM, GPU temperature 113
B.9 Dynamic scene with scaled time - FPS 113
B.10 Dynamic scenes Unity comparison - CPU, GPU 114
B.11 Dynamic scenes Unity comparison - RAM, VRAM, GPU temperature 115
B.12 Dynamic scenes Unity comparison - FPS 115
B.13 Multi-robot simulation - CPU, GPU 116
B.14 Multi-robot simulation - RAM, VRAM, GPU temperature 117
B.15 Multi-robot simulation - FPS 118

List of Tables

2.1 List of most recent ROS distributions 18
2.2 Supported graphics APIs based on the target platform 35

3.1 Technical specification of the RR1 robotic system 48

6.1 Hardware specifications of the test workstation 84

6.2 Comparison between Gazebo and Unity. The table follows a structure from [43] where the comparison has been made between V-REP, Gazebo, and ARGoS. The table is extended, and the Gazebo characteristics are updated based on findings from this thesis. 91
6.3 Summary of the most important results from the general performance experiments (number of objects is in the brackets) 95
6.5 Gazebo's real-time factor relative to number of simulated objects in a falling tower 96
6.6 Summary of the most important results from the multi-robot simulation experiments (number of simulated robots is in the brackets) 101

A.1 Definitions of digital twin characteristics 106

List of code listings

2.1 Example message definition 21
2.2 Example service interface definition 22
2.3 Example action interface definition 22
2.4 Example of a URDF file describing a simple robot with two revolute joints and a base with two cylindrical links 24
2.5 Fixing the base link to the world with a virtual fixed joint 32
2.6 Example of a Unity script 38
5.1 Inertial macro that returns a inertial block for specified box size 63
5.2 Custom macro for RR1 links that constructs the whole link block for a given part of RR1 64
5.3 Definition of the RR1's base_link using custom Xacro macros 65
5.4 Definition of the RR1's base_link in the generated URDF 65
5.5 Definition of RR1 shoulder_joint inside the rr1.urdf.xacro file 66
5.6 Macro used for joint control description 68
5.7 Added URDF description for ros2_control 69
5.8 Gazebo plugin that tells Gazebo to use ros2_control 70
5.9 Parameters for the controller_manager node 70
5.10 Configuration of the joint trajectory controller 71
5.11 Example of the event handler for starting the joint_state_broadcaster controller . 73
5.12 Coroutine from the JointTrajectoryController component following a received trajectory 79
5.13 Coroutine from the JointTrajectoryController component following a received trajectory 81
5.14 Instantiating multiple robots into Unity scene and setting correct ROS namespace for the ROS topics 81
6.1 Message interface with a payload size 32 Bytes 86

I would like to thank my supervisor for his guidance and insight into the robotic motion planning and development of the robotic manipulator RR1. This thesis would not be possible without his robot and passion for robotics. My biggest thanks goes to my girlfriend, who supported me throughout the work on this thesis and brought me snacks. Last but not least, thanks to my very supportive parents.

Declaration

In Praze on May 4, 2023 . . . .

Abstract

The Robot Operating System (ROS) is the most widely used framework for developing robotic applications. Also, several simulation tools exist that are used with ROS. Robot simulation is an essential part of robotics, making the development of hardware and robotic applications quicker, cheaper, and safer. Such simulation can also be used for academic demonstrations and research before fully developing the physical robot. Recently, powerful 3D development platforms like Unity have started to appeal to researchers in robotics. This work develops a ROS backend for a faculty-developed robotic manipulator Real Robot One (RR1), which allows the simulation of its digital twin prototype in Unity and Gazebo, a commonly used simulator in the ROS ecosystem. As one of the primary motivations behind the RR1 robotic arm is the research of multi-robot motion planning, the two simulation tools are compared with an emphasis on multi-robot simulation and user experience.

Keywords visualization, simulation, multi-robot motion planning, robotic manipulator, Unity, ROS

Abstrakt

Robot Operating System (ROS) je nejpoužívanější framework pro vývoj robotických aplikací. Jako nadstavby ROSu existuje také několik nástrojů pro robotické simulace. Simulace robotů je nezbytnou součástí robotiky, díky čemuž je vývoj hardwaru a robotických aplikací rychleší, levnější a bezpečnější. Simulace mohou být také použity pro akademické demonstrace a výzkum před samotným vyvinutím fyzického robota. V poslední době začaly výzkumníky v oblasti robotiky oslovovat výkonné 3D vývojové platformy, jako je Unity. Tato práce vyvíjí backend pro ROS pro fakultně vyvinutou robotickou paži Real Robot One (RR1). Navržený backend umožnuje simulaci prototypu jeho digitalního dvojčete RR1 v Unity a Gazebo, běžně používaném simulátoru v ekosystému ROS. Protože jednou z primárních motivací pro robotickou paži RR1 je výzkum plánování pohybu více robotů, jsou oba simulační nástroje porovnávány s důrazem na simulaci více robotů jak z hlediska grafického výkonu, tak vzhledem k uživatelskému komfortu.

Klíčová slova vizualizace, simulace, multirobotické plánování pohybu, robotická paže, Unity, ROS

Acronyms

AI	Artificial intelligence
API	Application programming interface
BSON	Binary JSON
CAD	Computer-aided design
DAE	Digital asset exchange
DDS	Data distribution service
DoF	Degrees of freedom
JSON	JavaScript object notation
LTS	Long-term support
NPC	Non-player character
QoS	Quality of service
RGB	Red, green, blue (color model)
ROS	Robot operating system
RPG	Role-playing game
RTF	Real-time factor
SDK	Software development kit
SLAM	Simultaneous localization and mapping
STL	Standard triangle language
TCP	Transmission control protocol
UDP	User datagram protocol
URDF	Unified rbotics description format
VR	Virtual reality
XML	Extensible markup language
Xacro	XML macros

Introduction

Digital twins and their simulation in virtual environments are essential to robotics as it shortens the development time of robotic hardware and applications and makes the process cheaper and safer. Robotic simulation also plays a significant role in academia and research, as robotic hardware is sometimes unavailable. Additionally, the field of robotics adapted robotic frameworks to develop robotic applications, making the software faster to develop, maintainable, and easy to share. One such robotic framework is the Robot Operating System, which has become a standard in robotic applications over the last few years. When it comes to simulation, the options for simulation software are much broader as there are many general or task-specific robotic simulators, many of which integrate with ROS. The most commonly used with ROS is Gazebo. However, in recent years, powerful 3D development platforms like Unity started to appeal to more researchers in the field of robotics.

This thesis aims to develop a digital twin prototype for a faculty-developed robotic manipulator with 6 degrees of freedom called RR1. The robotic arm is currently in development, so the digital twin could be used for design verification and development of control algorithms before the robot is finished. The prototype should have a ROS backend allowing its simulation and control. The thesis should also create a simulation prototype using Unity to explore the possibilities of Unity-ROS integration and its usability as a simulation tool for robotics. As one of the primary purposes of the robotic arm is the research of multi-robot motion planning, the performance and usability of Unity for multi-robot simulation should be assessed and compared with Gazebo.

Aim of the thesis

The thesis aims to create a prototype verifying the usability of Unity as a robotic simulation tool and compare its performance with Gazebo in a multi-robot simulation. This objective can be broken down into several subtasks:

- Develop a ROS backend for RR1 robotic arm and create a digital twin prototype.
- Create a simulation prototype of the digital twin in Gazebo and Unity with ROS integration.
- Compare the two tools and perform experiments assessing their performance in multi-robot simulation.

It has to be noted that the thesis does not aim to design and implement a complete robotic simulation for a specific use case but rather explore the capabilities of Unity, propose a solution for ROS integration and how the simulated robot is controlled from ROS, and assess its performance compared to Gazebo.

Expected outcome

Two hypotheses are explored and experimentally evaluated in this thesis. The first hypothesis is that the Unity platform is more performant with increasing scene complexity and the number of robotic arms simulated compared to the Gazebo. Another hypothesis is that Unity is more versatile and easier to use.

Structure of the thesis

First, the reader is provided with a theoretical background in the first chapter, like an introduction to robotics arms and digital twins, but also several concepts from motion planning. The second chapter provides an overview of the technologies utilized in the thesis, including the Robot Operating System, Gazebo, and Unity. In the third chapter, the faculty-developed robotic manipulator RR1 and the motivation behind its development are described. The fourth chapter bridges the theoretical and practical parts of the thesis by providing a rationale behind the developed prototype and the choices of technologies and reviews related work from the literature. The fifth chapter is dedicated to prototype realization, explaining various steps in the development process, from the robot description to its simulation in Unity. The sixth chapter explains the proposed performance experiments and the evaluation methods, compares the features and characteristics of the Gazebo and Unity, and evaluates the results of the experiments. Lastly, the conclusion provides an overview of the findings and work done in this thesis, evaluates the completion of goals, and proposes future steps for the developed prototype and research.

Chapter 1

Theoretical background

This chapter provides the reader with the necessary theoretical background and explains various terms referenced throughout the rest of the thesis. The first section defines several terms from robotics, specifically related to robotic arms (manipulators), as well as an example robotic arm that will be used to explain various topics from motion planning. The second section reviews simulation in 3D, including the basics of scene representation, real-time rendering, and physics simulation. Next, the third section is an overview of the core concepts of motion planning, which are often further specified for manipulators as RR1 is one. Then the concept of digital twins for the robotics domain is explored. The last section of this chapter briefly overviews physics engines, mainly comparing two coordinate representations used in them and how they affect their performance.

1.1 Robotic arms

Robotic arms are open-chain multi-body systems, also known as *serial mechanisms* [2]. Mechanically, these systems are constructed by connecting a set of rigid bodies, in robotics referred to as *links*, by *joints*. Such a system of interconnected links is called *kinematic chain* and permits relative motion between individual links. The kinematic chain connects a driven link to a fixed coordinate frame, which is usually at the base of the robotic arm. Robotic arms are also referred to as *manipulators*, as they are often used to manipulate objects in the environment. Figure 1.1 shows the *Standford Arm*, one of the first robotic manipulators designed and constructed.

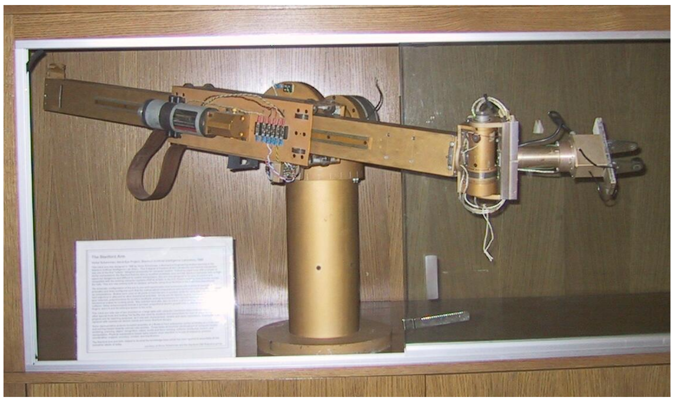


Figure 1.1 *Standford Arm* designed by Victor Scheinman in 1969 (image from [1])

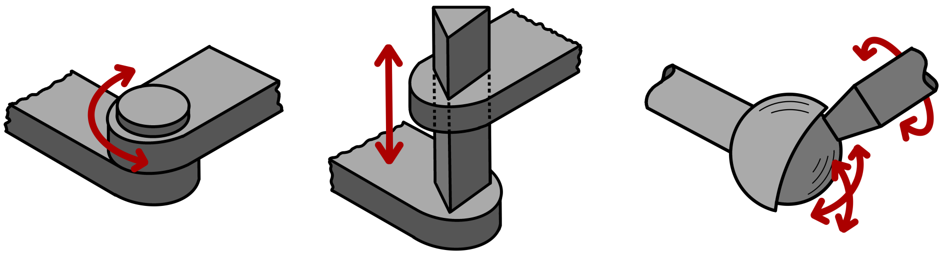


Figure 1.2 Illustration of three mechanical joints: revolute joint (left), prismatic joint (middle), and spherical joint (right)

1.1.1 Joints

Joints are usually actuated by motors, in which case they are referred to as *active joints*. *Passive joints*, on the other hand, are not actuated directly but are articulated implicitly by an actuated source of motion. Based on the complexity and type of motion, we can specify several common mechanical joints used in robotic arms. The two simplest joints with a single *degree of freedom*[1] are *revolute joints*, which rotate around a single axis, and *prismatic joints*, which allow translation along a single axis. An example of a more complex joint is a *spherical joint* with three degrees of freedom, allowing rotation around all three axes. All three mentioned joints are illustrated in Figure 1.2.

In the physical world, the motion of mechanical joins is usually constrained by certain limits. As a result, these limits affect the range of motion of the whole robotic system and have to be considered during the planning of movement, as the physical robot cannot move outside its range of motion.

1.1.2 End-effectors

Robotic manipulators are usually equipped with a device designated to manipulate or interact with the environment. This device is installed at the end of the robotic arm and is referred to as an *end-effector*, and it is the driven link at the end of the kinematic chain. Depending on the robot's application, the end-effector can be a *tool* or a *gripper*.

Tool end-effectors are usually designated for one specific task. They can be used in conditions that might be dangerous for a human or because high precision is needed for the task. The most recognizable industry applications for tool end-effectors are spot welding [3, 4], CNC machining [5, 6], or spray painting [7, 8], each utilizing a specific end-effector.

Gripper end-effectors [9] are usually used in pick-and-place applications or for other environment manipulation. There are various types of grippers for specific tasks or objects and materials they are designed to handle. One of the most common types of grippers is impactive grippers, which are used to grasp an object directly. These include jaws or claws with several fingers ranging from the most fundamental two fingers to five, like on a human hand, and beyond. These can further be categorized based on multiple factors. For example, based on the actuation type, impactive grippers are electric, pneumatic, hydraulic, and even manual. Based on the type of motion of the fingers, there are parallel, angular, or radial grippers. External impactive grippers grip an object around the exterior, but some use cases require internal grippers that grip an object by the interior. Apart from impactive grippers, other types of grippers might use suction, magnetism, or electroadheison to adhere to manipulated surfaces or pins and needles to penetrate the surface of manipulated objects physically.

[1]Degrees of freedom refer to the number of independent directions that a joint can move in. The term is further explained in Section 1.3.2.1 in regards to the whole robotic system.

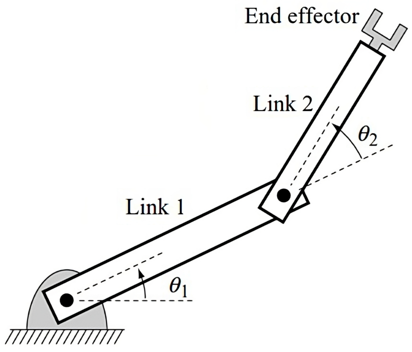


Figure 1.3 Simple two-joint plannar robotic arm with a gripper end effector (image from [10])

1.1.3 2R planar manipulator

Let us consider a two-joint planar robotic arm[2] shown in Figure 1.3. It is a simple manipulator that will be used as an example for the practical presentation of various motion planning concepts. The manipulator's base is fixed, so the only possible motion of the robotic arm is in its two revolute joints (2R), which also prevent relative motion between the links. For simplicity, let us assume that the two links can move over each other without any collision and that the range of motion of the joints is not restricted, allowing angles from the interval $[0, 2\pi) \subset \mathbb{R}$.

1.2 Simulation in 3D

Robotic simulation software is usually built as a *real-time 3D application*, where the physical simulation of the environment and the robot inside is also rendered to the user in real time. The *physics simulation* is the essential part of the simulation. Simulators usually allow execution in a headless mode, where only the simulation is performed without rendering the scene in a GUI of the application. However, the object and *scene representation* is also crucial for the physics, but also for the rendering.

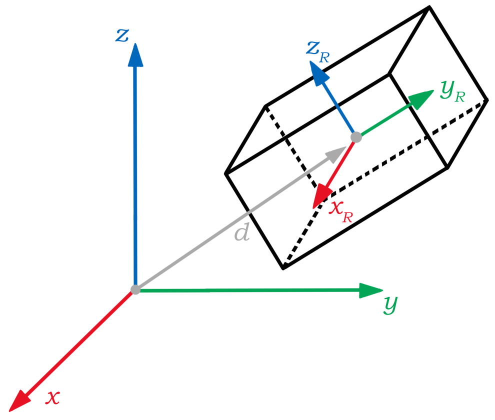
Rendering in real-time 3D applications is an exceptionally complex task on its own. It requires generating images in real-time, usually at a rate of 30 to 60 frames per second, which requires efficient algorithms and high computational capacity. The rendering process must usually be optimized for specific platforms to meet the user's output quality and efficiency standards.

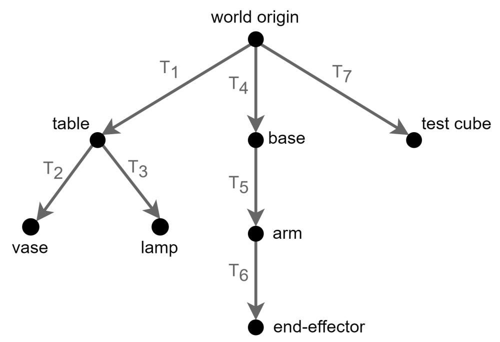
1.2.1 Scene representation

The scene in a simulation is a mathematical model of the world where individual objects are represented and manipulated. The representation of the scene must allow various processes necessary for simulation, like object manipulation, applying physical forces, collision detection, or animation of objects.

In a robotic simulation, robots and the scenes usually consist of *rigid bodies*, defined in Definition 1.1. These rigid body objects have properties like positions of the mesh vertices, normals, color for rendering, or physical properties like mass, a center of gravity, or inertial properties so they can participate in the physics simulation. The objects are positioned in the scene, and the position and orientation can be represented as a translation and rotation

[2]This example is borrowed from [10].





(a) Transformation of an object (adapted from [10]) **(b)** Scene graph example with robotic arm in it

Figure 1.4 Transformation of a rigid body in a scene and a scene graph example

transformation of the object's frame of reference relative to the origin of the scene, also called the world frame.

▶ **Definition 1.1** (3D rigid body). *Rigid body is a closed set of points from* $\mathbb{R}^3$ *such that the distance between any pair of its points remains constant in time, regardless of any motion and forces applied to it.*

1.2.1.1 Transformations

Transformations are used everywhere in the simulation, from rendering to physics and animation. There are several types of transformations, including *translation*, *rotation*, and *dilation*, but also more complicated ones like change of perspective and skewing. In a robotic simulation, where the world is realistically simulated, rotation and translation are mainly used, with rotation being a linear transformation. A transformation in 3D space can be mathematically represented as *Homogenous Transformation Matrix*

$$T = \begin{bmatrix} R & d \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} & R_{13} & d_1 \\ R_{21} & R_{22} & R_{23} & d_2 \\ R_{31} & R_{32} & R_{33} & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where R is a *rotation matrix* and d is a *displacement vector.* This transformation can be applied to a point in 3D space a as $a' = Ta$, where a and a' are the homogenous coordinates of the point. Most importantly, multiple transformations can be composed by multiplying their homogenous transformation matrices, resulting in a homogenous transformation matrix: $a' = T_3T_2T_1a = Ta$. Figure 1.4a shows a transformation of a rigid body's reference frame relative to a stationary world frame x-y-z.

1.2.1.2 Meshes

The models of rigid bodies in the scene are usually modeled as *triangle meshes* [11] that represent the surface of an object. Triangles are used because they are the simplest type of polygon that is always planar and remain triangles after most transformations. The graphics-acceleration hardware used for rendering is therefore designed around the triangle rasterization.

1.2.1.3 Scene graph

Objects in the simulated scene are logically ordered in a graph structure. The structure of choice is usually a tree graph that creates the scene hierarchy. It has several purposes, including transformation composition and application or partial spatial ordering. The spatial ordering can aid the rendering process as objects that are guaranteed to be outside the camera's view can be removed from the rendering process to increase efficiency. In the development process, the hierarchy also helps improve the management of the scene inside a scene editor, naming and grouping objects and activating or deactivating whole groups of objects.

Figure 1.4b shows an example of a simple scene with some objects and a small robotic arm. The position of the end-effector frame in the scene is a composition of transformations T_4, T_5, and T_6 relative to the world origin frame, and if the table frame is moved by changing the transformation T_1, the pose of vase and lamp frames also change.

1.2.2 Physics simulation

Physics simulation plays a vital role in robotic simulation, with the two most common components being *collision detection* and *rigid body dynamics*. However, in some simulation use cases, advanced physics models like fluid or cloth dynamics or spring-mass systems might be required.

The physics simulation goes through several steps. First, all the physical forces are applied to the objects, and then a simulation step is computed. In the simulation step, numerical integration is performed to approximate positions, velocities, and acceleration of objects and particles, collision detection occurs, and the detected collisions are resolved. After the simulation step, the objects in the scene are updated.

1.2.2.1 Collision detection

A *collision detection* system in a physics simulator aims to determine if any objects in the scene have come into contact and provide relevant information about the detected collisions. This information contains what entities are in contact, where is the contact point or if some objects are interpenetrated, in which case they are moved apart before the next frame is rendered to prevent unrealistic visual anomalies.

The collisions are detected on *collision meshes*, sometimes called collision models or simply *colliders*, which are 3D approximations of an object. Using the original mesh of an object as its collider is possible, but this would result in very inefficient collision detection. In 3D space, a collision between two spheres is the easiest to handle, and the computation is the most expensive for irregular meshes that are not convex. Typically, the collision mesh of a complex object is divided into several collision primitives like spheres and boxes, for which collision detection is computationally less difficult.

1.2.2.2 Rigid body dynamics

The *rigid body dynamics* system is mainly concerned with the *kinematics* of objects and how they move in the ambient space over time. The dynamics of their motion include the forces that affect them. In classical rigid body dynamics, the objects obey Newton's laws of motion and cannot be deformed in any way. Various constraints can be applied to the objects, such as joints.

There is a tight connection between rigid body dynamics and collision detection, as one of the most common constraints on the simulation is the non-penetration of the objects. Therefore, collision detection is needed so the rigid body dynamics system can provide realistic collision responses.

1.3 Motion planning

Motion planning plays a significant role in robotics, as one of the most fundamental tasks in robotic control is to plan a collision-free path or trajectory for a robot in an environment containing obstacles. Because the geometry of the obstacles and the robot itself can be complex, this simple task is computationally hard. This section should familiarize the reader with several concepts of motion planning, especially in the context of robotic arms. Additional concepts of motion planning can be found in [10, 12, 13], from where most of the following definitions are outlined.

1.3.1 Workspace

In most cases, robots are assumed to operate in planar ($\mathbb{R}^2$) or three-dimensional ($\mathbb{R}^3$) Euclidean ambient space, sometimes called the *workspace* ($\mathcal{W}$). Robots can physically move in this space, but the workspace will often contain obstacles preventing the robot from accessing certain parts of the workspace. Definitions 1.2, 1.3, and 1.4 formally define the space occupied by the robot's geometry, workspace obstacles, and the free workspace.

▶ **Definition 1.2** (Space occupied by the geometry of a robot in a 3D workspace)**.** *Let $\mathcal{W} = \mathbb{R}^3$ be a workspace, and let a robot be defined as a collection of m links $\mathcal{A}_1, \mathcal{A}_2, ...\mathcal{A}_m$, where each link $\mathcal{A}_i \subset \mathcal{W}$ is a rigid body. The space ocupied by the robot's geometry in the worskpace is then defined as $\mathcal{A} = \bigcup_{i=1}^{m} A_i \subset \mathcal{W}$.*

▶ **Definition 1.3** (Workspace obstacle region)**.** *Let $\mathcal{W} = \mathbb{R}^N$, where $N = 2$ or $N = 3$, be a workspace and $\mathcal{A}$ be the robot occupying and operating in it. Then, let the closed set of points $\mathcal{W}O_i \subset \mathcal{W}$ be the i-th obstacle in the workspace. The union of all obstacles in the workspace $\mathcal{O} = \bigcup_i \mathcal{W}O_i \subset \mathcal{W}$ is called the workspace obstacle region. When $\mathcal{A} \cap \mathcal{O} \neq \emptyset$, the robot is in collision with an obstacle from the workspace obstacle region.*

▶ **Definition 1.4** (Free workspace)**.** *Given a workspace $\mathcal{W}$ and its obstacle region $\mathcal{O}$, the set of points $\mathcal{W}_{free} = \mathcal{W} \setminus \mathcal{O}$ is the free workspace the robot can access.*

With robotic arms, the workspace often specifically refers to a set of points of the ambient space that are reachable by the end-effector, usually the tip of the attached tool or the position inside a gripper.

The example R2 robotic arm is planar, operating in 2-dimensional (2D) space, so its workspace can be illustrated in a 2D image. As shown in Figure 1.5a, the workspace is an annulus[3] for this particular manipulator because of the lack of constraints and Link 2 being shorter than Link 1. Note in the illustration how every point in the interior of the defined workspace can be reached in two ways: right-arm and left-arm configurations. This implies that the end-effector's position does not suffice as a configuration of the robot because it does not describe the location of all points of the manipulator.

1.3.2 Configuration space

For convenience, motion planning usually occurs in the space of all possible configurations of the robot called the *configuration space* and denoted $\mathcal{Q}$. A *configuration* q of a robotic system, which is a complete specification of the position of that system, is a point in this abstract space $\mathcal{Q}$ ($q \in \mathcal{Q}$). For robotic arms, the configuration space is sometimes referred to as the *joint space* as the configuration of a manipulator consists of parameters that correspond to the angles of the joints of the manipulator. This representation is advantageous because the robotic system

[3]Annulus is a 2D disk with a smaller disk removed from the center.

is mapped into a single point in space, regardless of its geometrical complexity. In the case of the example R2 robot, its configuration is defined by only two parameters, θ_1, and θ_2, as shown in Figure 1.3. Because of the simplicity of this robot, both the workspace and the configuration space can be visualized in 2D space, as illustrated in Figure 1.5. In Definitions 1.5, 1.6, and 1.7, the configuration obstacle region, free space and free path are formally defined.

▶ **Definition 1.5** (Configuration obstacle region)**.** *Let $\mathcal{W} = \mathbb{R}^N$, where $N = 2$ or $N = 3$, be a workspace and $\mathcal{A}(q) \subset \mathcal{W}$ be a closed set of all the points in the workspace occupied by the robot when in configuration q. The configuration obstacle region, denoted $\mathcal{Q}_{obs}$, is defined as $\mathcal{Q}_{obs} = \{q \in \mathcal{Q} | \mathcal{A}(q) \cap \mathcal{O} \neq \emptyset\}$. The individual obstacles $\mathcal{Q}O_i$ in the configuration space correspond to robot configurations that intersect a corresponding obstacle in the workspace: $\mathcal{Q}O_i = \{q | \mathcal{A}(q) \cap \mathcal{W}O_i \neq \emptyset\}$.*

▶ **Definition 1.6** (Free configuration space)**.** *Given a configuration space $\mathcal{Q}$ and its configuration obstacle region $\mathcal{Q}_obs$, the free configuration space (sometimes only free space) is defined as $\mathcal{Q}_{free} = \mathcal{Q} \setminus \mathcal{Q}_{obs}$*

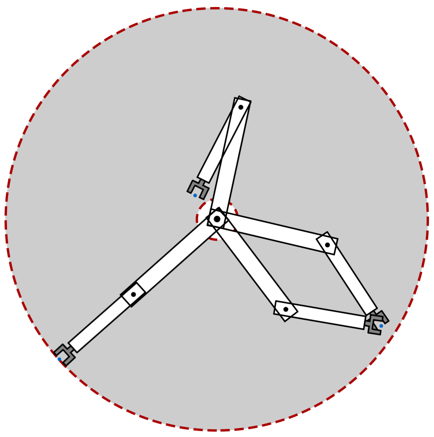
▶ **Definition 1.7** (Free path)**.** *A free path in a free configuration space $\mathcal{Q}_{free}$ is a continuous mapping $\tau : [0, 1] \rightarrow \mathcal{Q}_{free}$ and does not allow contact between the robot and any obstacle. A semi-free path allows contact between the robot and an obstacle boundary and is defined as a continuous mapping $\tau : [0, 1] \rightarrow cl(\mathcal{Q}_{free})$, where $cl(\mathcal{Q}_{free})$ denotes a closure of $\mathcal{Q}_{free}$.*

Even with a slightly more complex robotic system, the configuration space obstacles are much more challenging to compute. Therefore, grid-based representations of the configuration space are sometimes used. A test computation is performed for each grid cell to see if the robotic system collides with any obstacle in the workspace. When the grid cells are represented as pixels, color-coded per obstacle, we can visualize the configuration space, obstacles, and planned paths. Figure 1.6 shows such a visualization for the R2 robotic arm. On the left side of the figure, the robot's workspace with obstacles and a path of the robot's end-effector through the workspace are visualized. The start, end, and several in-between configurations are also depicted. The same workspace is illustrated in the middle of the figure with two different configurations. On the left side, the generated visualization of the configuration space with the obstacles, the robot's trajectory, and the configurations are shown in corresponding colors. Note how no free path exists between the two configurations from the middle of the figure, even though it is not apparent from the image of the workspace.

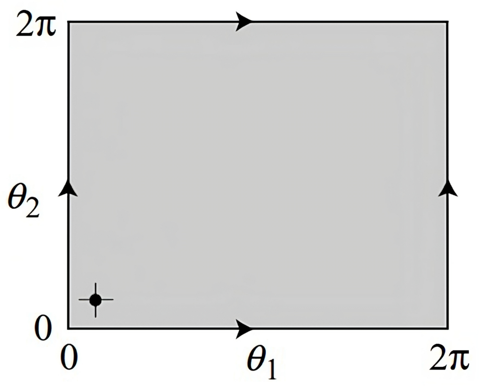
1.3.2.1 Degrees of freedom

The *degrees of freedom* (DoF) of a robotic system is the minimum number of parameters needed to specify its configuration. Thus it also corresponds to the dimension of the configuration space. Analogically to the configuration of a robotic arm, the DoF of a robotic manipulator usually equates to the sum of the degrees of freedom of individual joints in the kinematic chain. The 2R robot has two DoF as it has two revolute joints with one DoF each. Another example is the human arm, which has seven DoF. The shoulder joint is spherical with three DoF, the elbow is a simple revolute joint with one DoF, and the wrist can be substituted with a three-DoF spherical joint.

Every rigid body in the 3D ambient space has six DoF. Three degrees of freedom describe its position and allow translation along all three axes, and the other three allow rotation around these axes. Industrial robotic arms usually have six or seven DoF. A minimum of six DoF is needed to achieve any position and orientation of the end effector inside the robot's workspace. Adding more DoF creates redundancy in the system as a given end-effector pose can then be achieved with multiple configurations. The redundancy is useful when there are obstacles in the workspace and some configurations cannot be reached, but increasing the DoF of the robotic system makes motion planning increasingly challenging.



(a) Workspace of the R2 planar manipulator with several illustrated configurations



(b) Configuration space of the R2 planar manipulator with the configuration from Figure 1.3 indicated with a marker

Figure 1.5 Illustration of the workspace and configuration space of the example robotic arm (images from [10])

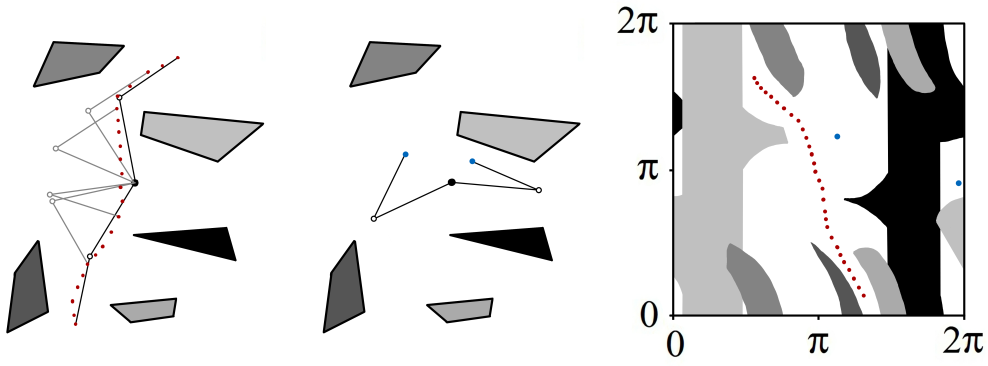


Figure 1.6 Visualization of robots workspace with end effector trajectory (left), two configurations with no existing free path between them (middle), and visualization of configuration space with obstacles and the trajectories (right) (images adapted from [10])

1.3.3 Forward kinematics

Forward kinematics is one of the kinematics tasks that bridge the configuration space and workspace of the robot. Specifically, it is a technique for determining the position and orientation of the end effector's frame in the workspace relative to the robot's base link based on its configuration. In other words, it converts a point in the configuration space to a point and its orientation in the workspace. Formally, a *forward kinematics map* $\phi : \mathcal{Q} \to \mathcal{W}$ is defined.

1.3.3.1 Solving forward kinematics

A direct way to compute the forward kinematics of a kinematic chain is by composition of transformations by multiplying homogenous transformation matrices of adjacent links. First, the coordinate frames of individual links are defined and transformation matrices are derived related to these coordinate frames. Then the transformation metrices can be multiplied to get the homogenous transformation matrix of the end-effector frame related to the base link coordinate frame. The rotation matrix and displacement vector can then be extracted from the result transformation matrix. Mathematically,

$$T_n^0 = T_1^0 T_2^1 ... T_n^{n-1} = \begin{bmatrix} R_n^0 & d_n^0 \\ 0 & 1 \end{bmatrix},$$

where T_n^{n-1} is the homogenous transformation matrix of the coordination frame of the n-th link relative to the $(n-1)$-th link, and the base kink being the 0th and the end-effector the n-th link of the kinematic chain.

The individual transformations for each pair of the subsequent frames can be found directly by deriving the rotation matrix and displacement vector of the transformation, which are parametrized by the angle of the joint. In practice, a more straightforward industry-standard method is used for finding these individual homogenous transformation matrices, called the *Denavit-Hartenberg Method* [12]. The method is faster and produces the same transformation matrix but obscures the meaning of the rotation matrix and the displacement vector.

1.3.4 Inverse kinematics

Inverse kinematics is the opposite task to forward kinematics. With a known position or orientation of the end-effector in the workspace, inverse kinematics is a problem of finding the configuration of joints to get the end-effector into that pose. This is a more difficult problem than forward kinematics because it is non-linear, and there is a possibility of multiple solutions or singularities, as it might be possible to achieve various positions in the workspace with multiple configurations. Formally, a *inverse kinematics map* $\phi^{-1} : \mathcal{W} \to \mathcal{Q}$ is defined.

1.3.4.1 Solving inverse kinematics

Various techniques and algorithms are available to solve the inverse kinematics, including analytical methods for simple kinematic chains, but primarily numerical methods and optimization algorithms. Machine learning techniques can also be used to teach models how to map end-effector pose to the configuration space. Generally, after the end-effector pose is defined, a set of non-linear equations is derived that describes the relationship between the configuration of the robotic arm and the end-effector position by solving the forward kinematics. Then the set of equations is solved. The solutions must be checked if the configurations are feasible, do not result in a collision, or violate any other constraints.

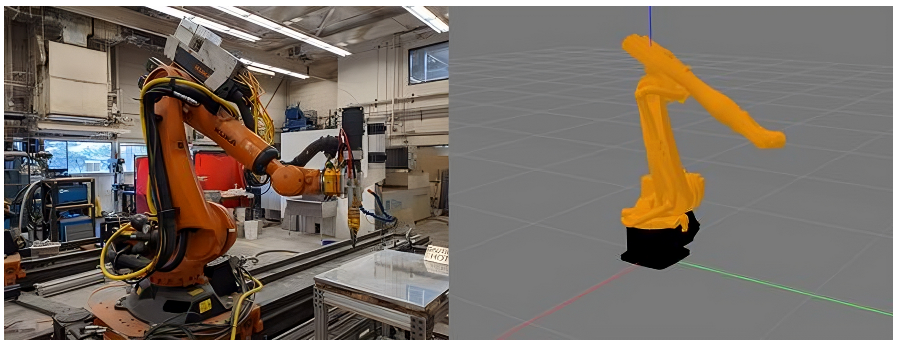


Figure 1.7 KUKA industrial robotic manipulator with its digital twin simulated in Gazebo (image from [17])

1.3.5 Motion planning problem

Depending on the ambient space, constraints, or application, there can be several variations of the motion planning problem. The most basic is the *geometric path planning problem* defined in Definition 1.8.

▶ **Definition 1.8** (Motion planning problem [13]). *Let $\mathcal{W} = \mathbb{R}^N$, where $N = 2$ or $N = 3$, be a workspace and $\mathcal{O} \subset \mathcal{W}$ an obstacle region inside that workspace. Given a robot in $\mathcal{W}$, defined as a collection of m links $\mathcal{A}_1, \mathcal{A}_2, ...\mathcal{A}_m$, the configuration space $\mathcal{Q}$ with both $\mathcal{Q}_{obs}$ and $\mathcal{Q}_{free}$ can be defined. For an initial configuration $q_I \in \mathcal{Q}_{free}$ and a goal configuration $q_G \in \mathcal{Q}_{free}$, find a (continuous) free path, $\tau : [0, 1] \rightarrow Q_{free}$, such that $\tau(0) = q_I$ and $\tau(1) = q_G$.*

The pair (q_I, q_G) is often called a *query*, and the solution to this problem is the *motion plan*, which is the collision-free path, τ, sometimes also referred to as the *trajectory*. By following the motion plan, the robot will move in the workspace from its starting position $\mathcal{A}(q_I)$ to its goal position $\mathcal{A}(q_G)$. Depending on the abstraction of the problem, the motion plan can be as simple as a sequence of actions the robot needs to perform or positions in a gird the robot needs to visit, to continuous curves in space.

1.4 Digital twin

Over recent years, the utilization and research of the concept of a *digital twin* [14] have been increasing considerably [15, 16], especially in the manufacturing domain. To some extent, it is an ambiguous and very general concept that is still evolving as products in various domains employ this concept in different forms, use varying technologies and terminology, and utilize the digital twin in various stages of the product's life cycle. In [15], authors have proposed unification of the terminology and definitions of some characteristics of the digital twin concept, which are summarized in Table A.1 directly from that publication. This section provides an overview of some relevant characteristics with an emphasis on robotics and robotic manipulators.

1.4.1 Motivation for digital twins

In the modern period of robotics, it is common to deploy digital replicas of the physical hardware in various simulations. Although having such a model of the robotic system could be viewed as a digital twin, it is missing several characteristics to be considered a digital twin, as will be defined later. However, the modeling and simulation of robots is a significant part of digital twins in robotics, and the motivation behind it remains unchanged. Here it is discussed from two perspectives: historical and modern.

One of the historical problems with robotics was that the design and development of both the robotic hardware and software were slow and expensive processes. This is because the procedure usually requires multiple iterations of testing and modifications of the physical system. On the one hand, building a robot is expensive, but testing it can also be expensive and time-consuming. Even for a robot with a simple task, the experiments might require hiring specialized personnel and building a custom testing site. Moreover, the tests could potentially be dangerous for the robot and the personnel. With increasingly more complex robotic systems with complicated or precise tasks and many sensors and algorithms running simultaneously, this process often requires more iterations and more sophisticated tests than before. Modeling and simulation of the robotic system solve most of these issues as it is a safer way to test the robot, the testing conditions are consistent, and it is possible to iterate over modifications faster. Additionally, it is possible to simulate various test environments and edge cases that might be very difficult to reproduce in the real world. After the iterative development process in the simulated environment, the robotic system can finally be tested and validated in the real world before its production phase.

Furthermore, due to the increasing complexity of the tasks the robots perform and a considerable number of parameters describing the state of the system and the workspace, the use of model-based predictive control based on machine learning or deep learning models is increasingly demanding. Training such models often require a large quantity of training data, which can be unattainable to obtain through physical execution only. Simulation can be used to effectively generate the training data, allowing deployment in various simulated environments or miscellaneous randomization of input parameters which increases the robustness of these models.

1.4.2 Core characteristics

Digital twins have some specific features and characteristics. There has to be a *physical entity* that operates in a *physical environment* and its virtual counterpart that exists in a *virtual environment.* The environments also correspond to each other as, in most cases, the virtual environment is a digital copy of the physical one. The entity and its corresponding environment, whether physical or virtual, is described by a set of *parameters* that can be measured through *metrology* in either of the worlds. The measured values define the state of that entity, and through the process of realization, this state can be changed in either of the worlds.

In the context of robotic arms, the physical entity is naturally the constructed physical manipulator itself. The physical environment refers to the space where the robotic arm is situated, including its workspace. Any relevant parameters of the environment affecting the physical entity should be captured. These can include anything from the geometric shape and size of the robotic arm, physical parameters of the joints, positions of objects in the workspace, the purpose and control of the arm, to external factors like ambient temperature and wind forces. These parameters can be continually fed into the virtual environment in the digital domain to get an accurate mirror of the physical environment. Then it is possible to perform more accurate simulations to optimize the manipulator's operation or make decisions. The number of parameters chosen to capture, their accuracy, and even the chosen level of abstraction for the virtual environment corresponds to the fidelity of the virtual counterpart.

1.4.3 Twinning

To consider the physical and virtual entities as twins, a cycle between the physical and virtual states must be achieved to synchronize the two. This cycle is called *twinning* or mirroring, and the rate at which the states synchronize is called the *twinning rate.* As is shown in Figure 1.8, this process requires two connections between the entities and environments.

The *physical-to-virtual* connection enables the state of the physical world to be transferred to the virtual environment. It consists of metrology in the physical environment and a realization in the virtual environment. First, the state of the physical entity is measured, whether by encoders

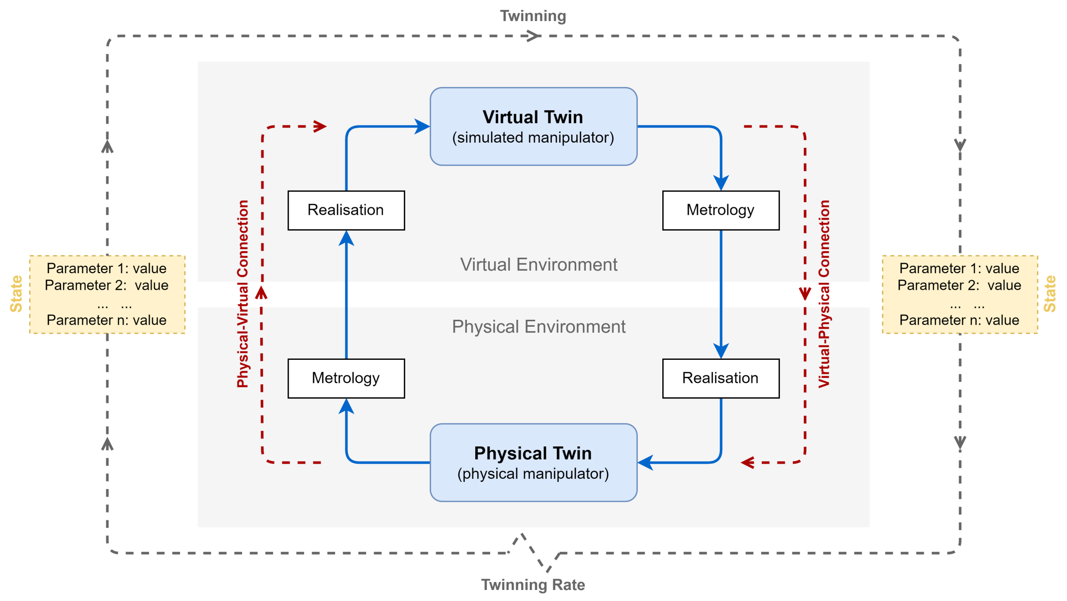


Figure 1.8 The *twinning* process between physical and virtual entities (adapted from [15])

in the joints or other sensors and cameras, then the state is realized by the virtual entity. So if the state changes in the physical world, it is mirrored in the virtual environment through this connection. This continuous mirroring is one of the main differences between a digital twin and the traditional modeling and simulation of robotic hardware.

Although sometimes omitted in the description of digital twins, the *virtual-to-physical* connection is the data flow from the virtual environment to the physical one. The physical entity must therefore contain functionality to realize state changes measured in the virtual entity. Such realization with robotic manipulators can be, for example, direct control of the joint actuators. This connection closes the loop with the physical-to-virtual connection, so the hypotheses that might be generated in the virtual environment can be directly tested in the physical environment. This enables a continuous optimization cycle of predicting physical states in the virtual environment and allows the digital twin to learn from its historical performance.

1.4.4 Life cycle

The concept of the life cycle of the digital twin in relation to the physical product's life cycle needs to be more unified throughout the literature. In various cases, a digital twin exists only during a specific phase of the product's life cycle, or there can even be numerous digital twin instances for various phases. In general, the digital twin starts its life cycle as a *digital twin prototype* in the concept and design phases of the physical product. During the product's life cycle, the digital twin is evolving, eventually surpassing the physical entity, as it can have potential value even after the product's retirement.

Different processes can run in the physical and virtual environments during the life cycle of the product and the digital twin object. The *physical processes* are activities performed in the physical environment by the physical entity. Examples of physical processes for a robotic manipulator include pick-and-place tasks, welding, or general control of the robotic arm. Because of these processes, changes in the state of the physical twin occur. *Virtual processes*, on the other hand, refer to activities performed in the virtual environment by the virtual twin. These processes include modeling and simulation, control optimization, design verification, edge-case

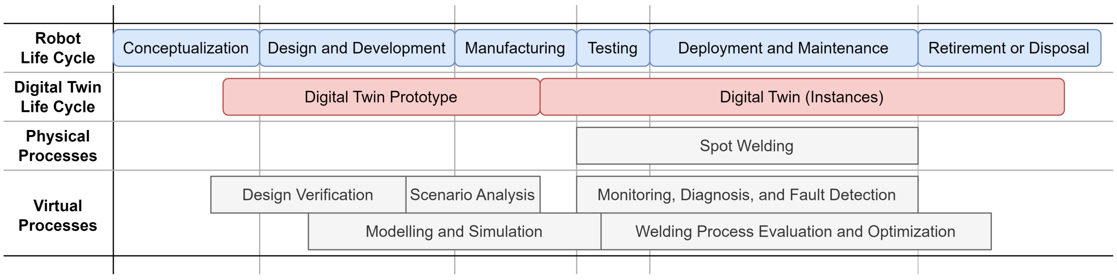


Figure 1.9 Example of a digital twin life cycle relative to the life cycle of a welding robotic arm with various processes and applications of the digital twin[4]

scenario analyses, and more. These processes can be used to analyze the changes in the state of the virtual entity or even realize them in the physical one. In Figure 1.9, there is an artificially constructed example of a life cycle of a welding robotic arm and its digital twin and various processes that might be applicable.

1.5 Physics engines

Physical forces and their effects on the environment and objects often need to be simulated in various domains and applications like robotic simulation, video games, film, and animation. Specialized software, called the *physics engine*, is used to computationally approximate these physical systems, including rigid body dynamics, soft body dynamics, fluid dynamics, collision detection, and more. Physics engines are usually used as middleware in other applications, such as game engines, 3D modeling applications, or scientific simulators, where their purpose can be fundamentally different.

In the context of this thesis, *real-time physics engines* are relevant because of their use in robotic simulations and game engines. There are numerous available physics engines, and because of their role as middleware in the software they are used in, they can frequently be interchangeable. However, as many of these engines were developed in different domains, they adapt various algorithms with their strengths and weaknesses for specific tasks or only support specific features. For example, some engines might specialize in fluid dynamics, and others in rigid body dynamics. Because of these differences, and their complexity and configurability, it is challenging to compare them or evaluate their general performance, even though it has been attempted in several studies [18, 19, 20]. Nevertheless, certain characteristics of physics engines are known to affect their performance in specific use cases substantially.

1.5.1 Coordinate representation

One such characteristic, particularly pertinent for the simulation of robotic arms, is *coordinate representation*. When representing a position of a physical object in space, *maximal coordinates* or *generalized coordinates* can be used.

1.5.1.1 Maximal coordinates

With *maximal coordinates*, sometimes called Cartesian coordinates, objects in 3D space are represented with all six DoF (see Section 1.3.2.1). This is an efficient representation for objects

[4]Note that the proportion of depicted lengths of the robot life cycle stages in the figure does not correspond to the proportion of actual durations of these stages. The figure mainly illustrates how various processes and the life cycle of the digital twin can overlap these stages.

that can freely move in the environment. However, when representing joints, their movement has to be appropriately limited by constraints. So for a simple revolute joint frequently used in robotic arms, additional constraints have to be added to restrict the original six DoF to one. The physics engine must try to satisfy these constraints when solving the calculations. The more constraints there are in the simulated system, the more constraints can be potentially violated, leading to inaccuracies that are often very noticeable when the simulation is displayed. This is especially problematic when simulating long kinematic chains. Physics engines like *Nvidia PhysX* [21], *ODE* [22], and *Bullet* [23] primarily use maximal coordinates.

1.5.1.2 Generalized coordinates

On the other hand, *generalized coordinates*, also called joint coordinates, can represent joints directly with the correct amount of DoF without additional constraints. So the same revolute joint will be described by one DoF instead of six DoF with constraints limiting the movement of the joint as was the case in the maximal coordinate representation. This representation is challenging to implement but significantly improves simulating kinematic chains. With considerably fewer constraints, the computations are more efficient and accurate. The generalized coordinate representation is used in engines like *DART* [24] and *Simbody* [25]. However, support for generalized coordinates has also been added to Nvidia PhysX and Bullet physics engines.

1.5.2 Precision-speed trade-off

Scientific simulators might require very high precision of the approximation, but the computation can potentially run for days. Game engines, conversely, need the simulation to be performed in real-time but can afford lower accuracy. However, high physical accuracy and fast simulation cannot be achieved simultaneously as this trade-off is inherent to the slow convergence of optimization algorithms used for these computations. With this trade-off in mind, physics engines can often be configured for an acceptable level of precision the specific application requires. Nonetheless, the simulation will always be a simplified approximation of the real world.

Chapter 2

Technological background

This chapter provides the reader with the necessary background regarding the technologies used in the thesis. Namely, the first section covers the *Robot Operating System (ROS)* and some of its essential components and tools, including visualization tools like *RQt* and *RViz*. The following section continues with a description of the *Gazebo* simulator and some background in robotics simulation. The third section overviews the *Unity* game engine, its key modules, and its role in robotics.

2.1 Robot operating system

Robot operating system (ROS), although in literature often labeled as middleware, is a free and open-source framework for developing robotics applications[1]. It is maintained by Open Robotics, formerly Open Source Robotics Foundation (OSRF), and its development started in 2007. Since then, it evolved dramatically, its community grew, and it became a standard in robotics. It is not an operating system in the traditional sense. It provides an abstract and structured communication layer of a heterogenous compute cluster that runs above the host operating system. This layer is used for explicit communication between components via message passing. ROS also aggregates an extensive set of libraries made by community contributors that can be used when building various types of robotic systems, with additional utilities for monitoring, logging, debugging, communication introspection, and more. All of this enables even small teams to build complex robotic systems.

However, ROS started to fall short when robotic applications started to turn into products because its initial design had very few production-grade features in mind. Reliability, system uptime, support for large-scale embedded systems, and security should have been prioritized. ROS struggled to consistently deliver data over lossy links like wifi or satellite because the communication architecture was built on TCP/IP, and the used peer-to-peer topology needed a lookup mechanism that became a single point of failure. There were many attempts to patch various issues with ROS, but none solved the core limitations of the overall architecture. Therefore, over the last few years, ROS was redesigned from the ground up to solve most of these shortcomings. The old architecture of ROS is now referred to as ROS 1 [26, 27], and the new one is ROS 2 [28, 29]. This redesign of ROS has started a massive community effort to migrate most of the libraries to ROS 2, which is still ongoing today. However, ROS 1 will be discontinued in 2025 with its last LTS distribution, as is shown in Table 2.1, which lists some of the most recent distributions of ROS. Because ROS 2 is the future of ROS, this chapter mainly describes the architecture and concepts of ROS 2.

[1]It can also be described as a robotics software development kit (SDK).

Table 2.1 List of most recent ROS distributions

Distribution	LTS	ROS version	Release	End-of-life
Iron Irwini	No	ROS 2	May 2023	-
Humble Hawksbill	Yes	ROS 2	May 2022	May 2027
Galactic Geochelone	No	ROS 2	May 2021	December 2022
Foxy Fitzroy	Yes	ROS 2	June 2020	May 2023
Noetic Ninjemys	Yes	ROS 1	May 2020	May 2025
Eloquent Elusor	No	ROS 2	November 2019	November 2020
Melodic Morenia	Yes	ROS 1	May 2018	May 2023
Lunar Loggerhead	No	ROS 1	May 2017	May 2019

2.1.1 Motivation for robotic frameworks

Writing software for robotic systems is difficult because different types of robots use varying hardware components and sensors, and thus code reuse is not trivial. The code base for a robotic system can be enormous and usually requires the expertise of multiple domains, which is unattainable by a single researcher. Additionally, the scope and scale of these systems continue to grow. Therefore researchers started creating various, usually single-purpose, frameworks that tried to create abstraction or provide architectural methods to decompose the whole system into manageable pieces. Only a few could rival ROS in its significance and usage in the industry.

ROS was initially designed to solve specific challenges in developing large-scale service robots and mobile manipulators. However, the resulting architecture was far more general and reusable for other types of robotic systems. Its modular structure promotes reusability and collaborative development. It enables the development of generic logging, visualizations, and playback capabilities and is lightweight but scalable.

2.1.2 ROS 2 overview

ROS 2 aimed to fulfill several design requirements like security, real-time computing, product readiness, diverse network architectures, or embedded systems support. Additionally, the design was guided by a set of principles: distribution, abstraction, asynchrony, and modularity. These principles have their trade-offs but generally lead towards benefits like code reuse, global scale collaboration, better fault isolation, and software testing. The architecture of ROS 2 follows an approach of distributed systems with several abstraction layers. Modularity is enforced on multiple levels, and the whole ecosystem is distributed across many decoupled packages, so the users can choose which parts to use or exchange various components. As shown in Figure 2.1, the abstraction is generally hidden behind a set of client libraries that provide core communication APIs and are implemented in many different languages, making ROS language agnostic. These client libraries are used to develop components in ROS, and the main two languages used are Python and C++. However, the community has developed and continues to maintain many client libraries for other programming languages. These client libraries depend on a common intermediate interface written in C called *rcl*. Underneath this interface is a middleware abstraction layer called *rmw* (*ROS MiddleWare*).

The communication architecture of ROS 2 has been redesigned and based on *Data Distribution Service (DDS)*, an open standard for communication used in critical infrastructures. DDS uses UDP protocol, which does not automatically re-transmit data. On top of this, ROS 2 introduced a set of *Quality-of-Service (QoS)* settings that can be used for communication optimization for available bandwidth and latency but also allowed for designing embedded and real-time systems. Some of the most common QoS settings allow configuring if message delivery is guaranteed (*reliability*) or if the messages are forgotten after they are sent (*durability*), or how

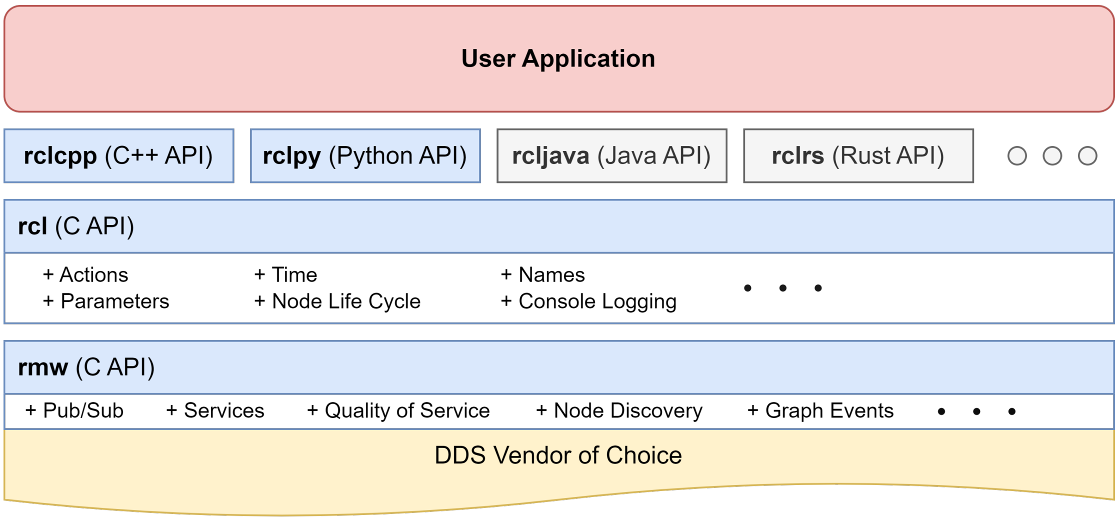


Figure 2.1 Client library API stack in ROS 2 (adapted from [28])

many messages are buffered when the network cannot keep up with the communication (*history*). The communication in ROS is asynchronous, creating an event-based system where each component can have a different frequency of providing and accepting data. As for security, DDS comes with its own security standard, but ROS 2 also provides additional tools for managing security infrastructure.

The ROS 2 ecosystem can be divided into three categories: *middleware*, *algorithms*, and *development tools*. The middleware category refers to the underlying communication infrastructure for sending data between components running in ROS. The second category aggregates libraries with implementations of standard algorithms used in robotics like planning, perception, or simultaneous localization and mapping (SLAM). Furthermore, ROS provides an abundance of command line and graphical development tools for logging, visualization, or other development processes.

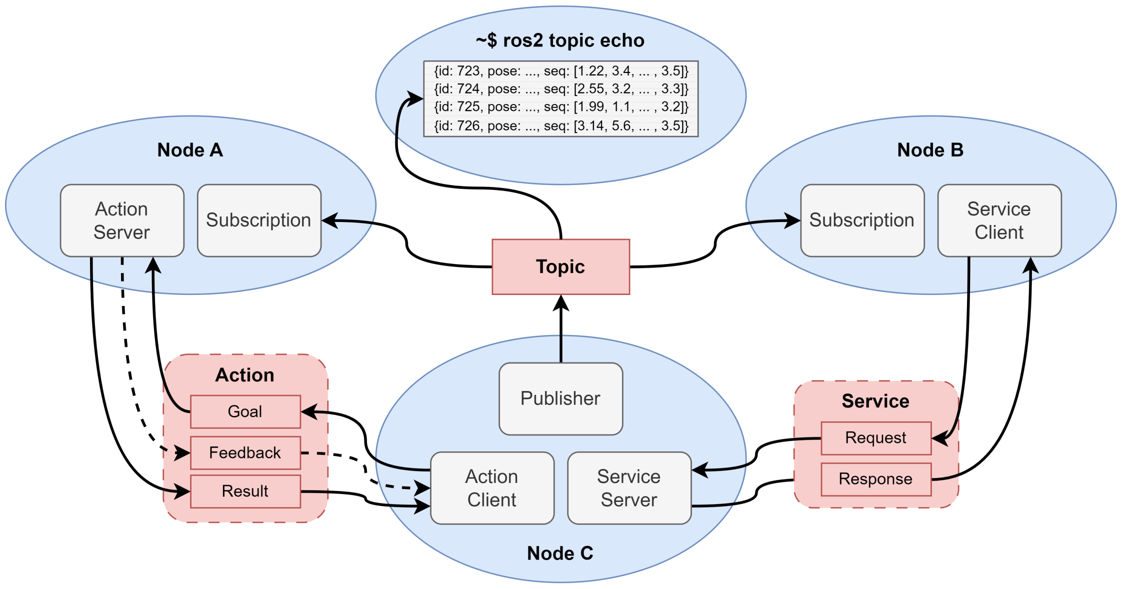
Unlike ROS 1, ROS 2 can integrate with the cloud making it possible to connect to cloud resources. ROS 2 should also have additional platform support for Windows and macOS, but Linux is still preferred and widely used for ROS development.

2.1.3 Concepts

This section provides an overview of some of the most fundamental concepts in ROS. Some of the concepts, like nodes and all the communication patterns, are depicted in Figure 2.2. The reader can refer to this visualization when reading about these concepts.

2.1.3.1 Nodes

Nodes are one of the fundamental concepts of ROS. It is an essential organizational unit allowing more straightforward reasoning about complex systems. In ROS 1, a node corresponded to a single process, but in ROS 2, multiple nodes can share a single process and use resources more efficiently. A robotic system typically comprises many nodes, each usually responsible for specific functionality. Nodes communicate explicitly via *message passing*, so the ROS components can have separate runtime execution contexts and be distributed over multiple heterogeneous systems. The term node arises from the fact that ROS systems are usually visualized at runtime as graphs with individual components as nodes and communication between them as arcs.



```
{id: 723, pose: ..., seq: [1.22, 3.4, ... , 3.5]}
{id: 724, pose: ..., seq: [2.55, 3.2, ... , 3.3]}
{id: 725, pose: ..., seq: [1.99, 1.1, ... , 3.2]}
{id: 726, pose: ..., seq: [3.14, 5.6, ... , 3.5]}
```

Figure 2.2 ROS 2 nodes and communication patterns with their interfaces (adapted from [28])

Such visualization can be found in Figure 2.3, where nodes are depicted as ovals and topics as rectangles.

Such a modular structure makes it possible to connect and disconnect nodes in runtime easily and thus dynamically modify the graph of nodes running in a system. This is especially useful when running nodes under active development alongside well-tested modules so that only the single node being developed and tested needs to be restarted repeatedly instead of the whole system. ROS 2 also brought a new pattern for managing the life cycle of nodes, which was not present in ROS 1. Nodes now have states like unconfigured, inactive, active, and finalized, which further enhance the system's management as a whole.

2.1.3.2 Messages

ROS nodes exchange data through *messages*, which are strictly typed data structures. The fields in a message can be either built-in primitive types, other messages, or even arrays of these. Messages are used in all three communication patterns ROS 2 provides: *topics*, *services*, and *actions*. Each pattern has to define its communication *interface* with message types[2], which is done using a language-neutral *interface definition language (IDL)*. An example of a simple message interface is shown in Code listing 2.1. This message has three fields. The first one is of a primitive type, and the second one is an unbounded dynamic array. The last field is of a message type, which has its own fields and can be defined in a different package.

Many commonly used messages are already available in ROS but users can provide their own custom interface definitions in the IDL format. A code required for communication in any used client library language is generated at compile time from the IDL interface definition. Because of this, ROS provides a language agnostic message passing scheme and components written in different languages can be mixed and matched.

2.1.3.3 Topics

Topics are an asynchronous communication pattern providing an anonymous publisher-subscriber architecture. It is the most straightforward and used pattern in ROS, allowing many-to-many

[2]A message is also an interface because topics, the simplest communication pattern, use simple messages.

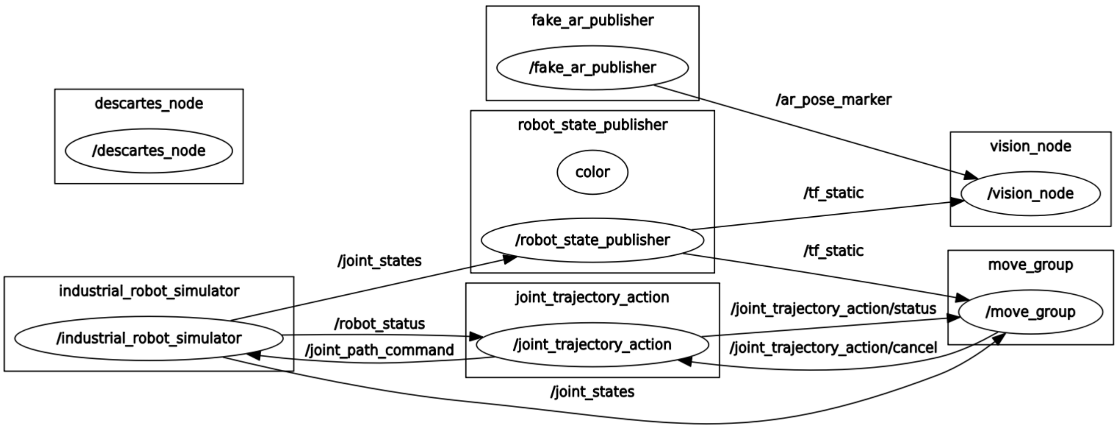


Figure 2.3 Visualization of nodes and topics running in ROS created by the *RQt* visualization tool (image from [30])

Code listing 2.1 Example message definition

```
uint32 id
float32[] seq
geometry_msgs/Pose pose
```

communication between nodes. The nodes can publish messages to topics, and other nodes can subscribe to topics to receive messages that were published there. Topics act as a bus for message exchange and are accessed by their name. A node can publish or subscribe to any number of topics simultaneously, and multiple concurrent *publishers* and *subscribers* for a single topic can exist. However, the message interface is strictly defined, so nodes can subscribe and publish only the defined message type. As the publishers and subscribers are unaware of each other's existence, the overall communication is anonymous. Because of this architecture, communication introspection can be done by simply subscribing to a topic that needs to be monitored, as shown in Figure 2.2.

2.1.3.4 Services

Services are a *request-response* communication pattern defined by a pair of messages. This pattern is analogous to web services. Unlike topics, only one node can advertise a service of a particular name, which is then referred to as a *service server*. Any number of other nodes can then send request messages to the advertised service, becoming *service clients*. ROS 2 makes it possible for the client's process not to be blocked during a call, and it can check for the response later. The client can ensure that a particular task was completed, as the request-response pair are associated together. The service interface defines the request-response message pair, as shown in Code listing 2.2.

2.1.3.5 Actions

Actions are an asynchronous communication pattern unique for ROS 2 that is best suited for goal-oriented, long-running tasks. They are similar to services with the difference that actions can be canceled anytime and provide periodic feedback during execution. The action interface has three parts: *goal*, *result*, and *feedback*. The goal and result are analogous to the service's request and response messages. An example of such an interface definition is shown in Code listing 2.3.

Code listing 2.2 Example service interface definition

```
# request
int32 a
int32 b
---
# response
int32 sum
```

Code listing 2.3 Example action interface definition

```
# Goal
geometry_msgs/Pose goal_pose
---
# Result
bool is_done
---
# Feedback
geometry_msgs/Pose current_pose
```

One node can advertise an action of a particular name as an *action server*. Then other nodes, as *action clients*, can trigger the action by sending a goal message. While waiting for the action to finish and receive the result message, they receive periodic feedback, usually reporting the progress. During that time, they can also choose to cancel the running action.

Internally, actions are built on topics and services. The periodic feedback is realized via a single topic. The goal and result messages are sent via services, where the receivers send an acknowledgment back to the sender in a response message. In Figure 2.2, action is depicted in a more simplified way.

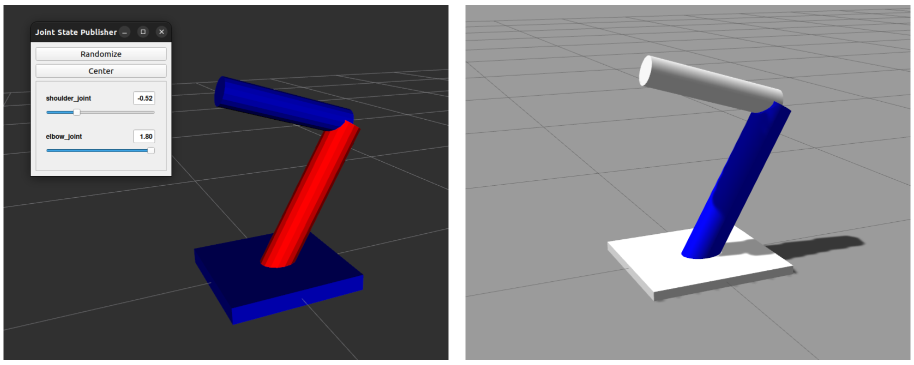
2.1.3.6 Parameters

Nodes can be configured via *parameters*. In ROS 1, parameters were global variables stored in a *parameter server*. However, in ROS 2, the parameters are implemented using service calls, and each node maintains its own set of parameters. The parameter type has to be declared in advance and is enforced. The values of parameters can be manipulated at runtime.

2.1.3.7 Packages

ROS applications are organized into *packages* that support the collaborative development of larger systems and are partially the result of the modular ecosystem of ROS. The definition of the ROS package is open-ended. In general, the ROS package is a container for ROS code, and in its minimal form, a package is a directory with an XML file describing it and listing its dependencies. A collection of such packages is a directory tree, where packages are at the leaves. In general, the ROS package is a container for ROS code. ROS provides utilities for building packages and their creation for a user-specified client library and specified dependencies.

As packages can aggregate a set of functionalities and components to support their configuration and startup, ROS provides a launch system. It makes it possible to define *launch files* that act as package executables. These launch files can be used to configure the system, including what nodes will run, where to run them, in what order they will start, or what arguments will be provided to them. Furthermore, the launch system is also responsible for monitoring the started processes, reporting their state, or reacting to their state changes. In ROS 1, launch files were defined in XML files, but in ROS 2, YAML and Python can also be used. Using Python scripts as launch files is especially powerful because it gives the developer more flexibility.



shoulder_joint -0.52
elbow_joint 1.80

(a) RViz visualization **(b)** RViz visualization

Figure 2.4 Simple robot example described in URDF visualized in RViz and Gazebo

2.1.4 Unified robotics description format

Unified robot description format (URDF) is an XML format for specifying robots in ROS, including their geometry, physical properties, control parameters, and more. The format is standardized, and libraries inside ROS know how to use it. Code listing 2.4 contains an example description of a simple robot constructed from basic shapes with a base link and two cylindrical links connected by two revolute joints. Figure 2.4 then shows its visualization in both RViz and Gazebo[3].

The `<link>` and `<joint>` XML blocks are two fundamental building blocks of robot description. The `<link>` block describes visual geometry, collision geometry, and physical properties like the mass and inertia of a link. The link geometry can be described using basic shapes, as in Code listing 2.4, but also complex meshes in standard formats like STL, DAE, or Wavefront (OBJ), with the first two being used the most. The `<joint>` block describes the joint's type and parameters, like their limits and axes of movement, and references both parent and child links. All measurements in URDF format use meters for distance, radians for angles, and kilograms for weight.

Other XML tags can define sensors or transmissions between actuators and joints. Some tools or packages can also extend the URDF format to include their specific description elements. A great example is Gazebo, which includes tags for configuring its plugins.

2.1.4.1 Materials

The URDF description can also define some simple materials that can be applied to links. The materials are usually solid colors, but a simple texture could also be used, although it is more complicated to define in the URDF. In the example code in Code listing 2.4, a blue material is defined and applied to the base link and one of the two cylindrical links. In the RViz visualization shown in Figure 2.4a, the two links are displayed in blue color, and the last link has a red color, which is the default RViz material. Unfortunately, these materials are ignored by Gazebo, as shown in Figure 2.4b, because Gazebo uses its own material definitions. This is done by defining a Gazebo reference for a specific link and specifying one of the materials defined by Gazebo, as seen at the end of Code listing 2.4. Gazebo's blue material has been applied to one of the robot's links, and in Gazebo, it appears in blue. However, the remaining URDF materials of the other

[3]RViz and Gazebo are described in Secions 2.1.6.2 and 2.2. Note that the robot described in Code Listing 4 could not spawn in a Gazebo scene. Gazebo requires the robot description to include physical properties of the links, like mass, collision geometry, and inertia tensors. The Gazebo visualization was obtained by extending the URDF considerably and is meant to show different material handling discussed in Section 2.1.4.1.

Code listing 2.4 Example of a URDF file describing a simple robot with two revolute joints and a base with two cylindrical links

```
<?xml version="1.0"?>
<robot name="simple_robot">
  <material name="blue">
    <color rgba="0␣0␣0.8␣1" />
  </material>

  <link name="base_link">
    <visual>
      <geometry>
        <box size="0.4␣0.4␣0.05" />
      </geometry>
      <material name="blue" />
    </visual>
  </link>

  <link name="arm_link">
    <visual>
      <origin xyz="0␣0␣0.25" rpy="0␣0␣0" />
      <geometry>
        <cylinder radius="0.05" length="0.5" />
      </geometry>
    </visual>
  </link>

  <link name="forearm_link">
    <visual>
      <origin xyz="0␣0␣0.2" rpy="0␣0␣0" />
      <geometry>
        <cylinder radius="0.04" length="0.4" />
      </geometry>
      <material name="blue" />
    </visual>
  </link>

  <joint name="shoulder_joint" type="revolute">
    <parent link="base_link" />
    <child link="arm_link" />
    <axis xyz="0␣1␣0" />
    <limit effort="10" velocity="1.0" lower="-1.1" upper="1.1" />
  </joint>

  <joint name="elbow_joint" type="revolute">
    <parent link="arm_link" />
    <child link="forearm_link" />
    <origin xyz="0␣0␣0.5" rpy="0␣0␣0" />
    <axis xyz="0␣1␣0" />
    <limit effort="10" velocity="1.0" lower="-1.8" upper="1.8" />
  </joint>

  <gazebo reference="arm_link">
    <material>Gazebo/Blue</material>
  </gazebo>
</robot>
```

links are ignored, and Gazebo displays them in its default white material. So, suppose the user wants to define the materials in the robot description directly. In that case, they usually have to define the materials twice and use the Gazebo's limited list of pre-defined colors.

A more flexible and straightforward approach is including the material data directly in the mesh file. The DAE format, unlike STL, contains information about materials and textures in the same file. These are automatically used in Gazebo and RViz when the robot description with the mesh is used[4]. The advantage of this approach is that different parts of the mesh can have different materials, and advanced texturing methods can be used to achieve the desired look.

2.1.4.2 Xacro

Robot descriptions in URDF format can quickly become extensive and unclear, even for simple robots. This is problematic, especially when the design of the robot is subject to change or the description file has to be maintained regularly. The *Xacro* package simplifies the writing and maintenance of robot descriptions by extending the URDF format with additional functionalities, reducing the amount of work that has to be done. The Xacro format is beneficial for development, but other packages and components usually do not support it. Fortunately, the Xacro package provides a command line utility that generates a URDF file out of a Xacro robot description. This conversion is often done directly in a launch file, where the URDF description will be used.

Xacro, which stands for XML macro, extends the URDF format by adding properties and macros, performing mathematical operations, and splitting the description into multiple description files. Properties in the Xacro format define constant values that can be referenced throughout the robot description by name. When the value needs to be changed, perhaps due to a design decision, it can be done in a single place. Macros serve a similar purpose as the properties. Oftentimes, various URDF elements are repeated multiple times throughout the robot description, and if the element has to be changed, it needs to be modified in multiple places. *Macros* can be used to define potentially parametrized URDF elements that can be reused in the robot description repeatedly. Xacro also makes it possible to split the description into multiple xacro files and compose the whole robot description by including them. For example, macros, properties, and materials could be in separate files and then be included in the main xacro file, which defines the links and joints of the robot.

2.1.4.3 Semantic robot description format

Semantic robot description format (SRDF) is a complement to URDF that is intended to specify additional semantic information about the robotic system that is not in the URDF file. For example, the SRDF file may include additional virtual or passive joints, joint and link groups, default robot configurations and poses, or additional collision-checking information for ignoring self-collisions of specific links. SRDF can be used for various applications, like specifying semantic information for motion planning in the MoveIt project (see Section 2.1.7). In the case of MoveIt, the SRDF file can be generated using a provided setup assistant with GUI.

2.1.5 ros2_control

The `ros2_control` package serves as a platform for connecting *hardware interfaces* and control algorithms, also called *controllers*. It decouples the two software groups and makes them communicate through a common interface. This allows easier maintenance and extendability of the robot control, and the user can pick and choose different controllers and hardware interfaces based on their needs. Moreover, as communication is done over a common interface, sharing and using existing controllers and hardware interfaces is easier.

[4]The two tools visualize colors differently, so they might not match the intended shade.

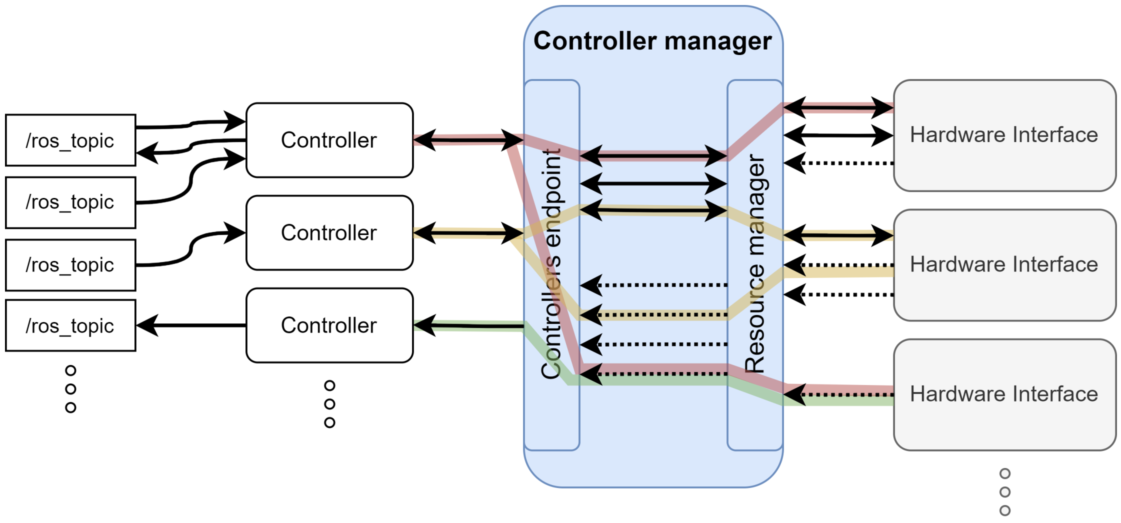


Figure 2.5 An illustrated example of controllers and hardware interfaces managed by the controller manager from the ros2_control package

The `ros2_control` does not need to know what type of robot is controlled. It only knows what hardware interfaces are available and what control algorithms should communicate with them. A component called the controller manager is responsible for managing available hardware interfaces and controllers and pairing them together.

2.1.5.1 Hardware interfaces

The *hardware interface*, sometimes called the hardware component, is software that directly communicates with the robot's hardware and exposes it to ROS in the standard `ros2_control` way. It is almost exclusively written in C++, as it is meant to be fast and create minimal latency between the hardware and the control algorithms. A hardware interface is specific to a particular piece of hardware and its API and capabilities, so usually, the same hardware interface cannot be reused on two different actuators from different manufacturers. Hardware manufacturers sometimes provide hardware interfaces for their products, but the robot users must often develop the hardware interface themselves.

The hardware interface represents the hardware by exposing *command interfaces* and *state interfaces*. Command interfaces are used for things in the hardware that can be controlled and have read and write capabilities. So for a motor that can be controlled by both velocity and torque, there would be one command interface for velocity control and one for torque control. However, in that example, they would not be used simultaneously. The state interfaces are used for things that can only be monitored and have only read capabilities. So the same motor might have encoders that can report the velocity and position of the robot. Therefore, there would be two state interfaces for velocity and position, and they could be used simultaneously because they only read the data. Hardware interfaces are also used for non-actuated components like sensors.

A robotic system can have multiple hardware interfaces, each exposing multiple command and state interfaces. The control manager has a component called the *resource manager* which is responsible for aggregating all the available interfaces and exposing them together so the controllers can access the whole list of available interfaces. This is illustrated on the right side of Figure 2.5. However, the resource manager has to know what hardware interfaces are available for the robot. These are listed in the URDF robot description directly inside a `ros2_control` description block.

2.1.5.2 Controllers

The ROS ecosystem uses the *controllers* to interact with `ros2_control`, and they will be designed to accommodate specific robot applications. On one side, they can subscribe to other ROS topics to listen for inputs like joint states or direct teleoperation. On the other side, they communicate with the hardware interfaces. The controllers can perform some computations to calculate the correct inputs for hardware interfaces. However, controllers do not necessarily need to control anything. They can also be used only to publish data from hardware interfaces to ROS topics. Because there are some typical applications in robotics, the `ros2_control` package already provides several controllers that can be used for them, but the user can write custom controllers.

The *controller manager* is responsible for managing the controllers. It will load specified controllers and match them with appropriate command and state interfaces, as shown in Figure 2.5. Only one controller can use one command interface. The state interfaces, however, can be shared between multiple controllers as they are read-only. If several controllers are loaded, the controller manager can be used to switch between them if needed. A YAML configuration is created to set up the controllers and provide the required parameters, and it is passed to the controller manager during startup. Besides the YAML configuration, the controller manager usually needs the robot description from the URDF file. It can be started via a node provided inside the `ros2_control` package, but the user can also write their own node that will start and use the controller manager. Once the controller manager is running, the user can interact with it using several available ROS services, command line tools, or specialized nodes.

2.1.6 Visualization tools

ROS provides a range of debugging and visualization tools for developers to use. Visualization is beneficial when working with ROS and in robotics in general. It helps with visual design verification during the design phase of a robotic system, but it is also used in the production phase when the system needs to be monitored. One of the more unique use cases for ROS is message visualization. Because the messages are often not in human-readable form or transmitted with a high frequency, simple communication introspection by subscribing to topics from the command line is insufficient. Fortunately, this modular architecture of ROS 2 allows for creation of generic visualization tools such as RQt and RViz.

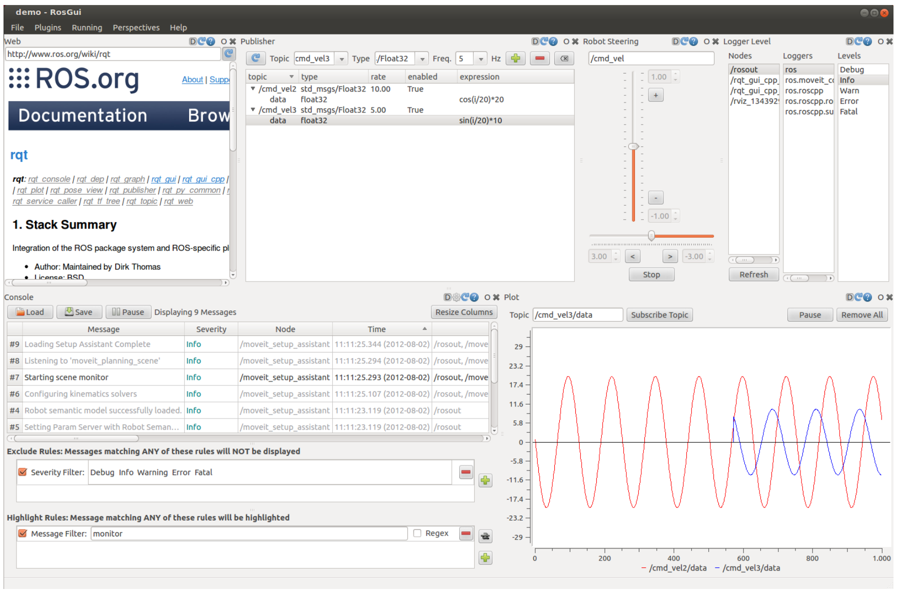
2.1.6.1 RQt

RQt [31] is a plugin-based visualization framework for ROS built using Qt [32]. Users can run all the existing GUI tools as dockable windows in RQt, allowing them to create custom visualization interfaces for their applications, as shown in Figure 2.6. The various tools, referred to as RQt plugins, can also be run as standalone applications, but RQt makes it easier to manage and view all the tools in a single window. Using either Python or C++, users can also create their own RQt plugins or easily turn existing Qt widgets into RQt plugins.

There are numerous RQt plugins to choose from. The `rqt_graph` is a standard plugin for visualizing the ROS computation graph and, optionally, some topic statistics. Figure 2.3 was exported from this plugin. The `rqt_console` provides a GUI for viewing and filtering ROS messages being published with the ability to display detailed information about any of them. Other plugins facilitate interactive Python console, web browser, image viewing, plotting functionalities, or means to publish messages or call services and actions.

2.1.6.2 RViz

RViz [34] is a powerful 3D visualization tool for ROS. Its strength and one of its primary uses is a visualization of what the robot sees through its sensors, like cameras and lasers. RViz does this by subscribing to appropriate topics and visually representing incoming messages like



http://www.ros.org/wiki/rqt

rqt

rqt: rqt_console | rqt_dep | rqt_graph | rqt_gui | rqt_gui_cpp | | rqt_plot | rqt_pose_view | rqt_publisher | rqt_py_common | r rqt_service_caller | rqt_tf_tree | rqt_topic | rqt_web

1. Stack Summary

Integration of the ROS package system and ROS-specific pl

- Author: Maintained by Dirk Thomas
- License: BSD

Topic cmd_vel3 Type ./Float32 Freq. 5 Hz

topic	type	rate	enabled	expression
▼ /cmd_vel2	std_msgs/Float32	10.00	True	
data	float32			cos(i/20)*20
▼ /cmd_vel3	std_msgs/Float32	5.00	True	
data	float32			sin(i/20)*10

/cmd_vel

1.00 + - -1.00

3.00 < > -3.00

Nodes	Loggers	Levels
/rosout	ros	Debug
/rqt_gui_cpp_	ros.moveit_c	Info
/rqt_gui_cpp_	ros.roscpp	Warn
/rviz_134392	ros.roscpp.ro	Error
	ros.roscpp.su	Fatal

Displaying 9 Messages

	Message	Severity	Node	Time	
#9	Loading Setup Assistant Complete	Info	/moveit_setup_assistant	11:11:25.344 (2012-08-02)	/rosout, /move
#8	Listening to 'moveit_planning_scene'	Info	/moveit_setup_assistant	11:11:25.294 (2012-08-02)	/rosout, /move
#7	Starting scene monitor	Info	/moveit_setup_assistant	11:11:25.293 (2012-08-02)	/rosout, /move
#6	Configuring kinematics solvers	Info	/moveit_setup_assistant	11:11:25.107 (2012-08-02)	/rosout, /move
#4	Robot semantic model successfully loaded.	Info	/moveit_setup_assistant	11:11:23.119 (2012-08-02)	/rosout
#5	Setting Param Server with Robot Seman…	Info	/moveit_setup_assistant	11:11:23.119 (2012-08-02)	/rosout

Exclude Rules: Messages matching ANY of these rules will NOT be displayed

☑ Severity Filter: Debug Info Warning Error Fatal

Highlight Rules: Message matching ANY of these rules will be highlighted

☑ Message Filter: monitor ☐ Regex

Topic /cmd_vel3/data

Figure 2.6 Multiple RQt plugins docked in RQt layout (image from [33])

images, point clouds, depth maps, meshes, and more. It is also great for visualizing URDF robot descriptions during their development or validating a robotic system's design. Another powerful feature of RViz is the support of custom visual markers that can be implemented for visual debugging purposes, like displaying a trajectory a robot is following. Interactive markers can also be implemented so the developer can use markers in RViz to interact with the robotic system. A great example is an interactive gizmo at the manipulator's end effector, through which an operator can control the robot from RViz. A screenshot from RViz is shown in Figure 2.7.

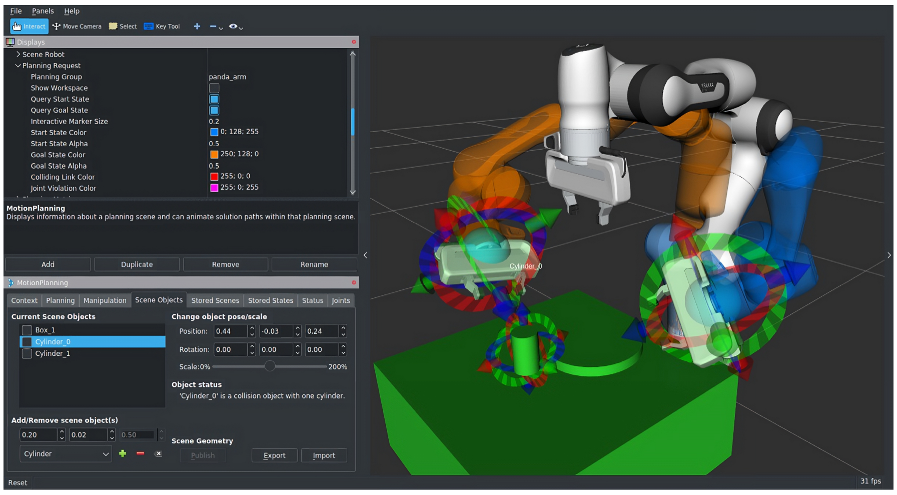
Like RQt, RViz is plugin-based, allowing users to develop custom tools or visualization types in C++ and arrange their windows into a custom layout specifically for their application. As with all other libraries and components of ROS, it also had to be ported to ROS 2. The ROS 2 version of RViz is referred to as RViz2, and fortunately, its migration is almost finished, and most of the plugins have been successfully ported.

Importantly, RViz is not a simulator. It can only visualize data and has no physics engine to compute interactions between objects in the environment. However, it is common to use RViz together with a simulation tool. The simulation tool is used to visualize and simulate the robotic system and its environment, and RViz is used to visualize message data or debugging markers.

2.1.7 MoveIt

MoveIt [36, 35] is a widely used open-source motion planning framework for ROS. It incorporates motion planning, manipulation, control, perception, and navigation. MoveIt can solve inverse kinematics and generate trajectories for robotic manipulators that can be directly published through standard interfaces to control robotic hardware or its digital twin. As shown in Figure 2.7, MoveIt provides RViz tools for out-of-the-box visualization and testing of various planning algorithms and control. The included planners in MoveIt use algorithms from the *Open Motion Planning Library (OMPL)* [37], *Trajectory Optimization for Motion Planning (trajopt)* framework [38], and *Pilz Industrial Motion Planner* [39]. MoveIt also provides a setup assistant with GUI that helps users to create a MoveIt configuration for their robotic manipulator.

As with all ROS tools, MoveIt is going through the migration process to ROS 2. The new port is called MoveIt 2, and the migration process is almost finished. However, the GitHub



Scene Robot
Planning Request
Planning Group panda_arm
Show Workspace ☐
Query Start State ☑
Query Goal State ☑
Interactive Marker Size 0.2
Start State Color 🟦 0; 128; 255
Start State Alpha 0.5
Goal State Color 🟧 250; 128; 0
Goal State Alpha 0.5
Colliding Link Color 🟥 255; 0; 0
Joint Violation Color 🟪 255; 0; 255

MotionPlanning
Displays information about a planning scene and can animate solution paths within that planning scene.

Current Scene Objects
Box_1
Cylinder_0
Cylinder_1

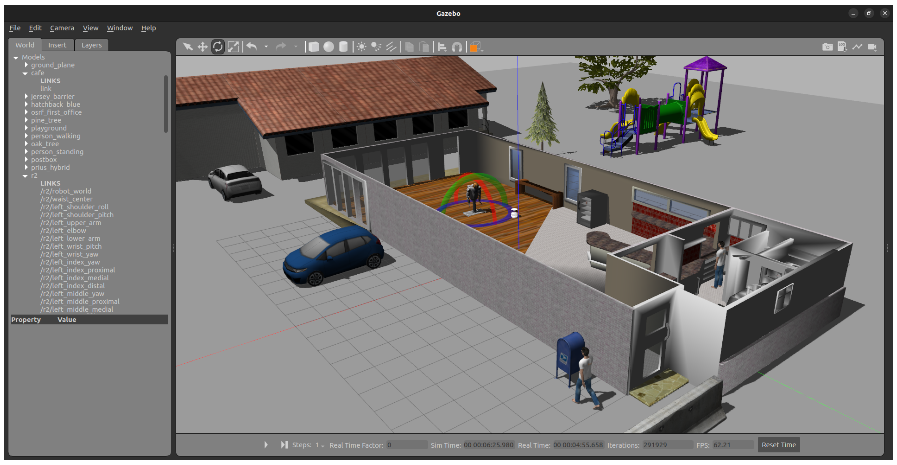
Change object pose/scale
Position: 0.44 -0.03 0.24
Rotation: 0.00 0.00 0.00
Scale:0% 200%

Object status
'Cylinder_0' is a collision object with one cylinder.

Add/Remove scene object(s)
0.20 0.02 0.50
Cylinder

Scene Geometry

Figure 2.7 RViz visualization of a robotic arm with MoveIt plugins (image from [35])



Models
ground_plane
cafe
LINKS
link
jersey_barrier
hatchback_blue
osrf_first_office
pine_tree
playground
person_walking
oak_tree
person_standing
postbox
prius_hybrid
r2
LINKS
/r2/robot_world
/r2/waist_center
/r2/left_shoulder_roll
/r2/left_shoulder_pitch
/r2/left_upper_arm
/r2/left_elbow
/r2/left_lower_arm
/r2/left_wrist_pitch
/r2/left_wrist_yaw
/r2/left_index_yaw
/r2/left_index_proximal
/r2/left_index_medial
/r2/left_index_distal
/r2/left_middle_yaw
/r2/left_middle_proximal
/r2/left_middle_medial

Property Value

Steps: 1 Real Time Factor: 0 Sim Time: 00 00:06:25.980 Real Time: 00 00:04:55.658 Iterations: 291929 FPS: 62.21

Figure 2.8 Screenshot from Gazebo

repository of MoveIt 2 [40] is overflowing with open issues and bug reports, so it can be currently challenging to integrate MoveIt 2 with a robotic application in ROS 2.

2.2 Gazebo

Simulation enables the emulation of physical environments and the testing of robotic systems in those environments. Section 1.4 mentions this is critical for developing mechanical hardware and applications, as it quickens the iterations between modifications. Robotic simulators are usually capable of simulating physics as well as sensors such as lasers. One such simulator that is extensively used in the industry and built with ROS in mind is *Gazebo*[5].

Gazebo [41] is a free and open-source 3D simulation software developed by its community led by Open Robotics, currently in its last version, Gazebo 11. Its development started in 2002, and

[5]Because of its redesign and migration process to ROS 2 (see Section 2.2.3.1), Gazebo is now referred to as *Gazebo Classic*. The new version was called *Ignition* or *Ignition Gazebo* but was renamed back to Gazebo in 2022. For convenience and clarity, this text uses the name Gazebo for the classic version of Gazebo and Ignition Gazebo for the new redesign of Gazebo.

it is a general-purpose robotic simulator that does not specialize in one type of robotics system. It has evolved alongside ROS, sharing a significant portion of its community. Therefore, one of its considerable advantages is its out-of-the-box integration with ROS. Figure 2.8 shows the default layout of the Gazebo simulator.

2.2.1 Components

Several components come into play when it comes to using Gazebo. Arguably, rendering and physics are the most critical aspects of any 3D simulation software, but the GUI and scene editor is also helpful. As ROS, Gazebo also follows a distributed architecture where all separate libraries are used for different simulator components, like physics, rendering, sensors, and GUI.

2.2.1.1 Rendering and physics

Gazebo uses the *Object-Oriented Graphics Rendering Engine (OGRE)* [42] for 3D rendering. It is an open-source real-time rendering engine that is scene-oriented. The GUI of Gazebo is built on the Qt library.

For the physics simulation, Gazebo uses its physics library, which provides a generic interface to physics simulation, including rigid bodies, colliders, and joints. By default, Gazebo uses the ODE physics engine, but the physics library has also been integrated with Bullet, Simbody, and DART. Unfortunately, the only way to switch between the physics engines is to build Gazebo from source code with a different physics engine.

2.2.1.2 Headless mode

Commonly, simulation software like Gazebo supports execution in *headless mode*, which refers to running the software without the graphical user interface. In the headless mode, the simulation runs in the background without rendering and displaying visual output. Because of this, some components like rendering can be omitted, spending fewer resources and possibly using them in the physics simulation. As no visuals have to be displayed, the simulation can run as fast as possible, which is useful when training machine learning models.

In addition to the separate libraries, Gazebo is divided into two separate processes: `gzserver` and `gzclient`. In regular use, these two processes run simultaneously and communicate with each other. While `gzserver` is responsible for simulating the physics, sensors, and rendering, the `gzclient` provides the graphical interface for visualization and interaction with the simulation. When using Gazebo in headless mode, only `gzserver` runs the simulation.

2.2.1.3 Models

Gazebo includes an extensive collection of pre-built models ranging from simple props to complex industrial robots and buildings in its *model library*. The model library is open to community contributions, so it gradually grows as users share their models. The library is accessible in Gazebo through the internet, so it takes a while until the list of models loads in the GUI, and when a model is selected, Gazebo has to download it. Custom user models saved on their computer will also appear in the model library after a path to the models is imported to Gazebo.

2.2.1.4 GUI and scene editor

The graphical user interface of the Gazebo is divided into several panels providing various functionalities to interact with the scene and the simulation. The central part of the simulator is the scene view, where simulated objects are rendered as they interact with the environment. Using a mouse, users can interact with this scene view to move, rotate and zoom the camera view or

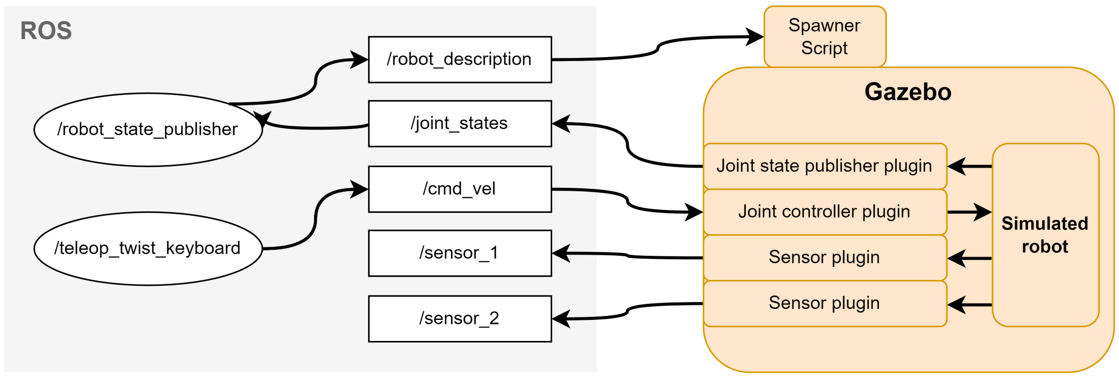


Figure 2.9 Illustration of Gazebo using plugins to interact with ROS

select objects. Two side panels can be displayed, resized, or hidden. The left panel provides the means to view the list of models in the scene, modify the parameters of the models, open the model library, or organize and manage visualization groups. The right panel, which is hidden by default, is used only for interaction with mobile parts when a model in the scene is selected. The upper toolbar exposes some of the most used tools for the scene editor, like moving, rotating, and scaling the selected objects, creating basic 3D shapes and light sources. Simulation data like time passed in simulation, the real-time factor (see Section 2.2.1.7), or camera FPS is shown in the bottom toolbar. It also provides interactive buttons to play or pause the simulation or manually step forward through the simulation. Overall the user interface of the Gazebo is simple and user-friendly.

2.2.1.5 Gazebo-ROS communication

Every time Gazebo wants to interact with something outside itself, including ROS, it needs to use plugins. *Plugins* are pieces of code that Gazebo can execute at a specified or appropriate time. For example, a plugin would be used to receive control inputs from ROS and move the simulated robot, another one would be used for reporting simulated joint poses back to ROS, and for any simulated sensor, a separate plugin would get its data and send it to the corresponding ROS topic. Figure 2.9 shows an example of a possible Gazebo-ROS interaction. When running Gazebo with ROS integrations, instead of running it directly, Gazebo provides a ROS package called `gazebo_ros` with a launch file that starts Gazebo with some of the ROS interactions for us. For example, Gazebo will automatically begin publishing some performance metrics to a ROS topic.

2.2.1.6 URDF extension

The `<gazebo>` element is an extension of the URDF format that can be used for specifying properties for Gazebo simulation. There are three types of the `<gazebo>` element. It can either reference one specific link or a joint, as shown in Code listing 2.4, or it can be specified without the reference property, which means it is related to the whole robot model. The element is mainly used to determine what control or sensor plugins run in Gazebo or specify additional physical or kinematic properties used in the simulation. Only Gazebo utilizes these XML elements, so when Gazebo is not running, the elements are ignored.

In some robotic systems, like robotic arms, it is desirable to have certain robot parts fixed to the environment. The example robotic arm defined in Code listing 2.4 would fall on its side in a Gazebo simulation due to movement or gravity because the base is not fixed to the ground. To fix this issue, the URDF would be extended by a virtual fixed joint that attaches the `base_link` to an additional link called `world`, as shown in Code listing 2.5.

Code listing 2.5 Fixing the base link to the world with a virtual fixed joint

```
<link name="world" />
<joint name="virtual_joint" type="fixed">
  <parent link="world" />
  <child link="base_link" />
</joint>
```

2.2.1.7 Simulation time

ROS keeps track of time, which by default uses the system clock, and all the running nodes can synchronize by using the *ROS API*. However, in simulation, it is common to pause or restart the *simulation time* or run the simulation at a different speed. If the ROS nodes are synchronized with the system time, some unexpected behavior might occur in the simulation. Therefore, all nodes in ROS have a parameter called `use_sim_time`, which, when set to `true`, the ROS API calls will use the time that is published into `/clock` topic. Gazebo and other simulation tools can publish the simulation time into this topic.

A standard metric in simulation is the *real-time factor (RTS)*, which describes how fast the simulation runs. A real-time factor equal to one means the simulation runs in real-time. A factor greater than one means that the simulation runs faster, which is very useful when the visualization of the simulation is not essential or the simulation is used in the training process of machine learning algorithms. On the other hand, a factor of less than one slows down the simulation time. Gazebo lowers the real-time factor if the simulation is too complex, and the physics engine could use more time for its computations.

2.2.2 SDF

The *SDFormat*, or *Simulation Description Format (SDF)*, is an XML format used in Gazebo for both robot and environment descriptions. It is similar to URDF as it can be used for an accurate description of a robot. While URDF can be used to describe a robotic system, SDF is more general and can be used to define a whole scene with multiple robots, objects, and light sources inside. Gazebo can then load these SDF files as scenes and models that can be imported into the environment. The advantage of describing models and scenes in the XML format is that they can be modified programmatically, making a simulation process easier to automate.

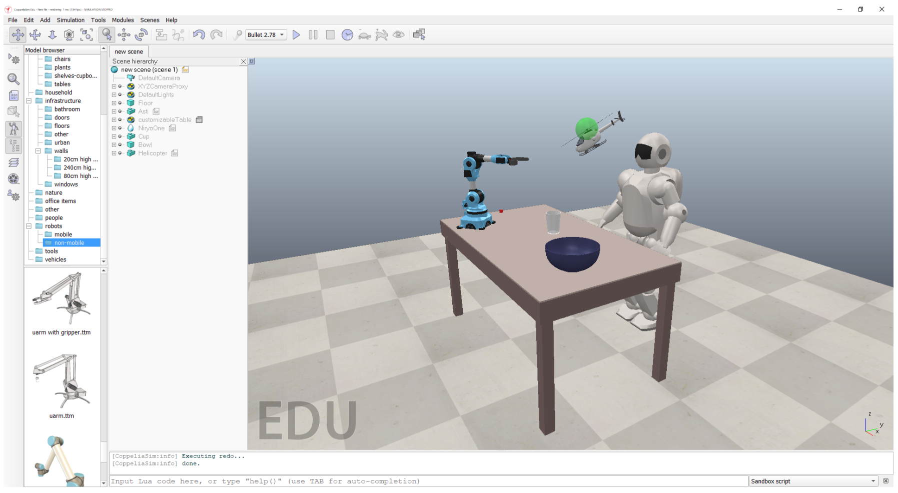
In ROS, the robots are described in the URDF format, which differs from SDF. It is still possible to use URDF description because Gazebo provides tools that can convert URDF to SDF format. Specifically, the ROS package `gazebo_ros` includes a spawner script, illustrated in Figure 2.9, that spawns a robot model in a running Gazebo scene. When the script is provided with a URDF file, it converts it to SDF.

2.2.3 Alternatives

There are many alternatives to choose from when it comes to robotic simulation, and comparative studies are often done to compare their features or simulation capabilities [43, 44]. The most notable include *CoppeliaSim* [45], *Webots* [46], and *ARGoS* [47]. There are also various domain-specific simulators which are usually modified or created for a specific application. Lastly, the *Ignition Gazebo* will soon replace Gazebo as its successor.

2.2.3.1 Ignition Gazebo

Like ROS 2, *Ignition Gazebo* is a complete architectural redesign of the classic Gazebo, bringing a more modular and distributed design with a new GUI overhaul. Ignition Gazebo has been



```
[CoppeliaSim:info] Executing redo...
[CoppeliaSim:info] done.
Input Lua code here, or type "help()" (use TAB for auto-completion)
```

Figure 2.10 Screenshot from CoppeliaSim simulator

developed alongside ROS 2 and is the successor of Gazebo, which will meet its end of life in 2025. Even though Ignition Gazebo is still not complete and bug-free, it has proven itself in several applications and continues to evolve. The architecture change has enabled several significant modifications compared to Gazebo. Its modularity allows the interchanging of various components, like renderers or physics engines as plugins, without building the framework from source code. Also, it is supported on all major operating systems, including Windows and macOS.

The rendering engine stayed the OGRE, although Ignition Gazebo uses a newer version of OGRE. The default physics engine used has changed from ODE to DART, and no other physics engines are currently supported. However, they will be added in the future as plugins.

2.2.3.2 CoppeliaSim

CoppeliaSim [45], formerly known and often found in literature as the *Virtual Robot Experimentation Platform (V-REP)*, is a versatile robotic simulation framework used in the industry developed by Coppelia Robotics. Unlike Gazebo, it is cross-platform, so it can be easily installed and run on operating systems other than Linux. It is plugin-based, providing various functionalities like motion planning, data visualization, or image processing as interchangeable components. Users can develop their own plugins and add-ons for the framework. Scene management and creation are also more versatile than in Gazebo, and it is possible to customize user interface elements. CoppeliaSim can also be integrated with ROS and provide other programming approaches to customize the simulation, like scripting and developing remote API clients. Multiple physics engines are available as libraries, including ODE, Bullet, or MuJoCo, providing more flexibility in the physics simulation. As shown in the screenshot of the simulator in Figure 2.10, the rendered visualization in CoppeliaSim has a particular, almost stylized, look. Mainly all sharp edges in the scene are outlined, making the visualization clearer and objects more distinguishable.

CoppeliaSim has a bit more complicated and unclear licensing since not all of its libraries are open source, and a commercial license might be required for commercial use. There are three versions of the software available: *CoppeliaSim Player*, *CoppeliaSim Edu*, and *CoppeliaSim Pro*. The problem is that the three versions are stated to be different, but the differences are not specified explicitly. For example, the CoppeliaSim Player only has simulation functionalities, but editing is limited. Moreover, the pricing for the CoppeliaSim Pro, which can be used commercially without any restrictions, is not publicly available. Because the CoppeliaSim Player versions seem to be limited compared to the pro version, and CoppeliaSim Edu cannot be used by research

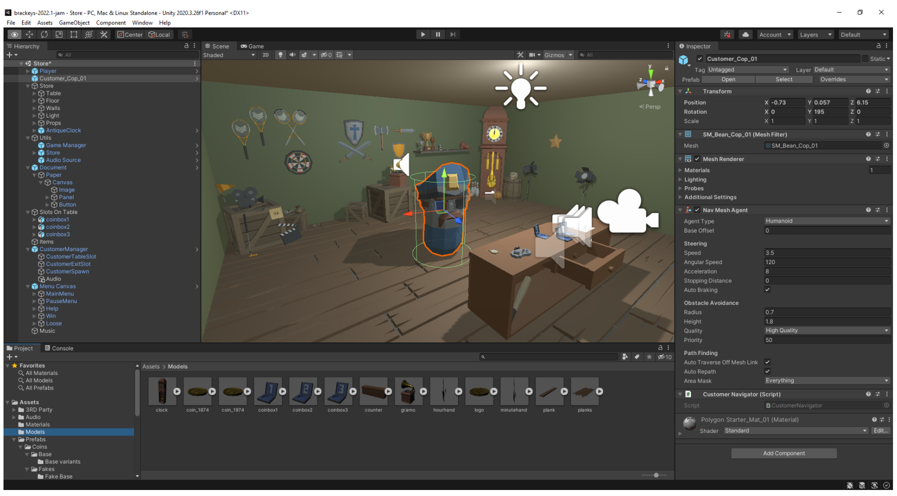


Figure 2.11 The default layout of the Unity Editor

institutions and non-profit organizations, Gazebo is used more in the research, and CoppeliaSim is more suited for commercial use.

2.3 Unity game engine

Unity [48], also formerly known as Unity3D, is a game engine developed by Unity Technologies since 2005 [49], which has grown into a modern real-time 3D development platform also extensively used across multiple non-game industries, including architecture, automotive, film, and more. Unity has flourished over the years because of its ease of use and gentle learning curve compared to similar tools, making it a preferable choice for interactive installations, data visualization, model prototyping, and research. Further advantages of Unity are a massive community and the abundance of official end community-created learning materials. The Unity community supports other developers and creates assets for others to use, thus increasing development speed and efficiency.

The applications in Unity are developed in C# programming language, and users can use various development tools included in the *Unity Editor* (shown in Figure 2.11), like a built-in profiler, debug console, version control, and more. Some use cases might not require programming at all. Unity's asset workflow makes it effortless to import, update, use, or even directly edit various assets. The 3D or 2D scenes are easily managed through the *scene view* or in the *hierarchy* window, and objects can be animated in the built-in animator. Various functionalities and scripts, called *components*, are already available and can be directly added to objects in the scene. The properties and methods from the scripts can be exposed in the *inspector* window so they can be configured or called easily from the editor directly, even during the run time.

Being multiplatform is another core feature of Unity, making it an excellent choice for a broad user base. Unity editor can build and deploy applications for almost any operating system for mobile, desktop, web, console, and VR platforms. Switching the target platform is usually a matter of two clicks in the build settings of the editor. On the other hand, the target platform for Unity Editor itself is Windows, with fairly decent support for macOS. The support for Linux could still be considered to be in a preview state[6], as it is limited to Ubuntu 18.04, Ubuntu

[6]In Unity, "preview" refers to a state in the development lifecycle of a package, which is considered somewhat experimental and risky to use in production.

20.04[7], and CentOS 7[8] [50], and users regularly encounter bugs or performance issues.

2.3.1 Main modules

The development platform provided by Unity is very flexible and modular, with its support of packages that can be imported and updated via a *package manager* accessible directly from the Unity editor. Unity Technologies provide a wide range of official packages available in the Unity Registry, which provide the core modules of the underlying engine and fundamental functionalities of the editor. User-made packages containing editor plugins, scripts, or assets are distributed through the official *Unity Asset Store* or published on GitHub and also can be imported through the package manager.

The lifecycle of a Unity package usually has three stages. First, it is *in development* and is not directly available to the users. When it is ready for Unity users to test and provide feedback, it enters the *preview* state. This stage is considered experimental, and it is risky to use in production. Some preview packages also require users to have additional training or expertise, thus, are not recommended to use in normal circumstances. Because of this, the preview packages are not discoverable in the package manager. After additional development and several testing and validation stages, preview packages enter the *verified* state and appear in the Unity editor.

2.3.1.1 Real-time rendering

As a game engine and 3D development platform, real-time rendering is a crucial module of Unity that has continuously improved to meet today's industry standards. Both 2D and 3D real-time rendering is supported, allowing the creation of nearly any type of interactive media. Unity provides several built-in render pipelines that can be used in different contexts and the required tools to create a custom render pipeline to suit the user's specific needs. Currently, the most versatile render pipeline is the *Universal Render Pipeline (URP)*, which makes it easy to optimize graphics across a range of target platforms, from mobile, including standalone VR headsets, to high-end computers. However, there are better choices for high-fidelity graphics trying to achieve photorealism[9].

Depending on their availability on the target platform, Unity supports several graphics APIs the user can choose from directly in the project settings inside the editor. The choice of graphics API that will be used will affect the application's performance. By default, Unity will automatically choose a preferred graphics API for the target platform, and the user will need to change this setting only in very specific use cases. Table 2.2 lists the supported graphics APIs based on the target platform.

Table 2.2 Supported graphics APIs based on the target platform

Platform	Supported graphics APIs
Windows	DirectX, OpenGL, Vulkan
Mac	Metal, OpenGL
Linux	OpenGL, Vulkan

2.3.1.2 Physics

Physics simulation is also an essential part of modern game engines. In games, physics can be used in many ways, such as procedural physics-based animations, projectile simulation, particle

[7]The current LTS version of Ubuntu is Ubuntu 22.04 LTS, released on April 2022.
[8]CentOS 7 will reach its end of life at the end of June 2024.
[9]For example, Unity precisely provides the *High Definition Render Pipeline (HDRP)* for this purpose.

systems, and rigid-body or soft-body simulations. The laws of physics can be altered to suit the needs of the specific game or application.

In object-oriented projects, Unity provides built-in 2D physics as an integration of the *Box2D* physics engine [51] and a built-in 3D physics, which integrates the *Nvidia PhysX* engine [21]. Users can also use third-party physics engines like Bullet [52] or MuJoCo [53] to replace the built-in physics. This might be because the third-party physics engine implements a feature that is not implemented in PhysX. As the physics engines improve over time, the physics simulations in Unity improve with them.

For data-oriented projects, Unity develops its own *Unity Physics* package, which is the default physics engine for Unity's *Data-Oriented Technology Stack (DOTS)*. Alternatively, the user can switch to an implementation of the *Havok* physics engine for Unity, which is subject to a specific licensing scheme and is available only for the users of Unity's Pro and Enterprise plans (see Section 2.3.3).

2.3.1.3 Virtual reality

Unity makes creating VR and AR experiences seamless by providing the *XR Interaction Toolkit* package. It is a high-level interaction system that provides cross-platform controller support with inputs, haptics, visual feedback, 3D and UI interactions, and more. Together with the URP and Unity's deployment workflow, it is relatively easy to develop and deploy VR and AR applications on standalone headsets, consoles, or computers with VR support.

2.3.1.4 Artificial intelligence

It is common for games to have various non-player characters (NPC) that need to be managed and controlled. A combination of pre-defined scenarios and behaviors evaluated in runtime is often used to create an illusion of intelligence for these characters. Techniques such as behavior trees, navigation meshes, path-finding, crowd simulation, or finite state machines are used to create these behaviors. Many of the path-finding features are provided in Unity within an *AI Module*.

Additionally, Unity developed *Unity ML-Agents Toolkit* [54], a project that enables the training of intelligent agents using various techniques like reinforcement learning or imitation learning, or any other method directly in the unity project. It can serve as an AI research platform or for creating NPC behaviors and automated testing.

Closely related is also the *Perception* package, which can be used for computer vision applications. It excels in generating large-scale synthetic datasets for the training and validation of computer vision algorithms.

2.3.1.5 Inputs and user interfaces

Unity provides a modern and mature *Input System*, which handles a significant amount of input capabilities with minimal integration and development done by the user. It is a high-level input module, and the user usually does not need to understand specific controllers and their layouts and haptic capabilities to implement generic controls for the application. The input system makes it easy to support connecting and disconnecting controllers during runtime, control remapping, or even *local co-op* multiplier[10].

Unity UI, a toolkit for designing and developing user interfaces, is also part of the Unity editor. It can be used to create and arrange UI elements directly from the editor and connect them to event systems that can trigger various user-defined functionalities.

[10]A type of non-networked multiplier, also known as *couch co-op*, where players play on a single device and use multiple controllers to control their individual characters.

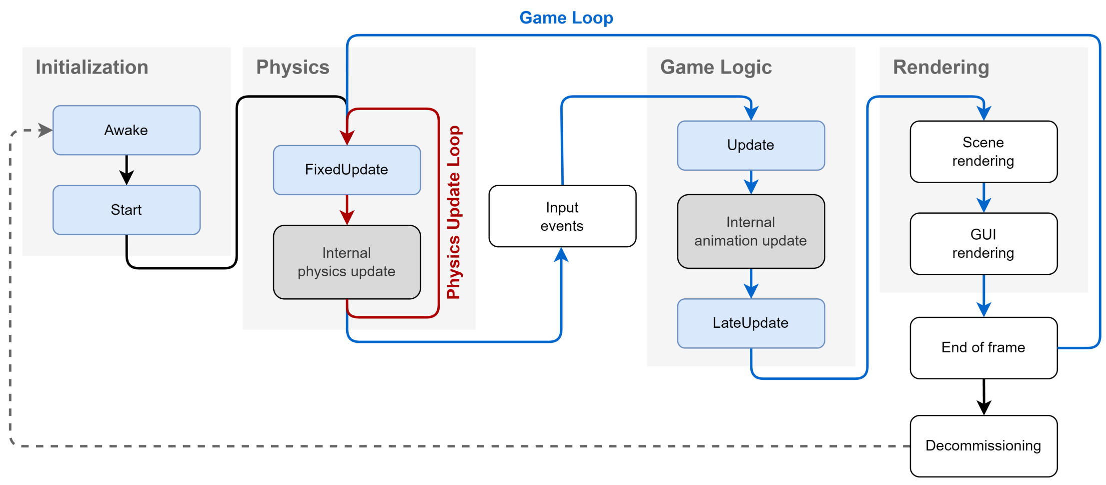


Figure 2.12 A simplified version of a script lifecycle flowchart in the Unity game engine

2.3.2 Game loop

Compared to other types of applications, one specific aspect of games, or other multimedia real-time applications, is their execution architecture. Games run in an active execution loop referred to as the *game loop* or, in a more general context, the *rendering loop*. Therefore game engines, including Unity, natively support and run under this architecture. The execution loop runs on the application's main thread and executes various event functions in a predetermined order. In this loop, the application continuously evaluates the physics interactions and input events, executes necessary logic, updates the state of the objects in the scene, and renders a new frame displayed to the user at the end of the loop. Generally speaking, the length of the individual loop iterations directly impacts the application's frame rate. If the developer executes a slow blocking operation in the code implementing the application's logic, the rendering would be blocked, negatively impacting the final user experience. Fortunately, most modern game engines offload some processes like physics computations, audio handling, or rendering to different threads to achieve current industry standards. In Figure 2.12, a simplified version of a script lifecycle flowchart with Unity's order of execution [55] is depicted.

This variable update frequency is inherent to this execution architecture but is not ideal for physics updates. Physics calculations generally benefit from a fixed update period, in other words, a fixed time step between updates. So the physics update loop seen in Figure 2.12 is, to some degree, independent of the main game cycle and is executed at a fixed rate. When the application's frame rate is higher than the fixed update rate, the physics update loop can be skipped during the game loop iteration. Otherwise, if the fixed time step is less than the frame update time, the physics update can be executed multiple times during one frame cycle. The fixed time step can be configured, and by default, Unity will update physics 50 times per second.

2.3.2.1 MonoBehaviour

Every Unity script derives from a base class called `MonoBehaviour`. The base class provides a framework that procures the script's life cycle, as shown in Figure 2.12, by providing hooks into events from the game loop. These include `Awake` and `Start hooks`, which run before the application enters the rendering loop, or the `Update`, `FixedUpdate`, and `LateUpdate` hooks, executed repeatedly. The `MonoBehavior` also allows attaching user-created scripts to objects in the scene as components. The base class also allows users to start, stop, and manage *coroutines*, a code that can run asynchronously over a longer period or wait for specific actions to complete.

Code listing 2.6 Example of a Unity script

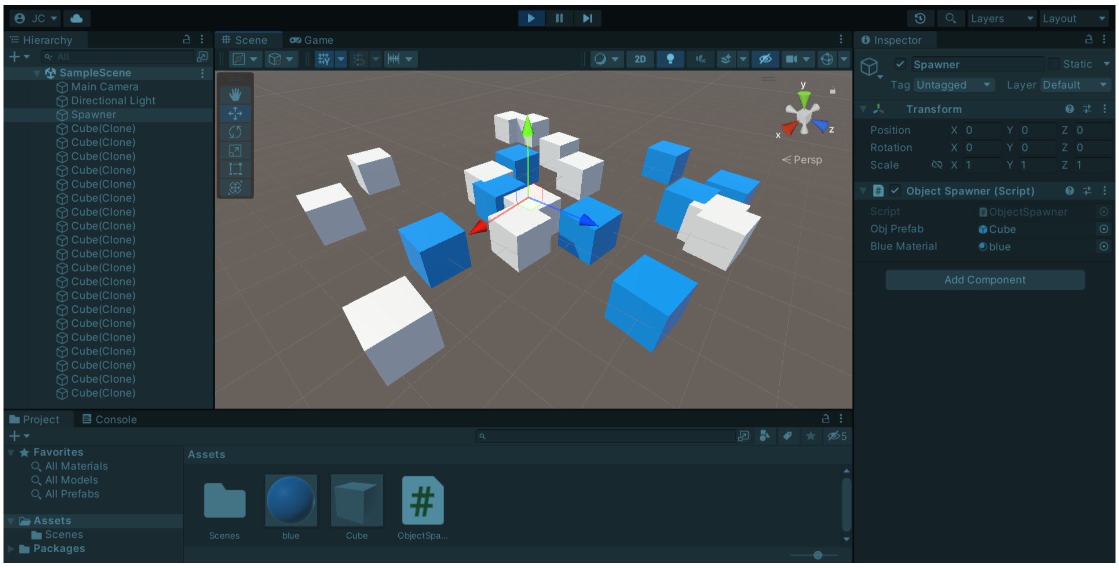
```
public class ObjectSpawner : MonoBehaviour {
    [SerializeField] private GameObject objPrefab;
    [SerializeField] private Material blueMaterial;
    private List<GameObject> instances;
    private float speed = 100f;

    private void Start() {
        instances = new List<GameObject>();
        for (var i = 0; i < 20; i++) {
            var randomPos = new Vector3(Random.Range(-5f, 5f), 0f,
                                        Random.Range(-5f, 5f));
            var instance = Instantiate(objPrefab, randomPos,
                                       Quaternion.identity);
            instances.Add(instance);
            StartCoroutine(ChangeMaterial(instance));
        }
    }

    private void Update() {
        foreach (var instance in instances)
            instance.transform.Rotate(
                                speed*Vector3.up*Time.deltaTime);
    }

    private IEnumerator ChangeMaterial(GameObject obj) {
        yield return new WaitForSeconds(Random.Range(0f, 10f));
        var renderer = obj.GetComponent<Renderer>();
        if (renderer != null)
            renderer.material = blueMaterial;
    }
}
```

An example of such a user-created script is in Code listing 2.6, where multiple cube objects are spawned into the scene at the application's start. For each instance of the object, a coroutine is started, which waits for a random amount of time before changing the object's material. The `Update` hook, executed every frame, rotates each object instance. A screenshot from running this script is shown in Figure 2.13. The picture also shows how the script is attached to an object in the scene called `Spawner` and how some properties are exposed and assigned through the Inspector window.

2.3.3 Plans and licensing

Unity offers a range of plans from Personal, which is free, to Enterprise, which is best suited for larger organizations [56]. The licensing is pretty lenient, especially for research purposes, which usually do not generate considerable profits. Unless the revenue or funding of a project exceeds 100,000 USD in the last 12 months, the free Personal plan for Unity is sufficient. Otherwise, one of the paid plans will be needed based on the revenue. Moreover, students and educators also have free access to the Pro version of the Unity Editor.

Naturally, different plans provide different features to the user. Most notable in the context of this project are technical support, better cloud diagnostics and collaboration tools, and Havok Physics for Unity.



Spawner Static
Tag Untagged Layer Default
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Object Spawner (Script)
Script ObjectSpawner
Obj Prefab Cube
Blue Material blue
Add Component

Figure 2.13 MonoBehavior and coroutine example in Unity

2.3.4 Unity for robotics

Using game engines in robotic applications is not uncommon [57, 58, 59, 60, 61], especially for simulation and visualization purposes. For some particular use cases, it might be impractical or even impossible to use conventional tools used in robotics. For example, applications using virtual reality [62, 63, 64] or mixed and augmented reality [65, 66] in the field of *human-robot interaction (HRI)*, or even for immersive teleoperation, are great examples where game engines are preferred to other simulators used in the industry. Amongst the game engines used in these robotic applications, Unity is a common choice because of its maturity, ease of use, gentle learning curve, and massive community support.

The main advantage of using game engines in robotics, specifically the Unity game engine, is that they are far more versatile than other specialized tools. As mentioned in the previous sections, Unity supports deployment to almost any platform and easily accessible VR and AR development. Rigging and animation can be used to animate robotic hardware, humans, or other objects in the scene to create complex scenarios for robotic simulation. Procedural generation of scenes and tasks can be implemented to train models for robot control, and the user interface can be tailored to a specific use case of the application. Additionally, modern game engines are constantly improving to keep up with the advancements and trends in real-time rendering and physics simulation. As game developers often need to build extensive scenes, game engines provide various tools and techniques for optimization so that the games can run smoothly and be less demanding on the hardware. All of this can be beneficial for robotic applications. Naturally, in some cases, a game engine might not be a suitable tool for a robotic application, as there can also be some downsides to using game engines. As the gaming industry has different needs than robotics, the tools for game development, including game engines, have their specific focus. One example that is especially relevant in robotics is physics simulation. Physics engines used in games were known for using algorithms that favor speed over physical accuracy or take advantage of various shortcuts as the physics in games usually does not need to depict reality accurately, and maximizing the frame rate is always desired. However, modern physics engines are so versatile and configurable that this is becoming a problem of the past, as is shortly dissected in Section 1.5. They can be fine-tuned for specific needs by advanced users. Unless the application needs a hyper-realistic physics simulation and the computational time is not limited, then using a game engine should not pose a significant disadvantage over conventional robotic simulators.

2.3.4.1 Unity-ROS integration

As ROS is becoming a standard in the robotics community, the number of robotic projects using Unity and integrating with the ROS framework is also increasing. All of these projects had to solve the problem of Unity-ROS integration and thus overcome one obstacle: ROS is built on message passing and Unity on frame-based execution. Over the years, there were several attempts to integrate Unity and ROS, and most of the solutions were developed for the specific needs of the project in which they were used. The overall general approach in these solutions was similar. On the ROS side, there would be some library converting ROS messages sent to some particular topics to other formats like JSON or BSON and sending them over some network connection. In the opposite direction, the library would receive JSON data, convert them to ROS messages and publish them to desired ROS topics. The `rosbridge`[11] library, in particular, was adapted for this in several projects using the WebSocket protocol for communication and was gradually transformed into a collection of packages called *RosBridgeSuite* which is still available and used today [68]. On the Unity side, a different library would set up a connection to ROS and define some Unity publishers and subscribers. In the main rendering thread, it would deserialize received JSON data into appropriate structures that could be used by other scripts running in the Unity project. In [69], the authors provide a short and comprehensive overview of the attempts to link Unity and ROS before 2018.

Although the general approach to Unity-ROS integration has mostly stayed the same, these solutions are generally not used today for various reasons. Many solutions created during that period were either abandoned or could not keep up with the ROS development and were not migrated to ROS 2. The exception is the rosbridge library, which is still used for communication outside of the ROS framework in general. However, the Unity side was not as mature, and the existing solutions were not generic enough to be used in multiple different robotic applications in Unity. So in 2018, Siemens started developing a set of open-source libraries and tools called ROS# (*RosSharp*) that also utilized the rosbridge library and could be used for communication between ROS and Unity. The ROS# library is still available and used today.

2.3.4.2 RosSharp

ROS# (RosSharp) [70] is a set of open-source libraries and tools developed by Siemens that can be used for Unity-ROS communication based on the RosBridgeSuite [68] package. It is also directly available in the Unity Asset Store [71] and can be imported to any Unity project. The ROS# project is split into generic interfaces for easy reuse in other non-Unity .NET applications and Unity-specific extensions of these interfaces used in Unity. The main library of ROS# is the RosBridgeClient which, together with the RosBridgeSuite, provides communication between ROS and Unity via WebSocket connections transferring JSON and BSON data. This is the only supported way of communication in ROS#, but additional protocols could be potentially added by implementing a provided protocol interface. Using ROS#, it is possible to create scripts in the Unity application that behave like publishers, subscribers, services, or even action servers. Because of the use of WebSocket protocol and RosBridgeSuite, the RosBridgeClient library has several external dependencies.

Figure 2.14 depicts the setup for the Unity-ROS integration using this solution. The ROS environment on the left side of the diagram illustrates some ROS applications running several nodes that communicate through topics[12]. The RosBridgeSuite also runs on the ROS side. Mainly two nodes are started by launching the RosBridgeSuite. The node called rosapi is responsible for

[11] Not to be mistaken with *ros1_bridge* [67], which is a package that enables message exchange between ROS 1 and ROS 2 over a network bridge.

[12] As is standard practice in ROS, nodes are depicted as ovals, topics as rectangles, and arrows indicate how nodes subscribe or publish messages to topics.

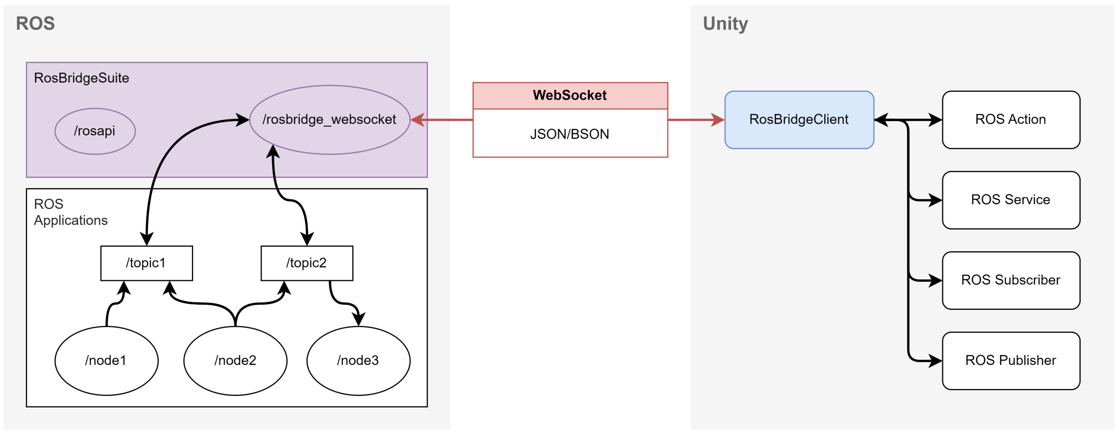


Figure 2.14 Unity-ROS integration using *ROS Sharp* and *RosBridgeSuite*

getting ROS meta-information about running nodes, available topics and parameters, and more. The node `rosbridge_websocket` handles the message conversion from and to JSON/BSON format and handles WebSocket connections. On the right side of the diagram, the Unity application is depicted. An instance of the RosBridgeClient library is running in the Unity application responsible for communicating with the `rosbridge_websocket` via a WebSocket connection and converting JSON messages to and from appropriate C# interfaces. Other scripts in the application can work with these deserialized messages but also use the RosBridgeClient to register topic subscriptions or publish messages. From the design of ROS, the ROS applications can be distributed over multiple machines, and so can the Unity application run on a different machine than ROS as long as an appropriate network connection can be achieved for communication.

Additionally, there are two other valuable libraries in ROS#. The *MessageGeneration* library generates C# source code for ROS interfaces, as it is commonly done for C++ and Python in ROS. This allows the use of these interfaces from Unity scripts. Furthermore, the *UrdfImporter* library is a URDF file parser that allows importing robots with their meshes to Unity from their robot description files in the URDF format.

2.3.4.3 Unity Robotics Hub

Around 2020, Unity Technologies started its own endeavors to make Unity a better platform for robotic simulation. They started with significant upgrades to the underlying physics engine by switching to PhysX 4.1, which is more capable of simulating real hardware and modeling kinematic chains. This upgrade allowed an addition of a new solver, called *Temporal Gauss-Seidel*, which better mitigates inaccuracies in joint simulation compared to the default Projected Gaus-Seidel solver. Another addition was physical articulations which can be used to model open kinematic chains and use joint coordinate representation, which also helps with the accuracy of the physics simulation (see Section 1.5). Physical articulations in Unity are constructed as a chain of objects with *Articulation Body* components attached to them. This component has replaced the previous way of modeling articulation chains as a combination of Rigid Body and Joint components.

Following the Unity game engine changes that improved robot simulation, Unity Technologies started working on the official Unity-ROS integration by creating a fork of the ROS# project into two separate packages - *ROS TCP Connector* [72] and *URDF Importer* [73]. Since then, these two packages are being developed in parallel with the ROS# project and are in the preview stage. Around the same time, *Unity Robotics Hub* [74] was created, a central GitHub repository for Unity's documentation, demo projects, resources, and tools for robotic simulation in Unity, which is under active development. Naturally, these robotics packages can also be used with

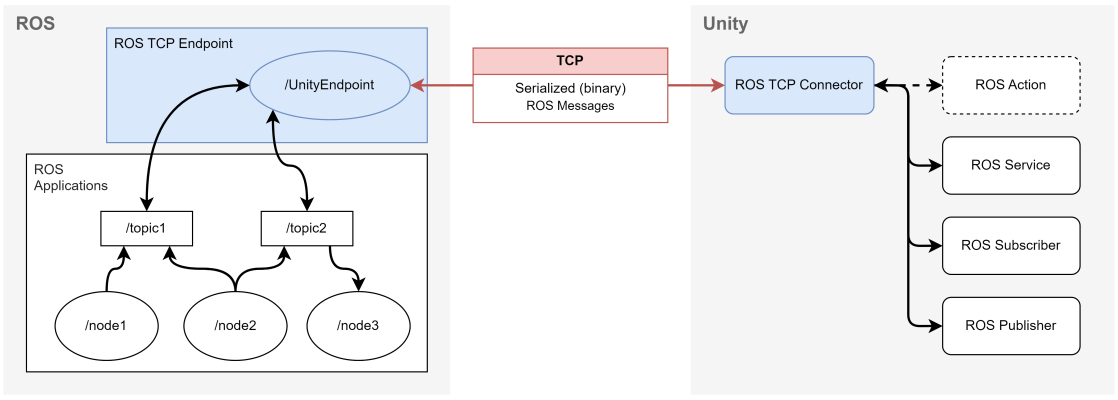


Figure 2.15 Unity-ROS integration using the *ROS TCP Endpoint* provided by *Unity Robotics Hub*

other assets and packages. The Unity team demonstrated this in a demo project combining robotics and the Perception package for training and deploying a deep learning model for object pose estimation in a pick-and-place scenario with a robotic arm.

Unity has made several changes and improvements to the Unity-ROS integration. As is shown in Figure 2.15, the general concept of the integration setup stayed the same. The main difference is that instead of RosBridgeSuite, direct TCP-based binary data communication is established. RosBridgeClient from ROS# has been replaced with the *ROS TCP Connector* package, which serializes and deserializes messages as ROS would internally do and handles both sending and receiving data. The original message generation functionality is also present but has been extended, so the generated C# classes from ROS interfaces now also include methods responsible for their serialization and deserialization. On the ROS side, the RosBridgeSuite has been replaced with a custom ROS package called *ROS TCP Endpoint* [75], created by Unity. This package creates a TCP endpoint that runs as a ROS node to accept and send messages from Unity as it works directly with the ROS TCP Connector. These changes increase the communication speed between Unity and ROS, which is especially beneficial when sending large messages like image data from cameras. As this integration works with both ROS 1 and ROS 2, the ROS TCP Endpoint package is available for both versions of ROS separately. ROS TCP Connector can seamlessly switch between ROS versions in its settings. As with the RosBridgeClient from ROS#, other scripts in the Unity scene can use ROS TCP Connector to publish messages or register subscriptions. With one exception that the support of ROS actions is currently limited.

The *URDF Importer* has been moved to a separate Unity package and incorporated a significant change. The robot from a parsed URDF file is imported into Unity using articulation bodies, unlike the URDFImporter from ROS#, which still uses standard Joint and Rigid Body components. This is a significant advantage for modeling open kinematic chains like robotic arms. On the other hand, the downside of articulation bodies is that they currently do not support cycles, so it is impossible to model closed kinematic chains. For this purpose, the URDFImporter from ROS# would have to be used.

In addition to these core packages adopted from the original ROS# project, the Unity team working on the Unity Robotics Hub will most likely provide other tools and improvements for robot simulation in Unity. The latest addition to the available tools is the *Unity Robotics Visualization* package, which can track and visualize incoming and outgoing ROS messages. A default visualization of standard interfaces is already provided, but also a set of APIs that can be used to create custom visualizations. Messages can be tracked in a user interface on the screen, but the strength of the package is to visualize more complex data directly in the scene, like point clouds and images. Creating such visualizations of ROS messages directly in the Unity simulation can replace the need to use RViz.

2.3.4.4 Unity editor as a simulator

The Unity game engine can be used for robotic simulation in two ways. The first approach is to develop a custom simulator that can be built and distributed using Unity as an underlying engine. The other approach is using the Unity editor for the simulation directly. Both approaches have their advantages and disadvantages. However, in this use case, unlike in game development, the user has the flexibility to combine or switch between both approaches to get the best of both worlds.

Developing a standalone simulator can be a complex and time-consuming endeavor. All the features required for the specific application must be developed and tested. So, for example, if the user needs a simulator similar to Gazebo, they need to develop all the scene controls like camera movement, management of objects in the scene, simulation pausing or restarting, and more. All these features may require unique controls or user interfaces. However, the benefit is that the simulator can be as lightweight or as complex as the user desires. Moreover, the final build runs more efficiently than running it from the editor because it does not need to run some of the overhead required for the editor.

Unity editor allows running the developed application directly from the editor in play mode. When the play mode is active, changes can be made to the scene or the exposed properties of all the objects in the scene but will be discarded when the play mode is terminated. This feature is used to test the application or to configure various properties interactively. So a robot simulation can be run or stopped anytime in the play mode without building it. The significant difference is that all the editor's functionalities and tools are available while the application runs. So the scene can be modified in the scene view or the hierarchy window as the application is running, and the profiler and statistics are accessible. The editor's user interface for scene management is already there, assets can be easily dragged into the scene, and even the properties of the robotic arm can be changed interactively. The downside is that the simulator may not need some of the tools and processes the editor is running, creating unnecessary overhead. Also, the editor may be more challenging for inexperienced users than a specifically designed user interface for the target simulation.

Combining these two approaches can be a great way to use Unity for robot simulation. Utilizing the available tools and the complexity of the Unity editor to develop scenes and custom functionalities and run various test scenarios can be very efficient. Specific scenes, tests, and scenarios can be individually built for specific purposes, like benchmarks or demonstrations, to get the most performance and precision from the available hardware.

2.3.5 Alternatives

In terms of game development, there are many alternatives to the Unity game engine. Most of them serve a specific purpose or support the development of a narrow niche of game genres. For example, *GameMaker* [76] for 2D games, *Ren'Py* [77] for visual novels, *RPG Maker* [78] for top-down pixel art role-playing games, and *Adventure Game Studio* [79] for point-and-click adventures, to name a few. The most direct competitor to Unity in terms of game engines also used as a real-time 3D development platform in other industries is the *Unreal Engine* [80]. Another aspiring game engine that became an excellent alternative to Unity during the last few years is the *Godot Engine* [81]. However, it is yet to be widely used in industries other than game development.

2.3.5.1 Unreal Engine

Unreal Engine [80] is a powerful game engine and real-time 3D creation tool developed by Epic Games with a long history since 1998. It is written in and also supports development in C++ programming language, but also supports visual scripting. As well as Unity, other industries have also adopted it, but even though their use cases overlap, they are different. Unreal Engine

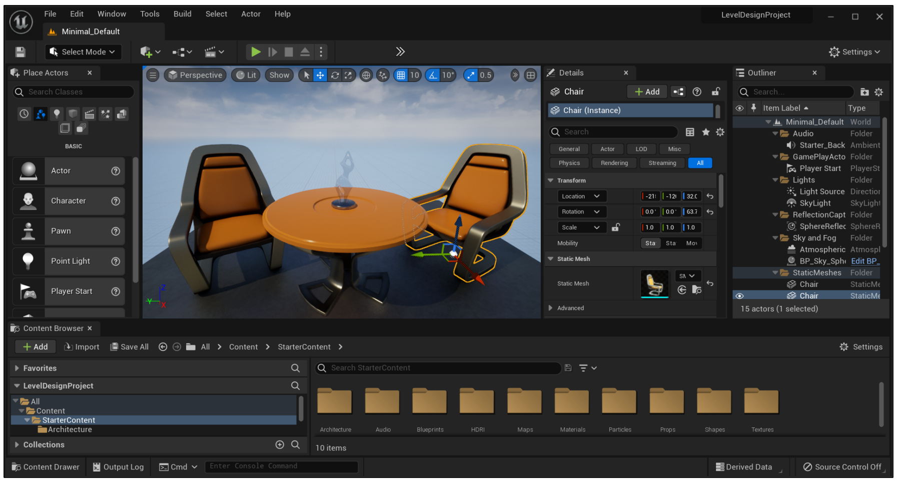


Figure 2.16 Screenshot of the *Unreal Engine* user interface in the minimal default interface layout (image from [82])

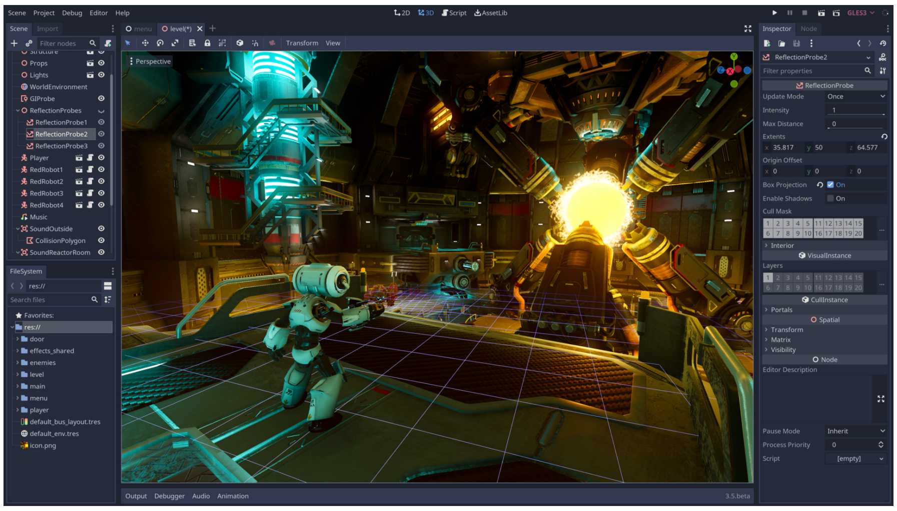
is more suited and praised for its capabilities to create projects that look hyper-realistic, and it is used significantly more by professional game studios compared to Unity. With its compatibility with other professional tools used in the gaming industry and the latest advancements in creating high-fidelity procedural assets and scenes, Unreal Engine is an excellent replacement for custom game engines created by professional game studios specifically for their projects. In these regards, Unreal Engine is more potent than Unity. On the other hand, when it comes to 2D games, even though Unreal Engine supports their development, Unity is a better fit as it has more and better 2D tools.

Unlike Unity, Unreal Engine is known to have a very steep learning curve and a more complicated user interface, which can be seen in Figure 2.16. Therefore Unity is more used among starting independent game developers, small studios, or research teams. Another interesting difference is in licensing, where the commercial use of Unreal Engine is based on a royalty model.

As for its use in robotics, Unreal Engine has also been utilized [57, 83] multiple times, but not as predominantly as Unity and even less often in combination with ROS [84]. Although the applications developed in Unreal Engine are programmed in C++, integration with ROS poses several challenges [85] and is far from seamless. For example, to ensure high performance, Unreal Engine has a custom and very strict C++ coding standard, and many features widely used in the ROS framework are unavailable. This makes it very difficult to integrate ROS and Unreal Engine directly, so usually, bridging techniques were used with libraries like rosbridge. Unreal Engine also uses the *Nvidia PhysX* engine to simulate physics.

2.3.5.2 Godot Engine

Godot Engine [81] is a relatively young cross-platform 2D and 3D game engine initially released in 2014. However, it is an interesting game engine to consider, especially in the future. It is a free and open-source game engine developed mainly by its growing community. Over its relatively short lifetime, it has already become one of the most relevant competitors to Unity. The project even received several grants and donations from companies like Microsoft, Facebook, and Epic Games to develop or improve various game engine features. The scripting in Godot can be done in C++, C#, or GDScript, a scripting language syntactically similar to Python. Additional community-supported languages like Rust, Haskell, Swift, and more can be used. However, unlike Unity and Unreal Engine, Godot uses the Bullet physics engine for 3D physics simulation, which is also free and open-source. Godot is also great for 2D games, with a dedicated 2D rendering



ReflectionProbe2

Update Mode Once
Intensity 1
Max Distance 0
Extents x 35.817 y 50 z 64.577
Origin Offset x 0 y 0 z 0
Box Projection On
Enable Shadows On
Cull Mask
Layers
Editor Description
Pause Mode Inherit
Process Priority 0
Script [empty]

Figure 2.17 Screenshot of the *Godot Engine* user interface in the default interface layout (image from [86])

engine. Figure 2.17 shows a screenshot from the editor.

Godot Engine has not yet been used in robotics research, but there are already working attempts to integrate ROS with this engine [87, 88]. Especially this project [88], which integrates Godot and ROS 2.

Chapter 3

Robotic arm RR1

"I think having a small-scale Industry 4.0 on the table could be great for research and testing."
– Pavel Surynek [89]

This chapter presents the faculty-developed desktop robotic arm *Real Robot One (RR1)* [89, 90]. In the first section, the motivation behind the development of this custom manipulator is explained. Later sections describe the robotic arm in more detail and provide technical specifications.

The robotic arm is currently in active development, with the first functional prototype[1] already built. The second prototype is being designed and constructed at the time of writing this thesis and will bring improvements in many aspects. Although the following chapters work with the models of the second prototype that are available from the RR1 GitHub repository [90], this chapter mainly describes the first prototype. Nevertheless, Section 3.3 will shortly disclose some changes the second prototype will adapt.

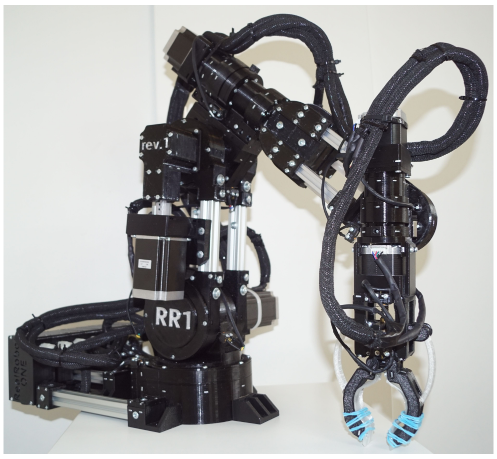
3.1 Motivation

The motivation behind the development of RR1 is to have a desktop robotic arm that is both similar to standard industrial manipulators [91, 92] and can be effectively produced in large numbers. Therefore the overall design of the robotic system aims to lower the cost of production and make it possible to construct at the faculty or even at home while maintaining the standard joint layout of robotic arms used in the industry. The smaller form factor of the robot decreases the build cost but also makes it safer to operate compared to large systems used in production lines. Nevertheless, RR1 is much bigger and more capable than some toy robotic arms used in academia but could still be categorized as a desktop-size manipulator.

The main goal for RR1 is to be usable in both research and academia but also have practical applications. The arm has enormous research potential in the field of motion planning, especially in the multi-robot setting, where many robotic manipulators cooperate precisely and without collisions. This topic has been extensively covered for mobile robots [93] but poses a more significant challenge for multiple robotic arms [94, 95] as the configuration space usually has many more dimensions.

Besides the research, RR1 could be used for academic demonstrations of AI and planning algorithms or student projects. It could also have additional practical applications as a laboratory robotic arm and enable remote manipulation.

[1]The first prototype is called "revision 1" or "rev. 1" in short. The second prototype is referred to as "rev. 2".


(a) Robotic arm Real Robot One (RR1)

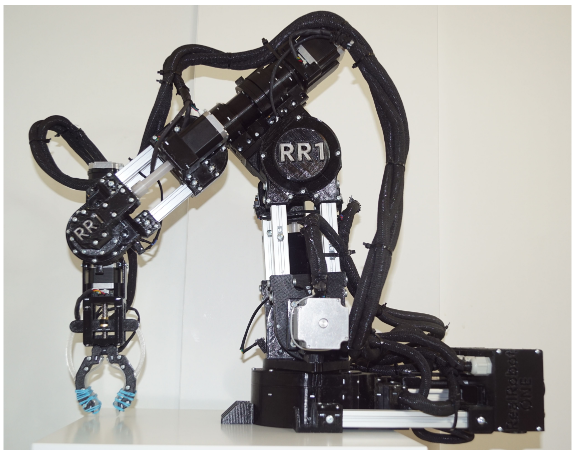

(b) Robotic arm RR1 from the profile

Figure 3.1 First functional prototype of RR1 robotic arm, called "rev. 1"

3.2 RR1 in detail

The robotic system of this manipulator consists of two parts: the robotic arm RR1 itself and a separate control computer called *Real Box One (RB1)*. The first prototype of RR1 can be seen in Figure 3.1, and the control computer in Figure 3.3. For a quick summary of the technical specification of this system, documented in the following sections, refer to Table 3.1.

Table 3.1 Technical specification of the RR1 robotic system [89]

Mass	14kg (RR1) + 8kg (RB1)
Reach	Approximately 80cm
Repeatability	TBD[2]
Payload	1kg tested
Actuators	4x NEMA 23 + 3x NEMA 17 stepper motors
Reducers	3D printed custom planetary gear reducers
Electronics	Arduino Due
Stepper drivers	7x DM556

3.2.1 Overview

RR1 is a six DoF desktop robotic arm with a gripper end-effector (see Section 3.2.3). It is actuated by stepper motors and should be capable of lifting two kilograms of payload[3]. One feature distinguishing this robotic arm from similar projects [96, 97, 98, 99] is that RR1 does not use belts to transfer power to the joints. This design choice simplifies certain aspects of the robot construction. Instead, all the torque transmission in RR1 is done via custom 3D-printed gear reducers.

The initial cost of constructing the first prototype was approximately 4,500 USD, including all the materials and electronics but also the cost of a new 3D printer and other tools. The

[2]Repeatability is in the process of experimental evaluation.

[3]This is only a theoretical payload limit. So far, the first prototype was tested with a 1kg payload, which it had no problem lifting.

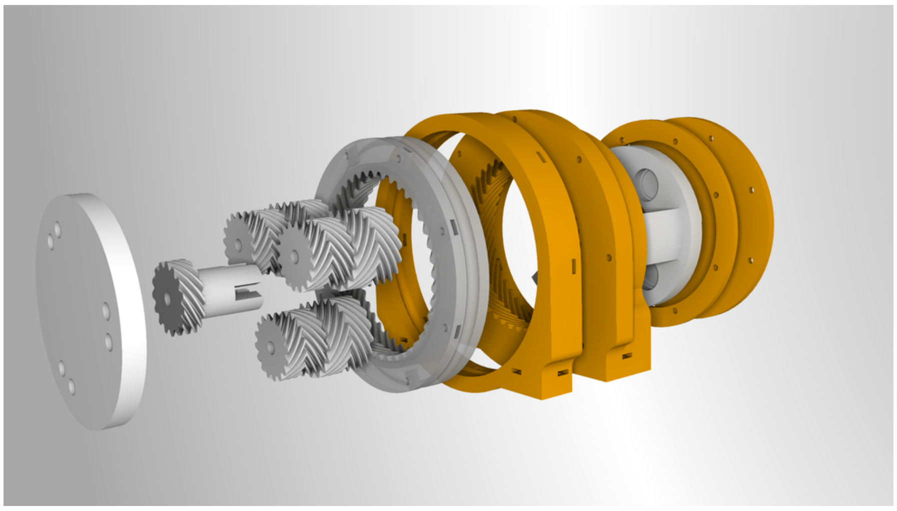


Figure 3.2 Model of split-ring planetary gearset used in the main lower joint of RR1 (model for the second prototype)

building cost of the consequent versions of the robotic arm will be lower as they will not require some of the aforementioned initial investments.

Several desktop robotic manipulators are related to RR1 with their size, design, construction process, or purpose. Namely the commercially available Niryo One [99] or its successor Niryo Ned2, open source 3D-printed robotic arm BCN3D Moveo [96], and the AR3 and AR4 [97, 98] manipulators.

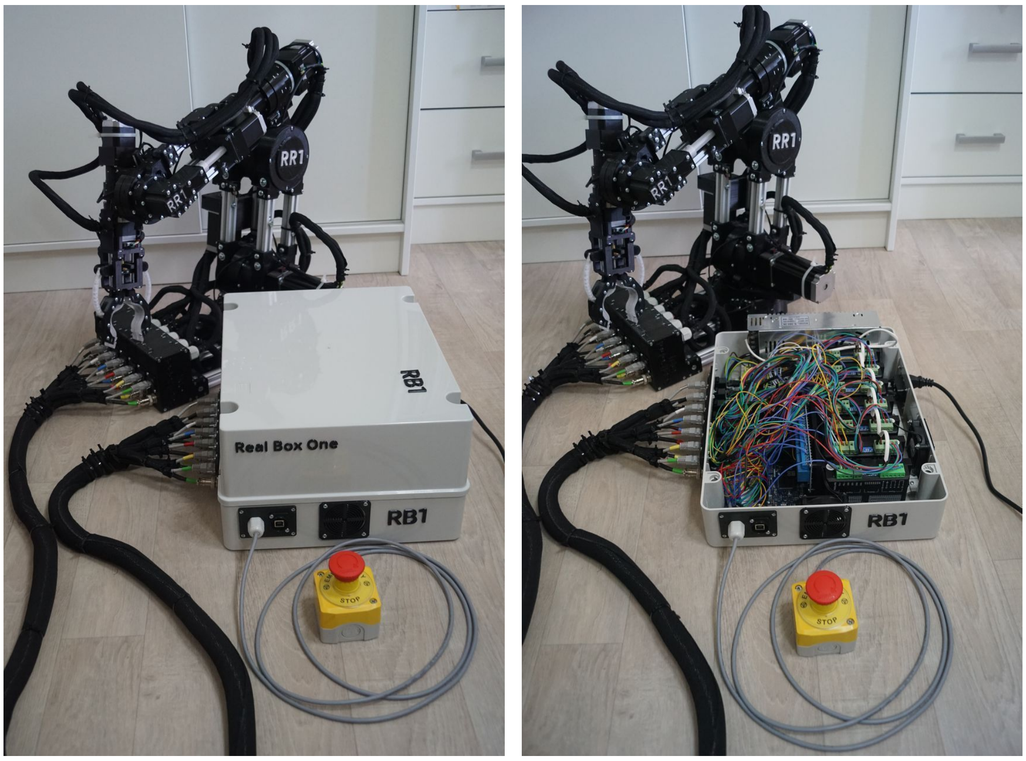
3.2.2 Joints and actuators

The robotic arm RR1 has six main joints that provide the manipulator with six degrees of freedom and one extra joint operating the end-effector at the end of the arm. The number of joints and their layout in the manipulator is standard in the robotic industry.

Each of the six main joints is actuated by a stepper motor. Namely, four motors NEMA 23 [100] and two smaller ones, NEMA 17 [101], are used. An accurate encoder is connected to every motor of the robot, so the joint angles are known at any given moment, creating an entirely closed-loop system. The torque transmission of every joint is done via a custom-built 3D-printed planetary gear reducer which can be seen in Figure 3.2[4]. Specifically, it is a split-ring compound planetary gear reducer consisting of three herringbone gears as planets and one middle gear connected to the stepper motor axle. The rotating ring that moves the connected link is fixed using bearing balls. Herringbone gears have the advantage of smoother and more precise rotation. At the same time, the split-ring design makes it possible to achieve significantly higher gear ratios than a standard planetary gear reducer. There are six gearboxes of various sizes, one for each joint. It consists of three herringbone gears as planets and one middle gear connected to the stepper motor axle. The rotating ring that moves the connected link is fixed using bearing balls. Herringbone gears [102] have the advantage of smoother and more precise rotation. There are six of these gearboxes of various sizes, one for each of the joints.

For example, the lower main joint of the manipulator uses the largest NEMA 23 stepper motor generating torque of 3Nm, which is connected to a planetary reducer with a gear ratio of 1:40. This should, in theory, provide a torque of 120Nm for the main joint.

[4]Note that the figure is the model of 3D-printed parts for the gearbox and does not contain bearing balls and bolts used in the assembly.



(a) Control computer Real Box One (RB1) **(b)** Interior of the control computer RB1

Figure 3.3 Control computer Real Box One (RB1) housing electronics needed to control RR1's actuators

3.2.3 End-effector

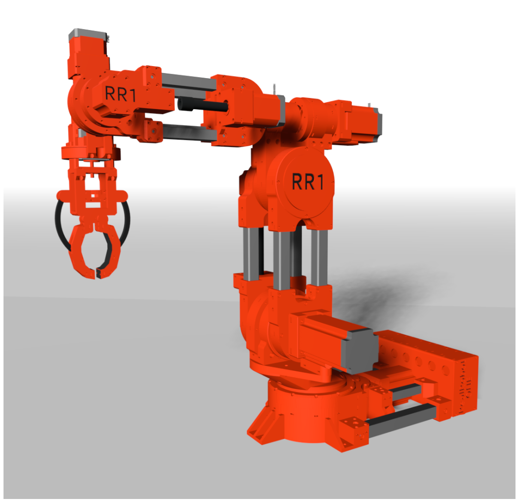
As shown in Figure 3.1, the end of the manipulator is equipped with a two-finger gripper end-effector, which can be used for object manipulation in the robot's workspace. It is an angular impactive gripper actuated by an additional NEMA 17 stepper motor.

The first prototype of RR1 showed some issues with the surface material of the gripper, causing smooth and heavier objects to slip out of the grip. The problem was temporarily mitigated with rubber bands, but it will be resolved in the next prototype of the robot (see Section 3.3).

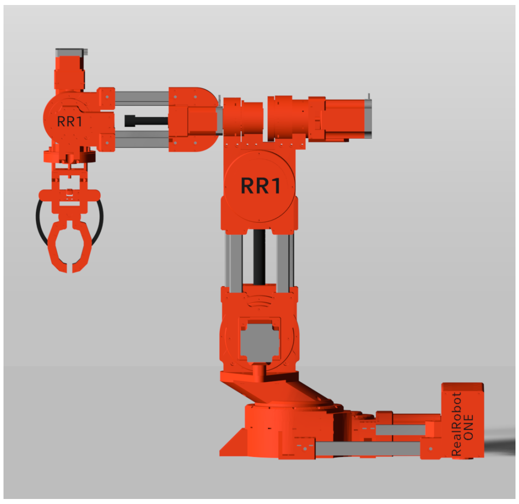
3.2.4 Control computer RB1

The robot's control computer RB1 supports modular design as it is separated from the main robotic arm. Having such a separate control unit allows many of the electronics to be offloaded from the body of RR1. However, the disadvantage of this approach is that the two modules of the robotic system have to be connected via potentially lengthy cables. These cables are subject to interference and have to be shielded and grounded. Both the exterior and interior of the RB1 can be observed in Figure 3.3. The RB1 is directly connected to the main robotic arm and houses electronics that control individual joint actuators. These electronics include *Arduino Due* [103], responsible for the general electronic control of all the motors, and seven 2-phase digital stepper drivers *DM556T* [104], each directly connected to one of the seven stepper motors in the arm.

It is important to note that this control computer will not perform any motion planning as the Arduino Due would not have the computational capacity to run complex planning algorithms.



(a) Second prototype of RR1



(b) Orthographic side view render of RR1 “rev. 2”

Figure 3.4 Render of the RR1 “rev. 2” in orange color

This process will be done on a separate machine, which will communicate with the Arduino Due in RB1 and send commands or planned trajectories to it. This separate machine can be a standard laptop or a computer connected directly to the Arduino via a serial connection.

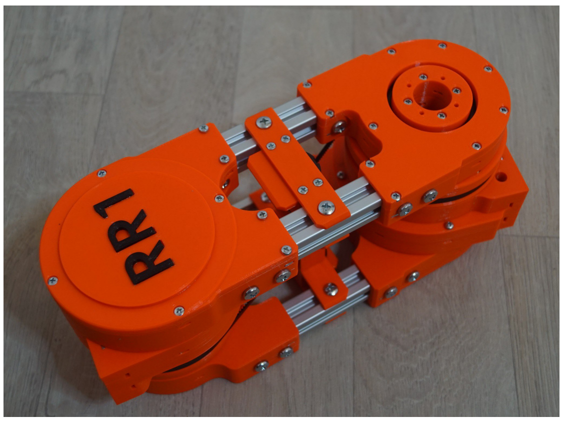
3.3 Second prototype

The second prototype of RR1, called “rev. 2”, will improve several aspects of its predecessor. The whole arm will be shifted a bit forward relative to the base, so it will become eccentric, as shown in Figure 3.4. This improves the arm’s reach and makes the robot’s workspace more natural. Such a configuration can often be seen in industrial robotic arms.

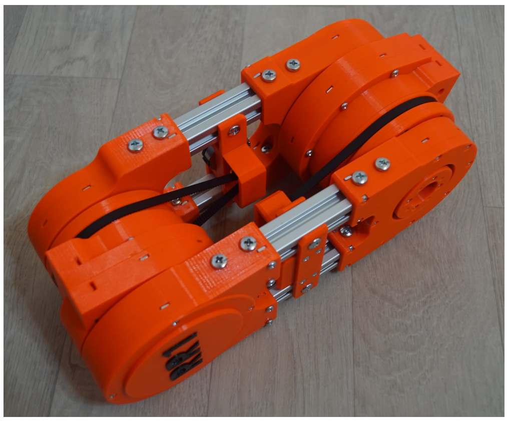
The planetary gearboxes of the joints overgone multiple improvements resulting in a significant reduction of backlash, which turned out to be a problem in the first prototype of the robot. Additional bearings were added to the rotating parts of the joints as one of these improvements, resulting in a slight increase in the robot’s weight. Another significant change in the joint design is that the encoders have been moved to the arm links and connected to the joints via belts, as shown in Figure 3.5b. Experiments with the first prototype showed that having the encoder on the motor shaft did not meet the accuracy expectations.

Additionally, more anti-slip components will be added to the gripper of the “rev. 2” model to mitigate the problems encountered with the first prototype, mentioned in Section 3.2.3.

There are already some plans for future iterations of this robotic system. For example, using cycloidal reducers instead of planetary ones is being considered. These reducers have higher torque capacity than the current planetary gearboxes but would increase the arm’s weight. Therefore some structural components might be machined out of metal for increased durability and payload limit.



(a) Upper arm link with two main joints



(b) Encoders moved from motor shafts to arm links

Figure 3.5 Upper arm link with two main joints for the second prototype of RR1

Chapter 4
Prototype rationale

This chapter serves as a bridge between the theoretical and practical parts of the thesis. It provides some rationale behind the developed prototype described in the next chapter, including its motivation, why certain technologies were chosen, and the expected results. The last section of this chapter explores previous work related to this thesis.

4.1 Prototype motivation

The prototype developed in this thesis should serve as a proof-of-concept for performant robotic simulation in Unity, mainly used for manipulators like RR1. The motivation is to use such a simulator to improve the development process and testing of the robotic arm itself and later in research of multi-robot motion planning, where several digital twins would operate simultaneously. Up to a hundred robots could be simulated in the planned multi-robot scenarios. As the design of RR1 should make this possible in the physical world in the future, the performance evaluation of Unity as a primary simulation tool for RR1 is highly prioritized. Additional goals are to use the simulation for academic demonstrations of core concepts of motion planning and robotics manipulators or to showcase the robotic arm at the faculty or various conventions. The potential use of virtual reality is also considered for several simulation scenarios. Because of these goals, the assessment of the performance and usability of Unity has to be enabled through the developed prototype.

As RR1 does not currently have any higher level of control or any robotic applications, a considerable portion of the prototype realization is the development of the ROS backend along with the robot description to enable simulation in ROS. The development of the ROS backend for RR1 has been approached in such a way that it is not designed only for simulation purposes but can be reused for the physical robot with minimal effort when its construction is finished.

4.2 Choice of technologies

ROS 2 was chosen as a robotic framework for RR1 as ROS has become a standard in the industry, and ROS 2 seems to be the future of robotic frameworks. Namely, the *Humble Hawksbill* distribution has been chosen (see Section 2.1). As it is the last LTS version of ROS 2 and the first ROS 2 distribution to have the standard five years long support period, it is an excellent choice for any new project in robotics. Although some libraries have yet to be migrated to ROS 2 or experience some issues after the migration, starting a project in ROS 1 is not recommended as it will be shortly discontinued. Specific ROS 2 components used in the prototype are mentioned in the next chapter.

For the simulation part, a game engine was used instead of a simulator specifically designed for robotic applications. The reason is that game engines should provide a wider range of functionalities compared to conventional simulators, including animations and easier scenario scripting, more flexible rendering pipelines, ability to build custom user interfaces and controls, and VR support. Moreover, a game engine will potentially provide more tools for optimization to increase performance in scenarios with big amounts of robots simulated simultaneously. Specifically, *Unity* game engine was chosen for the implementation. As explained in Sections 2.3.4 and 2.3.5. Of the two, Unity was chosen due to its reasonable learning curve, extensive community, and abundant learning materials. These attributes are important because multiple students or researchers might work on the simulator over time, and Unity is easier to get into without prior experience. With the Unity Robotics Hub, Unity also provides better support for robotic applications than the Unreal engine.

Gazebo was chosen for the performance comparison of the prototype implemented using Unity because it is a free and standard simulator choice in combination with ROS, and it is heavily used in the industry. Gazebo is also relatively easy to integrate within a robotic application as it works out-of-the-box with ROS. The second reason is that studies comparing various simulators commonly evaluate Gazebo, making the experiments and comparisons done in this thesis at least remotely extrapolatable to other simulation tools to compare to Unity. Even though Ignition Gazebo should soon become a more viable option than Gazebo after some initial issues are resolved, it is still immature and lacks comparison studies with other simulation tools.

4.3 Expectations

From the nature of the two compared platforms, it is evident that Unity will be more flexible and provide more functionalities than Gazebo due to the broader range of use cases in game development. However, it is also expected that for most of the features the platforms have in common, Unity will provide a better user experience or more functionality. For example, the efficient creation of complex scenes is more called for in game development than in robotic simulation. Although necessary in both fields, Unity will likely have better scene creation and management tools.

As for the performance, Unity is also expected to be more performant than Gazebo. Rendering and its performance are critical aspects of video games, so Unity provides more flexible and configurable rendering pipelines. Numerous optimization techniques can be used in game development, already incorporated and available in the Unity engine, so the performance can be improved when unacceptable.

4.4 Related work

Several comparison studies have been published comparing multiple simulation tools used in robotics. Some studies approached the comparison from a specific task or application perspective, while others performed general evaluations. Commonly, the underlying physics engines used in the simulators are evaluated, and the functionalities of different simulation applications are explored.

4.4.1 Simulator comparison for agricultural robotics

In [105] and [43], the authors considered numerous frameworks for simulation in agricultural robotics with ROS integration in mind. They compared a wide range of simulation platforms used in academia and industry but also reviewed customized frameworks specifically modified or created for the simulation of agricultural robots or farm machinery. The publications offer a short and comprehensive introduction to these simulation platforms. Gazebo, V-REP, and ARGoS[1] were additionally selected for in-depth feature and performance comparison.

The performance benchmarks were run in both the headless and GUI modes. The tests included 1, 5, 10, and 50 robots simulated in a small scene that contained only a 2D plane and a large scene with an imported model of an industrial building. Unfortunately, the simulated robots were not the same in each simulator, but the authors tried selecting robots of similar complexity from the models available in the libraries of the three simulators. Real-time factor, CPU, and RAM usage were selected metrics for the comparison.

Of the three simulators, ARGoS had the fewest features, the smallest model library, and no scene editor. However, it used the smallest number of resources, and in the simple scene, it could simulate a large number of robots more efficiently, making it a good choice for simple simulations of swarms of robots. The comparison of Gazebo and V-REP in these publications is more relevant for this work. Featurewise, V-REP was shown to be more sophisticated and flexible than Gazebo but also used more resources. However, Gazebo was not as limited as ARGoS. With the large scene, Gazebo performed the best regarding the real-time factor and consumed fewer resources than V-REP, which did not excel in any of the performance experiments. Moreover, the simulation of 50 robots in both scenes and the simulation of 10 robots in the large scene was not feasible in V-REP. This, however, could have resulted from a more complex robot model used in V-REP and low-end hardware used for the performance tests.

4.4.2 Gazebo, V-REP, and Unity quantitative study

From a thorough literature review, in [44], the authors concluded that the three most promising simulation tools for robotics are Gazebo, V-REP, and Unity. These three frameworks were then analyzed and compared in more detail, especially regarding their usability.

The experiment consisted of three experts, each with experience with only one of the three tools, and they had to complete a specified task in each simulator to asses their usability in a questionnaire. The assignment required them to install the software, create a simple scene with four walls, insert a mobile robot model inside the walls, and implement keyboard control of the robot.

The V-REP achieved an excellent score in the quantitative study, and both Gazebo and Unity achieved a decent score, with Unity graded the lowest, just behind Gazebo. However, a few discrepancies exist in their comparison of the task execution with the three tools. For example, V-REP was praised for its scripting feature and that ROS connection is not needed to control the mobile robot. A simple script was written in Lua programming language to control the robot. On the other hand, Gazebo does not support a scripting functionality, so the installation of ROS was required, which was described as challenging. Even though Unity also provides a scripting functionality, the choice of implementing the robot control directly in the scene was substantiated by the complexity of connecting Unity to ROS using the ROS# project. The authors found Unity to be intuitive regarding scene building and script implementation. The scene in both V-REP and Unity was created using resizeable blocks as walls, and in Gazebo, a build in wall builder seems to be utilized from the figures shown in the publication. Although the same mobile robot was used in the three simulators, the models were not identical. Especially in the case of Unity, the model is substantially different than in the other two simulators, and the authors were met with complications regarding its import to Unity and physical stability during the simulation.

[1]ARGoS is an open-source simulator explicitly developed for efficient real-time simulation of enormous swarms of robots.

4.4.3 Gazebo and Unity physics comparison

In thesis [106], the author analyzed the physics performance of the Unity simulation of a mobile robot. The results of the experiments were compared with real-world experiments and Gazebo simulations of the same scenarios provided by a third-party company. At the end of the thesis, a successful case study of a SLAM application for the used robot was performed in Unity. For both the experiments and the case study, ROS 1 was used, and the integration with Unity was done using the ROS# project. The experiments in Unity were performed in Windows, on top of which an additional virtual machine was running Linux and ROS. The Gazebo tests were performed on different hardware in Linux.

The results of the experiments showed that the overall behavior of Unity's physics simulation was satisfactory. Even though the simulations showed non-deterministic aspects, the general behavior of the simulated robot was more similar to the reality compared to Gazebo. Gazebo showed more idealistic behavior with almost no errors, which was impossible to achieve with the real robot experiments. Some configurable parameters that impacted the physics simulation in Unity were identified and experimented with during the investigations. However, it was shown that even small changes in the parameters of the physics engines, or changes in the robot model itself, can significantly impact the overall simulation results. Both underlying physics engines, PhysX and ODE, have many configurable parameters, making it very challenging to compare the physics performance of the two simulation tools directly.

Chapter 5

Prototype realization

This chapter describes the realization of the prototype in a systematic succession of steps needed to achieve the final proof of concept used for experimentation. The first section reviews the development process in a high-level overview, and the following sections explore specific steps in more depth. From the creation of the robot description for RR1 to the developed Unity components, the chapter covers common issues encountered in developing a robot simulation and design decisions made throughout the implementation process.

5.1 Overview

In reference to the concepts of digital twins explained in Section 1.4, the virtual entity for the second prototype of the RR1 robotic arm (revision 2) is created in this proof of concept. The entity is then simulated in a virtual environment inside Unity or Gazebo, which is one of the core processes of digital twins in robotics. However, because the second revision of RR1 still needs to be finished and constructed, twinning cannot occur. Therefore, to be precise, part of developing this proof of concept is creating a digital twin prototype. This digital twin prototype can be used for further development of the robotic hardware and jumpstart the preparations for future twinning and implementation of custom virtual processes specific to the digital twin of RR1.

5.1.1 Development steps

The developed prototype can be divided into three separate parts. First, the ROS backend must be developed for the RR1 robot, including the URDF robot description and control. This backend is mainly necessary for the prototype to simulate the robotic virtual entity in Gazebo and Unity. Nevertheless, the developed backend should serve as a good foundation for the future extension to control physical hardware as well.

The second step of development is the Gazebo simulation. This step is entangled with the first one, as additional launch files will be needed to start Gazebo simulations and spawn robot instances inside the simulation world, which will also be part of the ROS backend in a separate package.

Lastly, Unity is integrated with ROS using the official solution from Unity Robotics Hub, and a simulation prototype is developed. This includes custom controllers for the simulated arm and other Unity components used in the simulator.

5.1.2 Gripper simulation

From the prototype development's beginning, the gripper motion in the simulations was omitted. There were three reasons for this decision:

1. The separate models for the gripper fingers were unavailable in the RR1 GitHub repository [90], ruling out a robot description with functional gripper control. The repository contains models of completely assembled links of the articulation chain, and grippers were part of the last link.
2. Functional grippers in the simulation are unimportant for the performance comparison between Unity and Gazebo as they bring no additional value.
3. Gripper control and motion planning for robotic arms, including griping objects, is a separate topic outside of the scope of this thesis.

Nevertheless, almost at the end of the development process, the model of the last link was decomposed, and separate models of gripper fingers were made available in the repository. The gripper control was then added to the simulated robot to demonstrate the movement of the grippers. However, no additional simulated scenarios were developed to utilize the grippers in a pick-and-place application.

5.2 ROS package organization

Given the distributed architecture of ROS and how it encourages modularity, robotic applications are also usually distributed over several packages. The ROS backend for the RR1 manipulator also consists of several packages, as shown in Figure 5.1. The package organization follows some best practices and naming conventions used in ROS applications for robotic arms. This setup allows independent development of various application components that can be tested alongside already tested packages, increasing the whole application's maintainability in the process. The following subsections shortly describe all of the developed packages and the dependencies between them. Separate sections were dedicated to the main parts of the ROS backend, like robot description and control of the robotic arm. The Gazebo simulation is also part of the application as a separate package of the ROS backend.

The packages can be categorized into three levels, as also illustrated in Figure 5.1, with dependencies between the packages. The low-level packages contain mainly the robot description and definition of custom ROS interfaces for the application but do not provide any functionality. The mid-level depends on the low-level layer and provides some functionality, including algorithms, visualizations, and simulation in Gazebo. The high-level layer brings everything together, mainly with launch files orchestrating launches of multiple components at once for specific use cases.

5.2.1 rr1_bringup

The *bring-up package* is a high-level package used to bring together other packages in the application and house launch files with necessary configurations for orchestrating whole systems of ROS nodes. The launch files usually start nodes from other packages and provide the necessary parameters.

RR1's bring-up package acts as a single entry point for the end user, containing complete launch files that start subsets of the application modules. Namely, one launch file starts a Gazebo simulation of RR1, and another one is responsible for starting necessary nodes for simulation in Unity. An additional helpful launch file was created for RViz visualization of the robot and manual joint control, which is very useful when validating the robot description

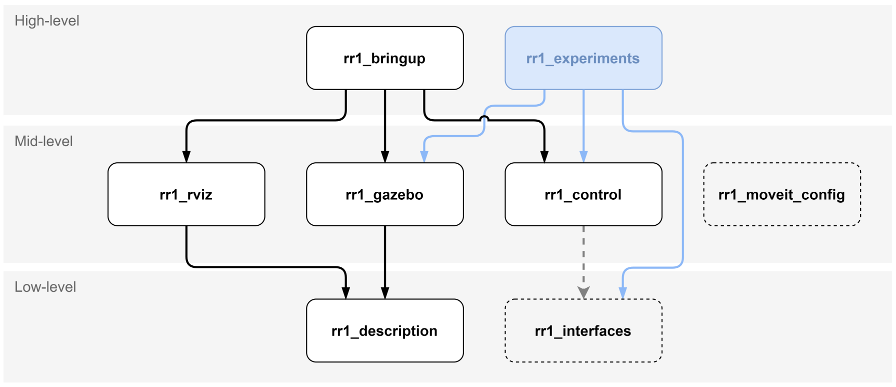


Figure 5.1 The RR1 ROS package structure

5.2.1.1 rr1_description

Robotic systems are commonly described in a separate low-level *description package* containing the URDF file and all the meshes and configurations related to the robot description. The package has no additional functionality and usually does not need to depend on any other package. Section 5.3, which covers the creation of the robot description for RR1, exclusively relates to this package.

5.2.2 rr1_rviz

The `rr1_rviz` package is a small package containing the launch file and configuration files for RViz. Other packages use this package to start RViz with saved configuration for visualization purposes during simulation. However, to visualize the robot itself, the path to the URDF file has to be provided, or an additional node publishing the description has to be started. RViz is an essential and straightforward tool for visualization during the development of the robot description.

5.2.3 rr1_gazebo

Robotic systems integrated with Gazebo often have a separate package like this one. Inside this package, specific configurations and launch files for starting Gazebo and spawning robot instances into the world are developed. SDF files with custom environments and models would also be present here, along with necessary mesh files.

5.2.4 rr1_control

The *control package* aggregates all control algorithms or custom controllers for the robotic system. It is also common to develop hardware interfaces inside this package. However, when the hardware interfaces are more complicated or the custom controllers are reusable for different robotic systems, the hardware interfaces can be moved into a dedicated package.

The `rr1_control` package currently contains implementations of ROS nodes and launch files that start them with desired parameters that emulate planning algorithms sending control trajectories and commands to the robotic arm. These nodes can be used to test the controllers of the arm and for experiments with scenarios simulating the robotic arm. Uniquely to the other packages, this one is hybrid, supporting both C++ and Python implementations. The reason for this is that it is expected that the hardware interfaces for the robotic arm, which are strictly implemented in C++, will be added to this package once the physical manipulator has been constructed. Other nodes and scripts are implemented in Python.

5.2.5 rr1_interfaces

Sometimes the robotic application needs custom messages, services, or actions in addition to the standard ones existing in ROS. These are usually defined in a separate package dedicated to custom interfaces. The custom interfaces are primarily needed for custom control, planning algorithms, or custom-made sensors used in the robotic hardware. Therefore, packages like `rr1_control` will often depend on this package.

In the current RR1 application, there is no particular need for custom interfaces. Nevertheless, the deployment infrastructure has been prepared in this package if the need arises in the future. The package was utilized to create custom messages used in the experiments (see Section 6.2.1).

5.2.6 rr1_moveit_config

Robotic arms are commonly integrated with the MoveIt framework. During the integration process using the setup assistant provided by MoveIt, a *configuration package* is created, commonly named with the suffix `_moveit_config`. Unfortunately, for RR1, this integration was created and successfully tested only in ROS 1, as the MoveIt 2 framework still has some issues. Nevertheless, the MoveIt integration is not essential for the performance tests and the developed prototype.

5.2.7 rr1_experiments

On top of the developed ROS backend for RR1, another package called `rr1_experiments` was created. It contains launch files, node implementations, and other Python scripts used in the experimentation phase. The package shows how the components are decoupled and can be reused in other packages. It also contains scripts for measuring the targeted metrics, measured data themselves, and Jupyter notebooks analyzing the measured data. The individual experiments and their results are described in Chapter 6.

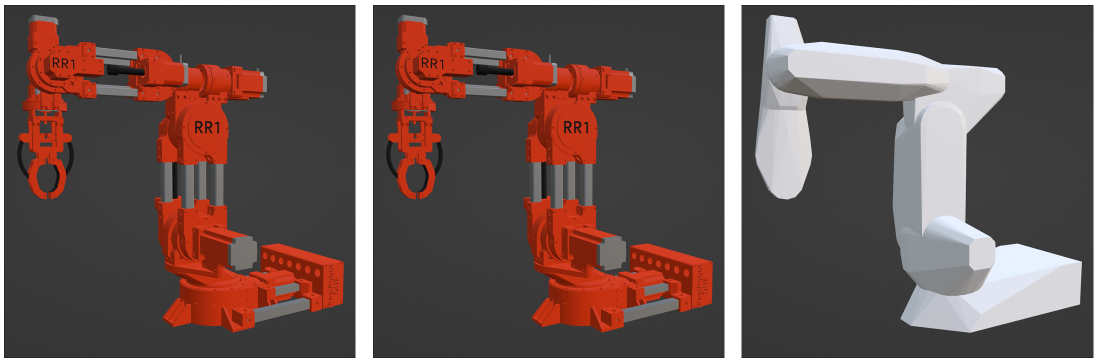
5.3 Robot description

To simulate a virtual entity of RR1, a URDF robot description has to be created. Before this step, the available models had to be assessed and modified for use in the URDF, as discussed in Section 5.3.1. After the meshes are prepared, the joints and visuals of the links can be described in the URDF. This allows visualization in RViz, but additional collision and inertial features of the links and kinematic properties of the joints have to be specified to enable simulation in Gazebo. Lastly, when the visual and physical properties of the robotic arm are described in the URDF, additional control specification is usually included in the robot description.

5.3.1 Model preparation

The *original mesh* for the RR1 robotic arm is available in the RR1 GitHub repository [90], where it is divided into separate STL files as individual links of the robot[1]. Although the STL format is supported in Gazebo and other visualization or simulation tools and can be directly used in the URDF format, two issues must be addressed: mesh complexity and model materials.

[1]As mentioned in Section 5.1.2, the gripper fingers were decoupled from the last link later in the prototype development.



(a) Original mesh (1,140,430 triangles) **(b)** Decimated mesh (456,167 triangles) **(c)** Collision mesh (1,100 triangles)

Figure 5.2 RR1 meshes used for the robot description

5.3.1.1 Mesh complexity

The original model for each of the links is a union of individual components designed in CAD. As the models were created for 3D printing, they are unnecessarily complex for simulation. The original mesh, shown in Figure 5.2a, has 1,140,430 triangles. Using such a detailed model does not bring any value to the simulation but creates an increased load on the hardware. As the main goal is a multi-robot simulation, the mesh should be simplified as much as possible.

Using Blender, the original mesh for each link was decimated by 60%. This factor was chosen as it simplified the mesh as much as possible without altering the visual of the robot significantly. The resulting mesh contains 456,167 triangles and is shown in Figure 5.2b. Further simplification should be possible if desired but would require a considerable amount of modeling and result in a simplified visual appearance of the robotic arm in simulation.

5.3.1.2 Textures and materials

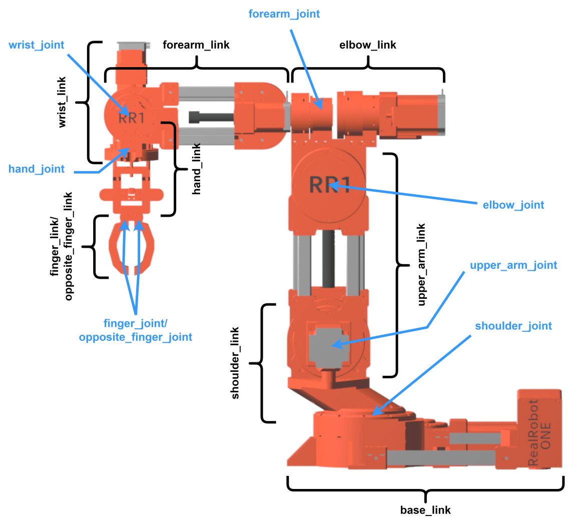
Using the STL file format is common in 3D printing and CAD. However, the STL format does not save the texture and material information of the mesh, which is not ideal for simulation purposes, where the model is used as a visual asset in the scene. It is possible and common to use the STL format to specify a mesh in URDF, and as discussed in Section 2.1.4.1, a simple material can be defined and assigned to it. However, this is not attainable for a complex model such as RR1 because the links would need to be broken down into individual components with different materials.

Several materials were created in Blender and assigned to various parts of the mesh to achieve the correct visual appearance of the robot shown in Figure 5.2. The models of the individual links were then exported in the DAE format, which saves the material together with the mesh and is supported by ROS, Gazebo, and Unity.

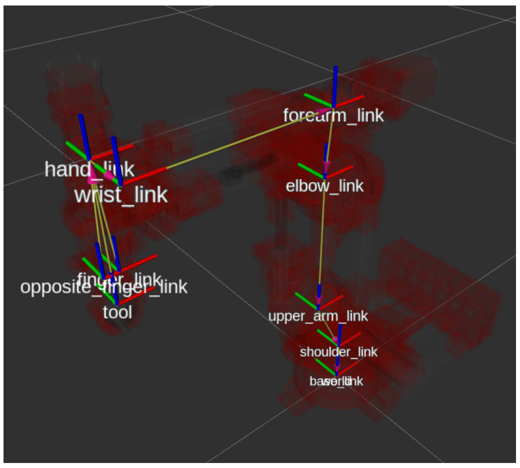
5.3.1.3 Collision mesh

Collision mesh is used by the physics engine to detect collisions between objects in the scene. This collision mesh must be defined for the simulated robot to interact with other physical objects in the simulated environment. As mentioned in Section 1.2.2.1, this collision model should be as simple as possible.

The collision mesh for RR1 was created in Blender for every link of the manipulator by creating a convex hull around the link and then decimating the mesh. The result is a collision model for RR1 with 1,100 triangles, shown in Figure 5.2c. As the colliders are only used in



(a) Links and joints of RR1



(b) RR1 transforms in RViz

Figure 5.3 Links and joints of RR1 and their transform origins visualized in RViz

collision detection, there is no need for materials or textures. Therefore, the collision mesh for each link was exported as an STL file and can be used in ROS, Gazebo, and Unity. Potentially, the collision mesh could be further optimized by dividing the links using basic 3D objects.

5.3.2 RR1 description

The robot description for RR1 robotic arm has to describe all of its nine links and eight joints shown in Figure 5.3a. The link description specifies the visual mesh, collision model, and inertial properties. The joints connect individual links and specify the transform positions of the child link, as shown in the visualization in Figure 5.3b. Additional kinematic properties and joint limits are also specified. Following subsections describe the URDF description of RR1 in more detail, starting with auxiliary macros and adherent Xacro files and then tying everything together with the base link description example. Further sections of the robot description regarding the robot control are later mentioned in Section 5.4.

5.3.2.1 Xacro structure

Due to the complexity of the robotic arm, *Xacro* format was utilized to make the robot description easier to develop and maintain. The URDF file was divided into several Xacro files, as shown in Figure 5.4, logically dividing various description components. The main Xacro file of the robot description is called rr1.urdf.xacro and includes several other Xacro files. Before the robot description is used in ROS, the Xacro command is used to generate a single URDF file that ROS supports.

5.3.2.2 Inertial macros

Inertial properties of a link include its mass and moment of inertia tensor. These properties are specified inside an XML block called `<inertial>`, which is part of the `<link>` block. Gazebo requires all links to have these properties specified. Otherwise, it would not spawn the model inside the scene.

The inertia tensor is provided by some modeling programs, or it can be experimentally measured on the physical hardware. As in this prototype, the links are sometimes substituted with

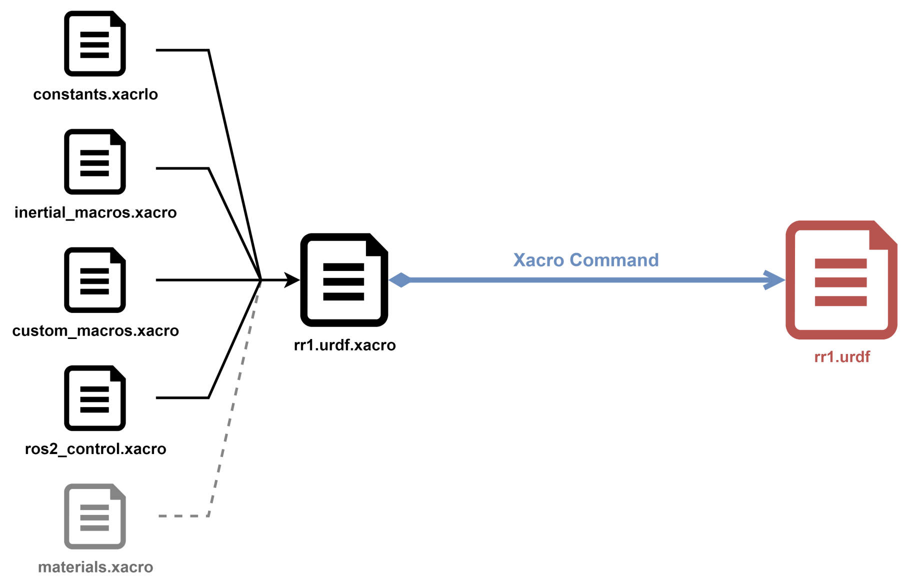


Figure 5.4 File structure of the robot description

Code listing 5.1 Inertial macro that returns a inertial block for specified box size

```
<xacro:macro name="inertial_box" params="mass␣x␣y␣z␣*origin">
  <inertial>
    <xacro:insert_block name="origin" />
    <mass value="${mass}" />
    <inertia
      ixx="${(1/12)␣*␣mass␣*␣(y*y+z*z)}" ixy="0.0" ixz="0.0"
      iyy="${(1/12)␣*␣mass␣*␣(x*x+z*z)}" iyz="0.0"
      izz="${(1/12)␣*␣mass␣*␣(x*x+y*y)}" />
  </inertial>
</xacro:macro>
```

basic geometry like cylinders or boxes, whose moment of inertia tensors are easy to compute from known formulas.

Instead of performing the calculations of inertia matrices on the side and then inserting the results into a URDF file, Xacro format makes it possible to create macros that perform the computations during the generation of the final URDF file. In the `inertial_macros.xacro` file, several inertial macros have been created for some basic geometric shapes. A macro computing inertial matrix for a cuboid is shown in Code listing 5.1. The macro takes the mass and dimensions of the cuboid as parameters, as well as an `<origin>` element, and returns the whole `<inertial>` block defined inside the macro. The `<origin>` element defines an offset of the center of mass from the transform of a given link. The inertia matrix, defined inside the inertia XML element, is calculated from the parameters. Because the inertia matrix is a 3x3 rational matrix, it can be represented by only six elements:

$$I = \begin{pmatrix} \mathbf{ixx} & \mathbf{ixy} & \mathbf{ixz} \\ ixy & \mathbf{iyy} & \mathbf{iyz} \\ ixz & iyz & \mathbf{izz} \end{pmatrix}$$

■ **Code listing 5.2** Custom macro for RR1 links that constructs the whole link block for a given part of RR1

```
<xacro:macro name="rr1_link"
params="name␣mesh_file␣*origin␣*inertial">
  <link name="${name}">
    <visual>
      <xacro:insert_block name="origin" />
      <geometry>
        <mesh scale="${model_scale}"
          filename="package://path/to/${mesh_file}.dae" />
      </geometry>
    </visual>
    <collision>
      <xacro:insert_block name="origin" />
      <geometry>
        <mesh scale="${model_scale}"
          filename="package://path/to/${mesh_file}.stl" />
      </geometry>
    </collision>
    <xacro:insert_block name="inertial" />
  </link>
</xacro:macro>
```

5.3.2.3 RR1 link macro

Nine links need to be described in the RR1 URDF and have the same XML structure. Visual mesh is prepared in a corresponding DAE file with the correct materials. The `<collision>` element must be specified for the simulations, and the collision mesh is prepared in a corresponding STL file. Both `<visual>` and `<collision>` elements share the same origin, which can be used to offset the mesh from the link transform. The positions of joints in the articulation chain define the transform position and rotation. Lastly, the `<inertial>` block has to be included inside the `<link>` block.

A custom macro called `rr1_link` macro has been developed to make the robot description easier to navigate, read, and maintain. The macro is shown in Code listing 5.2. It is provided with a link name, mesh file name, `<origin>` element, and `<inertial>` block as parameters and returns a complete `<link>` description block[2].

5.3.2.4 Links

With the two macros shown in Code listings 5.1 and 5.2, all the links can be easily defined. The link definitions are in the rr1.urdf.xacro file, and Code listing 5.3 shows an example definition of the `base_link` using the created macros. When the final URDF file is generated using the Xacro command, this description is expanded as shown in Code listing 5.4[3]. Figure 5.5 then shows the resulting visual, collision, and inertial components of the `base_link` visualized in RViz.

All other links can be defined in the same way. The mass of the individual links is unknown because the second prototype of the RR1 has not been completed yet. Currently, mass estimations based on the initial prototype have been used in the URDF description and can be easily modified in the inertial macro call. The origin poses throughout the whole description, including joint origins, are heavily dependent on the original robot model. Their definition is an iterative

[2]Note that the file paths have been shortened to fit the code listing, and the model_scale is a Xacro property defined elsewhere and used for all the meshes.

[3]Again, the code has been formatted to fit the width of the page, and the paths were shortened.

Code listing 5.3 Definition of the RR1's base_link using custom Xacro macros

```
<xacro:rr1_link name="base_link" mesh_file="01_RR1_extended_base">
  <origin rpy="0␣0␣0" xyz="-0.052␣-0.013␣0.179" />
  <xacro:inertial_box mass="4.0" x="0.36" y="0.15" z="0.09">
    <origin rpy="0␣0␣0" xyz="0.105␣0␣0.0375" />
  </xacro:inertial_box>
</xacro:rr1_link>
```

Code listing 5.4 Definition of the RR1's base_link in the generated URDF

```
<link name="base_link">
  <visual>
    <origin rpy="0␣0␣0" xyz="-0.052␣-0.013␣0.179"/>
    <geometry>
      <mesh scale="0.001␣0.001␣0.001"
        filename="package://path/to/01_RR1_extended_base.dae" />
    </geometry>
  </visual>
  <collision>
    <origin rpy="0␣0␣0" xyz="-0.052␣-0.013␣0.179"/>
    <geometry>
      <mesh scale="0.001␣0.001␣0.001"
        filename="package://path/to/01_RR1_extended_base.stl" />
    </geometry>
  </collision>
  <inertial>
    <origin rpy="0␣0␣0" xyz="0.105␣0␣0.0375"/>
    <mass value="4.0"/>
    <inertia ixx="0.010199" ixy="0.0" ixz="0.0"
      iyy="0.045899" iyz="0.0"
      izz="0.050699"/>
  </inertial>
</link>
```

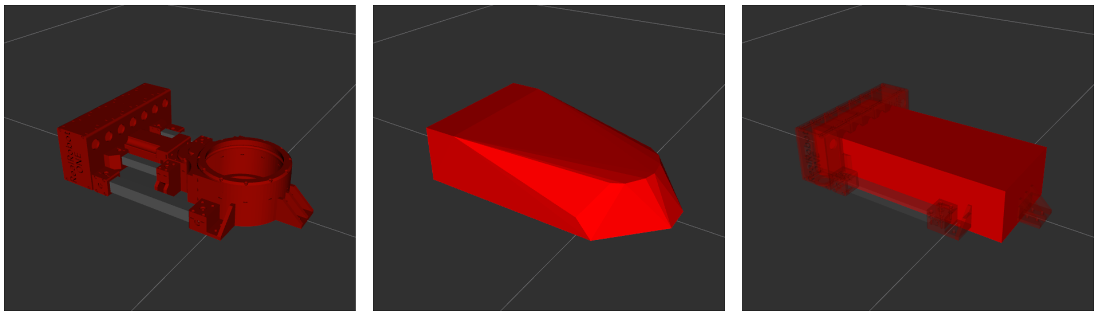

(a) base_link visual **(b)** base_link collision **(c)** base_link inertial

Figure 5.5 RR1 base link visualized in RViz with visual mesh, collision mesh, and inertial

Code listing 5.5 Definition of RR1 shoulder_joint inside the rr1.urdf.xacro file

```
<joint name="shoulder_joint" type="revolute">
  <parent link="base_link" />
  <child link="shoulder_link" />
  <origin xyz="0␣0␣0.07" />
  <axis xyz="0␣0␣1" />
  <limit lower="-${PI*3/4}" upper="${PI*3/4}"
    effort="${effort}" velocity="${velocity}" />
  <dynamics damping="${damping}" friction="${friction}" />
</joint>
```

process of measuring and modifications specific to the robotic system described and a particular model used[4].

5.3.2.5 Joints

Between the links, joints have to be defined using a `<joint>` description block. In Code listing 5.5, the `shoulder_joint` definition is presented. The `<origin>` element defines the position of the child link transform as a relative offset from the parent link transforms, creating a transform tree as visualized in Figure 5.3b. Additionally, the axes of movement and kinematic properties of the joint are specified. In the `shoulder_joint` example, xacro properties effort, velocity, damping, and friction are used. These properties are defined in the rr1.urdf.xacro file as the robot's parameters. The Xacro parameter PI is one of the constant parameters defined in constants.xacro file.

Concluding the joint definitions, the RR1 description is complete, making it possible to visualize the robot in RViz and spawn it in Gazebo. However, no control has been defined, so the robotic arm will fall to the ground under gravity in the simulated environment in Gazebo.

5.4 Robot control

The primary part of robot control is the ability to move it by sending commands according to some planned trajectory. But in general, the robot should also report back the state of its joints to consider them in the planning process or to visualize the robot's pose in visualization tools like RViz. If there were any additional sensors in the system, it would also be desirable to receive data from the sensors as well.

For the simulated robot in Gazebo, all of these tasks can be solved relatively quickly and easily by Gazebo plugins, which are discussed in Section 2.2.1.5. Numerous Gazebo plugins are available to provide these functionalities. A plugin called `gazebo_ros_joint_state_publisher` can be used to report simulated joint configurations to the `/joint_states` topic, and a plugin called `gazebo_ros_joint_pose_trajectory` can be used to publish control trajectories to a topic called `/set_joint_trajectory` that the plugin is subscribed to. Unfortunately, although straightforward, this solution is Gazebo-specific. Later, switching between the control of the virtual entity and the physical RR1 robot should be possible when the hardware is ready. So a different solution would need to be implemented for the physical robot. Therefore, the `ros2_control` package, explained in Section 2.1.5, is used in this prototype. It is more complicated to set up but standardly used, and it allows the same control code to be used for both Gazebo and the real robot, minimizing future modifications.

[4]For example, the position of the link's origin, also referred to as pivot in some modeling applications, and the rotation in the original link mesh significantly impacts the positioning in the URDF.

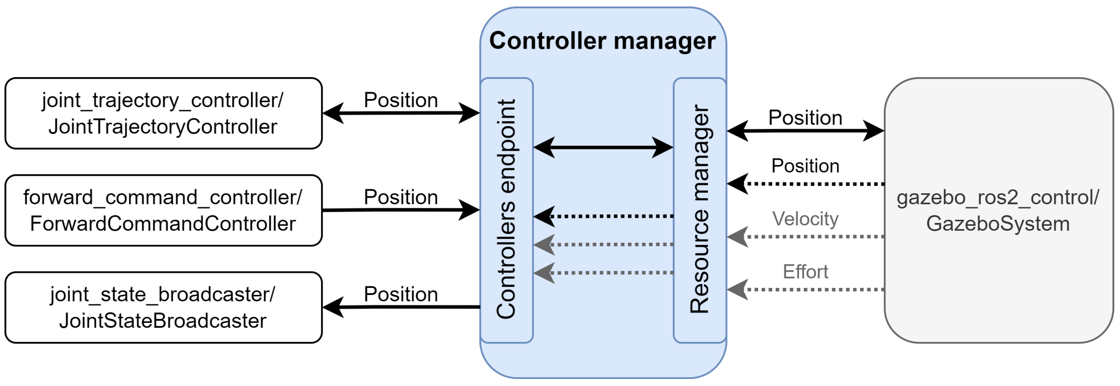


Figure 5.6 Overview of the RR1 control setup with ros2_control

5.4.1 Overview

To allow control of the RR1 robotic arm using `ros2_control`, the URDF has to be extended to tell the resource manager what hardware interfaces will be used and which joints will be controlled. Conversely, a YAML configuration has to be created for the controller manager, which will describe controllers and their configuration. As shown in Figure 5.6, the RR1 control setup follows a similar structure as illustrated in Section 2.1.5.

5.4.1.1 Hardware interfaces

In the case of the virtual entity in Gazebo, the only hardware interface that will be used is one provided specifically for Gazebo, called `gazebo_ros2_contro/GazeboSystem`. A position command interface is exposed for each of the joints in the arm, which sends position commands to the hardware joints. Three state interfaces for position, velocity, and effort were exposed, although not all are currently used.

When the physical robotic arm is completed, additional hardware interfaces will be developed, and the exposed interfaces will mostly depend on the capabilities of the hardware actuators.

5.4.1.2 Controllers

As simple robotic arm control is a common application in robotics, all used controllers are available in `ros2_control`. Namely, three controllers were set up and can be used:

1. The `joint_state_broadcaster` is a controller which reports joint states into `/joint_states` and `/dynamic_joint_states` topics, which makes it possible to visualize the robot in RViz. It supports all types of hardware interface types.

2. The `joint_trajectory_controller` is the primary controller used for RR1 control in this prototype. It can receive joint trajectories that the robot will follow, which can be sent over a specific topic or action interface. It supports joints with position, velocity, or effort control interfaces.

3. The `forward_command_controller` is a secondary controller for this prototype which can be used to receive commands and forward them to the hardware interfaces. Currently, it is used for position commands that can set joints to provided positions. Unlike the previously mentioned `joint_trajectory_controller`, the robot does not continuously follow a trajectory but snaps directly into a specified position.

Code listing 5.6 Macro used for joint control description

```
<xacro:macro name="ros2_control_joint"
params="name␣min_lim␣max_lim">
  <joint name="${name}">
    <command_interface name="position">
      <param name="min">${min_lim}</param>
      <param name="max">${max_lim}</param>
    </command_interface>
    <state_interface name="position">
      <param name="initial_value">0.0</param>
    </state_interface>
    <state_interface name="velocity">
      <param name="initial_value">0.0</param>
    </state_interface>
    <state_interface name="effort">
      <param name="initial_value">0.0</param>
    </state_interface>
  </joint>
</xacro:macro>
```

5.4.2 Extending the URDF

A separate Xacro file called `ros2_control.xacro` was created for all robot description elements connected to robot control and included in the main Xacro file. Most importantly, the `<ros2_control>` element must be added, and the hardware interface and controlled joints must be specified inside it. Secondly, because of using Gazebo, a `<gazebo>` element has to be added, specifying a plugin used by Gazebo to integrate with the `ros2_control`.

5.4.2.1 Joint control macro

State interfaces and the control interface, including limits, must be specified for each of the eight joints. Therefore another custom macro has been created to shorten the contents of the `<ros2_control>` description element. The macro, called `ros2_control_joint`, is shown in Code listing 5.6. It takes the joint name and the joint limits as parameters and returns a complete `<joint>` block used inside the `<ros2_control>` tag.

5.4.2.2 ros2_control URDF element

The added element is shown in Code listing 5.7. From this element, the resource manager of `ros2_control` receives the information on what hardware interfaces are used and which joints are exposed through command and state interfaces. The hardware interface, in the `<ros2_control>` element referred to as plugin, is specified using a hardware tag. It implements how to talk to the Gazebo simulation like it is real hardware. Then, the command and state interfaces for all the joints are described using the `ros2_control_joint` macro.

5.4.2.3 Gazebo plugin

As discussed in Section 2.2.1.5, Gazebo uses plugins to interact with ROS. In this case, a plugin has to be added that tells Gazebo to use `ros2_control`. Not only does this plugin set up things on the Gazebo's end to control the simulated robot, but it also starts its own controller manager node. So when using this plugin, the users do not need to start it on their own, and they can

Code listing 5.7 Added URDF description for ros2_control

```
<ros2_control name="GazeboSystem" type="system">
    <hardware>
      <plugin>gazebo_ros2_control/GazeboSystem</plugin>
    </hardware>

    <xacro:ros2_control_joint name="shoulder_joint"
      min_lim="-${PI*3/4}" max_lim="${PI*3/4}" />
    <xacro:ros2_control_joint name="upper_arm_joint"
      min_lim="-${PI/8}" max_lim="${PI/8}" />
    <xacro:ros2_control_joint name="elbow_joint"
      min_lim="-${PI/2}" max_lim="${PI/2}" />
    <xacro:ros2_control_joint name="forearm_joint"
      min_lim="-${PI*3/4}" max_lim="${PI*3/4}" />
    <xacro:ros2_control_joint name="wrist_joint"
      min_lim="0" max_lim="${PI}" />
    <xacro:ros2_control_joint name="hand_joint"
      min_lim="-${PI*3/4}" max_lim="${PI*3/4}" />
    <xacro:ros2_control_joint name="finger_joint"
      min_lim="0" max_lim="${PI*3/8}" />
    <xacro:ros2_control_joint name="opposite_finger_joint"
      min_lim="0" max_lim="${PI*3/8}" />
  </ros2_control>
```

just spawn the controllers they want to use[5]. When starting the controller manager, the plugin will also provide the robot description to it automatically. However, the YAML configuration file has to be specified as a parameter, as shown in Code listing 5.8. The configuration file is discussed in Section 5.4.3. Additionally, the used hardware interface is identified.

Because of the need to simulate multi-robot scenarios, a namespace and topic remapping are done. A Xacro argument has been created, which can be specified when running the Xacro command and defaults to value "rr1". The value of this argument is used as a namespace, so the started controller manager node and any other node are started under this namespace. The `/tf` topic is also remapped to include the namespace. So for a namespace "rr1", an `rr1/controller_manager` will be started, and the instance of the robot will publish its transforms into the `/rr1/tf` topic. This allows spawning multiple robot instances under different namespaces and controlling them independently via their own controllers.

5.4.3 Controller configuration

In ROS, YAML files are commonly used to configure parameters for nodes. In `ros2_control`, parameters are provided in such a YAML file for the controller manager and specified controllers. This configuration file has been created inside the `rr1_description` package.

Code listing 5.9 shows the beginning of the configuration file, including parameters for the controller_manager node. The first line contains a wildcard for the namespace, indicating that the same parameters are used for all the nodes configured in this file that run under any namespace. This allows running multiple controller nodes for individual instances of the simulated RR1 robot. The names and types of discussed controllers are specified in the controller_manager configuration, as well as some other parameters like update rate, which will also limit the update rate of individual controllers. After the controllers are defined here, they can be configured

[5]Note that when Gazebo is not used, like when controlling a real robot, the controller manager node has to be started manually, for example, from a launch file.

Code listing 5.8 Gazebo plugin that tells Gazebo to use ros2_control

```
<gazebo>
  <plugin filename="libgazebo_ros2_control.so"
  name="gazebo_ros2_control">
    <robot_sim_type>
      gazebo_ros2_control/GazeboSystem
    </robot_sim_type>
    <parameters>
      $(find rr1_description)/config/rr1_controller.yaml
    </parameters>
    <ros>
      <namespace>/$(arg ns)</namespace>
      <remapping>/tf:=/$(arg ns)/tf</remapping>
    </ros>
  </plugin>
</gazebo>
```

Code listing 5.9 Parameters for the controller_manager node

```
/**:
  controller_manager:
    ros__parameters:
      update_rate: 50  # Hz
      use_sim_time: true

      joint_state_broadcaster:
        type: joint_state_broadcaster/JointStateBroadcaster
      forward_position_controller:
        type: forward_command_controller/ForwardCommandController
      joint_trajectory_controller:
        type: joint_trajectory_controller/JointTrajectoryController
```

individually, as shown in Code listing 5.10 for the `joint_trajectory_controller`[6]. In this application, the joint_state_broadcaster controller does not need to be configured. The controlled joints must be listed for the other two controllers, and interface command or state interfaces must be specified. This allows the `controller_manager` to assign interfaces to the controllers correctly. Each controller can have their own specific parameters.

5.4.4 Sending control trajectories

The actual act of controlling the robot is currently done by test nodes developed inside the `rr1_control` package, as mentioned in Section 5.2.4. One node can be used for sending position commands to the `forward_position_controller` and another node for sending trajectories to the `joint_trajectory_controller`. Both of the nodes have their YAML configuration files which contain the testing data and some additional parameters like frequency of message publishing. Launch files are also provided for the nodes making it also possible to configure the topic where the control commands are published. Therefore in a multi-robot setting, multiple instances of the nodes can be started, each sending commands to a different controller.

Additionally to these nodes, two scripts have been implemented to make it easier for the user to interact with the controller manager. One script can be used to switch between the

[6]Note that the configuration for the joint_trajectory_controller node is incomplete. Additional controller-specific parameters are configured in the YAML file.

Code listing 5.10 Configuration of the joint trajectory controller

```
joint_trajectory_controller:
  ros__parameters:
    joints:
      - shoulder_joint
      - upper_arm_joint
      - elbow_joint
      - forearm_joint
      - wrist_joint
      - hand_joint
      - finger_joint
      - opposite_finger_joint
    command_interfaces:
      - position
    state_interfaces:
      - position

    state_publish_rate: 50.0
    action_monitor_rate: 20.0
    ...
```

controllers that have been launched, and another script can be used to publish joint positions via a command line.

5.5 Gazebo simulation

The prototype Gazebo simulation is tightly connected to ROS, as it extends the URDF format and is the only entity currently possible to control with `ros2_control`. Therefore, some of the preparations for the Gazebo simulation have been done in Section 5.4. The remaining task is creating launch files for correctly orchestrating appropriate ROS nodes and Gazebo scripts. Specifically, Gazebo has to be started, and the virtual entity has to be spawned with all the `ros2_control` nodes, as shown in Figure 5.7. Optionally, RViz can also be started to visualize the robot's state inside the Gazebo simulation. When the controlled robot is simulated, additional nodes that publish control trajectories can be started.

5.5.1 Starting Gazebo

As Section 2.2.1.5 mentions, a launch file from the `gazebo_ros` package is used when running Gazebo with ROS integration. However, because the simulation uses custom meshes for the RR1 robot, Gazebo has to be able to find those meshes. Therefore, a custom launch file that starts Gazebo has been created in the `rr1_gazebo` package. The launch file modifies some path environment variables used by Gazebo to locate models and plugins so that they include paths to the models used in the RR1 description. Then, the launch file from the `gazebo_ros` package is included and configured to start a world provided inside the `rr1_gazebo` package[7].

[7]The world file contains only a default empty world scene but can be overwritten by another world file. The deployment process has been developed, so the world file is shared within ROS when the package is built, and the Gazebo launch file can find and open it.

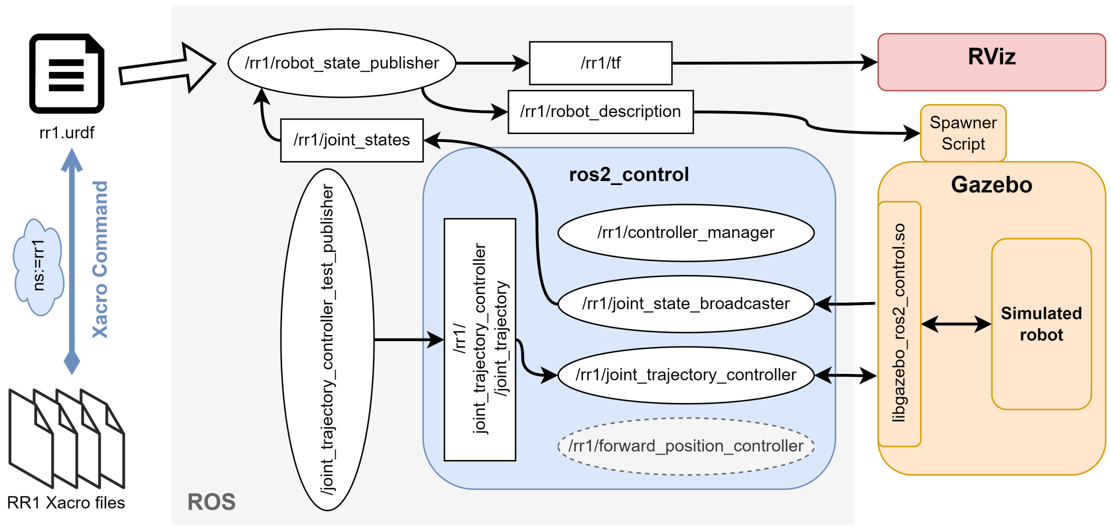

rr1.urdf
ns:=rr1
Xacro Command
RR1 Xacro files
ROS
/rr1/robot_state_publisher
/rr1/tf
/rr1/robot_description
/rr1/joint_states
/joint_trajectory_controller_test_publisher
ros2_control
/rr1/joint_trajectory_controller/joint_trajectory
/rr1/controller_manager
/rr1/joint_state_broadcaster
/rr1/joint_trajectory_controller
/rr1/forward_position_controller
RViz
Spawner Script
Gazebo
libgazebo_ros2_control.so
Simulated robot

Figure 5.7 RR1 Gazebo simulation landscape

5.5.2 Spawning RR1

Once Gazebo runs, a robot instance, including the controllers, can be spawned. This includes running several nodes, from which some have dependencies on others and have to be started in a specific order. Therefore a launch file is created inside the `rr1_gazebo` package to orchestrate the process of spawning a robot into Gazebo.

5.5.2.1 Starting robot_state_publisher

Firstly, a `robot_state_publisher` node has to be started to publish the robot description into a `/robot_description` topic. As illustrated on the left side of Figure 5.7, a single URDF description file for the RR1 robot is generated from the Xacro files using the Xacro command. A namespace can be provided as an argument for the command because the Xacro description has a namespace argument. The output URDF file of this command is used as a parameter for the `robot_state_publisher`, which then publishes its copy to the `/robot_description` topic continually. This makes it possible for other nodes to retrieve the robot description from that topic.

5.5.2.2 Spawning RR1 in Gazebo

A spawner script provided in the `gazebo_ros` package is used to spawn the robot. The spawner script retrieves the robot description from the `/robot_description` topic, spawns the simulated robot, and starts the plugin specified in the URDF. Arguments can be provided to the spawner script to specify the position where the entity should be spawned, as well as its name that will be used in the Gazebo scene hierarchy. The plugin, discussed in Section 5.4.2.3, also starts the `controller_manager` node from `ros2_control`. The plugin provides the path to the controller YAML configuration file, as it was specified in the URDF, and the `controller_manager` also retrieves the robot description from the `/robot_description` topic.

5.5.2.3 Starting controller nodes

After the robot is spawned in Gazebo and the `controller_manager` node is running. A spawner script of the controller manager is used to spawn the individual controller nodes. Because the

Code listing 5.11 Example of the event handler for starting the joint_state_broadcaster controller

```
joint_state_broadcaster = Node(
  package="controller_manager",
  executable="spawner",
  arguments=[
    "joint_state_broadcaster",
    "--controller-manager", f"/{NAMESPACE}/controller_manager"
  ]
)

joint_state_broadcaster_event = RegisterEventHandler(
  event_handler=OnProcessExit(
    target_action=spawn_robot,
    on_exit=[joint_state_broadcaster]
  )
)
```

controller manager knows about the available controllers and their configurations from the provided YAML file, the spawning is straightforward. However, the `controller_manager` node has to be running to spawn the controllers.

Although the spawner script will wait a while for the controller manager to start, if it is not running, the control manager may become active after this waiting period. Special event handlers are created in the launch file to start actions after other actions are finished. So, for example, the node responsible for starting a controller can be run after the Gazebo spawner script has finished its execution.

With these events set up, the `joint_state_broadcaster` node is started first, after the `controller_manager` node has been started, as shown in Code listing 5.11. The joint state publisher starts publishing the joint states into the `/joint_states` topic. The robot state publisher uses these joint states to update and publish the transforms of RR1, which RViz can use, for example. In the same way, the two controllers, `forward_position_controller` and `joint_trajectory_controller`, are started after the joint state broadcaster is running. However, the `forward_position_controller` is purposefully started in an inactive state because two controllers cannot control the same hardware interfaces. However, when it is started as inactive, the controller manager can switch between them if needed.

Independently on this launch process, a test node from the `rr1_control` package can be started to publish trajectories or position commands to appropriate topics. The virtual entity simulated in Gazebo starts the trajectory execution immediately once the controller is started and the control commands are published to the topic.

5.5.3 Multi-robot simulation

Several choices have been made to make the multi-robot simulation of RR1 easier inside the Gazebo. First, the URDF is parametrized with a namespace name, which is then used inside the Gazebo plugin specification. The plugin uses the namespace to run the `command_controller` and other related nodes under this namespace. Additionally, the launch file for spawning the robot inside Gazebo demonstrates the use of namespaces. Using namespaces is also illustrated in Figure 5.4.2.3, where the default namespace `rr1` is used. Lastly, it is possible to configure the topic used by the testing control nodes from the `rr1_control` package.

To run a multi robot simulation in Gazebo, a launch file has to generate multiple URDF files, each parametrized with a different namespace. A robot state publisher has to be started for each robot instance. These robot state publishers are used to spawn all robot instances inside Gazebo

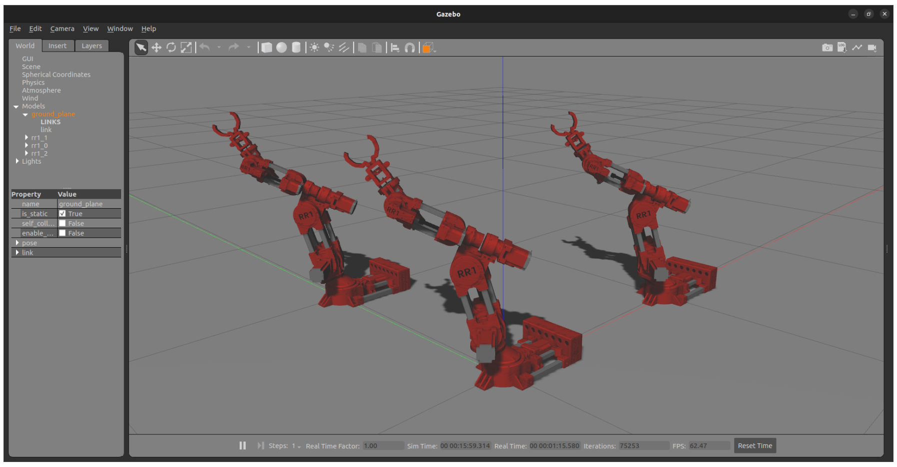


Figure 5.8 Multi-robot simulation scenario in Gazebo

on specified locations, also resulting in multiple controller managers being started in appropriate namespaces. For each controller manager, the controllers required for the application must be spawned. This is possible because the controller YAML configuration file uses a namespace wildcard as shown in Code listing 5.9. Finally, nodes publishing trajectories to the topic the controllers are subscribed to can also be spawned. Such a multi-robot Gazebo simulation is shown in Figure 5.8.

5.6 Unity simulation

The Unity simulation is not so entangled with ROS as using Gazebo but still takes advantage of some of the previous work. Mainly, the URDF robot description can be imported directly into the Unity project, creating a persistent asset that can be manually or programmatically instantiated multiple times into the scene. Figure 5.9 illustrates a classic single-robot simulation in Unity similar to the one in Gazebo, using a joint trajectory controller, with other available controllers indicated. However, most of the controllers and other ROS publishers had to be implemented for Unity, as the Unity Robotics Hub does not provide them. These controllers are implemented as components that can be attached to the robot asset so they are instantiated with the robot model. Once the Unity-ROS integration is established, ROS nodes can communicate with the controllers attached to the simulated instance inside Unity, as shown in Figure 5.10.

5.6.1 Unity-ROS integration

Unity packages from the Unity Robotics Hub are used for the Unity-ROS integration, which is explained in Section 2.3.4.3. Namely, the ROS TCP Connector on the Unity side and the ROS TCP Endpoint package on the ROS side are used for communication between ROS and Unity. All the developed publishers or subscribers implemented in Unity, including the robot controllers, use a ROS TCP Connector instance to subscribe or publish messages to topics inside the ROS ecosystem. The Unity-ROS communication, done over a single ROS node called UnityEndpoint, is shown in Figure 2.15. In Figure 5.9, however, the communication stream is decomposed, making it more apparent which Unity components communicate with which ROS topics.

In the `rr1_bringup` ROS package, a launch file for running the Unity simulation has been prepared. The launch file starts the UnityEndpoint node from the ROS TCP Endpoint package and optionally can also start RViz to visualize the state of the simulated RR1 instance. The testing nodes from the `rr1_control` package can be started to send testing trajectories to Unity.

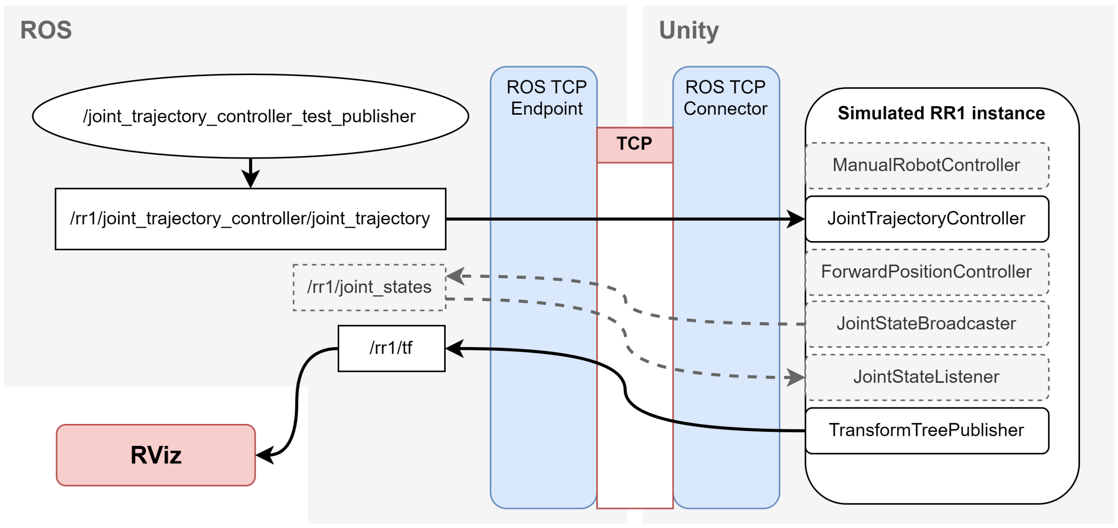
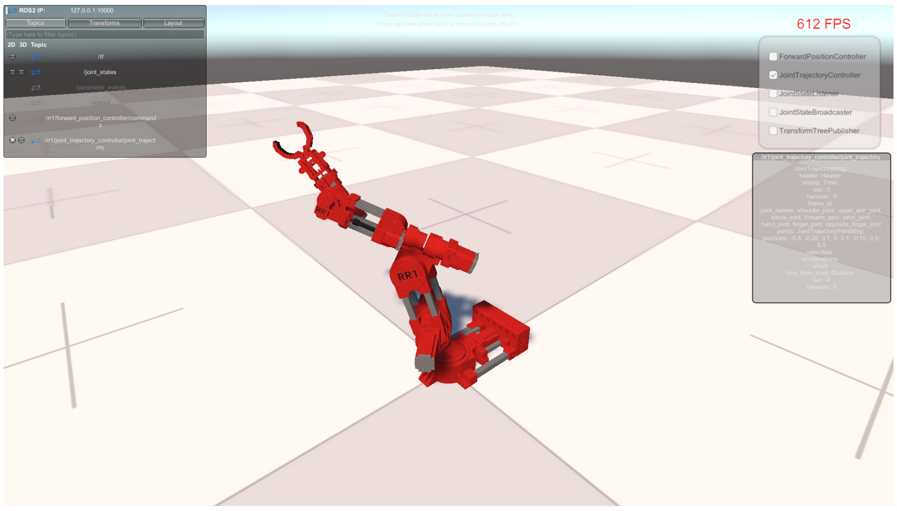


Figure 5.9 RR1 Unity simulation landscape


ROS2 IP: 127.0.0.1:10000
Topics
Transforms
Layout
(Type here to filter topics)
2D 3D Topic
/tf
/joint_states
/parameter_events
/rosout
/rr1/forward_position_controller/commands
/rr1/joint_trajectory_controller/joint_trajectory
612 FPS
ForwardPositionController
✓ JointTrajectoryController
JointStateListener
JointStateBroadcaster
TransformTreePublisher
RR1

```
/rr1/joint_trajectory_controller/joint_trajectory
JointTrajectoryMsg:
header: Header:
stamp: Time:
sec: 0
nanosec: 0
frame_id:
joint_names: shoulder_joint, upper_arm_joint,
elbow_joint, forearm_joint, wrist_joint,
hand_joint, finger_joint, opposite_finger_joint
points: JointTrajectoryPointMsg:
positions: -0.8, -0.28, 0.7, 0, 2.1, -0.15, 0.5,
0.5
velocities:
accelerations:
effort:
time_from_start: Duration:
sec: 4
nanosec: 0
```

Figure 5.10 RR1 simulation prototype in Unity

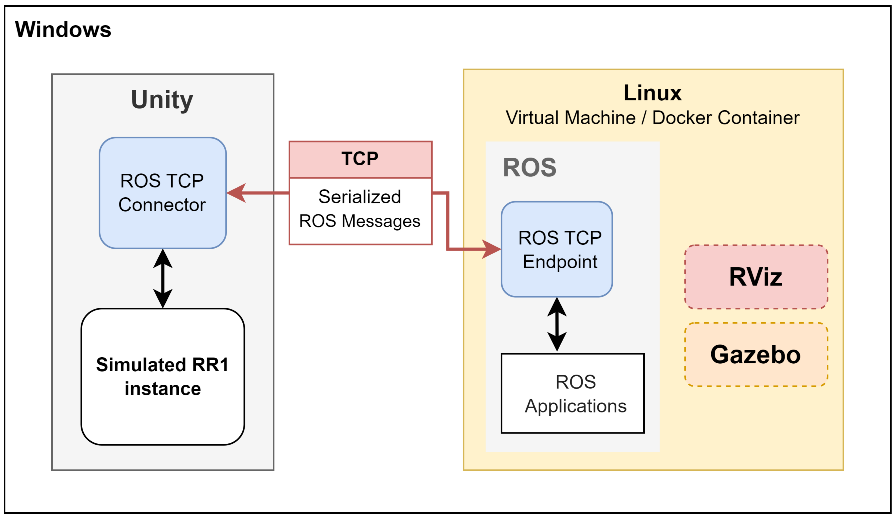


Figure 5.11 Unity-ROS integration on Windows

Because the Unity controllers can use the same topics used by the `ros2_control`, the simulation can run in Gazebo and Unity simultaneously.

5.6.1.1 ROS docker image

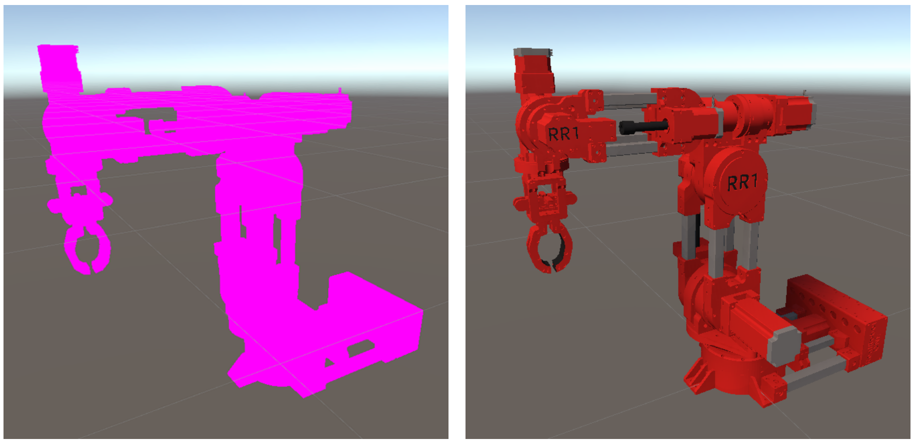
The Unity simulation can be run in Linux, together with ROS. However, Unity editor can encounter some issues with certain Linux distributions, and overall, Unity development is more convenient on Windows. On the other hand, the Windows support of ROS is limited, and the ROS development on Windows is unfavorable as the community is used to Linux from ROS 1. Nevertheless, running ROS on a Linux virtual machine or inside a Docker container running on Windows is possible, as depicted in Figure 5.11.

Both approaches using a virtual machine and a Docker container were used during the development and testing of the prototype simulation. It allows running ROS, as well as RViz or Gazebo on Windows. However, both solutions lack GPU acceleration, so running RViz and Gazebo yields very low FPS and is not recommended. However, no issues were encountered facilitating ROS and its communication with Unity running on Windows.

Running a virtual machine has a downside: the user must install Linux and all ROS dependencies independently. In Docker, all setup processes can be automatized for the user by providing a complete Dockerfile. Therefore, the use of Docker is recommended for the simulation, and a custom Docker image has been developed specifically for the RR1 robotic arm. When built, the Docker image installs ROS 2 Humble with all dependencies and development tools. The ROS TCP Endpoint from Unity Robotics Hub and all the RR1 ROS packages are cloned from GitHub, and the ROS workspace is built and loaded. When users start a container from this image, they can immediately run any launch file developed for RR1, including the Unity integration.

5.6.2 Importing robot into Unity

Using the URDF importer from the Unity Robotics Hub, the RR1 robot can be imported into the Unity project. The importer does not support the Xacro format, so the robot description has to be converted to a single URDF file using the Xacro command beforehand. After that, the `rr1_description` package can be copied to the project's assets folder, and then the URDF file



(a) Materials imported by URDF Importer appear pink **(b)** Fixed materials after importing the URDF

Figure 5.12 Shader issue encountered when using Unity's Universal Render Pipeline instead of the Built-in Render Pipeline

can be imported using the URDF importer[8]. The importer will construct the articulation chain in the scene with all the visual and collision meshes. One manual step is needed for a robotic arm like this: the `base_link` object has to be located in the articulation chain and set its link component to be immovable. The articulation body component must also be set as a base link of the articulation chain.

The robot model can be dragged back to the assets folder from the hierarchy window, which creates a prefab from the game object[9]. Depending on the application, different components can be added to the prefab. In the case of these simulated robotic arms, the components can be robot controllers, for example.

5.6.2.1 Material issues in URP

The Universal Render Pipeline (URP) is used in the prototype instead of the Built-in Render Pipeline. The URP provides more flexibility when it comes to graphics, and it better accommodates virtual reality projects. However, the two render pipelines use different shaders, affecting how materials are displayed. When the robot is imported into a project that uses URP, the materials will appear pink, as shown in Figure 5.12a. The problem is that a shader from the Built-in Render Pipeline is assigned to the materials. Unity provides an automated way to convert materials to URP, which switches the used shader one made for the URP. However, before this step, the materials must be extracted from the mesh files into separate material files so that the shader change can be saved inside the Unity project. This process is easily accessible from the inspector window inside the Unity editor. Figure 5.12b shows the result after the material conversion.

5.6.3 Unity robot controllers

In the current state of the prototype, several robot controllers are available for the simulated RR1 robotic arm. Most were implemented specifically for this proof of concept because they

[8]Note that Unity has to be able to find the meshes on paths specified inside the URDF file. For this particular robot description, this can be achieved by dragging the single URDF file outside the rr1_description directory.

[9]Prefabs are game objects saved together with all their components and settings. These prefabs can then be instantiated multiple times in the scene.

were unavailable in the Unity Robotics Hub. However, they are also generic enough to be used in other robotic arms imported via the URDF importer. The following subsections describe their use and functionalities.

5.6.3.1 Manual robot controller

Once a robotic arm URDF is imported, the URDF importer automatically attaches a component called RobotController to it, developed by the Unity Robotics Hub. This component can be used for direct keyboard control of the robotic arm by using arrows to switch between individual joints and adjust their angles.

The controller used Unity's old input system, which is hard to maintain and operate in a larger project. In this prototype, the component has been modified to use the new input system and renamed to ManualRobotController.

5.6.3.2 Joint state listener

The *JointStateListener* is a custom controller component subscribing to the `/joint_states` topic and receiving joint state messages from ROS. The fixed update call mirrors the joint states in the simulated RR1 instance, resulting in a similar behavior to RViz visualization. Although, RViz listens to transforms, which are updated by the robot state publisher from the received states. This controller can be used to visualize the state of the real robot as its encoders will report the joint states to ROS.

5.6.3.3 Joint state broadcaster

On the other hand, the *JointStateBroadcaster* component publishes the simulated joint states into the `/joint_states` ROS topic. It is a custom Unity controller corresponding to the `joint_state_broadcaster` used with the `ros2_control` package. In the Gazebo simulation, this controller was used to report joint states to ROS, which the robot state publisher used to update the transforms. RViz could then visualize the simulated robot by subscribing to the updated transforms. However, the Unity prototype can publish the updated transforms directly without needing a robot state publisher running in ROS. So there is currently no need to use the JointStateBroadcaster component.

5.6.3.4 Joint trajectory controller

Corresponding to the `joint_trajectory_controller` from `ros2_control`, the *JointTrajectoryController* is a custom component used as a primary controller in the simulation prototype. It subscribes to a configurable topic, receiving messages of the JointTrajectory message type, as the `ros2_control` does, and has the same behavior as the `joint_trajectory_controller`.

Upon receiving a trajectory, it starts a coroutine following the specified trajectory until it is completed or until a new trajectory is received. Code listing 5.12 shows a SubscriberCallback method, which is called when a new trajectory message arrives at the subscribed topic. A new instance of the FollowTrajectory coroutine is started. The coroutine follows the individual trajectory points with motion durations specified in the received message. In each update, when the coroutine execution continues, based on the elapsed time in the trajectory execution, a target position for each of the joints is computed using linear interpolation and set for all joints simultaneously.

5.6.3.5 Forward position controller

The *ForwardPositionController* is a custom Unity component equivalent to its `ros2_control` counterpart. It can subscribe to a ROS topic that is configurable via launch arguments and

Code listing 5.12 Coroutine from the JointTrajectoryController component following a received trajectory

```
private void SubscriberCallback(JointTrajectoryMsg message) {
  if (currentTrajectoryExecution != null)
    StopCoroutine(currentTrajectoryExecution);

  currentTrajectoryExecution = StartCoroutine(
                               FollowTrajectory(message.points));
}

private IEnumerator FollowTrajectory(
        JointTrajectoryPointMsg[] points) {
  foreach (var point in points) {
    var motionDuration = (float)(point.time_from_start.sec
            + point.time_from_start.nanosec * NanoToSec);

    var initialPoint = GetJointPositions();
    var elapsedTime = 0.0;
    while ((elapsedTime += Time.deltaTime) < motionDuration) {
      var targetPoint = Mathd.Lerp(initialPoint, point.positions,
                                   elapsedTime / motionDuration);
      SetJointTargets(targetPoint);
      yield return null;
    }
    SetJointTargets(point.positions);
  }
  currentTrajectoryExecution = null;
}
```

receive position control commands. When a message is received, it snaps the simulated joints into the requested positions.

5.6.3.6 Transform tree publisher

In one of the demo projects developed by Unity Robotics Hub, a *TransformTreePublisher* component has been developed and used in a mobile robot simulation. This component was adapted in the RR1 simulation prototype and can publish the transform tree of the simulated articulation chain into the `/tf` topic. RViz can read these transform messages to visualize the state of the simulated entity if needed.

5.6.4 Other components

A few additional components were developed for the Unity simulation prototype. In this section, some of the most notable are described.

5.6.4.1 GUI

The ROS TCP Connector package already comes with some essential GUI elements displaying the status of the Unity-ROS connection and messages being passed. A custom GUI panel was also created to enable turning different controllers on and off in run time and switching between them. This panel can be seen in the top-right corner of Figure 5.10.

5.6.4.2 ROS clock publisher

In section 2.2.1.7, the need for a simulator to keep track of time instead of ROS was explained. In the case of the simulator being Unity, the *ROSClockPublisher* component can be used to publish the Unity time to the `/clock` topic periodically. The Unity time can be scaled based on the application's need allowing the simulation to run faster or to be slowed down.

5.6.4.3 Camera control

In the play mode of the application or the build, the camera control from the scene view is unavailable, as it is a feature of the Unity editor. Therefore a custom camera control has been developed using the new Unity input system. The camera control allows free movement around the simulated robotic arm.

5.6.4.4 Frame rate counter

Because the performance of the simulation is one of the main points of interest in this prototype realization, a component tracking the actual frame rate of the simulation has been implemented. The frame rate value is then visibly displayed in the GUI, as shown in Figure 5.10.

Code listing 5.13 shows the code of the component. Because the time that elapses between individual frames can be very variable and dynamic, the frame rate is updated four times per second. Between these frame rate updates, the game loop will execute many times, and the average elapsed time between individual frames is more stable.

5.6.5 Multi-robot simulation

Running a multi-robot simulation in Unity is less complicated compared to Gazebo. For the Gazebo simulation, a substantial number of nodes had to be started for each robot instance. In Unity, because a prefab of the robot has been created, this prefab can be dragged into the

Code listing 5.13 Coroutine from the JointTrajectoryController component following a received trajectory

```
public class FPSCounter : MonoBehaviour {
  [SerializeField] private Text label;
  [SerializeField] private float fps = 0.0f;

  private int frameCnt = 0;
  private float dt = 0.0f;
  private float updateRate = 4.0f;  // 4 updates per sec.

  public void Update () {
    frameCnt++;
    dt += Time.deltaTime;
    if (dt > 1.0f / updateRate) {
      fps = (frameCnt / dt) * Time.timeScale;
      frameCnt = 0;
      dt -= 1.0f / updateRate;

      label.text = (int)fps + "␣FPS";
    }
  }
}
```

Code listing 5.14 Instantiating multiple robots into Unity scene and setting correct ROS namespace for the ROS topics

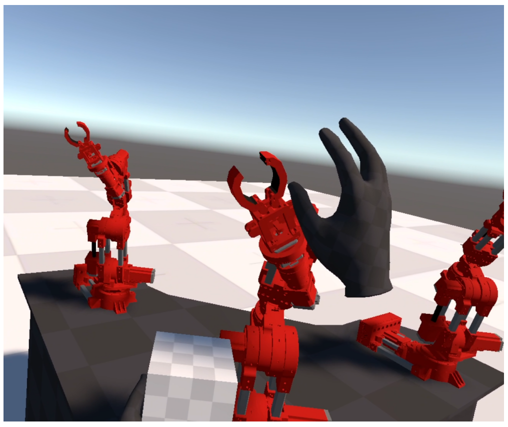
```
List<Vector3> positions = generateRobotPositions();
for (var i = 0; i < robotCount; i++) {
  string ns = $"{namespacePrefix}_{i}";
  var robot = Instantiate(RobotPrefab, positions[i],
              RobotPrefab.transform.rotation, ParentTransform);
  robot.name = ns;
  var jtc = robot.GetComponent<JointTrajectoryController>();
  jtc.TopicName = $"/{ns}{TopicName}";
}
```

scene multiple times or instantiated programmatically, as indicated in Code listing 5.14. In this code example, the JointTrajectoryController components are used for the simulated robot control. Therefore the namespace of the ROS topic the controllers subscribe to has to be set up correctly, so the robots can be controlled independently. It is also possible for all the controllers to subscribe to the same topic.

On the ROS side, appropriate nodes must be started to send trajectories to the correct topics. This was also done in the Gazebo simulation via the testing nodes from the `rr1_control` package. A launch file can be written to start a corresponding number of these nodes, publishing trajectories to appropriate topics used by the RR1 instances simulated in Unity. A multi-robot simulation example in Unity can be seen in Figure 5.13b.

5.6.6 VR simulation

In order to demonstrate the flexibility and capabilities of simulation in Unity, a multi-robot scene was developed in virtual reality using the XR Interaction Toolkit mentioned in Section 2.3.1.3. In the scene, the user can take control of two hands that are animated and physically interact


(a)

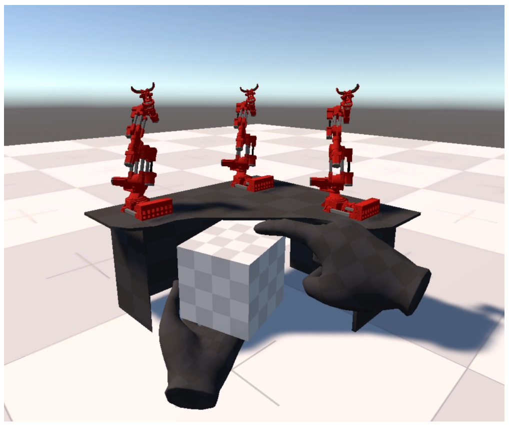

(b)

Figure 5.13 Screenshots from multi-robot simulation in virtual reality

with the robotic arms. There is also an interactable object the user can grab. Figure 5.13 shows screenshots from the VR scene. The VR simulation was tested with the Oculus Quest 2 headset wirelessly connected to a computer.

The robotic arms in the scene were programmatically spawned with configured JointTrajectoryController components and are controlled from ROS by the test nodes in the `rr1_control` package. The setup was the same as in the multi-robot simulation. Only the VR control was added on top.

Chapter 6
Experiments

In this chapter, the experiments performed with Gazebo and Unity are described, and the results are presented. The first section overviews the experimentation approach, including used metrics, their measurement, and the hardware on which the experiments were done. The second section describes tested scenarios and their purpose. Before reviewing the results of the tested scenarios, the third section introduces a comprehensive comparison between Gazebo and Unity, which is also essential when choosing the right tool for multi-robot simulation. The last two sections then present the results of tested scenarios.

6.1 Overview

The primary purpose of the proposed experiments is to evaluate the performance of Gazebo and Unity in relation to the scene complexity and their usability for the simulation of many robotic arms in a multi-robot scenario. Additionally to the experiment scenarios and test configurations, the functionalities and features of both simulators are taken into consideration.

6.1.1 Performance metrics

Several metrics were selected and monitored throughout the tests to evaluate the performance of Gazebo and Unity. However, not all could be obtained programmatically or from a single script. Moreover, one of the commonly used metrics used to compare robot simulators, the real-time factor, is unavailable in Unity. The following subsections discuss some of these metrics, and the methods of measuring them are described in Section 6.1.3.

6.1.1.1 Frame rate

The *frame rate* is a commonly used metric for graphical applications. It measures the number of frames generated in one second of execution. Generally corresponding with how fast and responsive the application is, it is a straightforward and understandable measurement for the end user to evaluate the overall performance of applications and compare their performance.

Increasing the visual complexity of a scene results in a decrease in frame rate as the individual frames take longer to render. For game engines like Unity, where other operations are also part of the rendering loop, blocking operations or extensive computations will negatively impact the frame rate as well. In Gazebo, where the simulation and graphical interface are separated into two running processes, the impact of physics complexity might not affect the frame rate at all.

It is also important to note that when it comes to optimizing and profiling the application's performance during development, the frame rate is not a good metric. To find problems in the execution and potential optimizations, measuring frame times between individual frames and identifying overloaded frames is better.

6.1.1.2 Real-time factor

The *real-time factor (RTF)* is a standard metric used to compare the performance of robotic simulators and is explained in Section 2.2.1.7. Gazebo reports the RTF in its GUI, and the metric is also published into a ROS topic. When the physics simulation is too complex, Gazebo will decrease the RTF, slowing the simulation time to give the physics engine more time to perform the calculations. It is not possible to enforce a particular RTF in Gazebo.

Conversely, Unity does not have such a mechanism and will always run in real-time ($RTF = 1$) unless explicitly set differently. Specifically, Unity's time scale can be changed. By default, it equals one but can be decreased or increased, making the simulation time slower or quicker, respectively. However, the fixed delta time, the amount of time between physics updates, must be scaled appropriately[1]. These settings can also be done programmatically. Therefore, a Time-Manager component has been created for benchmark purposes to accommodate the time scale changes if needed.

6.1.1.3 Hardware utilization

Additional metrics were measured, providing insight into the *hardware utilization* of the two simulators. Namely, CPU utilization was measured individually for each CPU core, and RAM, GPU, and VRAM usage were measured. The temperature of the CPU and GPU were also monitored.

6.1.2 Workstation specifications

All the experiments were done on a single workstation specified in Table 6.1. The tests were performed on Windows 10 Education and Ubuntu 22.04.2 LTS, installed in a dual-boot configuration. When ROS had to be used for Windows tests, a Docker image based on the same Ubuntu version was used.

Table 6.1 Hardware specifications of the test workstation

Motherboard	ASUS TUF GAMING B550-PLUS
CPU	AMD Ryzen 7 3700X (3.6 GHz)
CPU Cores	8 Cores (16 logical processors)
RAM	32 GB 3600 MHz DDR4
GPU	GeForce RTX 2060 SUPER
GPU VRAM	8 GB GDDR6

6.1.3 Experiment and measurement methods

All test scenarios, specified in Section 6.2, were performed in Gazebo on Linux, Unity on Linux, and Unity on Windows. Each of the test configurations was run as a separate simulation. Precisely, a simulation process was not reused to run a different scenario or increasingly complex scene. After the measurements were performed for a specific test configuration, the simulation

[1]Lowering the time scale without scaling down the fixed delta time results in a slowed simulation, where physics objects visibly snap between positions as their update rate is low.

was terminated completely, including any ROS nodes, before the next test. Between the tested configurations, the temperature was monitored, and the hardware components were allowed to cool down to avoid thermal throttling. Mainly the GPU was cooled down below 40°C, during which time the CPU already cooled down below 30°C. The whole system was restarted between individual scenarios or when switching to a different simulator.

A Python script was developed for metrics measurements, which runs during the simulation and can be manually triggered to log measured values. It has been further modified for Gazebo and Unity tests, and the tests run on Windows because different metrics can be measured in those situations. Each of these versions is described in the following subsections. An additional lightweight Python script was developed for the CPU and GPU temperature monitoring between test runs.

6.1.3.1 Gazebo measurements

The Python script measuring the metrics uses psutil and gpustat Python libraries to retrieve information about the system utilization. The psutil library is used to track the utilization of individual CPU cores, CPU temperature, and RAM usage. The gpustat library, on the other hand, is used to log the GPU utilization and temperature and the VRAM usage. For Gazebo specifically, the script retrieves the current RTF from a topic called /performance_metrics, where Gazebo publishes it.

The script waits for the user to start measuring when several samples are taken at a specified rate. However, retrieving the RTF from a ROS topic takes considerable time, decreasing the possible sampling rate significantly. For each test configuration, twenty samples are measured and logged, and then averaged during the analysis process.

6.1.3.2 Unity measurements (Linux)

For Unity on Linux, the only modification is skipping the RTF measurement, as Unity does not provide this metric. Because the script does not need to retrieve any message from a ROS topic, the sampling rate can be significantly higher. However, it is kept at a similar pace to Gazebo measurements.

6.1.3.3 Unity measurements (Windows)

On Windows, the script has to be further modified. The psutil library was not able to retrieve information about the CPU temperature. Therefore this metric has been removed. The script used for temperature monitoring between test runs uses the same API. However, the tests on Linux showed that by the time the GPU cools down sufficiently, the CPU also cools down significantly. So only the GPU temperature was closely monitored on Windows.

6.1.3.4 Frame rate measurements

The frame rate is a bit more complicated to obtain in Gazebo. It is reported individually for any sensor in the simulated world, such as a single camera in the experiment scene. The camera's frame rate is shown only in the Gazebo's GUI on the right side of the bottom bar. By default, the frame rate is limited to 62 FPS every time Gazebo starts, and although the camera object is specified in the world's SDF file, the frame rate limit property is not present. However, it can be changed in the camera's properties from the left panel in Gazebo. Therefore, before each experiment, the frame rate limit is manually set to 1000 FPS, allowing the reported frame rate to increase to the value limited by the scene complexity and available hardware. The camera's frame rate is not published to any ROS topic as a performance metric, so it is recorded separately upon visual observation.

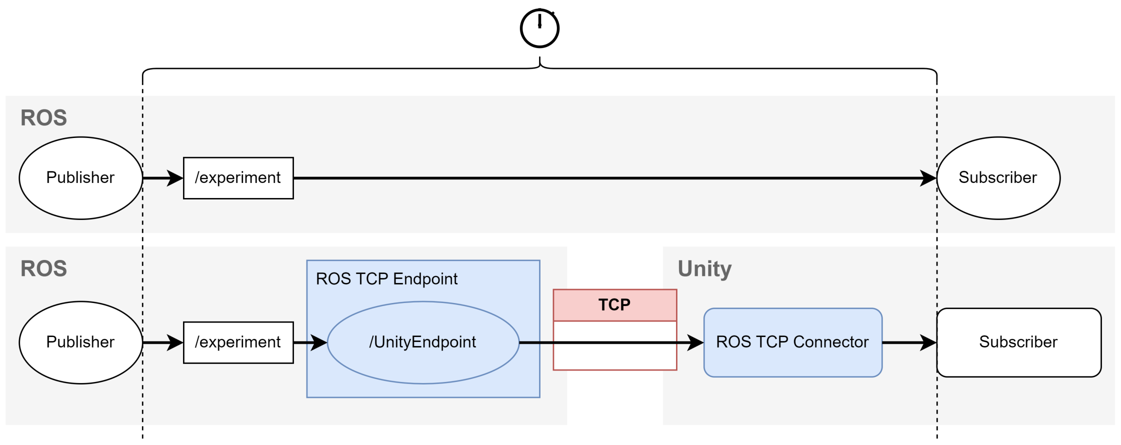


Figure 6.1 Illustration of the TCP endpoint latency experiment

Code listing 6.1 Message interface with a payload size 32 Bytes

```
float64 timestamp
byte[32] payload
```

In Unity, the frame rate is not limited by default. To create a similar environment for experiments as in Gazebo, a simple component has been created to monitor the frame rate, as discussed in Section 5.6.4.4. The metric has been recorded the same way as for Gazebo.

6.2 Scenarios

Several performance experiments have been performed to evaluate the performance of Gazebo and Unity. Because the immediate goal of the selected simulator is a multi-robot simulation, the primary test scenario is a multi-agent performance test with RR1 robotic arms. Before this test, a set of task-unrelated tests were done to test the general performance and scalability of the two simulation tools. The scenarios are described in the following subsections.

6.2.1 TCP endpoint latency

As discussed in Section 5.6.1, the simulation prototype in Unity is integrated with ROS using the packages provided by the Unity Robotics Hub. The communication between Unity's components and ROS nodes is done via TCP and a single ROS node. Gazebo, on the other hand, communicates with ROS directly. The TCP endpoint latency experiment was performed to measure the latency created by the additional communication step and how it behaves with the increasing size of the messages sent. The test compares the latency of a message sent from ROS to ROS and ROS to Unity, as illustrated in Figure 6.1.

6.2.1.1 Message interfaces

The test uses custom message interfaces generated with a Python script inside the rr1_interfaces ROS package. The messages must also be imported into Unity, so the C# interfaces can be generated. The message consists of a timestamp, which the publisher provides as a time when the message was published and a payload. The payload ranges in size from 2 Bytes to 16 Mebibites. An example interface is shown in Code listing 6.1 with a payload of size 32 Bytes.

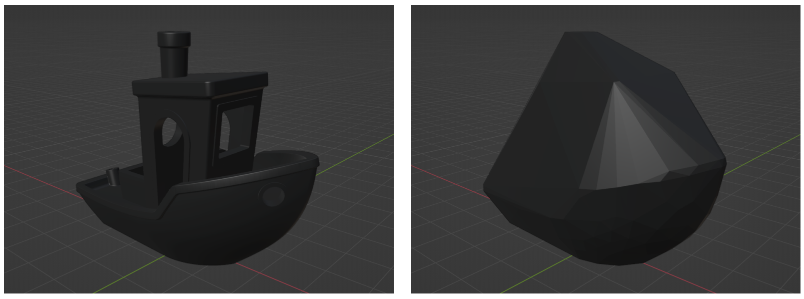

(a) Visual mesh (250,000 triangles) **(b)** Collision mesh (256 triangles)

Figure 6.2 Model of 3DBenchy boat used in experiments

6.2.1.2 Test method

The experiment is performed on Linux only. A new publisher and subscriber nodes are launched for every message size for the ROS to ROS communication. The publisher sends a message of a given size with a timestamp and a payload. When the subscriber receives the message, it logs how long it took for the message to arrive based on the timestamp in the message. For each message size, 1000 messages are sent with sufficient time spacing, so a message is sent after the subscriber has received the previous message.

For the ROS to Unity communication, a publisher node and the ROS TCP Endpoint are launched in ROS. The subscriber runs as a simple Unity component in a clean Unity project with an empty scene. Unfortunately, the project could not run in headless mode due to some build issues. However, no camera was present in the scene, so nothing was being rendered.

6.2.2 Static scene

The static scene experiment tests the general performance of Gazebo and Unity with the increasing complexity of static geometry in the scene. No dynamic objects are in the scene, so no physics simulation is needed.

The test is performed by running a Gazebo or Unity simulation with a static test scene and measuring the selected metrics during the run. Each test configuration creates a more complex scene, ranging from an empty scene with the ground plane to a complex static scene with 125 million triangles.

6.2.2.1 Static objects

A 3D boat model called 3DBenchy, shown in Figure 6.2a, was used to emulate the increasing scene complexity. This model is commonly used for benchmarks of 3D printers and was selected because of its higher complexity than other 3D models used in benchmarks. Its mesh was slightly modified to contain exactly 250,000 triangles, which is also conveniently close to one-half of the complexity of the RR1 model. In this particular experiment, only the visual mesh is used.

Figure 6.2b also shows a collision mesh created for the 3DBenchy model. It was created the same way as the collision meshes for RR1. However, the collision mesh is not used in the static scene experiment.

6.2.2.2 Scene creation

In Unity, the process of scene creation was straightforward. A prefab was created from the boat model, and a simple spawner component was implemented that instantiates a given number of

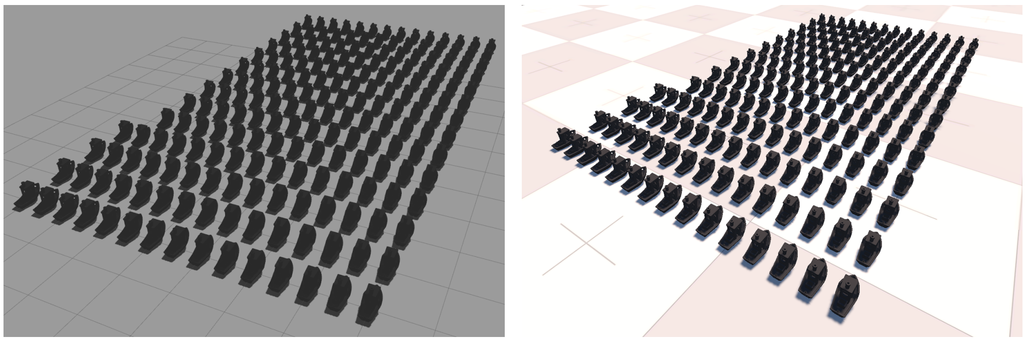

(a) Static scene experiment in Gazebo **(b)** Static scene experiment in Unity

Figure 6.3 Static scene experiment in Gazebo and Unity with 200 static models of 3DBenchy

boat models at the start of the application. The number of boats was given as a command line argument when starting the built application, and the objects are placed in as compact a rectangle formation as possible, as shown in Figure 6.3. Scenes ranging from 0 to 500 boats were tested.

For Gazebo, the same scenes were created and tested. However, the process of scene creation was far more complicated. Gazebo does not have a very flexible scene editor, and for a use case like this, the only possibility is to create an SDF world description programmatically. This is sometimes praised in literature as a great feature, but creating an SDF world description is inconvenient compared to Unity's instantiation process. First, an SDF model of the boat had to be created, which was complicated by Gazebo's crashes. Then a simple scene with the boat was created, which served as a base for a custom world generator Python script, creating an SDF file with multiple copies of the boat's XML description blocks. A Python script was created to orchestrate the test. First, it starts the world generation script, which generates a world description with a given number of boat models. Then it starts Gazebo with the world description file loaded.

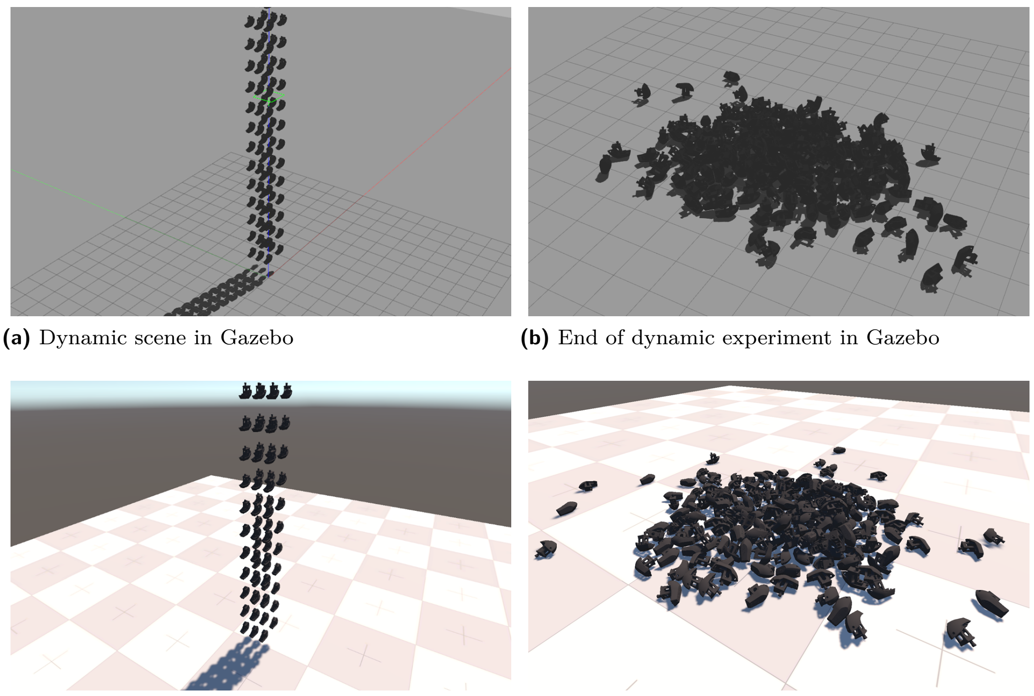
6.2.3 Dynamic scene

The dynamic scene experiment tests the general performance of the two simulators with the increasing complexity of dynamic geometry in the scene. This time, the boat objects are simulated rigid bodies with collision meshes. Therefore, on top of the static geometry that needs to be rendered, the physics engine has to compute the physical forces applied to the objects and resolve collisions between them.

The testing method is similar to the static scene experiment, but the objects are placed in a tall tower formation that falls to the ground when the simulation starts. The start and end of the experiment are shown in Figure 6.4. This tower collapse creates a large number of collisions that the physics engine has to resolve. The measurement of the metrics is manually started after the top of the tower falls onto the heap of fallen objects. Again, scenes with 0 to 500 objects are tested.

6.2.3.1 Dynamic objects

The model of 3DBenchy is used together with the collision mesh shown in Figure 6.2b. The collider was deliberately created in a way that it contains 256 triangles. If the mesh was more complex, Unity would print out warnings into the console that the mesh should be simplified or divided into multiple colliders. Naturally, the same mesh is used for Unity and Gazebo.



(a) Dynamic scene in Gazebo

(b) End of dynamic experiment in Gazebo

(c) Dynamic scene in Unity

(d) End of dynamic experiment in Unity

Figure 6.4 Dynamic scene experiment in Gazebo and Unity with 200 dynamic models of 3DBenchy

6.2.3.2 Scene creation

The scenes in Unity and Gazebo were constructed in the same way as for the static scene experiment, but the models were placed into a tall tower formation. Several constraints were placed on the column creation to ensure some collisions between individual objects. For example, if there are fewer than 20 objects in the test configurations, the tower is constructed with only a single column. As the number of objects increases, the number of tower columns increases up to 6, as shown in Figure 6.4a or Figure 6.4c.

Because the frame rate limit in Gazebo has to be manually modified before the experiment starts, the simulation is paused in the SDF world description. When Gazebo loads it on launch, the simulation time is stopped and can be manually started when the experiment is ready. The same feature was implemented in Unity for convenience. The simulation starts paused and can be unpaused by pressing the spacebar, as in Gazebo.

6.2.4 Multi-robot simulation

The multi-robot simulation experiment is the most relevant benchmark for this thesis. The previous tests in Gazebo and Unity used ROS only to launch Gazebo and retrieve its real-time factor during simulation. In the multi-robot simulation experiment, the ROS infrastructure needed to start the simulation and control the individual robots is launched. One of the test configurations is shown in Figure 6.5.

The individual test configurations spawn multiple instances of the RR1 robot in an empty scene, ranging from 1 to 120 robots. The robots are spawned from the URDF described in Chapter 5. Only the joint trajectory controller is used to control the robot movements, and one controller is started for every single robot instance. The metrics are measured as in the previous

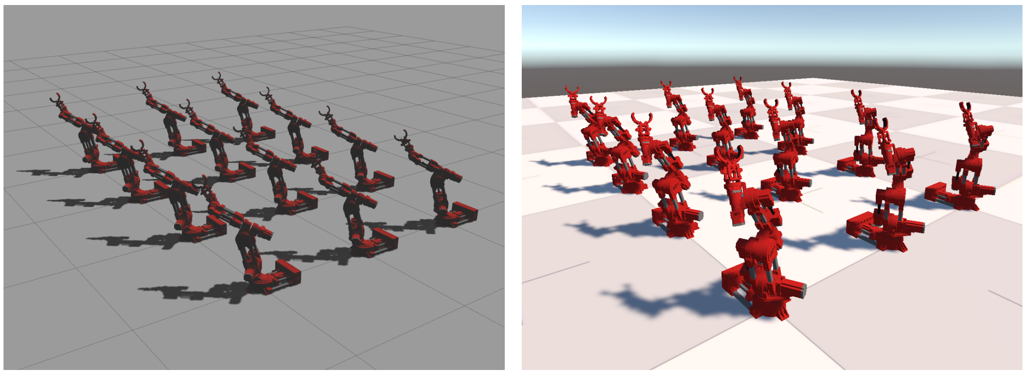

(a) Twelve robots simulated in Gazebo **(b)** Twelve robots simulated in Unity

Figure 6.5 Multi-robot simulation experiment with twelve robots simulated in Gazebo and Unity controlled from ROS

Gazebo and Unity experiments. The measurement starts once all the ROS nodes are started and all simulated robots follow their trajectories.

6.2.4.1 Gazebo

The structure of the Gazebo simulation is explained in Section 5.5. Once Gazebo is started, the robot instances are spawned from the URDF through the robot state publisher node. A separate controller manager with a joint trajectory controller is started for each of them. Each of the robot instances uses its own ROS namespace. Test joint trajectories are published from the nodes implemented in the rr1_control package.

6.2.4.2 Unity on Linux

With Unity on Linux, the simulation is performed as explained in Section 5.6. In Unity, the robotic arms with only the JointTrajectoryController component are spawned using a custom spawner component. On the ROS side, the ROS TCP Endpoint is started to establish communication between Unity and ROS, and nodes from the rr1_control package are started to publish joint trajectories to the individual robot instances.

6.2.4.3 Unity on Windows

The main difference with the experiment on Windows is that ROS runs in a Docker container mentioned in Section 5.6.1.1. Everything else is the same as in the Linux experiment.

6.3 Simulator comparison

During the practical preparation of the prototype and the experiments, features, and characteristics of Gazebo and Unity have been observed and compared. The comprehensive comparison is available in Table 6.2. The table is adapted from [43], where a similar comparison has been made between Gazebo, V-REP, and ARGoS so that the results can be extrapolated to other simulators. The list of characteristics has been extended, and the description of the Gazebo was updated based on the experience from this thesis. The following subsections then highlight and further develop some of the characteristics from the table.

Table 6.2 Comparison between Gazebo and Unity. The table follows a structure from [43] where the comparison has been made between V-REP, Gazebo, and ARGoS. The table is extended, and the Gazebo characteristics are updated based on findings from this thesis.

Gazebo	**Unity**
Supported on Linux and is best used on Ubuntu distribution. Can be run from a VM or Docker container on Windows but without GPU acceleration.	The editor is best used on Windows or macOS. Linux is supported, but issues can be encountered on certain distributions. Applications can be built for any OS.
Built-in capabilities	
The ODE physics engine is available by default. Gazebo can be built from code with a different physics engine.	The Nvidia PhysX (3D) and Box2D (2D) physics engines are available by default. Other engines can be imported as packages or installed as plugins.
Does not include a code editor. External code editing tools have to be used separately.	Does not include a code editor but has a great integration with the most commonly used IDEs.
Includes a scene editor, but is very limited. It is possible to interact with scene objects. However, several basic functionalities are missing, like selecting multiple objects at once.	Includes a feature-rich scene editor, which is very intuitive to use.
The scene can be manipulated during the simulation but does not return to the original state when the simulation is restarted.	The scene can be manipulated from the scene window or hierarchy window in the play mode. When the play mode is exited, all changes roll back.
Has a built-in wall editor, which can be used to build textured walls into the scene quickly. This can be useful for mobile robot simulation.	Walls have to be built and textured manually from resizable cube objects. However, there are packages available that provide dynamic wall creation.
Mesh manipulation is not supported.	Mesh can be fully manipulated programmatically. Minimal mesh manipulation is possible in the editor by default, but some packages are available for complete mesh manipulation.
Simulation outputs include log files or saved video frames as pictures.	No out-of-the-box outputs are included. However, logging and camera capture can be easily implemented.
No particle systems.	Has built-in particle systems.
Has a model editor which can be used to create an SDF model from simple shapes or from imported meshes. It is very unintuitive and sometimes crashes Gazebo.	Has a very mature asset system with drag-and-drop capabilities which flawlessly integrates with other modules in Unity. Prefabs of imported meshes can be created, which can be dragged into the scene or instantiated programmatically.
Animating objects is not possible.	Has a mature built-in animation system that can be used to animate objects in the scene.
VR support is limited to Oculus Rift VR headset using the OculusVR SDK (not tested).	Extensive VR support and commonly used to develop VR applications. Supports a long list of VR headsets.

continues on next page

Can be run in a headless mode.	It should be possible to build headless applications commonly used as game servers. However, some build issues prevented verifying this.
No audio support.	Audio sources can be placed into the scene or generated programmatically, which are then picked up by the camera.
No custom inputs for the user.	Has a robust built-in input system that allows users to create applications with custom controls and controller support.
Robot and other models	
Includes a diverse model library, including various objects, buildings, and robots. The models are pretty simple in complexity.	Does not include a model library. Models can be found on Unity Asset Store and then imported from the package manager into the project. Models vary in their complexity, and not all of them are free.
Custom robot models can be imported from URDF or SDF descriptions.	Custom robot models can be imported from URDF using the URDF Importer package from Unity Robotics Hub.
To add custom object models to the scene, an SDF model must first be created manually by writing its XML code (which did not work properly) or using the model editor inside Gazebo (often crashed).	Mash files can be simply dragged from the file explorer into the project files in the editor and are ready to be used in the scene. On Linux, the files had to be imported through a context menu in the editor because the drag-and-drop did not work.
Imported meshes cannot be changed or optimized. Third-party 3D modeling applications have to be used.	Unity packages are available on Unity Asset Store or GitHub that can be used to optimize a mesh or perform other modifications.
Programming methods	
The scene (world) is saved as an XML file (SDF). Creating scripts to modify the XML file and then run simulations is possible.	The scene is saved in Unity's custom scene file format, which is saved as an asset in the project. Scenes can be divided into subscenes and loaded dynamically. The file format makes it hard to version with standard version control tools. However, Unity provides a robust API that makes it possible to construct and modify scenes programmatically, so there is no need for external scripts.
Lacks scripting capabilities. Functionality can be added only as compiled C++ plugins. Additionally, some functionality can be added with ROS applications.	Scripting is possible in C# language, which allows quick testing. Unity provides an API that also makes it possible to extend the editor's functionalities and create plugins or UI elements.
Visual scripting is not supported.	Visual scripting is supported.
No analysis tools for developers.	Provides multiple analysis tools, including profiler, physics and frame debuggers, and more.
User interface (UI)	

continues on next page

The GUI application often froze and sometimes crashes the whole Gazebo, especially when using the model editor or importing URDF models.	No freezing issues were experienced. On Linux, the editor printed an error into the console because of a known issue, but the performance and usability were unaffected.
The UI capabilities are relatively limited. Some issues were encountered, especially with object manipulation in the scene and changing properties.	The UI is intuitive for people with some experience with game engines, and the UI layout is fully configurable. No issues were encountered.
The model library is not distributed with the application but is downloaded on demand. The user has to wait until the model library loads the list of models.	Unity does not have a model library. However, when models are imported from the package manager or copied to the project, they are always available in the assets folder.
The model library is a long list of models that is hard to navigate.	The user can structure the assets folder of a Unity project however they want to improve the navigation.
Community and documentation	
A fairly comprehensive documentation with some step-by-step tutorials.	An extensive documentation that is easy to navigate and read. Unity provides a lot of official courses and tutorials [107], but the community creates a lot of tutorial content as well.
Gazebo seems to have a large community, but it is distributed over too many communication canals.	Unity has a massive community that is concentrated around the Unity Forum. The forum has a subforum specifically for robotics.

6.3.1 Scene manipulation

Both tools have a scene editor, where individual objects in the scene can be manipulated. Objects can be added, removed, moved, scaled, or rotated. Unity provided a significantly more feature-rich and intuitive experience. Gazebo was missing several important features that made using the scene editor inconvenient. For example, multiple objects cannot be selected at once. In Unity, multiple objects can be selected and manipulated simultaneously.

In Gazebo, it is possible to manipulate the scene during simulation. However, the scene does not return to its original state when the simulation is restarted. In Unity, the scene can also be modified inside the scene or hierarchy window in the play mode. Once the play mode is exited, all changes roll back to the original state.

6.3.2 Adding custom models

Adding custom models to the Unity project is relatively easy. On Windows, the mesh files can be dragged from the file explorer into the Unity Editor's project explorer or the assets folder in the file system. This drag-and-drop feature did not work on Linux, and files had to be imported using a context menu inside the editor. Once the meshes are in the assets folder, they can be dragged into the scene and used or instantiated programmatically.

In Gazebo, the process is more complicated. A mesh file of an object cannot be used directly. First, a model has to be created, which is an SDF file that describes the model and its meshes. This should be possible to do outside of Gazebo, but it did not work as expected. Fortunately, Gazebo's model editor can be used to import the meshes and save the model in SDF format. However, this process is unintuitive, and Gazebo crashed several times in the model editor. Once the model is created, it has to be imported into Gazebo by providing a path to its directory, and then it can be instantiated into the scene.

6.3.3 VR capabilities

The VR support for Unity is extensive, and Unity is commonly used to develop VR applications. A considerable portion of Unity's community also works with VR, providing assets and learning materials. The VR support was successfully tested on this prototype as mentioned in Section 5.6.6. The support was tested on Oculus Quest 2 VR headset.

Gazebo states VR support in its documentation using the OculusVR SDK and is only limited to Oculus Rift VR headset. This feature was impossible to test as only the Oculus Quest 2 headset was available for development.

6.3.4 Application issues

Gazebo experienced multiple issues with the GUI application. The application froze several times, especially when using the model editor or importing a robot using URDF, sometimes completely crashing the application. No such issues were experienced with Unity.

6.3.5 Community support

Unity has a massive community centered around the Unity Forum, where anyone can ask questions relevant to Unity development. The community also creates and shares a lot of tutorials and free assets. Although the community of robotic enthusiasts using Unity is less predominant, a separate subforum on Unity Forum is dedicated to robotics. An additional channel, more suited for shorter questions rather than more extended discussions, is Uniy Answers.

Gazebo, together with ROS, also has a relatively large community. However, the communication is distributed over many channels, and it is unclear what the primary hub is. There are Gazebo Answers, Gazebo Community, ROS Answers, ROS Discourse, and Robotics Stack Exchange, where Gazebo Answers and ROS Answers seem to be the main hubs for getting support and asking questions.

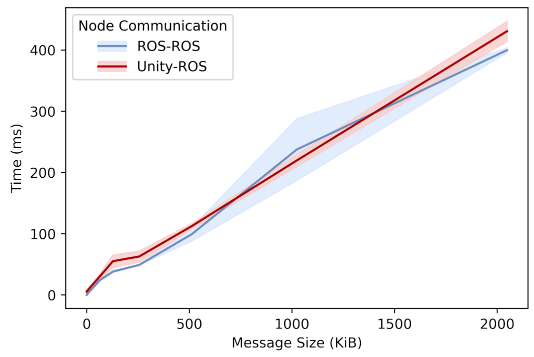
From the interactions on these forums, Unity did subjectively better. Four threads were created in the Robotics subforum on Unity Forum, and all were answered or sparked a more extensive discussion. Some additional questions were asked in other subforums, leading to discussions. However, more people visit those subforums, so getting an answer is more expected. Five threads were created on Gazebo Answers and ROS Answers, and only one was answered.

6.4 Performance evaluation

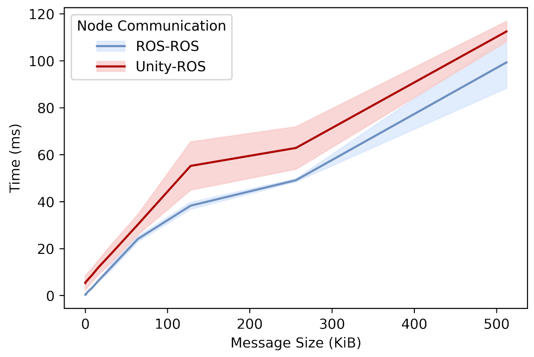
This section presents the results of the experiments discussed in Section 6.2. All the recorded metrics have been plotted, and the reader can find all the figures in Appendix B. In this section, only the most relevant or discussed plots are shown.

6.4.1 TCP endpoint latency

The TCP endpoint latency experiment showed that the added layer in communication between ROS and Unity does add some latency overhead. As shown in Figure 6.6a, the latency overhead is not increasing with the message size but stays relatively constant. The latency might be considered when working with sensors that generate large amounts of data, like cameras. However, for the use case of this prototype, where the message sizes are not very big, the added latency is negligible. Figure 6.6b shows the latency for smaller message sizes.



(a) Complete results of the experiment scenario



(b) Results for small message sizes

Figure 6.6 Results of the TCP endpoint latency experiments

Table 6.3 Summary of the most important results from the general performance experiments (number of objects is in the brackets)

Simulator	Average CPU (>=20)	RAM (10)	RAM (100)	VRAM (10)	VRAM (100)	FPS (10)	FPS (100)
Gazebo	16.34%	1.88 GiB	2.12 GiB	579 MB	584 MB	558	71
Unity (Linux)	9.93%	1.65 GiB	1.65 GiB	609 MB	721 MB	488	59
Unity (Windows)	3.03%	4.84 GiB	4.87 GiB	449 MB	458 MB	562	62

6.4.2 General performance

The results of the general performance tests in Gazebo and Unity are divided into several subsections based on the measured metrics. The CPU and GPU temperatures were disregarded as meaningful metrics because the temperature was allowed to decrease between individual test runs, and the tests were only running for a short time so that the temperature could truly reflect the load on the hardware. However, some interesting observations can be made in the temperature plots shown in Appendix B. The GPU temperature during Gazebo runs is considerably higher than in the Unity runs. Although it might not be the only reason, it is believed that this is the result of the Gazebo running for a short while before the measurements were captured, as the camera frame limit has to be changed manually before every experiment. The full experiment results are plotted in Figures B.3, B.4, B.5, and B.6. Additional tests, where the simulation time in Unity was scaled down, are explained in Section 6.4.2.5, and the measurements are plotted in Figures B.7, B.8, B.9, B.10, B.11, and B.12.

6.4.2.1 Real-time factor

The real-time factor measured in the two experiments in Gazebo shows how it is only affected by the complexity of the physics simulation in the dynamic scene. As shown in Figure 6.7 and Table 6.5[2], Gazebo will decrease the real-time factor to accommodate this complexity and give the physics engine more time to perform the computations.

[2]Only a selection of the test configurations is listed in the table to fit the page.

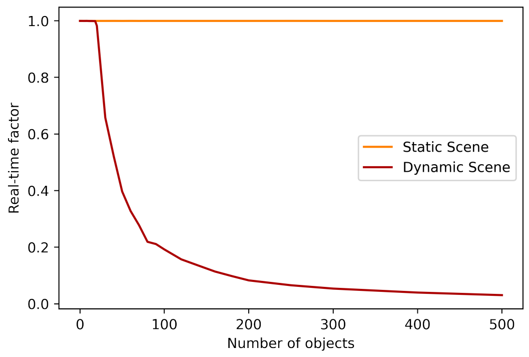


Figure 6.7 Effects of static and dynamic scene complexity on the real-time factor in Gazebo

Table 6.5 Gazebo's real-time factor relative to number of simulated objects in a falling tower

Number of objects	0	10	20	50	100	200	500
RTF	1.00	1.00	0.98	0.40	0.19	0.08	0.03

6.4.2.2 Frame rate

Figure 6.8 shows the measured frame rate during the static and dynamic scene experiments in Gazebo and Unity. Interestingly, the performance of the two simulators is almost identical. Unity starts at a much higher frame rate with small scene complexity but drops quickly as the scene gets more complex and then follows the Gazebo's trajectory. When the difference in frame rate between Gazebo and Unity is plotted, shown in Figures 6.8c and 6.8d, it seems that Unity is a bit less performant than Gazebo. Nevertheless, the difference is negligible.

Another interesting finding is that the difference between a static and a dynamic scene does not affect the frame rate very much. This was expected in Gazebo, as the GUI and simulation processes are separated. Nevertheless, it was expected to affect Unity's performance. In this case, it seems that the physics simulation is not as CPU-demanding to slow down the frame rate of the whole application.

6.4.2.3 CPU and GPU utilization

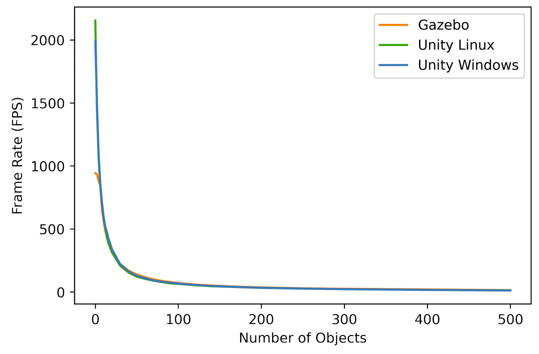
The CPU performance was recorded per each logical processor (threads). Figure 6.9 shows the utilization of the four most loaded CPU threads. Gazebo shows expected behavior. Only the GUI process fully utilizes a single thread during the static scene test. Once physical objects in a dynamic scene are simulated, the Gazebo's process responsible for simulation starts fully utilizing another thread. In the plot showing the overall CPU utilization, shown in Figures B.4a and B.4b, this seems like the dynamic scene doubles the CPU utilization.

Unity tested on Linux fully utilizes a single thread, as was expected. However, on Windows, the CPU utilization is very low. This was expected to be an issue in the psutil library on Windows, but the result was confirmed in the performance tab of the Windows Task Manager. It seems that on Windows, Unity's load is very well balanced to all the available cores.

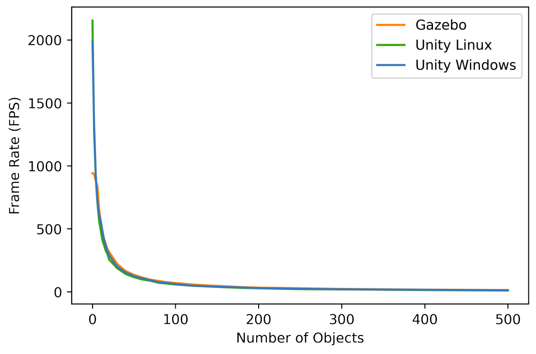
Gazebo and Unity utilize the GPU fully, regardless of the complexity of the scene. Although Gazebo seems to utilize the GPU less in the dynamic scene, as shown in Figures B.4c and B.4d.

6.4.2.4 RAM and VRAM usage

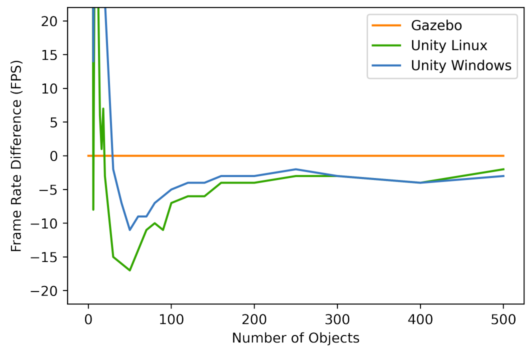
The RAM and VRAM usage show more interesting results, as shown in Figure 6.10. The Unity on Windows experiments shows high RAM usage compared to the tests performed on Linux



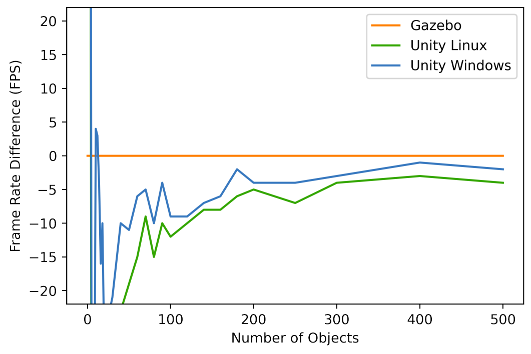
(a) Frame rate (static scene)



(b) Frame rate (dynamic scene)

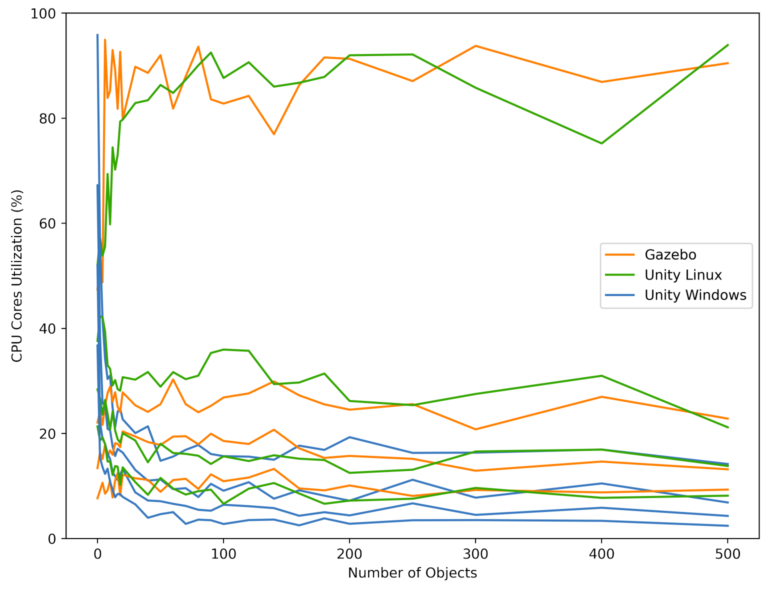


(c) Frame rate difference (static scene)

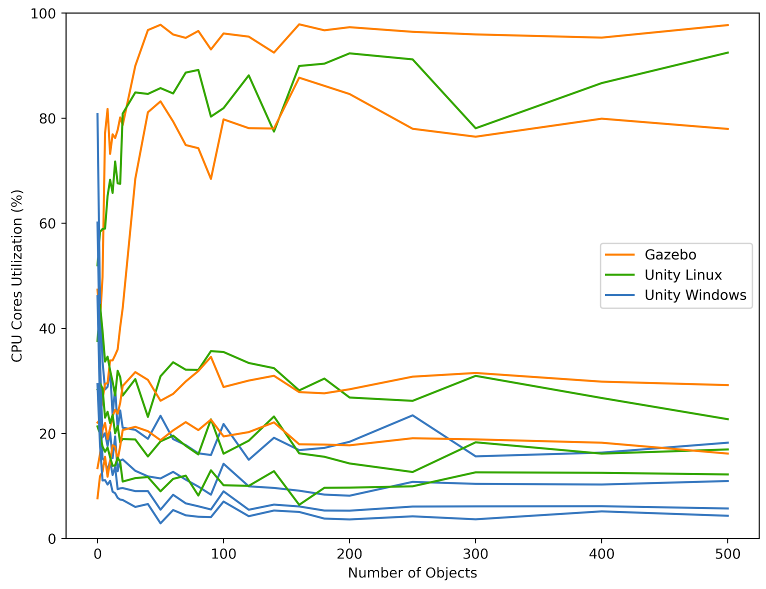


(d) Frame rate difference (dynamic scene)

Figure 6.8 Frame rate measured during the static and dynamic scene experiments with a difference compared to Gazebo's frame rate

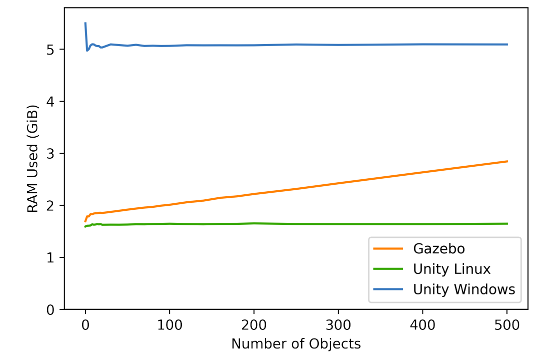


(a) CPU cores utilization (static scene)

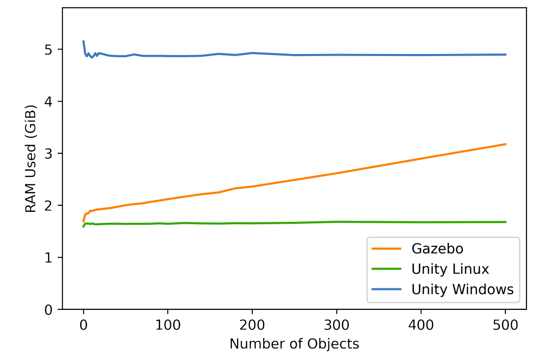


(b) CPU cores utilization (dynamic scene)

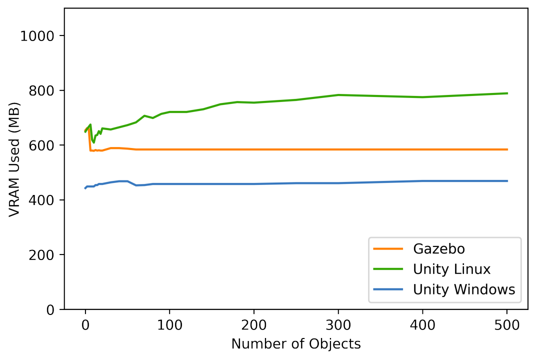
Figure 6.9 CPU utilization measured during the static and dynamic scene experiments



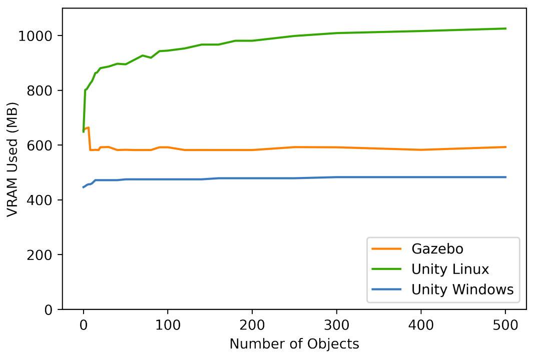
(a) RAM used (static scene)



(b) RAM used (dynamic scene)



(c) VRAM used (static scene)



(d) VRAM used (dynamic scene)

Figure 6.10 RAM and VRAM usage meassurements during the static and dynamic scene experiments

because Windows alone uses more RAM. Without the simulation, Linux used around 1.5 GB of RAM and Windows around 4.8 GB. The results show that with increasing scene complexity, Gazebo increases its RAM consumption linearly, unlike Unity which keeps it constant.

In the case of VRAM usage, Unity tested on Linux showed some increase, but Gazebo and Unity on Windows stayed constant. However, the curve in Figure 6.10d seems to be affected by some other process, as additional experiments with the dynamic scene showed a curve more similar to that in Figure 6.10c. Nevertheless, the increase in VRAM usage is present. This different behavior compared to Windows may be caused by different graphical APIs being used.

6.4.2.5 Scaled time in Unity

An additional set of experiments was performed to check if Unity's performance does not benefit from scaled-down time. The dynamic scene experiments were rerun in Unity, both on Linux and Windows. However, the time was scaled down by the RTF measured in Gazebo for the same test configuration. The results are plotted in Figures B.7, B.8, and B.9, where the results are compared with Gazebo, as in the previous cases. In Figures B.10, B.11, and B.12, a direct comparison between the Unity runs is made, with scaled time and without scaled time. The tests showed that scaling the time down did not affect the performance[3].

However, scaling the time down improved the quality of the physics simulation significantly. As Unity does not do this independently, unlike Gazebo, the physics simulation takes shortcuts

[3]Note that as the results for Linux showed no impact on performance, only a few test configurations were tested to verify the same expected behavior on Windows.

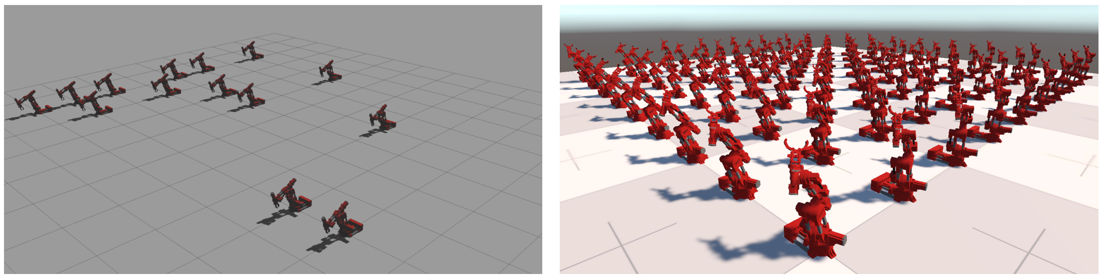

(a) Eighty robots simulated in Gazebo **(b)** Eighty robots simulated in Unity

Figure 6.11 Multi-robot simulation experiment with eighty robots simulated in Gazebo and Unity controlled from ROS, where Gazebo was unable to spawn all the robots and their controllers

to meet the real-time execution criterion. When the user scales down the time, the collisions between objects are resolved with more accuracy, which was visually apparent in the simulation.

6.4.3 Multi-robot simulation

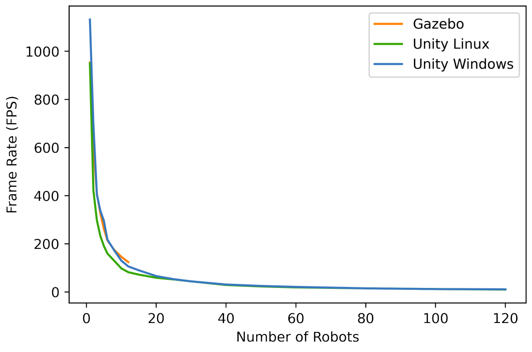
The multi-robot simulation experiments, where up to 120 robots were simulated, were expected to yield similar results as the dynamic scene tests, as the robots participate in the physics simulation. Although this seems to be the case, a few differences in the performance results should be pointed out. Also, some issues were observed that need to be addressed, like Gazebo's problem of spawning many robots and the desynchronization of ROS nodes. All plottet meassurements can be find in Figures B.13, B.14, and B.15.

6.4.3.1 Gazebo limits

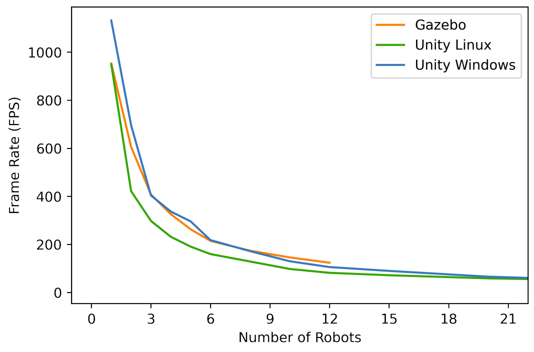
During the Gazebo experiments, a problem was encountered where Gazebo could not spawn more than 12 robots. When a test configuration with a little over 12 robots was tested, Gazebo would usually spawn only 12 robots. The robots not spawned successfully were always random instances from the whole set. Sometimes, even the 12 robots test would be inconsistent, with one robot missing or spawned without its controllers. When the number of robots in the test configuration was large, Gazebo usually spawns more than 12 robots, as shown in Figure 6.11a, where 14 robots out of 80 were spawned. However, the spawned robots usually do not start their controllers successfully in such cases. On the other hand, Figure 6.11b shows the same experiment in Unity, where all 80 robots were spawned and are executing a trajectory.

6.4.3.2 ROS node desynchronization

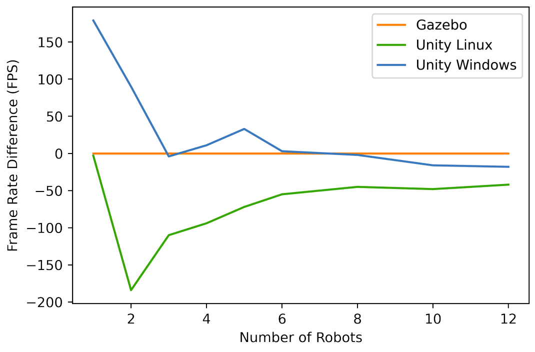
One issue arose with the increasing number of simulated robots. The test trajectories are sent from simple ROS nodes implemented in the rr1_control package, which sends one trajectory to the controller every six seconds. The first trajectory is also sent after six seconds after the node starts. However, when many nodes are started, ROS starts them sequentially, taking some time. Because there is no synchronizing mechanism, some nodes send trajectories sooner than others. With large numbers of nodes, it can also happen that some nodes will send trajectories before all of the nodes are successfully started. Therefore in a real scenario, the nodes should wait until after all the nodes are started successfully and then synchronize before sending trajectories.



(a) Frame rate (multi-robot)



(b) Frame rate (multi-robot, zoomed)



(c) Frame rate difference (multi-robot)

Figure 6.12 Frame rate measured during the multi-robot simulation experiments with a difference compared to Gazebo's frame rate

6.4.3.3 Performance

With the limited measurements from Gazebo experiments, the performance is similar to the dynamic scene experiments. However, as shown in Figure 6.12, Unity on Linux performs worse than Gazebo and Unity on Windows, with a considerably lower frame rate for the simulation of a few robots.

The other differences are in the RAM usage. During Unity experiments on Linux, the RAM consumption increases with the number of robots as ROS nodes are being started in addition to the Unity simulation. As shown in Figure 6.13, a large amount of RAM is consumed during Windows experiments because of Docker. Even though the Docker container did not need this much RAM, the WSL 2 backend used for Docker uses the available RAM by default. This behavior can be configured, however.

Table 6.6 Summary of the most important results from the multi-robot simulation experiments (number of simulated robots is in the brackets)

Simulator	Max robots	FPS (1)	FPS (3)	FPS (10)	FPS (100)	Min CPU	Max CPU
Gazebo	12	606	326	146	NA	11.15%	16.96%
Unity (Linux)	120	422	232	59	12	9.50%	17.59%
Unity (Windows)	120	696	337	66	12	5.32%	83.51%

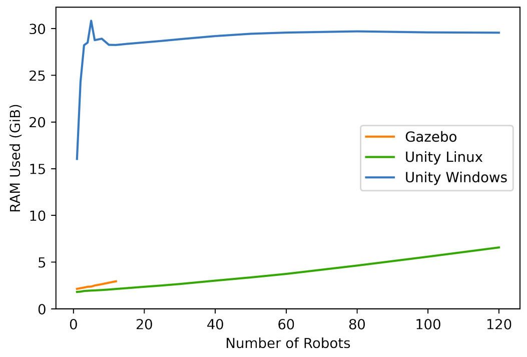


Figure 6.13 RAM usage meassurements during the multi-robot simulation experiments

6.5 Summary of results

Overall, the performed experiments showed that the performance of Unity and Gazebo is very comparable when using their default rendering and physics engines with the default configurations. On the other hand, Unity is more configurable than Gazebo regarding physics simulation settings and has robust rendering pipelines that are interchangeable, scriptable, and configurable. Unfortunately, in the multi-robot simulation experiment, Gazebo could not spawn more than 12 robots, which was a significant downfall, rendering it unusable for multi-robot simulation with many robots. Not to mention how Unity compares to Gazebo in usability and how Gazebo experienced severe issues, including crashes that Unity did not have.

Regarding the features and characteristics, the results in Section 6.3 revealed that Gazebo is inferior to Unity in both usability and available features. Unity is a feature-rich tool that is very versatile and extendable through packages. On the other hand, Gazebo is very limited, being a robotics simulation tool. What Gazebo is also missing, compared to Unity, is a range of analytics tools and optimization methods that can be used to enhance the performance in certain use cases or specific types of scenes. Importantly for the multi-robot simulation of RR1, Unity provided better scene editing and scripting capabilities that made spawning a large number of robots inside a scene very straightforward.

Conclusion

This work has contributed to the development of the robotic arm RR1 by producing a ROS backend with control capabilities for the arm and a digital twin prototype that allows simulation. The robot was used in a multi-robot simulation in Gazebo and a prototype simulation created in Unity, which resulted in a direct comparison between the two simulators and a set of performance experiments.

In Chapter 1 and Chapter 2, the reader was familiarized with the theoretical background and technologies used in this thesis. Chapter 3 introduced the faculty-developed robotic manipulator RR1 with an in-depth description. Chapter 4 discussed related work and provided a rationale for the prototype development and chosen technologies. In Chapter 5, the prototype was realized, including developing the ROS backend for RR1, Gazebo simulation, and Unity simulation prototype. Lastly, Chapter 6 compared the two simulations and evaluated the performed experiments.

The prototype simulation showed that it is possible to integrate Unity with ROS and use it for multi-robot simulation with many robotic arms. The prototype alone could be used for a visual demonstration of the robot or concepts of motion planning in academia and could be extended for specific research use cases.

Review of the thesis aims

All of the subtasks set in this thesis's introduction were successfully done. The subtasks were the following:

- Develop a ROS backend for RR1 robotic arm and create a digital twin prototype.
- Create a simulation prototype of the digital twin in Gazebo and Unity with ROS integration.
- Compare the two tools and perform experiments assessing their performance in multi-robot simulation.

The first two tasks were done in Chapter 5. The URDF robot description for RR1 has been developed, and the robot control was implemented using the ros2_control package, minimizing the work needed when migrating to the physical robot prototype. With the robot description done, the Gazebo simulation could be performed as described in Section 5.5. Lastly, a simulation prototype was created in Unity, verifying the possibility of Unity-ROS integration and Unity's usability for multi-robot simulation. This process is described in Section 5.6. The last goal of the thesis, the performance experiments and comparison between Gazebo and Unity, was performed in Chapter 6. The feature and characteristics comparison was made in Table 6.2 and further elaborated in Section 6.3, and the results of the performance experiments were summarized in Section 6.5.

Future work

The prototype showed that using Unity for a multi-robot simulation is possible. However, it can be improved or serve as a base for a specific simulation use case. The simulation performance can also be improved by creating a simplified model of RR1. Additionally, some more significant additions and research topics are possible:

- Simulating the gripper in a pick-and-place simulation.
- Research of multi-robot motion planning algorithms for robotic arms.
- Synchronizing the ROS control of robotic arms with many simulated instances.

Transfer into practice

The prototype developed in this work has already been used to demonstrate the RR1 robotic arm at a faculty-hosted career fair. Its expected transfer further into practice is becoming the primary software, both the simulator and ROS control backend, for the faculty-developed manipulator RR1. It will aid the completion of the second prototype of the robot and make the development of the subsequent prototypes quicker. Then, together with the robotic hardware, it is expected to become educational equipment for robotics courses in academia and enable large-scale multi-robot motion planning research for robotic manipulators, which is the central vision for RR1.

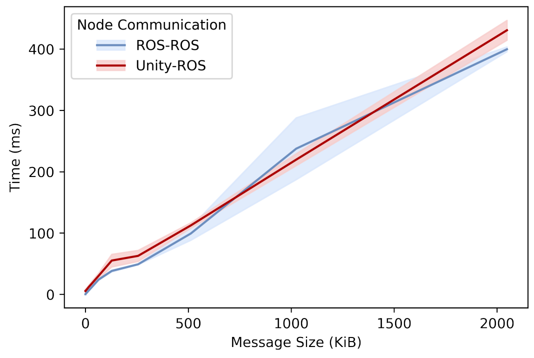
Appendix A
Large figures

Table A.1 Definitions of digital twin characteristics proposed in [15] (Table reprinted from [15])

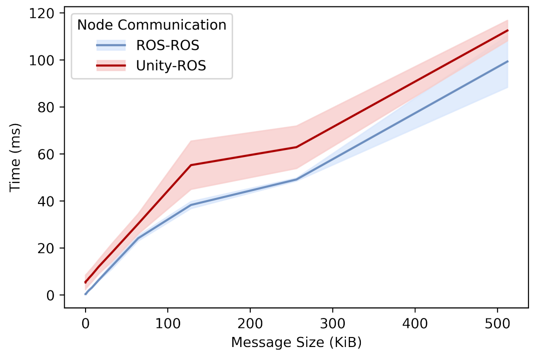
Characteristic	Definition
Physical Entity/Twin	The physical entity/twin that exists in the physical environment
Virtual Entity/Twin	The virtual entity/twin that exists in the virtual environment
Physical Environment	The environment within which the physical entity/twin exists
Virtual Environment	The environment within which the virtual entity/twin exists
State	The measured values for all parameters corresponding to the physical/virtual entity/twin and its environment
Metrology	The act of measuring the state of the physical/virtual entity/twin
Realisation	The act of changing the state of the physical/virtual entity/twin
Twinning	The act of synchronising the states of the physical and virtual entity/twin
Twinning Rate	The rate at which twinning occurs
Physical-to-Virtual Connection/Twinning	The data connections/process of measuring the state of the physical entity/twin/environment and realising that state in the virtual entity/twin/environment
Virtual-to-Physical Connection/Twinning	The data connections/process of measuring the state of the virtual entity/twin/environment and realising that state in the physical entity/twin/environment
Physical Processes	The processes within which the physical entity/twin is engaged, and/or the processes acting with or upon the physical entity/twin
Virtual Processes	The processes within which the virtual entity/twin is engaged, and/or the processes acting with or upon the virtual entity/twin

Appendix B

Experiment plots



(a) Complete results of the experiment scenario



(b) Results for small message sizes

Figure B.1 Results of the TCP endpoint latency experiments

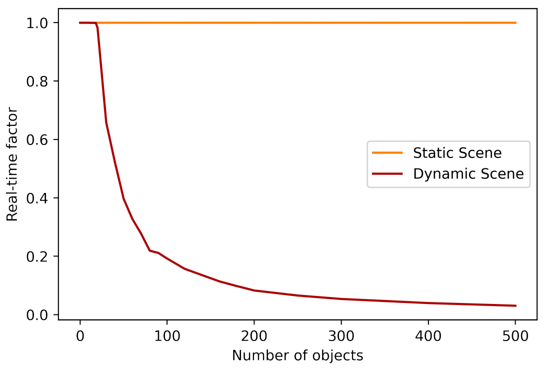
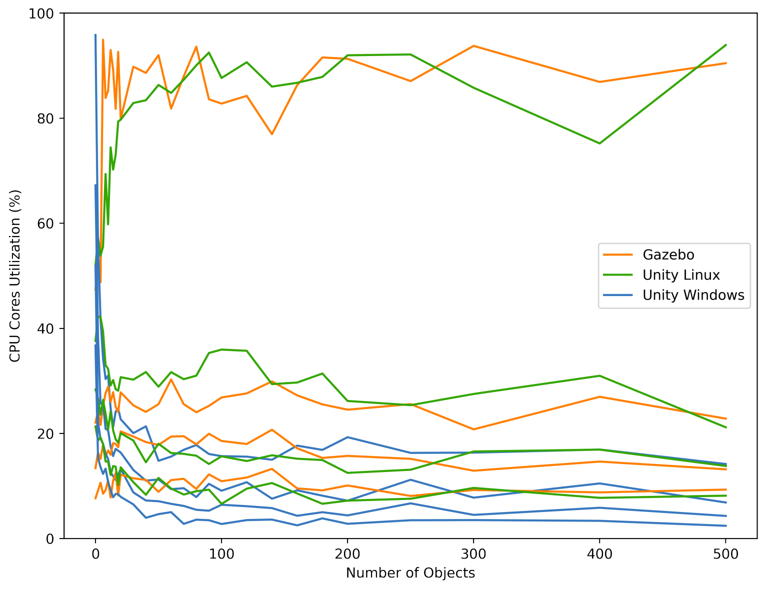
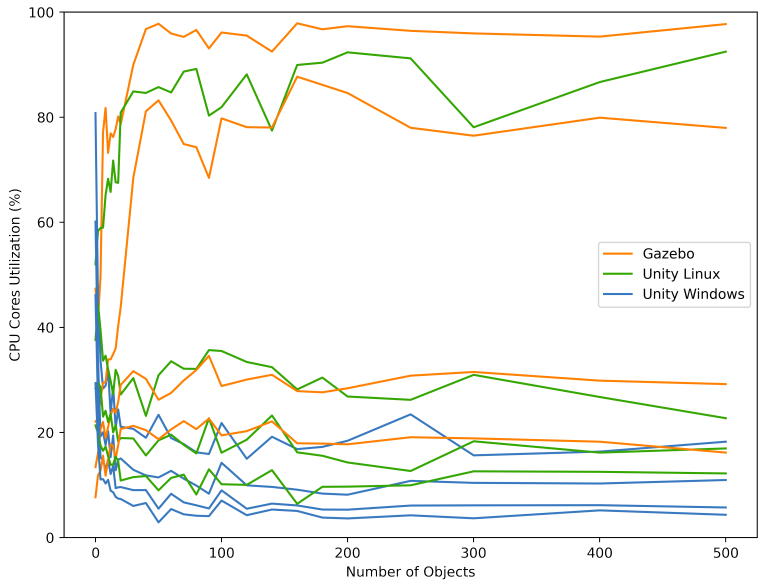


Figure B.2 Effects of static and dynamic scene complexity on the real-time factor in Gazebo

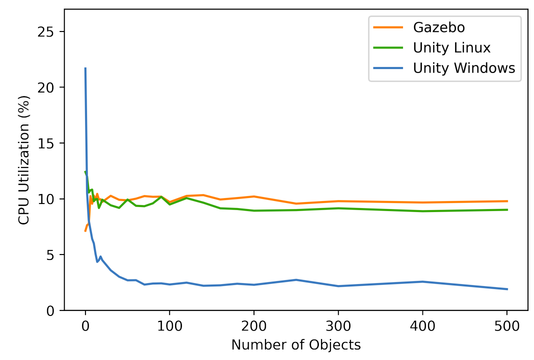


(a) CPU cores utilization (static scene)

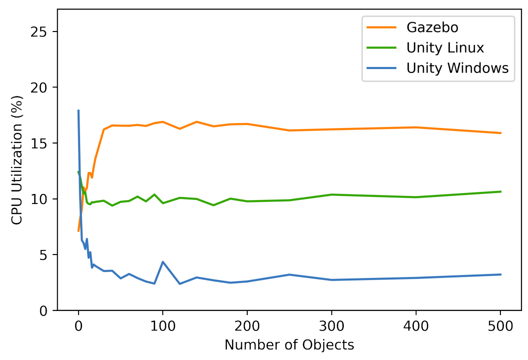


(b) CPU cores utilization (dynamic scene)

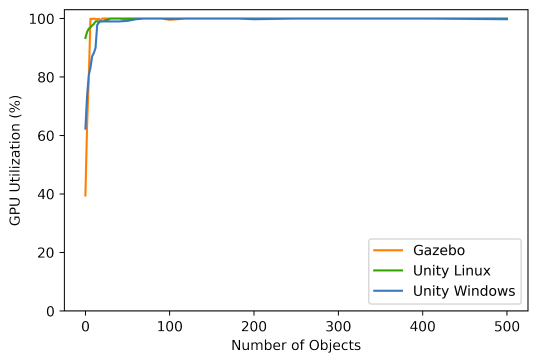
Figure B.3 CPU cores utilization measured during the static and dynamic scene experiments



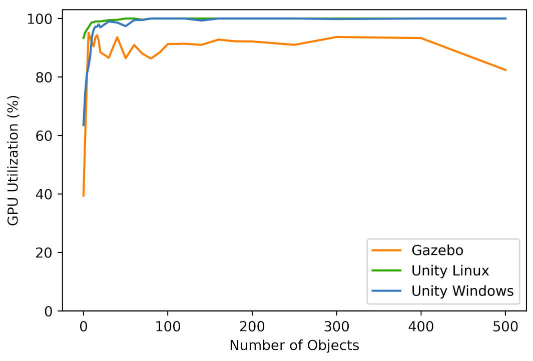
(a) CPU utilization (static scene)



(b) CPU utilization (dynamic scene)

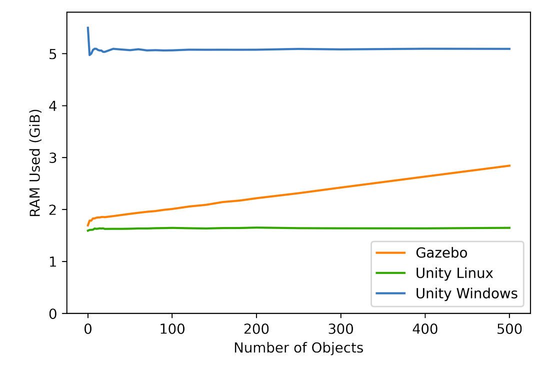


(c) GPU utilization (static scene)

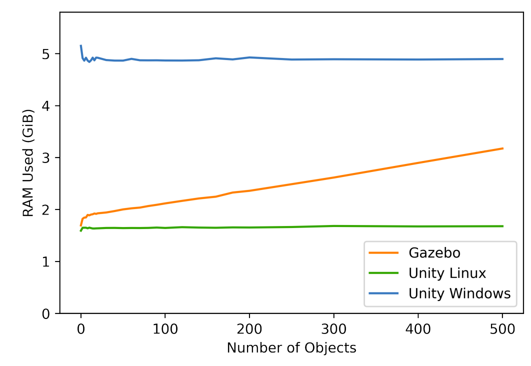


(d) GPU utilization (dynamic scene)

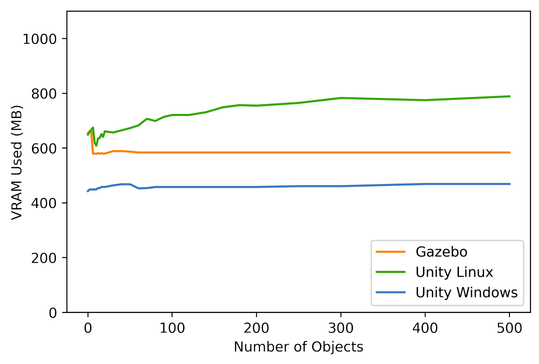
Figure B.4 CPU and GPU utilization measured during the static and dynamic scene experiments



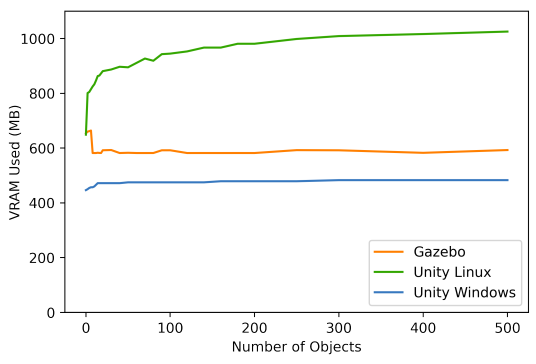
(a) RAM used (static scene)



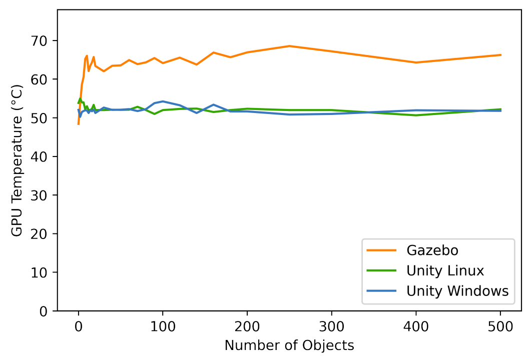
(b) RAM used (dynamic scene)



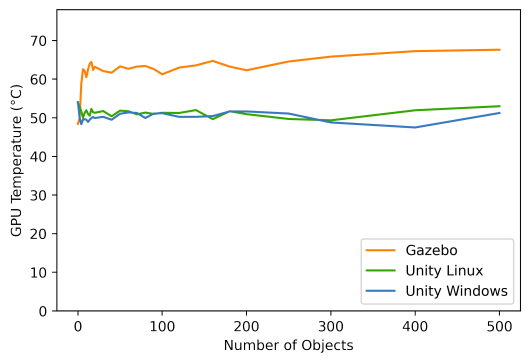
(c) VRAM used (static scene)



(d) VRAM used (dynamic scene)

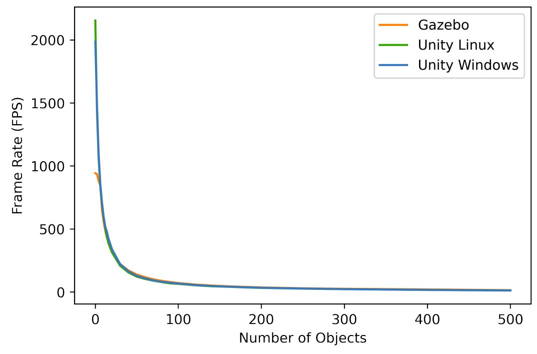


(e) GPU temperature (static scene)

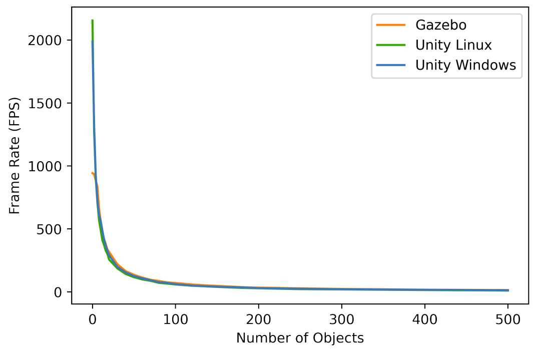


(f) GPU temperature (dynamic scene)

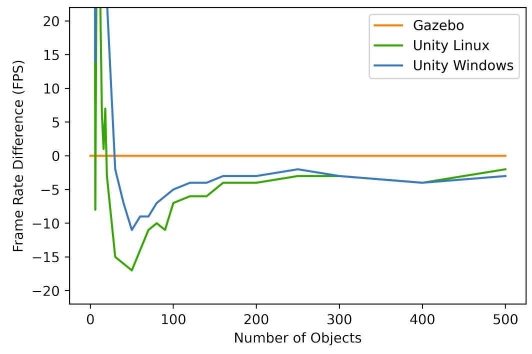
Figure B.5 RAM and VRAM usage meassurements and GPU temperature during the static and dynamic scene experiments



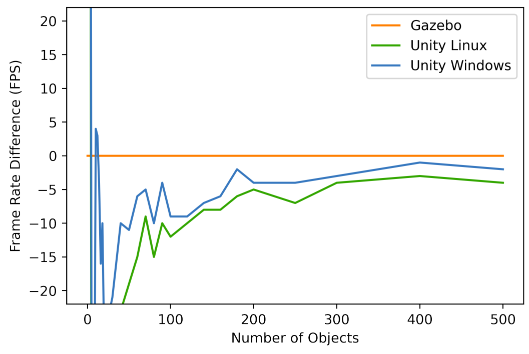
(a) Frame rate (static scene)



(b) Frame rate (dynamic scene)

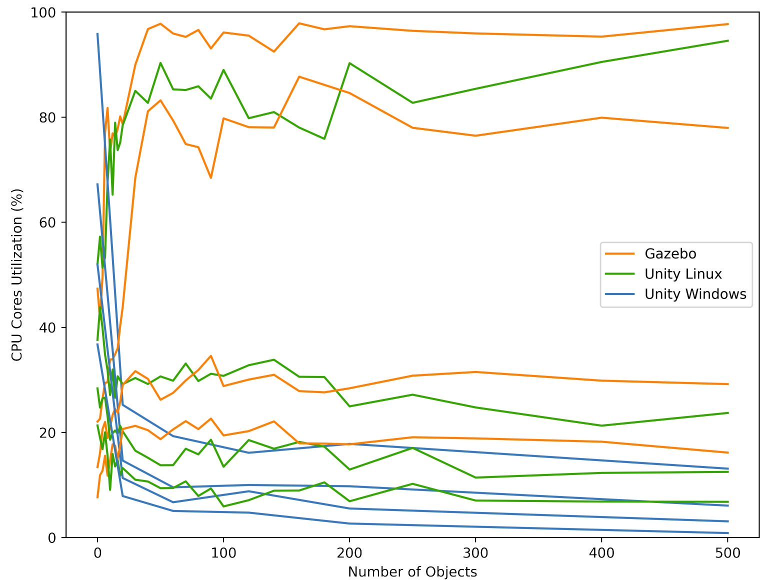


(c) Frame rate difference (static scene)

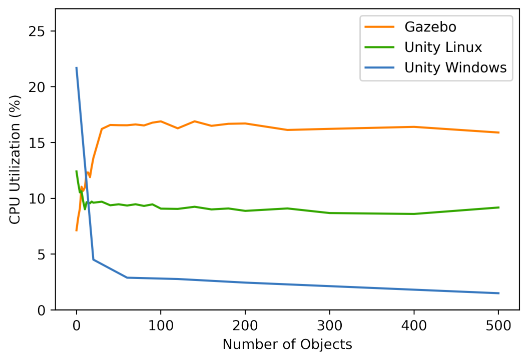


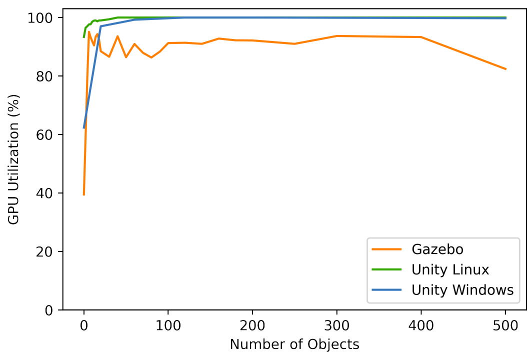
(d) Frame rate difference (dynamic scene)

🟦 **Figure B.6** Frame rate measured during the static and dynamic scene experiments with a difference compared to Gazebo's frame rate



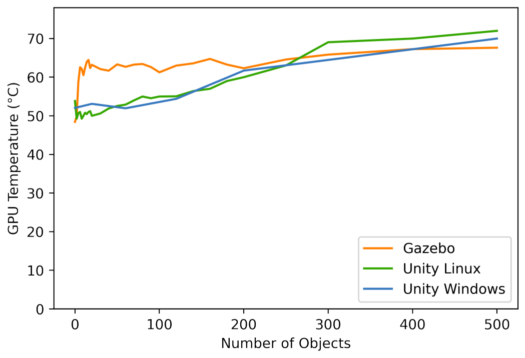
(a) CPU cores utilization (dynamic scene, scaled time)



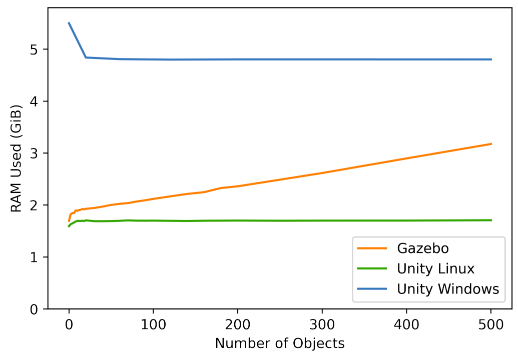


(b) CPU utilization (dynamic scene, scaled time) **(c)** GPU utilization (dynamic scene, scaled time)

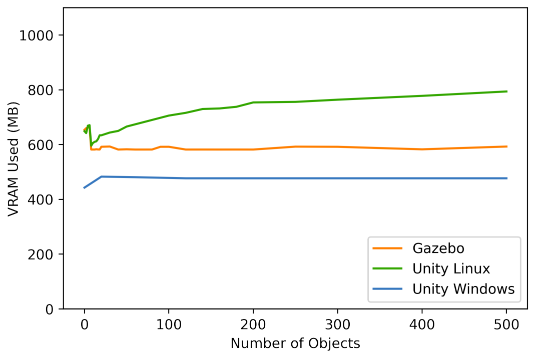
Figure B.7 CPU and GPU utilization measured during the dynamic scene experiment with Unity scaled down the time based on meassured RTF in Gazebo



(a) GPU temperature (dynamic scene, scaled time)

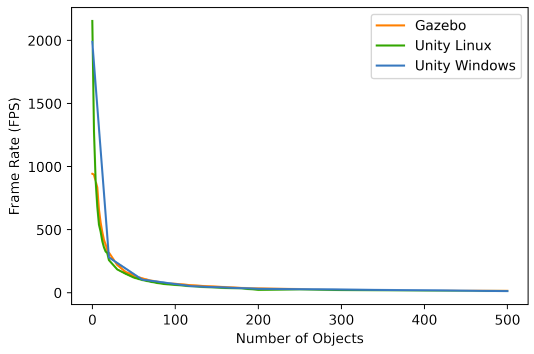


(b) RAM used (dynamic scene, scaled time)

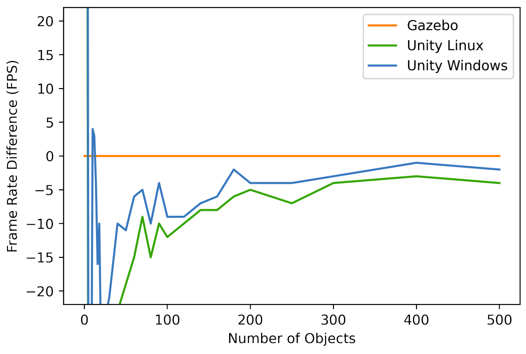


(c) VRAM used (dynamic scene, scaled time)

Figure B.8 RAM and VRAM usage meassurements and GPU temperature during the dynamic scene experiment with Unity scaled down the time based on meassured RTF in Gazebo

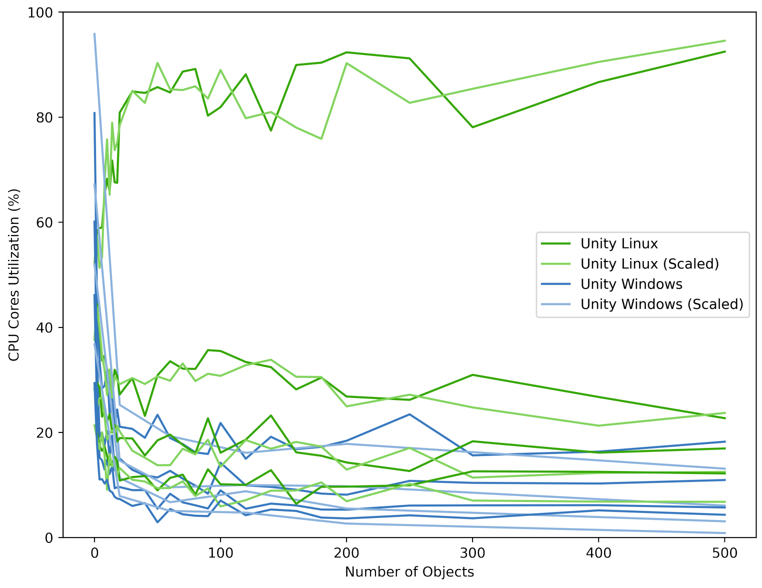


(a) Frame rate (dynamic scene, scaled time)

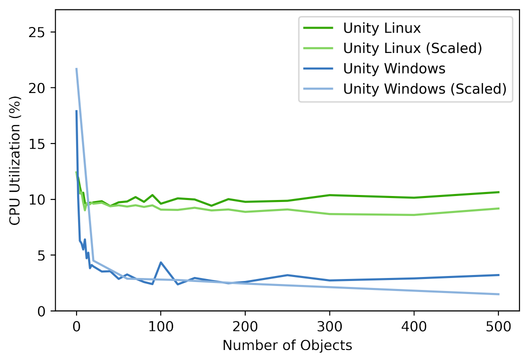


(b) Frame rate difference (dynamic scene, scaled time)

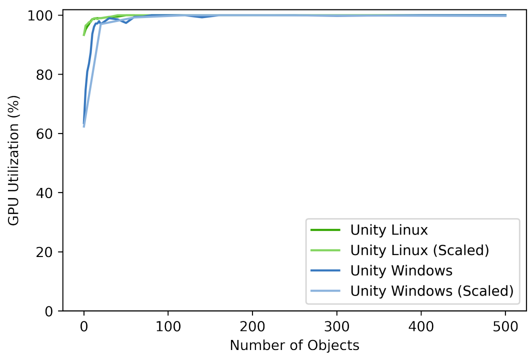
Figure B.9 Frame rate measured during the dynamic scene experiment with Unity scaled down the time based on meassured RTF in Gazebo with a difference compared to Gazebo's frame rate (less configurations tested for Unity on Windows)



(a) CPU cores utilization (Unity time scale)

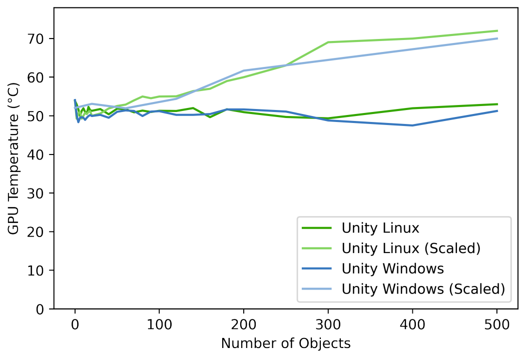


(b) CPU utilization (Unity time scale)

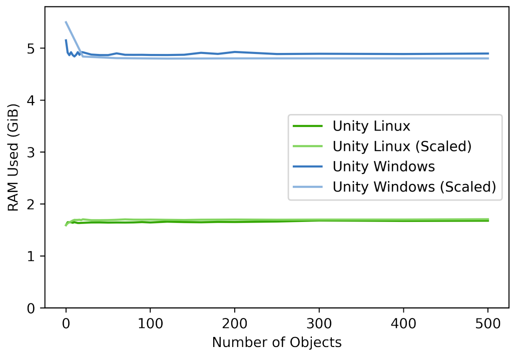


(c) GPU utilization (Unity time scale)

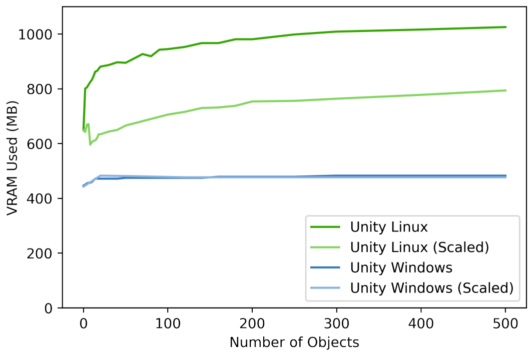
Figure B.10 CPU and GPU utilization measured during the dynamic scene experiments comparing Unity with and without scaled time



(a) GPU temperature (Unity time scale)

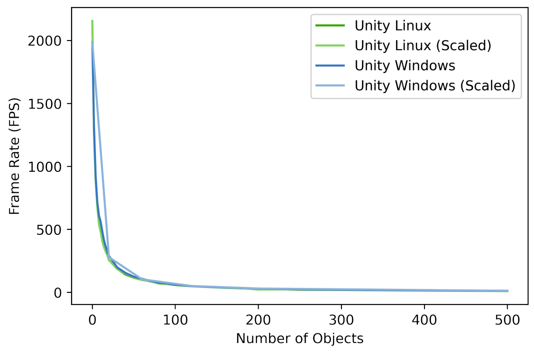


(b) RAM used (Unity time scale)

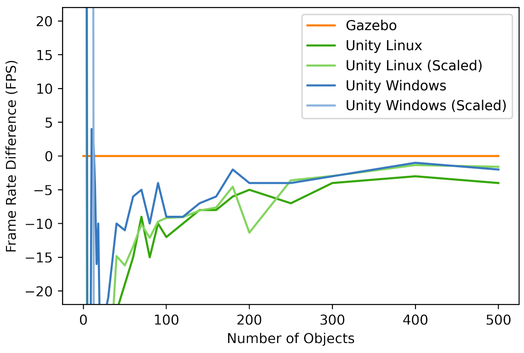


(c) VRAM used (Unity time scale)

Figure B.11 RAM and VRAM usage meassurements and GPU temperature during the dynamic scene experiments comparing Unity with and without scaled time

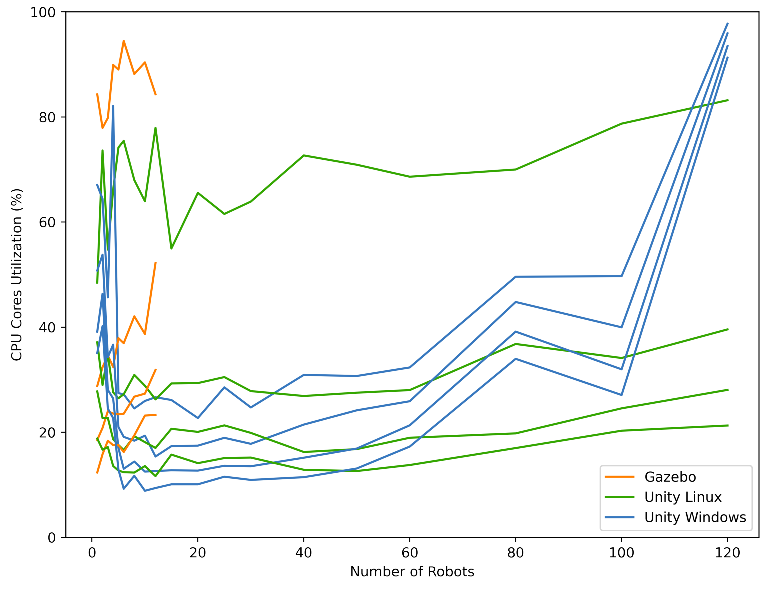


(a) Frame rate (Unity time scale)

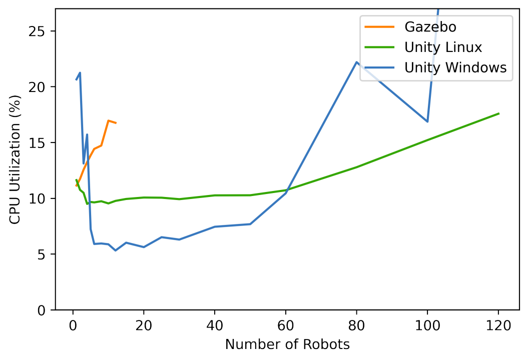


(b) Frame rate difference (Unity time scale)

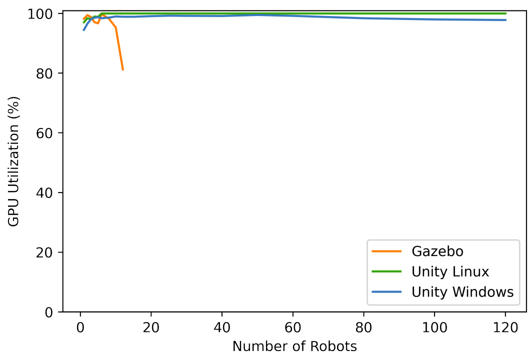
Figure B.12 Frame rate measured during the dynamic scene experiments comparing Unity with and without scaled time with a difference compared to Gazebo's frame rate (less configurations tested for Unity on Windows with scaled time)



(a) CPU cores utilization (multi-robot)

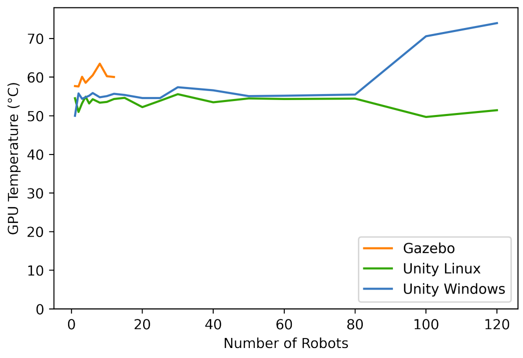


(b) CPU utilization (multi-robot)

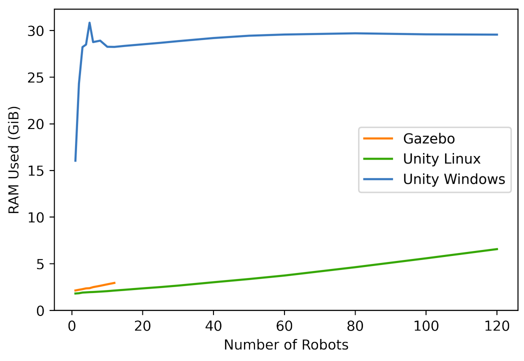


(c) GPU utilization (multi-robot)

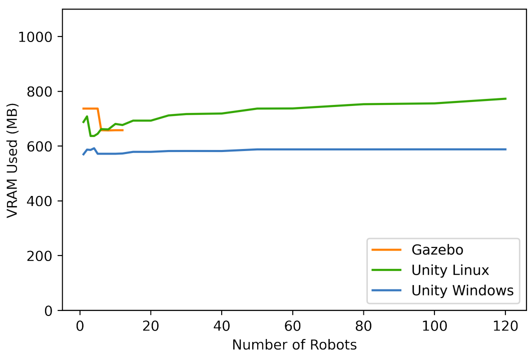
Figure B.13 CPU and GPU utilization measured during the multi-robot simulation experiments



(a) GPU temperature (multi-robot)

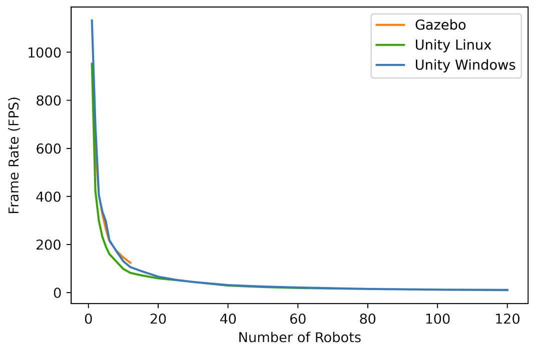


(b) RAM used (multi-robot)

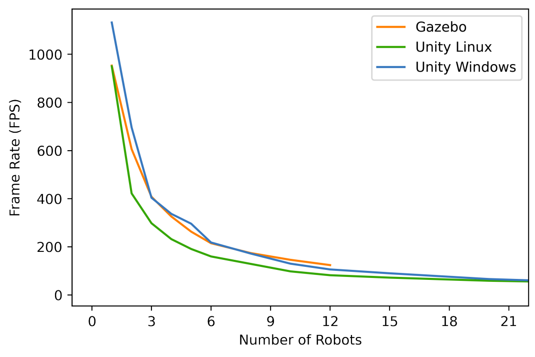


(c) VRAM used (multi-robot)

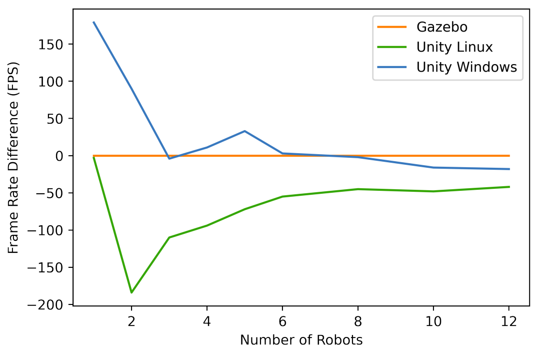
Figure B.14 RAM and VRAM usage meassurements and GPU temperature during the multi-robot simulation experiments



(a) Frame rate (multi-robot)



(b) Frame rate (multi-robot, zoomed)



(c) Frame rate difference (multi-robot)

■ **Figure B.15** Frame rate measured during the multi-robot simulation experiments with a difference compared to Gazebo's frame rate

Bibliography

1. *Robots and their Arms* [online]. 2023. [visited on 2023-05-04]. Available from: http://infolab.stanford.edu/pub/voy/museum/pictures/display/1-Robot.htm.
2. VIRGALA, Ivan; KELEMEN, Michal; PRADA, Erik. Kinematics of Serial Manipulators. In: VOLOŞENCU, Constantin; KÜÇÜK, Serdar; GUERRERO, José; VALERO, Oscar (eds.). *Automation and Control*. Rijeka: IntechOpen, 2020, chap. 7. Available from DOI: 10.5772/intechopen.93138.
3. BOŽEK, Pavol. Robot path optimization for spot welding applications in automotive industry. *Tehnicki vjesnik/Technical Gazette*. 2013, vol. 20, no. 5, pp. 913–917.
4. PELLEGRINELLI, Stefania; PEDROCCHI, Nicola; TOSATTI, Lorenzo Molinari; FISCHER, Anath; TOLIO, Tullio. Multi-robot spot-welding cells for car-body assembly: Design and motion planning. *Robotics and Computer-Integrated Manufacturing*. 2017, vol. 44, pp. 97–116. ISSN 0736-5845. Available from DOI: https://doi.org/10.1016/j.rcim.2016.08.006.
5. PÉREZ, Rodrigo; GUTIÉRREZ, Santiago C; ZOTOVIC, Ranko. A study on robot arm machining: Advance and future challenges. *Annals of DAAAM & Proceedings*. 2018, vol. 29.
6. ABBAS, Adel T; ALY, Mohamed F; HAMZA, Karim. Optimum drilling path planning for a rectangular matrix of holes using ant colony optimisation. *International Journal of Production Research*. 2011, vol. 49, no. 19, pp. 5877–5891.
7. GLEESON, Daniel; JAKOBSSON, Stefan; SALMAN, Raad; EKSTEDT, Fredrik; SANDGREN, Niklas; EDELVIK, Fredrik; CARLSON, Johan S.; LENNARTSON, Bengt. Generating Optimized Trajectories for Robotic Spray Painting. *IEEE Trans Autom. Sci. Eng.* 2022, vol. 19, no. 3, pp. 1380–1391. Available from DOI: 10.1109/TASE.2022.3156803.
8. GASPARETTO, Alessandro; VIDONI, Renato; PILLAN, Daniele; SACCAVINI, Ennio. Automatic Path and Trajectory Planning for Robotic Spray Painting. In: *ROBOTIK 2012 - Proceedings for the conference of ROBOTIK 2012, 7th German Conference on Robotics, 21-22 May 2012, International Congress Center Munich (ICM) in conjunction with AUTOMATICA, Munich, Germany*. VDE-Verlag, 2012. Available also from: http://www.vde-verlag.de/proceedings-de/453418039.html.
9. MONKMAN, Gareth J; HESSE, Stefan; STEINMANN, Ralf; SCHUNK, Henrik. *Robot grippers*. John Wiley & Sons, 2007.
10. CHOSET, Howie; LYNCH, Kevin M.; HUTCHINSON, Seth; KANTOR, George; BURGARD, Wolfram; KAVRAKI, Lydia; THRUN, Sebastian. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, 2005. ISBN 0262033275.
11. GREGORY, Jason. *Game engine architecture*. crc Press, 2018.

12. LAVALLE, Steven M. *Planning Algorithms*. Cambridge University Press, 2006. ISBN 9780511546877. Available from DOI: `10.1017/CBO9780511546877`.

13. KAVRAKI, Lydia E; LAVALLE, Steven M. Motion planning. In: *Springer handbook of robotics*. Springer, 2016, pp. 139–162.

14. GRIEVES, Michael. Digital twin: manufacturing excellence through virtual factory replication. *White paper*. 2014, vol. 1, no. 2014, pp. 1–7.

15. JONES, David; SNIDER, Chris; NASSEHI, Aydin; YON, Jason; HICKS, Ben. Characterising the Digital Twin: A systematic literature review. *CIRP Journal of Manufacturing Science and Technology*. 2020, vol. 29, pp. 36–52.

16. LIU, Mengnan; FANG, Shuiliang; DONG, Huiyue; XU, Cunzhi. Review of digital twin about concepts, technologies, and industrial applications. *Journal of Manufacturing Systems*. 2021, vol. 58, pp. 346–361.

17. LIANG, Ci-Jyun; MCGEE, Wes; MENASSA, Carol; KAMAT, Vineet. Bi-Directional Communication Bridge for State Synchronization between Digital Twin Simulations and Physical Construction Robots. In: 2020. Available from DOI: `10.22260/ISARC2020/0205`.

18. EREZ, Tom; TASSA, Yuval; TODOROV, Emanuel. Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx. In: *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2015, pp. 4397–4404.

19. RÖNNAU, Arne; SUTTER, F; HEPPNER, Georg; OBERLÄNDER, Jan; DILLMANN, Rüdiger. Evaluation of physics engines for robotic simulations with a special focus on the dynamics of walking robots. In: *2013 16th International Conference on Advanced Robotics (ICAR)*. IEEE, 2013, pp. 1–7.

20. BOEING, Adrian; BRÄUNL, Thomas. Evaluation of real-time physics simulation systems. In: *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*. 2007, pp. 281–288.

21. NVIDIA CORPORATION. *PhysX SDK* [online]. 2023. [visited on 2023-03-18]. Available from: `https://developer.nvidia.com/physx-sdk`.

22. SMITH, Russell. *Open Dynamics Engine* [online]. 2008. [visited on 2023-04-01]. Available from: `http://www.ode.org/`.

23. COUMANS, Erwin. *Bullet Real-Time Physics Simulation* [online]. 2013. [visited on 2023-04-01]. Available from: `https://pybullet.org/`.

24. TECH, Georgia; UNIVERSITY, Carnegie Mellon. *Dynamic Animation and Robotics Toolkit* [online]. 2012. [visited on 2023-04-01]. Available from: `https://dartsim.github.io/`.

25. SHERMAN, Michael; EASTMAN, Peter. *Simbody: Multibody Physics API* [online]. 2012. [visited on 2023-04-01]. Available from: `https://simtk.org/projects/simbody`.

26. PYO, Yoonseok; CHO, Hancheol; JUNG, Leon; LIM, Darby. *ROS Robot Programming (English)*. ROBOTIS, 2017. ISBN 9791196230715. Available also from: `http://community.robotsource.org/t/download-the-ros-robot-programming-book-for-free/51`.

27. QUIGLEY, Morgan; CONLEY, Ken; GERKEY, Brian; FAUST, Josh; FOOTE, Tully; LEIBS, Jeremy; WHEELER, Rob; NG, Andrew Y, et al. ROS: an open-source Robot Operating System. In: *ICRA workshop on open source software*. Kobe, Japan, 2009, vol. 3, p. 5. No. 3.2.

28. MACENSKI, Steven; FOOTE, Tully; GERKEY, Brian; LALANCETTE, Chris; WOODALL, William. Robot Operating System 2: Design, architecture, and uses in the wild. *Science Robotics*. 2022, vol. 7, no. 66, eabm6074. Available from DOI: `10.1126/scirobotics.abm6074`.

29. OPEN ROBOTICS. *ROS 2 Documentation* [online]. [visited on 2023-04-06]. Available from: `https://docs.ros.org/en/humble/index.html`.

30. ROS-INDUSTRIAL REVISION. *Using rqt Tools for Analysis* [online]. 2017. [visited on 2023-05-03]. Available from: `https://industrial-training-master.readthedocs.io/en/melodic/_source/session6/Using-rqt-tools-for-analysis.html`.

31. OPEN ROBOTICS. *Overview and usage of RQt* [online]. [visited on 2023-04-08]. Available from: `https://docs.ros.org/en/humble/Concepts/About-RQt.html`.

32. QT GROUP. *Qt Framework* [online]. [visited on 2023-04-08]. Available from: `https://www.qt.io/product/framework`.

33. OPEN ROBOTICS. *rqt: Package Sumary* [online]. [visited on 2023-04-08]. Available from: `http://wiki.ros.org/rqt`.

34. OPEN ROBOTICS. *RViz* [online]. [visited on 2023-04-08]. Available from: `http://wiki.ros.org/rviz`.

35. PICKNIK ROBOTICS. *MoveIt Homepage* [online]. [visited on 2023-04-08]. Available from: `https://moveit.ros.org/`.

36. COLEMAN, David; SUCAN, Ioan; CHITTA, Sachin; CORRELL, Nikolaus. *Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study*. 2014. Available from arXiv: `1404.3785 [cs.RO]`.

37. ŞUCAN, Ioan A.; MOLL, Mark; KAVRAKI, Lydia E. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*. 2012, vol. 19, no. 4, pp. 72–82. Available from DOI: `10.1109/MRA.2012.2205651`. `https://ompl.kavrakilab.org`.

38. SCHULMAN, John; HO, Jonathan; LEE, Alex X; AWWAL, Ibrahim; BRADLOW, Henry; ABBEEL, Pieter. Finding locally optimal, collision-free trajectories with sequential convex optimization. In: *Robotics: science and systems*. Berlin, Germany, 2013, vol. 9, pp. 1–10. No. 1.

39. PICKNIK ROBOTICS. *Pilz Industrial Motion Planner* [online]. [visited on 2023-04-08]. Available from: `https://ros-planning.github.io/moveit_tutorials/doc/pilz_industrial_motion_planner/pilz_industrial_motion_planner.html`.

40. PICKNIK ROBOTICS. *moveit2* [online]. GitHub [visited on 2023-04-08]. Available from: `https://github.com/ros-planning/moveit2`.

41. KOENIG, Nathan; HOWARD, Andrew. Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. Sendai, Japan, 2004, pp. 2149–2154.

42. OGRE3D TEAM. *Object-Oriented Graphics Rendering Engine cite* [online]. 2008. [visited on 2023-04-13]. Available from: `https://www.ogre3d.org/`.

43. PITONAKOVA, Lenka; GIULIANI, Manuel; PIPE, Anthony; WINFIELD, Alan. Feature and performance comparison of the V-REP, Gazebo and ARGoS robot simulators. In: *Towards Autonomous Robotic Systems: 19th Annual Conference, TAROS 2018, Bristol, UK July 25-27, 2018, Proceedings 19*. Springer, 2018, pp. 357–368.

44. DE MELO, Mirella Santos Pessoa; SILVA NETO, José Gomes da; DA SILVA, Pedro Jorge Lima; TEIXEIRA, João Marcelo Xavier Natario; TEICHRIEB, Veronica. Analysis and comparison of robotics 3d simulators. In: *2019 21st Symposium on Virtual and Augmented Reality (SVR)*. IEEE, 2019, pp. 242–251.

45. ROHMER, Eric; SINGH, Surya PN; FREESE, Marc. V-REP: A versatile and scalable robot simulation framework. In: *2013 IEEE/RSJ international conference on intelligent robots and systems*. IEEE, 2013, pp. 1321–1326. www.coppeliarobotics.com.

46. MICHEL, Olivier. Cyberbotics ltd. webots™: professional mobile robot simulation. *International Journal of Advanced Robotic Systems*. 2004, vol. 1, no. 1, p. 5.

47. PINCIROLI, Carlo; TRIANNI, Vito; O'GRADY, Rehan; PINI, Giovanni; BRUTSCHY, Arne; BRAMBILLA, Manuele; MATHEWS, Nithin; FERRANTE, Eliseo; DI CARO, Gianni; DUCATELLE, Frederick, et al. ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems. *Swarm intelligence*. 2012, vol. 6, pp. 271–295.

48. UNITY TECHNOLOGIES. *Unity homepage* [online]. 2023. [visited on 2023-03-16]. Available from: `https://unity.com/`.

49. HAAS, John K. A history of the unity game engine. *Diss. Worcester Polytechnic Institute*. 2014, vol. 483, no. 2014, p. 484.

50. UNITY TECHNOLOGIES. *Install the Unity Hub* [online]. 2021. [visited on 2023-03-16]. Available from: `https://docs.unity3d.com/hub/manual/InstallHub.html`.

51. CATTO, Erin. *Box2D* [online]. 2023. [visited on 2023-03-18]. Available from: `https://box2d.org/`.

52. DEANE, Ian. *Bullet Physics For Unity* [online]. 2017. [visited on 2023-03-25]. Available from: `https://assetstore.unity.com/packages/tools/physics/bullet-physics-for-unity-62991`.

53. MUJOCO. *MuJoCo Documentation: Unity Plug-in* [online]. [visited on 2023-03-25]. Available from: `https://mujoco.readthedocs.io/en/latest/unity.html`.

54. JULIANI, Arthur; BERGES, Vincent-Pierre; TENG, Ervin; COHEN, Andrew; HARPER, Jonathan; ELION, Chris; GOY, Chris; GAO, Yuan; HENRY, Hunter; MATTAR, Marwan; LANGE, Danny. Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*. 2020.

55. UNITY TECHNOLOGIES. *Order of execution for event functions* [online]. 2021. [visited on 2023-03-21]. Available from: `https://docs.unity3d.com/Manual/ExecutionOrder.html`.

56. UNITY TECHNOLOGIES. *Choose the plan that is right for you* [online]. 2023. [visited on 2023-03-16]. Available from: `https://store.unity.com/compare-plans`.

57. CARPIN, Stefano; LEWIS, Mike; WANG, Jijun; BALAKIRSKY, Stephen; SCRAPPER, Chris. USARSim: a robot simulator for research and education. In: *Proceedings 2007 IEEE International Conference on Robotics and Automation*. 2007, pp. 1400–1405. Available from DOI: `10.1109/ROBOT.2007.363180`.

58. OHASHI, Osamu; OCHIAI, Eiji; KATO, Yuka. A Remote Control Method for Mobile Robots Using Game Engines. In: *2014 28th International Conference on Advanced Information Networking and Applications Workshops*. 2014, pp. 79–84. Available from DOI: `10.1109/WAINA.2014.23`.

59. BARTNECK, Christoph; SOUCY, Marius; FLEURET, Kevin; SANDOVAL, Eduardo B. The robot engine—Making the unity 3D game engine work for HRI. In: *2015 24th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*. IEEE, 2015, pp. 431–437.

60. SITA, Enrico; HORVÁTH, Csongor Márk; THOMESSEN, Trygve; KORONDI, Péter; PIPE, Anthony G. Ros-unity3d based system for monitoring of an industrial robotic process. In: *2017 IEEE/SICE International Symposium on System Integration (SII)*. IEEE, 2017, pp. 1047–1052.

61. PAN, Junhao; ZHUO, Yong; HOU, Liang; BU, Xiangjian. Research on simulation system of welding robot in Unity3d. In: *Proceedings of the 15th ACM SIGGRAPH Conference on Virtual-Reality Continuum and Its Applications in Industry-Volume 1*. 2016, pp. 107–110.

62. KUTS, Vladimir; OTTO, Tauno; TÄHEMAA, Toivo; BONDARENKO, Yevhen. Digital twin based synchronised control and simulation of the industrial robotic cell using virtual reality. *Journal of Machine Engineering*. 2019, vol. 19, no. 1, pp. 128–145.

63. CODD-DOWNEY, Robert; FOROOSHANI, P Mojiri; SPEERS, Andrew; WANG, Hui; JENKIN, Michael. From ROS to unity: Leveraging robot and virtual environment middleware for immersive teleoperation. In: *2014 IEEE International Conference on Information and Automation (ICIA)*. IEEE, 2014, pp. 932–936.

64. WHITNEY, David; ROSEN, Eric; ULLMAN, Daniel; PHILLIPS, Elizabeth; TELLEX, Stefanie. Ros reality: A virtual reality framework using consumer-grade hardware for ros-enabled robots. In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 1–9.

65. HÖNIG, Wolfgang; MILANES, Christina; SCARIA, Lisa; PHAN, Thai; BOLAS, Mark; AYANIAN, Nora. Mixed reality for robotics. In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2015, pp. 5382–5387. Available from DOI: `10.1109/IROS.2015.7354138`.

66. LIU, Yuzhou; NOVOTNY, Georg; SMIRNOV, Nikita; MORALES-ALVAREZ, Walter; OLAVERRI-MONREAL, Cristina. Mobile delivery robots: mixed reality-based simulation relying on ros and unity 3D. In: *2020 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2020, pp. 15–20.

67. OPEN ROBOTICS. *ros1_bridge* [https://github.com/ros2/ros1_bridge]. GitHub, 2018 [visited on 2023-03-25].

68. ROBOT WEB TOOLS. *rosbridge_suite* [online]. [visited on 2023-03-25]. Available from: `http://wiki.ros.org/rosbridge_suite`.

69. HUSSEIN, Ahmed; GARCÍA, Fernando; OLAVERRI-MONREAL, Cristina. ROS and Unity Based Framework for Intelligent Vehicles Control and Simulation. In: *2018 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*. 2018, pp. 1–6. Available from DOI: `10.1109/ICVES.2018.8519522`.

70. SIEMENS. *ros-sharp* [online]. GitHub, 2018 [visited on 2023-03-25]. Available from: `https://github.com/siemens/ros-sharp`.

71. SIEMENS. *ROS#* [online]. [visited on 2023-03-25]. Available from: `https://assetstore.unity.com/packages/tools/physics/ros-107085`.

72. UNITY TECHNOLOGIES. *ROS TCP Connector* [online]. GitHub, 2020 [visited on 2023-03-29]. Available from: `https://github.com/Unity-Technologies/ROS-TCP-Connector`.

73. UNITY TECHNOLOGIES. *URDF Importer* [online]. GitHub, 2020 [visited on 2023-03-29]. Available from: `https://github.com/Unity-Technologies/URDF-Importer`.

74. UNITY TECHNOLOGIES. *Unity Robotics Hub* [online]. GitHub, 2020 [visited on 2023-03-29]. Available from: `https://github.com/Unity-Technologies/Unity-Robotics-Hub`.

75. UNITY TECHNOLOGIES. *ROS TCP Endpoint* [online]. GitHub, 2020 [visited on 2023-03-29]. Available from: `https://github.com/Unity-Technologies/ROS-TCP-Endpoint`.

76. OVERMARS, Mark; YOYO GAMES. *GameMaker* [online]. 2023. [visited on 2023-03-18]. Available from: `https://gamemaker.io/en`.

77. ROTHAMEL, Tom. *Ren'Py* [online]. 2023. [visited on 2023-03-18]. Available from: `https://www.renpy.org/`.

78. *RPG Maker* [online]. 2023. [visited on 2023-03-18]. Available from: `https://www.rpgmakerweb.com/`.

79. JONES, Chris. *Adventure Game Studio* [online]. 2023. [visited on 2023-03-18]. Available from: `https://www.adventuregamestudio.co.uk/`.

80. EPIC GAMES. *Unreal Engine* [online]. 2023. [visited on 2023-03-18]. Available from: `https://www.unrealengine.com`.

81. LINIETSKY, Juan; MANZUR, Ariel. *Godot Engine* [online]. 2023. [visited on 2023-03-18]. Available from: `https://godotengine.org/`.

82. EPIC GAMES. *Unreal Engine 5.1 Documentation: Level Editor* [online]. [visited on 2023-03-26]. Available from: `https://docs.unrealengine.com/5.1/en-US/level-editor-in-unreal-engine/`.

83. SHAH, Shital; DEY, Debadeepta; LOVETT, Chris; KAPOOR, Ashish. *AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles.* 2017. Available from arXiv: `1705.05065 [cs.RO]`.

84. YOUNG, Parker; KYSAR, Sam; BOS, Jeremy P. Unreal as a simulation environment for off-road autonomy. In: DUDZIK, Michael C.; JAMESON, Stephen M. (eds.). *Autonomous Systems: Sensors, Processing, and Security for Vehicles and Infrastructure 2020.* SPIE, International Society for Optics and Photonics, 2020, vol. 11415, 114150F. Available from DOI: `10.1117/12.2559006`.

85. JIANG, Fan; HAO, Qi. Pavilion: Bridging Photo-Realism and Robotics. In: *2019 International Conference on Robotics and Automation (ICRA).* IEEE, 2019, pp. 8285–8290.

86. LINIETSKY, Juan; MANZUR, Ariel. *godot* [online]. GitHub, 2014 [visited on 2023-03-26]. Available from: `https://github.com/godotengine/godot`.

87. RUDYVIC. *ROS-Websocket* [online]. GitHub, 2021 [visited on 2023-03-26]. Available from: `https://github.com/godotengine/godot`.

88. FLYNN, Evan. *godot* [online]. GitHub, 2021 [visited on 2023-03-26]. Available from: `https://github.com/flynneva/godot_ros`.

89. SURYNEK, Pavel. *RR1: Real Robot One - a DIY Desktop Robotic Arm* [online]. 2022-06. [visited on 2023-03-12]. Available from: `https://hackaday.io/project/185958-rr1-real-robot-one-a-diy-desktop-robotic-arm`.

90. SURYNEK, Pavel. *RR1* [https://github.com/surynek/RR1]. GitHub, 2022 [visited on 2022-11-10].

91. KUKA. *KR 210 R2700 extra* [online]. KUKA Deutschland GmbH, 2022 [visited on 2023-03-24]. Available from: `https://www.kuka.com/-/media/kuka-downloads/imported/6b77eecacfe542d3b736af377562ecaa/0000182736_en.pdf`.

92. FANUC. *FANUC Robot M-800iA/60* [online]. Fanuc Corporation, 2022 [visited on 2023-03-24]. Available from: `https://www.fanuc.co.jp/en/product/catalog/pdf/robot/RM-800iA(E)-01b.pdf`.

93. HEIDEN, Eric; PALMIERI, Luigi; BRUNS, Leonard; ARRAS, Kai O.; SUKHATME, Gaurav S.; KOENIG, Sven. Bench-MR: A Motion Planning Benchmark for Wheeled Mobile Robots. *IEEE Robotics and Automation Letters.* 2021, vol. 6, no. 3, pp. 4536–4543. Available from DOI: `10.1109/LRA.2021.3068913`.

94. CHEN, Jingkai; LI, Jiaoyang; HUANG, Yijiang; GARRETT, Caelan Reed; SUN, Dawei; FAN, Chuchu; HOFMANN, Andreas G.; MUELLER, Caitlin; KOENIG, Sven; WILLIAMS, Brian C. Cooperative Task and Motion Planning for Multi-Arm Assembly Systems. *CoRR.* 2022, vol. abs/2203.02475. Available from DOI: `10.48550/arXiv.2203.02475`.

95. ZHANG, Hejia; CHAN, Shao-Hung; ZHONG, Jie; LI, Jiaoyang; KOENIG, Sven; NIKOLAIDIS, Stefanos. A MIP-Based Approach for Multi-Robot Geometric Task-and-Motion Planning. In: *18th IEEE International Conference on Automation Science and Engineering, CASE 2022, Mexico City, Mexico, August 20-24, 2022.* IEEE, 2022, pp. 2102–2109. Available from DOI: `10.1109/CASE49997.2022.9926661`.

96. BCN3D TECHNOLOGIES. *BCN3D MOVEO: A fully Open Source 3D printed robot arm* [online]. 2016. [visited on 2023-03-12]. Available from: `https://www.bcn3d.com/bcn3d-moveo-the-future-of-learning-robotic-arm/`.

97. PURDON, Kyla; SETATI, Tiro; MARAIS, Stephen. Manufacturing and Evaluation of the Open-Source AR3 Robot Arm for Educational Uses. In: *2021 Rapid Product Development Association of South Africa - Robotics and Mechatronics - Pattern Recognition Association of South Africa (RAPDASA-RobMech-PRASA)*. 2021, pp. 01–05. Available from DOI: `10.1109/RAPDASA-RobMech-PRAS53819.2021.9829064`.

98. ANNIN, Chris. *Annin Robotics homepage* [online]. 2016. [visited on 2023-03-12]. Available from: `https://www.anninrobotics.com/`.

99. NIRYO. *Niryo One Mechanical Specifications* [online]. 2018. [visited on 2023-03-24]. Available from: `https://ozrobotics.com/wp-content/uploads/2020/11/Niryo-One-Mechanical-Specifications.pdf`.

100. NANOTEC. *SCA5618 – Stepper motor – NEMA 23* [online]. 2023. [visited on 2023-03-14]. Available from: `https://en.nanotec.com/products/2749-nema-23-stepper-motor-sca5618`.

101. NANOTEC. *ST4118 – Stepper motor – NEMA 17* [online]. 2023. [visited on 2023-03-14]. Available from: `https://en.nanotec.com/products/250-st4118-stepper-motor-nema-17`.

102. ERRICHELLO, Robert. Herringbone Gears. In: *Encyclopedia of Tribology*. Ed. by WANG, Q. Jane; CHUNG, Yip-Wah. Boston, MA: Springer US, 2013, pp. 1638–1639. ISBN 978-0-387-92897-5. Available from DOI: `10.1007/978-0-387-92897-5_583`.

103. ARDUINO. *Arduino Due Documentation* [online]. 2023. [visited on 2023-03-12]. Available from: `https://docs.arduino.cc/hardware/due`.

104. STEPPERONLINE. *Digital Stepper Drive DM556T: User Manual* [online]. 2017. [visited on 2023-03-12]. Available from: `https://www.omc-stepperonline.com/download/DM556T.pdf`.

105. R SHAMSHIRI, Redmond; HAMEED, Ibrahim A; PITONAKOVA, Lenka; WELTZIEN, Cornelia; BALASUNDRAM, Siva K; J YULE, Ian; GRIFT, Tony E; CHOWDHARY, Girish. Simulation software and virtual environments for acceleration of agricultural robotics: Features highlights and performance comparison. 2018.

106. KONRAD, Anna. Simulation of Mobile Robots with Unity and ROS: A Case-Study and a Comparison with Gazebo. In: 2019.

107. UNITY TECHNOLOGIES. *Unity Learn* [online]. 2023. [visited on 2023-04-29]. Available from: `https://learn.unity.com/`.

Contents of enclosed SD card

- `readme.txt` .... file with short description of SD card contents
- `builds` .... builded Unity applications for Windows and Linux
- `src` .... directory with source codes and models
  - `Models` .... Blender files and exported 3DBenchy meshes
  - `ROS Docker` .... custom Docker image for ROS Humble
  - `ROS Packages` .... all ROS packages implemented for RR1
  - `thesis` .... directory with LaTeX source codes of the thesis
  - `Unity Projects` .... Unity projects files and implementations
- `text` .... directory with the text of the thesis
  - `thesis.pdf` .... the thesis text in PDF format
- `visdoc` .... directory with visual documentation of the prototype