Fakultät für Informatik
Lehrstuhl für Robotik, künstliche Intelligenz und Echtzeitsysteme

Building a Modular Robot - Robot Visualization and Simulation

Yusuf Güleray, Sven Schepp, Peijie Liu, Leonard Klüpfel

Master-Praktikum *Building a Modular Robot* SoSe 2020

Advisor:	Constantin Dresel, M.Sc.; Stefanie Manzinger, M.Sc.
Supervisor:	Prof. Dr.-Ing Matthias Althoff
Submission:	27. August 2020

Visualization and Simulation Modular Robots in Gazebo and RViz

Yusuf Güleray, Sven Schepp, Peijie Liu, Leonard Klüpfel
Technische Universität München
Email: yusuf.guleray@tum.de, schepp@in.tum.de, ge35qem@mytum.de, leonard.kluepfel@tum.de

***Abstract*—This paper describes tools capable of realistic robot simulation and visualization that enable conducting simple and easy tests which facilitate efficient development processes. We outline the specific tasks that were to be achieved through RViz and Gazebo. Furthermore, features for trajectory visualization in RViz, and an approach to simulate a pick-and-place experiment in Gazebo are explained. Finally their results are presented, wherein the pre-defined objectives concerning visualization and simulation have been achieved. The paper concludes with an outlook of future work for both applications.**

I. Introduction

A realistic robot visualization and simulation gives rise to the opportunity to efficiently test and develop a system. This requires applications capable of imitating the behavior of real-world robots in great detail including environment manipulation. The importance of the ability to quickly simulate tests of a manipulator without its physical presence or the opportunity to access it, has been emphasized by the COVID-19 pandemic and the resulting social distancing and safety measures. Further, such simulations and visualizations allow for easier, cheaper, reproducible, and safer deployment of tests when compared to experiments performed by physical robots.

Industrial manipulators are exposed to frequently changing environments which requires re-occurring robot evaluations and calibrations [1]. To simplify this process, modular robots can be deployed [1]. The practical course building a modular robot (PCMR) aims to develop such a system, motivated by reductions in deployment costs for industrial robots [1] through modularity. Thereby, the PCMR consists of a Trajectory and Path Planning (TP), Self-Collision Checking and Avoidance (SC), Computer Vision (CV), Fail-Safe Planning (FS), Robot Control Device and User Interface (RC), and Robot Visualization and Simulation (RV) team [1]. In this setting, simulating and visualizing robots becomes increasingly important as described above. This report presents the work achieved by the RV team.

The progress of last semester's team yielded a rudimentary visualization of modular robots in RViz. Efforts concerning simulations in Gazebo resulted in incorrectly spawned non functioning robots [2]. We build upon the results of last year's team for both the RViz and Gazebo sub-projects as defined in our task description [3].

To provide interfaces for a wide range of requests and to simultaneously work on the given problem statement, the RV team analogously subdivided into two groups focusing on one tool respectively. The two groups are referred to as: RViz team and Gazebo team within the following sections. Thus, the overall task of the RV team can be further defined by two statements. First, creating a realistic robot simulation in a Gazebo environment which obeys the laws of physics and that the robot can interact with. Second, providing an integrable visualization RViz-toolkit that displays customized features of the robot behavior such as trajectory visualization.

After having outlined the motivation and purpose of a robot simulation and visualization, the next section provides the reader with a solid understanding of the scope of the project. Section II covers the specific features and goals which were set to be achieved in this practical course. Sections IV and III explain the chosen approaches and details concerning all implemented features. The results are evaluated in section V. Finally, section VI describes an elaborated outlook of possible future improvements and additional features, which next semesters students might consider working on.

II. Task Description

This section provides an overview of the goals and specific features, which were to be implemented by the RViz and Gazebo team. The first set of objectives has been derived from the innate task description of the RV project [3]. The second goal set is composed of custom requests made by other teams within the practical course as well as the RV team's self defined objectives. Below, the goals for both groups are defined, starting with those concerning the RViz team.

A. Tasks of the RViz Team

Based on the given problem statement from [3], improving upon the existing RViz visualization was stated as an expected working step. Thereby, no explicit details concerning features were mentioned. Thus, the RViz team's objectives consist of requests from members of other PCMR teams as well as self imposed goals. The RViz team defined the tasks of implementing a trajectory visualization and building customized

visualizations for the other teams. The former is subdivided into the following assignments:

- Define trajectories based on work-space coordinates,
- Define trajectories based on joint-space coordinates,
- Design a trajectory visualization interface.

During collaborations with other teams, custom visualization requests were received, such as:

- Provide an image stream through RViz,
- Build user tests for the evaluation of the control device.

B. Tasks of the Gazebo Team

The innate objectives for the Gazebo team were defined as spawning and controlling a modular robot as well as creating manipulative environment objects. Additionally, replicating the *Bulbs and Slides* experiments was defined as a self-imposed goal. This demonstration consists of robots placed on a table, moving light bulbs. The bulbs are situated on top of slides and are picked and placed iteratively from one slide to the other. The experiment is divided in to two versions. The first is executed by a 6 degree of freedom (DoF) modular robot, while the second trial is conducted by two 3 DoF robots. This describes a real-world use case in form of a pick-and-place motion, which is a common task in industrial settings. This combines all of the presented objectives in one experiment. A break-down of these goals results in the following list of specific features:

- Auto-generate modular robots that are manipulable in Gazebo,
- Add a control plugin through which modular robots can be controlled,
- Create models that can be placed in the environment and acted upon by a modular robot.

We additionally define the following self imposed goals:

- Equip all modular robots with Gazebo compatible colorization,
- Build and attach a Gripper to any valid robot such that grabbing objects is possible,
- Spawn and control multiple robots simultaneously,
- Rebuild the *Bulbs and Slides* environment used for the *Bulbs and Slides* experiment,
- Reproduce the grabbing and moving of bulbs using one six degree of freedom manipulator as presented in the *Bulbs and Slides* experiment,
- Reproduce the same demonstration using two three degree of freedom robots.

The following sections describe the modeling aspects of our projects starting with the RViz package. We present the architectures, modules, and functionalities of both packages separately while justifying our modeling choices. Details concerning the code of the implementation as well as execution steps are not discussed within this document. However, our wikis within the RViz- and Gazebo-package provide details concerning the implementation as well as execution instructions.

III. Visualization in RViz

The package `modrob\_visualization\_rviz` allows for visualization of the modular robot in RViz. Moreover, visualisations of the trajectories based on *work-space* and *joint-space* coordinates as well as *image streams* integrated in graphical user interfaces (GUI) are included in this package.

A. Trace Visualization

The `trace_visualizer` feature visualizes passed points (trace) of the end-effector frame. It samples the position of the end-effector frame at each time step and displays a line strip between the current and previous position for each sampled point.

During each cycle, the pose of the end-effector is obtained by the `tf/transform_listener` and passed to a `tf::StampedTransform` object. The Cartesian-Coordinates relative to the world frame are obtained through conversion to `geometry_msgs::Point`. Further, the Cartesian-Coordinates are inserted into an array containing all the previous points and finally displayed in form of a `visualization_msgs::Marker::LINE_STRIP`. Thereby, each point is connected with its predecessor. Special care should be taken while choosing an adequate sampling rate for the given task. Low sampling rates might result in very jagged paths while a high sampling frequency can lead to inefficient utilization of memory.

B. Work-Space Trajectory Visualization

One of the main design decision made concerning the work-space trajectory visualization is defining an architecture capable of transmitting and storing the trajectory data. The `bag` file format is selected due to the following reasons:

- Capability of writing and reading the bag files on the fly,
- Native file format for storing the serialized message data,
- Ease of creation through tools such as `rosbag`,
- Efficiency in both recording and playback.

The bag file containing the trajectory of the robot can be manually created by the user and visualized online. Additionally, a dedicated feature for recording trajectories to a bag file was built. The `trajectory_3Dpoints_recorder` feature records an array of sampled end-effector frame's position to a bag file. The position of the end-effector frame is obtained through the same method employed by `trace_visualizer`, see section III-A. The trajectory points are hereby written to a bag file using `rosbag` without being visualized.

The `trajectory_3Dpoints_visualizer` feature visualizes the trajectory that is recorded to a bag file. Each point of the trajectory is extracted by `rosbag`, where the contained points are connected to each other by and visualized as a `visualization_msgs::Marker::LINE_STRIP`. This process can be compared to the `trace_visualizer` displaying the trajectory, see section III-A .

C. Joint Space Visualization: Forward Kinematics with KDL

The following subsections describe all features that are part of the the joint-space trajectory visualization. This approach presents an alternative to the processes based on Cartesian-Coordinates described above.

As the points contained in `visualization_msgs::Marker::LINE_STRIP` are based on Cartesian-Coordinates, a translation from joint-space to work-space needs to be performed. Therefore, the end-effector position in Cartesian-Coordinates needs to be acquired. Two tools suitable for this purpose are `TF transforms` and the Kinematics and Dynamics Library (KDL). The former solely provides the current end-effector position by inferring it from the robots URDF description and the *current* joint angles while the latter performs forward kinematics for *any set* of joint angles and a given robot definition. Thereby, upcoming milestones of a trajectory can be visualized before the end-effector assumes its locations. This complies with the original goal of trajectory visualization which is independent of moving the robot itself. Therefore, KDL is chosen to perform all forward kinematics on planned joint-space trajectories where all trajectories are received from the TP team's interface.
The robot description is fetched from the `ROS parameter server` during the initialization phase. Subsequently, the number of moving joints is defined. Our demonstrations employ a four degree of freedom manipulator, since such a robot is featured in test scenarios described by the TP team. Further, the Cartesian position is derived by the `KDL::TreeFkSolverPos_recursive` of the end-effector and the result saved as a C++ vector for each configuration of joint-coordinates.

D. Message Type Adaptation

A common interface with the TP team had to be built to allow communication and transmission of trajectory-relevant message types. Further, compatibility with both Moveit and KDL should be preserved. In the following, the interface's structure as well as its flow of data is described. The overall structure can be seen in figure 1.

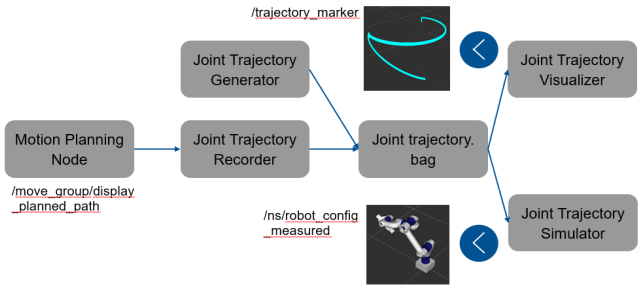


Fig. 1. Structure of the nodes implemented for joint-space trajectory visualization

The motion planning node of the TP team publishes joint-space trajectory messages on the `/tp_rv_topic` topic. The `Joint_Trajectory_Recorder` listens on this topic and creates individual joint-space trajectory points. These points are then saved within a ROS bag file, which the `joint_trajectory_visualizer` is reading from. The joint-space trajectory is simultaneously converted to a C++ vector of type `trajectory_msgs::JointTrajectoryPoint`.
Additionally, all trajectory information needs to be transformed into KDL compatible data types, before forward kinematics is performed. The resulting Cartesian positions are then saved in form of a `KDL::Frame`. These are finally converted to `geometry_msgs::Point` and inserted into the `visualization_msgs::Marker::LINE_STRIP`.

E. Visualization of Robot Movement Originating from Joint-Space Trajectory

To verify trajectories generated by the previously described processes, a visualization featuring a four DoF modular robot following a sample trajectory is performed. The `joint_trajectory_simulator` deploys a similar approach to section III-D. However, it converts the joint-space configuration into `RobotConfigMeasured`. These messages are published onto `/ns/robot_config_measured` which allows us to visualize the modular robot's movements following the joint-space trajectory. Should the physical robot be available, it can be measured and its positions can be fed into the visualization replacing the calculated ones described above. In either case, the robot's traveled path and its desired trajectory are compared either visually or by measuring their divergence. It can thereby be determined whether a robot is following the pre-defined trajectory.

The received trajectories solely contain positional data (no velocities), this poses an issue for the visualization of robot's movements. High granularity of the planned trajectory results in an overabundance of sampled points. Whereby, increasing the granularity will lead to longer travel times over the same distance, due to pauses at every milestone. Possible solutions are: a reduction of the pause duration or clipping a subset of configuration points. Therefore, a dedicated strategy has to be chosen which prevents decreases in performance of both the visualization as well as the divergence testing. One implementation of such a strategy is to keep only the last of every five configuration points.

F. Image Stream from RViz

This feature streams images of the RViz visualizations. It is achieved by setting up the extrinsic and intrinsic parameters of multiple virtual cameras in the environment of the robot and adjusting the lighting accordingly. The images published by the *image stream* can be used in a web-based GUI or recorded for the purpose of off-line analysis. The plugin $rviz\_camera\_stream$ allows the extrinsic parameters: rotation and translation of the camera frame to be adjusted relative to a given frame through the $static\_transform\_publisher$. The given relative frame moves within the world frame while the camera remains constant with respect to its coordinates

within the given relative frame. This can be attributed to `tf's` tree structure. It enables the creation of views such as the end-effector perspective, which renders images from a camera that is located at the end-effector frame. Finally, the intrinsic parameters such as: height, width, distortion model, and parameters as well as the focal lengths of the camera should be assigned. This information is published by $sensor\_msgs/CameraInfo.msg$. An example setup for the camera block can be found in figure 2. By replicating the code block in the figure 2, multiple cameras can be added.

```
<group ns="camera_topView">
  <node pkg="tf" type="static_transform_publisher" name="camera_broadcaster"
      args="0 0 2.4 -0.7071068 -0.7071068 0 0 part0 camera_topView 10" />
  <node name="camera_info" pkg="rostopic" type="rostopic"
      args="pub camera_info sensor_msgs/CameraInfo
      '{header: {seq: 0, stamp: {secs: 0, nsecs: 0}, frame_id:
      'camera_topView'},
      height: 720, width: 1280, distortion_model: 'plumb_bob',
      D: [0],
      K: [800.0, 0.0, 640, 0.0, 800.0, 360.0, 0.0, 0.0, 1.0],
      R: [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0],
      P: [800.0, 0.0, 640, 0.0, 0.0, 800, 360, 0.0, 0.0, 0.0, 1.0, 0.0],
      binning_x: 0, binning_y: 0,
      roi: {x_offset: 0, y_offset: 0, height: 720, width: 1280, do_rectify:
      false}}' -r 2"
      output="screen"/>
</group>
```

Fig. 2. The code block for setting up the top view camera in the launch file $rviz\_camera\_stream.launch$. The extrinsic parameters are assigned by the $static\_transform\_publisher$. The intrinsic parameters are assigned by the $camera\_info$ block.

The image stream `rviz_camera_stream.launch` has four principle views: top, front, right and orthographic. Additionally, a fifth camera located at the end-effector frame was created to visualize the end-effector's perspective.

By default, RViz has a single spot-light focused on the current view of the user in its GUI. This causes all of the views that are significantly different from the user's to have shadows displayed on the model of the robot. In order to resolve this issue, a custom lighting configuration had to be implemented. The $rviz\_lighting$ plug-in provides ambient, directional, point, and spot-lights, allowing for optimizations of image quality in different configurations. For the visualized robot and the given camera configuration, the optimal image quality was achieved by using the directional lighting from all six directions.

G. End-Effector Centered Views

The main disadvantage of fixed camera views is that they only provide valuable information when the robot is in a specific configuration. For instance, when the end-effector is on the opposite side of the frontal camera, the robot's actions can not be inferred from the image. To resolve this issue, end-effector-centered views have been implemented. They are positioned in fixed coordinate frames relative to the end-effector. Thus, their composition is independent of the robot's configuration. However, the orientations of these camera frames are assigned such that their xy-plane and the world frame are always in parallel. This provides a more comfortable perspective for the viewer. The z-axis (camera axis) points at the end-effector so that the end-effector is always at the center of the image.

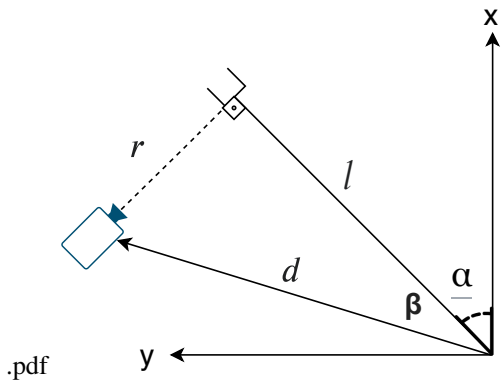


Fig. 3. This figure depicts the position of the side view camera which follows the end-effector. l is the distance between end-effector and the base, r is the assigned distance between the camera and the end-effector, and d is the distance between the base and the camera.

The positioning of the side end-effector frame can be seen in figure 3. Calculations of the Cartesian-Coordinates as well as Roll, Pitch, Yaw angles of the camera relative to the world frame can be seen below:

$$\alpha = atan2(y_{EE}, x_{EE}) \tag{1}$$

$$\beta = atan2(r, l) \tag{2}$$

$$d = \sqrt{l^2 + r^2} \tag{3}$$

$$x_{camera} = d\cos(\alpha + \beta) \tag{4}$$

$$y_{camera} = d\sin(\alpha + \beta) \tag{5}$$

$$z_{camera} = z_{EE} \tag{6}$$

$$Roll = -\pi/2, Pitch = 0, Yaw = \pi + \alpha \tag{7}$$

H. User Tests of the Control Device

The Robot Control and User Interface Team requested an RViz visualization through which they can evaluate the user's performance with their control device. The first user test: `RC_usertest_endpoint`, visualizes an end-point as a `visualization_msgs::Marker::SPHERE` at a given position. Afterwards, it measures the Euclidean distance between the end-point and the end-effector, which is later published onto the topic `distance_to_endpoint`. Additionally, the color of the sphere changes to indicate that the user has achieved the task, when the measured distance is below a threshold value. The second user test: `RC_usertest_endpoint`, visualizes an end-point and a trajectory. While the end-point visualization and measurement is the same as the `RC_usertest_endpoint`, `RC_usertest_endpoint` measures the Euclidean distance between the visualized trajectory and the end-effector. Subsequently, this measurement is published onto the topic `distance_to_endpoint`.

IV. Simulation in Gazebo

As described in section I, providing a simulation, the results of which can be accurately traced back to real life robot-behavior, is the main goal of the Gazebo team. The following subsections describe modules and mechanisms built to facilitate a realistic simulation in Gazebo.

A. Architecture

The interactions between modules are organized within our simulation architecture. In the following, brief descriptions of all models as well as the connections between them are presented.

The package `modrob_visualization` allows for realistic simulations of any chosen configuration of the modular robot. This includes an optional visualization, displayed in the Gazebo-Client as well as the calculation of all physical properties necessary to move the given robot. Thereby, properties including friction, damping, inertia, gravity, and maximum positional values for each joint and link are considered, since they influence the robots behavior in real-world environments. We built our system adherent to the architecture in figure 4 in order to facilitate a realistic simulation in Gazebo.

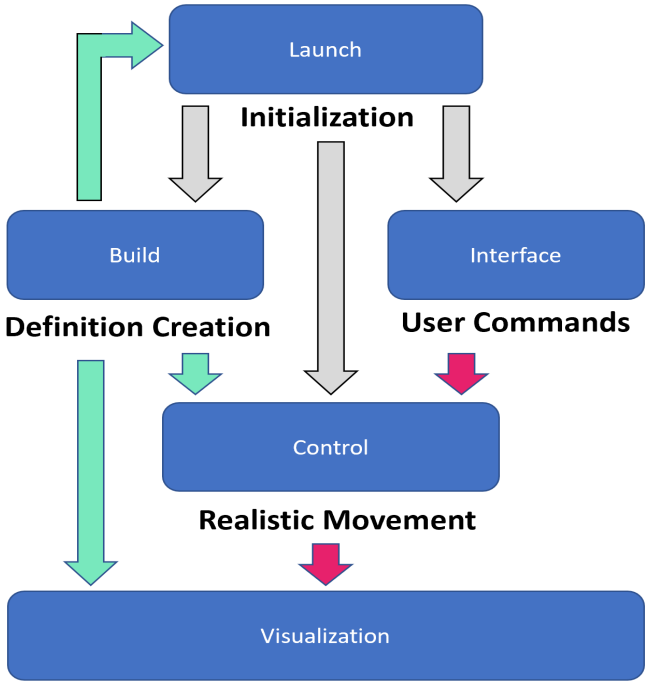


Fig. 4. A figure displaying all connections between the different modules. The grey arrows indicate launch commands that initialize nodes. The green arrows indicate the definition of control parameters, creation of launch-files for control nodes, and the building of robot descriptions as URDFs. The red arrows describe the flow of user given control inputs that are translated to realistic movements through physics simulations within the *Control-Module*.

Our Systems consist of five main modules: Build, Visualization, Interface, Control, and Launch, depicted in figure 4. These contain ROS-nodes that exchange information among each other through ROS-messages, where communication takes place within and between modules. The *Launch-Module* presents the only exception to this rule, since it is responsible for initializing all desired nodes and supplies them with parameters. This is managed by a hierarchically organized structure of ROS-launch-files that our wiki describes in detail. The *Build-Module* creates the desired robot-definitions from module-orders and base-positions which are provided by the user. This module is only executed if new robot descriptions in URDF-form are to be constructed. Wherefore, it is not necessary to execute nodes within this module to simulate previously defined robots.
The URDF robot definitions are supplied to Gazebo which consists of two subsystems: gzserver and gzclient. The gzserver can be described as a back-end-like structure, responsible for calculating simulation data and facilitating communication with external nodes, while the gzclient provides a graphical user interface by the means of which the robot can be moved and inspected. The latter represents our *Visualization-Module*. The URDF descriptions are internally converted to the Gazebo compatible sdf-format. This is performed automatically at the start of every simulation and is carried out by the gzserver.
Moving robots through the GUI in Gazebo version `9.0.0` is limited to force-inputs on a joint by joint basis and therefore neither precise nor efficient. To simplify the transmission of movement-data in the form of joint angles, we built an *Interface-Module* dedicated to each spawned robot that allows custom external nodes to communicate with Gazebo.
The *Control-Module* is responsible for receiving the desired joint-angles and controlling the position of the robots' joints accordingly. Thereby, all specified parameters influence the difference between desired and measured positions. Thus, proper tuning is advised. This phenomenon is explained in more detail in the Parameter Tuning section of our wiki.

B. The Building Process

All simulation dependent definitions such as: robot URDFs, control parameters, and control launch files are created by the *Building-Module*. It consists of two nodes: `create_urdf_and_gazebo` and `robot_description_publisher`. The latter of which interprets the user-given module orders (e.g. $[59, 3, 55, 63, 57, 5]$) by referring to the respective entries in the module database and transmits the base-coordinates as well as the robot-definitions to `create_urdf_and_gazebo`, where a Gazebo compatible URDF-file is created. The creation-script `create_urdf_and_gazebo` then builds a URDF-file from the received robot description. This process was already present in the winter-semester 2019 version of the package and is described in more detail in [2]. We solely present changes made to the creation script that were necessary to enable proper Gazebo simulations.

The previously implemented creation script was incapable

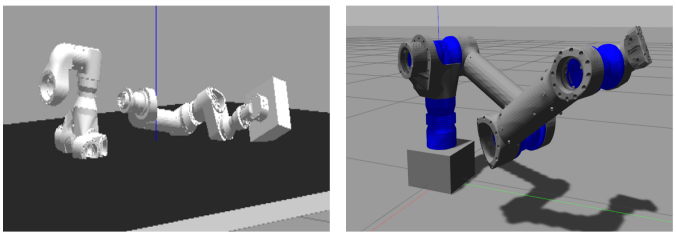


Fig. 5. An illustration of the simulation-quality before (left) and after (right) the implementation of gazebo-compatibility within the URDF-definitions.

of producing a stable robot within an arbitrary Gazebo environment, as seen in figure 5. The following changes had to be implemented to facilitate Gazebo compatible URDF-definitions:

- Connecting robot-bases to the world frame to counteract toppling,
- Adding <*gazebo*> tags that contain friction and damping values extracted from *Modules.csv*,
- Adding <*gazebo*> tags that contain Gazebo compatible colorization information.

As stated in section II-B, spawning multiple robots simultaneously is required to simulate the *Bulbs and Slides* experiment. The `robot_description_publisher` was therefore updated such that receiving multiple module orders and base positions is made possible. Similarly, `create_urdf_and_gazebo` was updated, enabling it to receive an array of robot definitions that can be treated as a queue within the creation process.

In the next step, <gazebo> tags containing the `gazebo_ros_control` interface and according transmission tags for each joint were added to facilitate position control, examples of which are depicted in 6.

```
<gazebo>
    <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
        <robotNamespace>/modrob0</robotNamespace>
        <robotSimType>gazebo_ros_control/DefaultRobotHWSim</robotSimType>
    </plugin>
</gazebo>

<transmission name="trans1">
    <type>transmission_interface/SimpleTransmission</type>
    <joint name="joint1">
        <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface>
    </joint>
    <actuator name="motor1">
        <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface>
        <mechanicalReduction>160.000000</mechanicalReduction>
    </actuator>
</transmission>
```

Fig. 6. An illustration of the added tags, where the *gazebo_ros_control* interface is defined in the top illustration while a transmission for *joint1* is added below.

Simulating multiple robots simultaneously can lead to collisions during the spawning process as well as the simulation phase. These can cause malfunctions of the system, displayed in figure 9. To counteract this, a minimum base distance(Δ_{min}) enforcing mechanism was implemented within `create_urdf_and_gazebo` that is described in more detail in section IV-E.

As controllers have to be provided for every created robot and since the number of robots can be defined by the user, all launch scripts responsible for spawning and controlling robots need to be updated to reflect the number of created robots. Therefore, all launch scripts and *.yaml* parameter files that change depending on the number of robots are automatically generated by `create_urdf_and_gazebo`. This implies that if different robots are to be simulated, a new run of the build-process has to be executed as defined in our wiki. An instance of Gazebo including the GUI is started after the creation process is finished for all robots. This can be avoided by setting the according flags to *false*.

C. The End-Effector

To manipulate its environment, an end-effector had to be designed and attached to the robot arm. Previously, the simulation of the robot has been equipped with a dummy placeholder tool which lacked physical properties and was not visualized in Gazebo. Figure 7 displays the resulting gripper in Gazebo holding a bulb. This demonstrates its grasping capability.

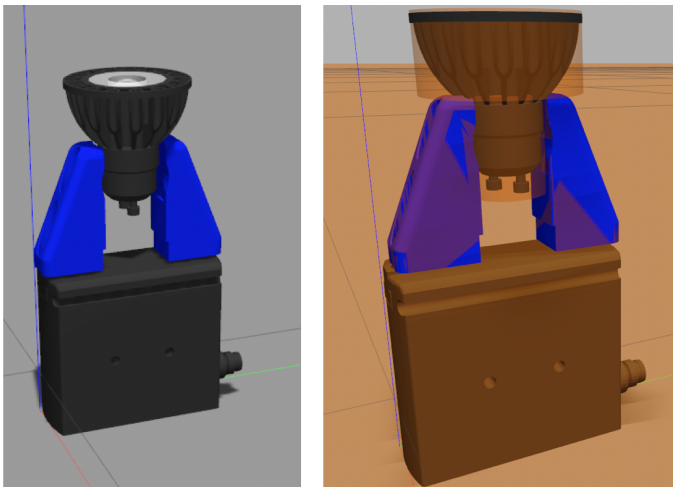


Fig. 7. Simulation of the gripper holding a bulb in Gazebo (left). Gripper grasping a bulb and visualized collision geometries (right).

'The gripper' we refer to in the following sections consists of a base and two fingers, where both fingers are connected to the base via prismatic joints. In the URDF-description of the robot both fingers as well as the base are characterized as links with visual properties. This structure allows for translational motions of the fingers along the base's y-axis. Objects can thereby be grasped by applying force to both fingers.

The base consists of the electric end-effector MPXM1612 from Gimatic. It can enact a total gripping force of up to $68N$. However, the maximum applicable forces had to be reduced to $3N$ to enable proper and consistent grasping of objects.

Gripping forces as high as $7N$ were tested successfully, where higher values resulted in crashes.
The fingers are 3D printed polylactic acid items and have been provided to us as STL-files. Before building the gripper, both modules have been adjusted and re-scaled using Blender. Additionally, for both items the origins were changed to simplify the building process.
The gripper is manually integrated in the creation process in section IV-B, where it is attached to the last link. The gripper can be included in the simulation by setting the parameter `gripper:="true"` in the `urdf_and_ gazebo.launch` file. This parameter is set to `"false"` by default.
This approach allows for a simple and modular implementation of the gripper as part of the robot arm. The end-effector can be controlled using transmission and position controllers, analogously to the main robot arm, as described in section IV-F.

D. Objects and the Bulb Experiment Environment

As stated in section II-B, the main goal of the Gazebo team is the simulation of the *Bulbs and Slides* demo application. Thus, the setting of the experiment had to be replicated. The environment thereby consists of a table on which a short slide, a long slide and several light bulbs, situated on top of the slides, are placed. The simulated setting, including two robots is displayed in figure 8.

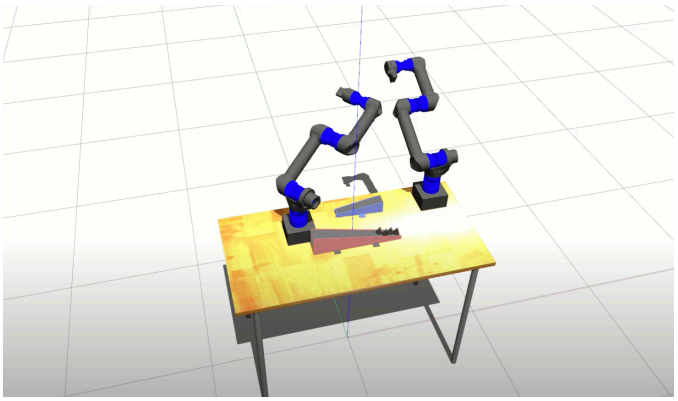


Fig. 8. Replicated *Bulbs and Slides* experiment setting in Gazebo.

We received model files for the bulbs and both slides as well as a layout-plan for the single robot experiment, which is enclosed in the Appendix VII. However, to simulate the second *Bulbs and Slides* demonstration, an additional robot had to be placed within the environment. The second robot's base position is chosen such that both modular robots keep a safety distance between each other. The underlying mechanisms are explained in detail in section IV-E. Additionally, the resulting layout plan is enclosed in the Appendix VII.

The table center is positioned at the world coordinates, within the *Bulbs and Slides* experiment:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{8}$$

To facilitate the grasping of bulbs, their collision geometries have been changed to two stacked cylindrical shapes. Moreover, the bulbs' and both slides' friction values have been adjusted to allow for a realistic grasping and sliding behavior.

E. Spawning Robots

As previously stated in section IV-A and shown in section IV-D, spawning multiple robots is enabled by our updated creation scripts. However, intersections of the collisions of two robots during the spawning process can lead to malfunctioning simulations. To avoid such scenarios, a scheme was implemented in `create_urdf_and_gazebo` that enforces a minimum distance($\Delta_{\mathtt{min}}$) between robot bases during spawn, reliant on a $safety\_distance$ parameter provided by the user.

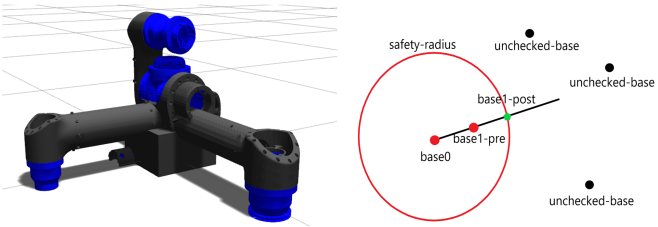


Fig. 9. An illustration of a crash caused by a collision (left) and the scheme employed to enforce a minimum base distance.

The following changes to the chosen base positions are carried out, should the distance between two bases within the x-y-plane be less than $\Delta_{\mathtt{min}}$. A robot's base position is moved along the line connecting the two bases until $\Delta_{\mathtt{min}}$ is reached. This process is iteratively repeated until no conflicts remain. The new base position `base1_post` is calculated as follows:

$$\Delta_{\mathtt{p}} = \sqrt{(\mathtt{base1\_pre_x} - \mathtt{base0_x})^2 + (\mathtt{base1\_pre_y} - \mathtt{base0_y})^2}, \tag{9}$$

$$\mathtt{base1\_post} = \mathtt{base0} + \frac{\Delta_{\mathtt{p}}}{\Delta_{\mathtt{min}}} \cdot (\mathtt{base1\_pre} - \mathtt{base0}), \tag{10}$$

where `base0`, `base1_pre`, and `base1_post` are the positions of base zero, base one before the shift, and after the shift. $\Delta_{\mathtt{p}}$ is thereby the current Euclidean distance of two bases. We can deduce from these equations that setting $\Delta_{\mathtt{min}} = 0.0$ functions as an off-toggle for this feature and is implemented as the default value. This function does not however check for intersections between robot models and the environment since such collisions, most often, do not lead to crashes but should be avoided nonetheless.

F. Controlling Robots

As discussed in section IV-A, integrating the `gazebo_ros_control` plugin enables the use of PID-position controllers for each joint of a robot via the previously described transmissions in section IV-A. We have added predefined transmissions of type *SimpleTransmission* and with the *EffortJointInterface* as our hardware-interface, seen in figure 6, since these are currently the only interfaces supported by Gazebo.

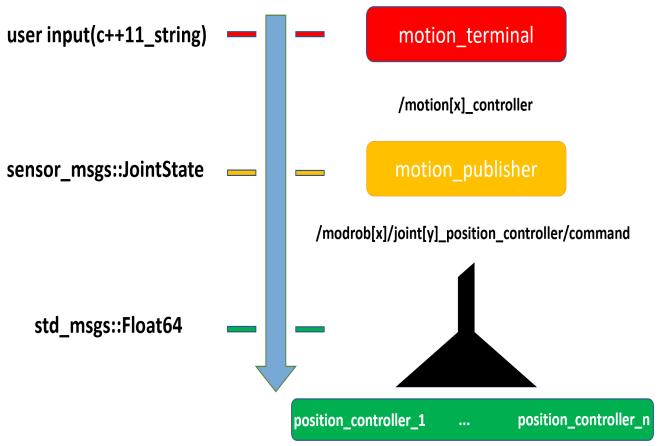


Fig. 10. An illustration of the hierarchical motion publishing architecture including the manually operated motion-terminal. Additionally, the lower level motion-publishers for each robot and its position controllers for each joint is illustrated.

Multiple joints can be moved simultaneously, facilitated by splitting our *Control-Module* into three hierarchical levels. The lowest layer and thus the most similar to directly publishing onto `joint_state`, is given by the position controller topics for each joint of a robot. They are enabled by `gazebo_ros_control`. The number of launched control-nodes, named `modrob[x]/joint[y]_position_controller`, thus depends on the number of robots and how many joints each robot possesses. They receive messages of type `std\_msgs::Float64`. However unlike directly publishing onto `joint_state`, joint angles are first passed through a position controller before they are applied to the robot's joints. This allows for more realistic simulations given the PID-parameters of the position controllers are tuned correctly.

Assuming a robot with n joints, this implies that n publishers of type `std_msgs::Float64` have to be set up to control such a robot. Our second abstraction layer circumvents this issue by distributing joint positions defined in the position field of `sensor_msgs::JointState` to the respective position controllers, akin to a reversed funnel as seen in figure 10. Thereby, only one message of type `sensor_msgs::JointState` containing all joint angles has to be sent to the respective robots `motion_publisher` node to control all joints at once. User defined nodes such as any type of trajectory generator, can be docked onto a robots `motion_publisher` to conveniently simulate complex motions.

The third abstraction layer is given by the `motion_terminal`, which receives user inputs in string-form via a terminal which converts these messages into `sensor_msgs::JointState`. These are then provided to the chosen robots *motion_publisher*. A `motion_terminal` can only control one robot at a time, however, the user has the ability to select which robot is currently controlled by typing $\#[x]$, where $[x]$ is the ID of the desired robot, as illustrated by figure 11. This can be performed at any time such that the user is capable of switching between different robots using only a single `motion_terminal`. This feature was built for the purpose of testing and debugging movements and allows for fast simulations of hand-crafted trajectories. By launching additional `motion_terminals` in different terminals, multiple robots can be hand-controlled simultaneously.

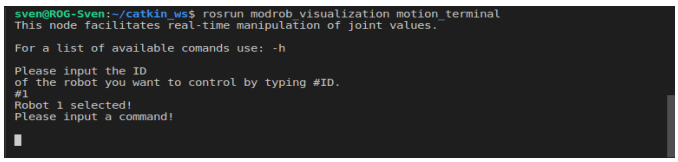


Fig. 11. An illustration of the *motion_terminal*, where the user currently selects the robot with ID `1` to be controlled.

Concrete examples and in depth descriptions of our *Control-Module* are detailed in the Motion Interface section of our wiki.

Controlling a robot in a predefined simulation will lead to unrealistic movement if parameters such as the PID-parameters of the moving robots' joints are not tuned correctly or if robots collide with each other or the environment. While the latter can be avoided by steering the robots clear of collisions through methods such as *self collision checking* and *trajectory planning*, the former needs to be addressed before any simulation takes place. We suggest spawning un-tuned robots within the simulation environment to determine whether the default joint angles lead to collisions during the spawning procedure and adapting them accordingly within the respective `modrob[x].urdf` file. After initial collisions have been ruled out, PID-parameter tuning using Ziegler Nichols or similar methods that can be performed by hand using the `motion_terminal` should be employed. We have applied a method for which a sinusoidal input is published onto a joint, while its response is measured. Tuning was performed by iteratively adjusting parameters until the measured and commanded input aligned as closely as possible.

V. Results

Within this section, all goals defined in the task description II are compared to the final results achieved in summer 2020s PCMR. The outcome of the practical course is evaluated based on whether the goals provided by the initial problem statements as well as the objectives we set ourselves were achieved. In the following subsections, evaluations of the RViz visualization and the Gazebo simulation are presented separately.

A. RViz: Results

The goals of the RViz team were to implement trajectory visualization features based on work-space and joint-space coordinates, in addition to creating applications according to requests from other teams. The specific tasks of the RViz team, referenced in the following part, are described in section II-A.

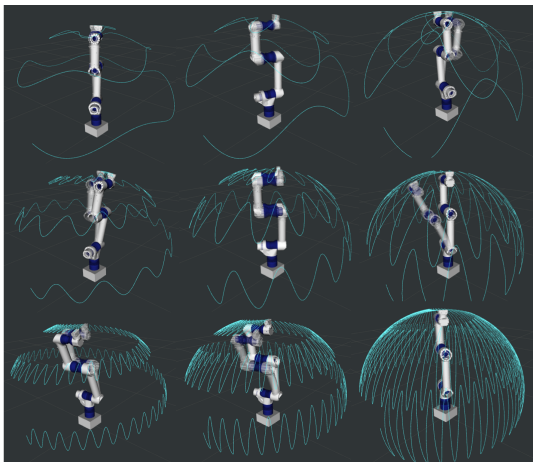


Fig. 12. Sample work-space trajectories that were created using the *dummy_trajectory_publisher* feature.

As described in section III-B, work-space trajectories can be recorded to and read from `bag` files. These files contain the work-space trajectories and can be manually generated. Additionally, we provide the `trajectory_3Dpoints_recorder` node for this purpose as well as the `trajectory_3Dpoints_visualizer` node capable of visualizing the contained information. Sample trajectories, that are displayed using this approach can be seen in figure 12.

The trajectory curve as well as the visualization based on inputs from the TP team prove the success of the joint-space approach. This can be further confirmed by comparing our results with the Trajectory and Path Planning team's outcomes, as depicted in figure 13. Thus, the validity and correctness of the calculated forward kinematics is verified. Further, the messages are successfully adapted from Moveit to KDL and `modrob_workstation`.

The `joint-space trajectory recorder` documents the trajectory in a bag file, while the

Trajectory.PNG

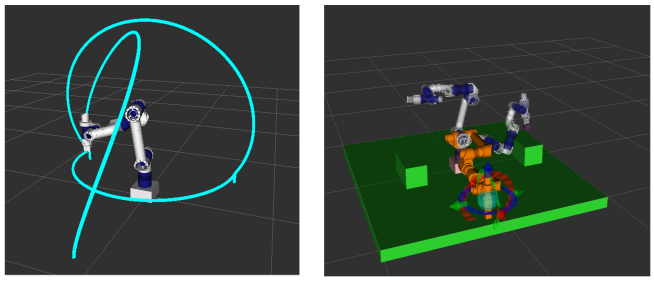


Fig. 13. Trajectory Visualization based on joint-space Coordinates as well as visualization of the modular robot following this trajectory (left). Visualization of Motion Planning performed by the TP team (right).

`joint-space trajectory visualizer` is rebuilding it from said bag file. Both processes result in no loss of data.

As previously mentioned in section III-E, the number of joint-space trajectory points greatly influences the visualization and thus needs to be tuned. The following setting has yielded the best results concerning the velocity of the visualized robot: receiving up to 3700 joint-space trajectory points, combined with a pause duration of 0.003 seconds between individual points, as pictured in figure 14.

```
if(ros::ok())
{

        measured_pub.publish(robotConfigMeasured);
        ros::Duration(0.003).sleep();
}
```

Fig. 14. The Pause duration between individual trajectory point is set to 0.003 for optimal visualization.

The first custom visualization request encompasses an image stream from RViz, proposed by the RC team. This task was achieved by the `rviz_camera_stream` as described in section III-F. It provides four principle views: top, front, right and orthographic as well as a fifth view located at the end-effector frame. It additionally resolves lighting issues through incorporation of directional lighting from `rviz_lighting`. The results are displayed in figure 15.

The image stream was further improved through added views in `rviz_camera_stream_3` that are end-effector-centered, as described in section III-G. These new perspectives have a fixed position relative to the end-effector frame. They thus provide more informative images for the viewer, as displayed in figure 16.

A second custom visualization request is constituted by user tests for the RC team. As described in section III-H, the user's performance was evaluated by measuring the

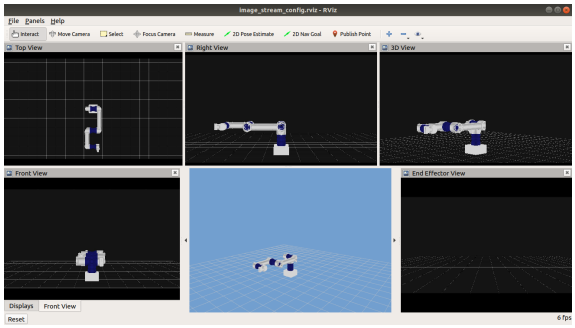


Fig. 15. The figure displays the image stream from RViz. Starting at the bottom-left image in clockwise direction: front, top, right, orthographic and end-effector views

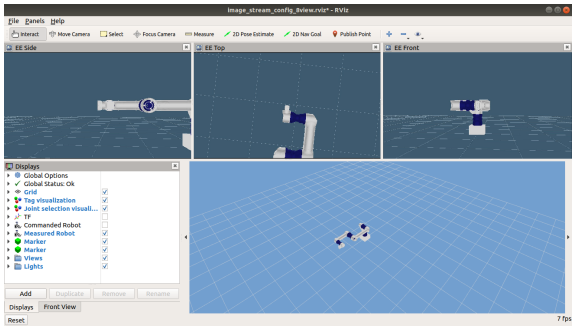


Fig. 16. Images generated by the end-effector following views. Left to right: side, top and front end-effector-centered view

distance between the end-effector and a given end-point as well as the distance between the end-effector and a pre-defined trajectory. The RC team have used the provided implementation for their tests. An example visualization of the `RC_userters_trajectory` can be seen in figure 17.

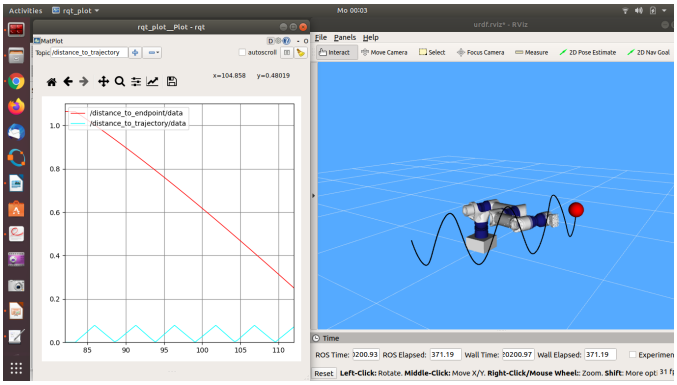


Fig. 17. A demonstration of the *RC_userters_trajectory* feature. The red plot displays the distance between the end-effector and the end-point (left), while the turquoise plot presents the difference between the end-effector and the trajectory (left). The black curve represents the trajectory to be followed by the user, where the red sphere is the goal (right).

In conclusion, all tasks that were decided on at the start of the semester were achieved. The combination of both trajectory visualization features is capable of visualizing a variety of trajectories stemming from different domains. Moreover, we provided visualizations according to custom requests from other teams. Additionally, we have improved the visualizations beyond what was initially described through features such as the end-effector-centered views.

B. Gazebo: Results

The innate objective of the Gazebo simulation was to produce realistic models of modular robots defined by arbitrary valid module orders within Gazebo. Additionally, environment models should be added that robots can interact with [3]. The specific goals are defined in section II-B.

As described in section IV-B, we are able to build modular robots from arbitrary valid module orders and visualize them in Gazebo. They can be manipulated through Gazebo as well as the *Motion-Interface* that we provide. Therefore, both the auto-generation and control objective were achieved through integration of the `gazebo_ros_control` plugin which provides realistic control inputs for all joints of any robot after parameter tuning has been carried out. As shown in section IV-D, we have reproduced the *Bulbs and Slides* environment as defined by [4] including realistically tuned friction values and hand crafted collisions. Additionally, a Gazebo model of a doughnut was designed in blender. We managed to prove that our system is capable of simulating manipulations of objects other than bulbs, by incorporating the doughnut in grasping experiments, depicted in figure 18. Therefore, all innate goals have been achieved.

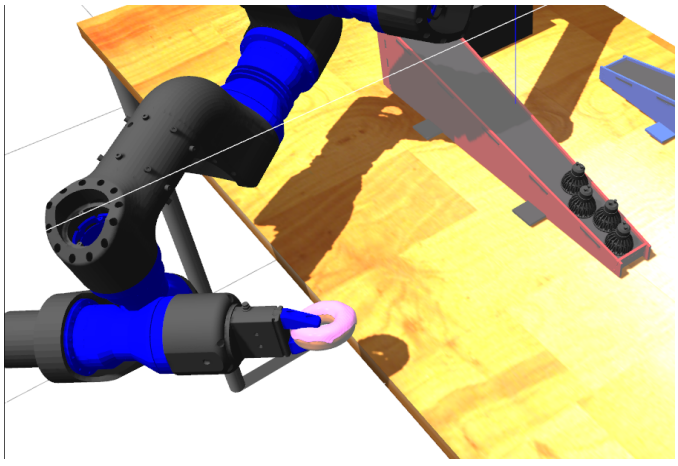


Fig. 18. An illustration of the six degree of freedom manipulator robot employed in the *Bulbs and Slides* experiment while grabbing a handcrafted doughnut. The doughnut model was produced in blender and is available as an .obj file within the models directory.

In the following paragraph, our personal objectives are evaluated. As mentioned in section IV-B, all robots share a predefined gazebo-friendly color scheme. The links are

painted in *Gazebo/DarkGrey* while the joints are colored in *Gazebo/Blue*. We built a gripper capable of grabbing and holding onto objects as defined in section IV-C. It can be attached to any modular robot by adding a flag to the creation-launch-command. Section IV-F proves that the user can spawn any amount of robots by supplying the *Build-Module* with as many module orders and base positions as desired. We describe in section IV-A that the control scripts are adapted to the number of robot definitions given. This enables the simultaneous control of any number of robots through their own motion interface within our *Interface-Module*, facilitated by the adaptive *Launch-Module* as shown in section IV-F. In section IV-D we describe the setup we used to define the environment of the *Bulbs and Slides* experiment. A snapshot of which is depicted in figure 19.

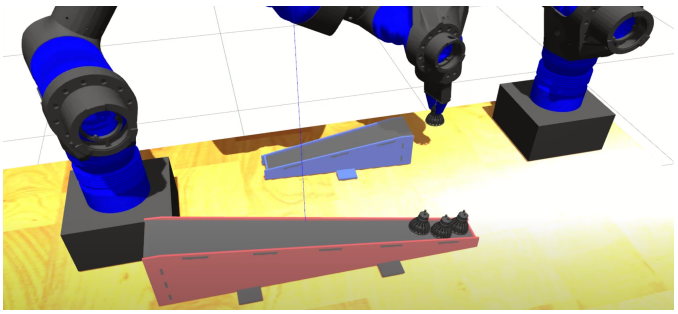


Fig. 19. A snapshot of the six degree of freedom robot's movement during the *Bulbs and Slides* experiment. The system is currently carrying a bulb from the red slide to the blue slide and is not yet hovering above its destination.

To prove the validity of our approach we performed a demonstration during which a six degree of freedom manipulator (module order defined as: $[59, 3, 55, 63, 64, 4, 66, 4, 67, 5, 56, 5]$ + gripper) performed a pick-and-place motion where the robot moved next to a bulb, grabbed it with its gripper, moved it towards a smaller blue slide and finally let go of the bulb while hovering over the slide resulting in a recreation of the movement presented in the real live demonstration of the single robot *Bulbs and Slides* experiment. We further provide proof that our manipulator and end-effector combination is capable of grabbing objects other than bulbs, as seen in figure 18.

Due to time constraints, no simulation of the two four degree of freedom robots performing a pick-and-place motion was recorded. However, the success of the six degree of freedom version proves the effectiveness and realism of our approach, which implies the same should be the case for the two robot setup. Finally, a self-built model of a doughnut was successfully manipulated. The latter was not part of any of the two goal sets and provides an additional asset that can be used in further simulations. In general, the results concerning a realistic simulation using Gazebo can be regraded as very promising, since we fulfilled all innate and all but one of the self imposed objectives. Therefore, simulating processes using our system was proven to lead to realistic results that can be used when referring to real life versions of the simulated robots.

VI. Conclusion

The importance of visualizing and simulating robotic systems is discussed in the preceding sections. Thereby, simplification and multiplicity of experimental reproduction as well as ease of access constitute the most prominent advantages when compared to real life demonstrations. We have outlined the innate objectives in form of expected working steps that we received from our advisors. Further, additional personal objectives of the RViz and Gazebo team were described respectively. We provide two packages: `modrob_visualization_rviz` and `modrob_visualization_gazebo`, where the former is used to **visualize** the movement of modular robots in RViz, while the latter performs realistic physics-incorporating **simulations** of robot manipulator behavior in Gazebo. Their modules and mechanisms are described in detail in section IV. Both teams achieved all innate objectives. Further, the RViz team managed to accomplish all self imposed goals. The Gazebo team has produced a realistic simulation of a six DoF modular robot performing a pick-and-place task within the pre-defined *Bulbs and Slides* environment.

Both packages allow for smooth integration within the PCMR project, wherein we have provided a test environment to support the other teams. This proved to be crucial since access to the real robot was seldom achievable. The interfaces we designed are upwards compatible with future versions of the PCMR project, since the modularity of our architecture allows for simple integration of new features.

Both the Gazebo and RViz teams have identified aspects that can be considered future work and thereby an initial starting point for next semester's RV students. The RViz team has examined the following areas of the existing trajectory visualization which are believed to further improve the RViz package within the PCMR project:

- Visualizing the trajectories' direction,
- Clipping irrelevant parts of the trajectory, i.e. highlighting upcoming segments and neglecting those that the modular robot has already passed,
- Reducing the number of milestones if the given trajectory is too granular.

The Gazebo team has identified the following features as contributory to the robot simulation and the PCMR project:

- Implementing and attaching the end-effector using XML macros to allow for an even higher degree of modularity,
- Applying inverse kinematics to the motion interface in cooperation with the TP team, enabling control through Cartesian-Space-Coordinates instead of joint angles,
- Controlling and regulating the robot's PID controllers with an advanced analytical approach, for instance, using the Ziegler-Nichols method,

- Simulating the *Bulbs and Slides* demo application with two 3 DoF robots,
- Executing additional experiments with different graspable objects, such as the provided doughnut model.

Please note that these proposals are not meant to be exhaustive, they rather present opinions and suggestions for improvements based on critical reflections of our own achievements.

We firmly believe that our applications can support the future progress of this practical course. Further, by taking inspiration from the previously mentioned recommendations even more feature-rich versions of both packages could be built.

VII. Appendix

References

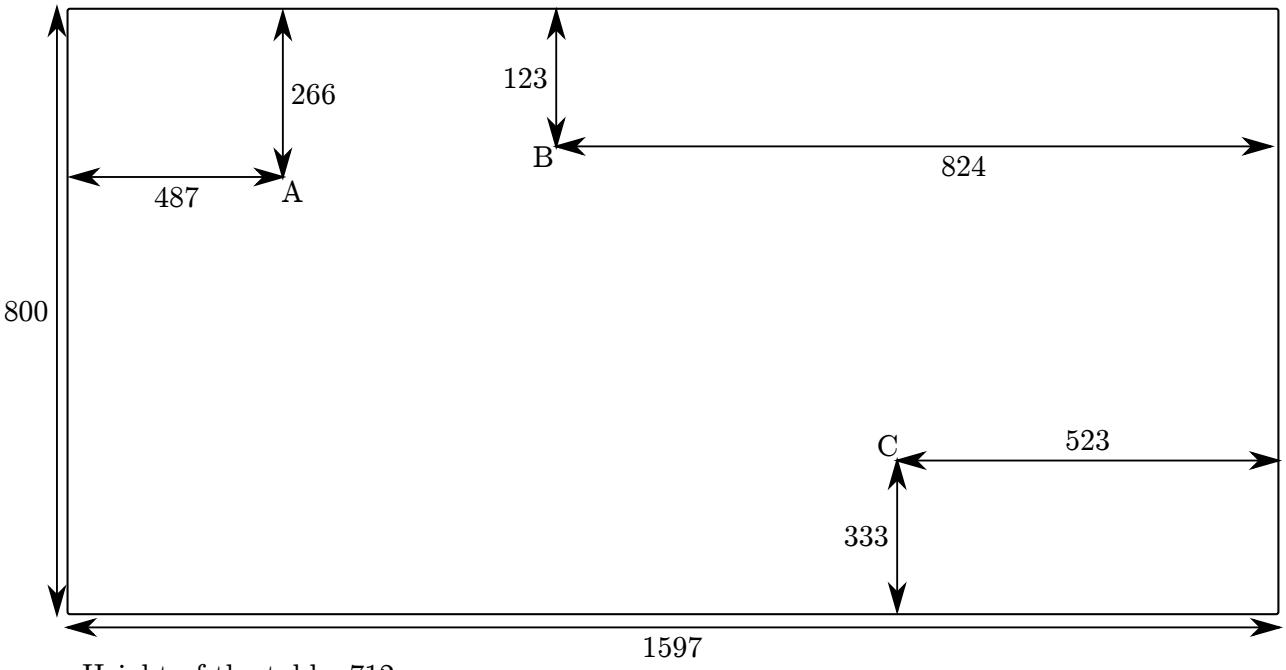
[1] M. Althoff and S. Liu, *Buidling a Modular Robot - Course Description*, URL:https://www.in.tum.de/i06/teaching/ss-18/practical-course-building-a-modular-robot/

[2] P. Goldbrunner and T. Piltz and T. Karlgren and A. Schultz, *Robot Visualization*, Final Report of PCMR, 2019

[3] R. Hölzl and M. Riedel and C. Dresel and S. Manzinger and P. Maroldt and M. Mayer and S. Liu, 2_$topics$, Material of PCMR, 2020

[4] C. Dresel, $bulbs\_and\_slides$, Material of PCMR, 2020

Demo Application: Bulbs and Slides

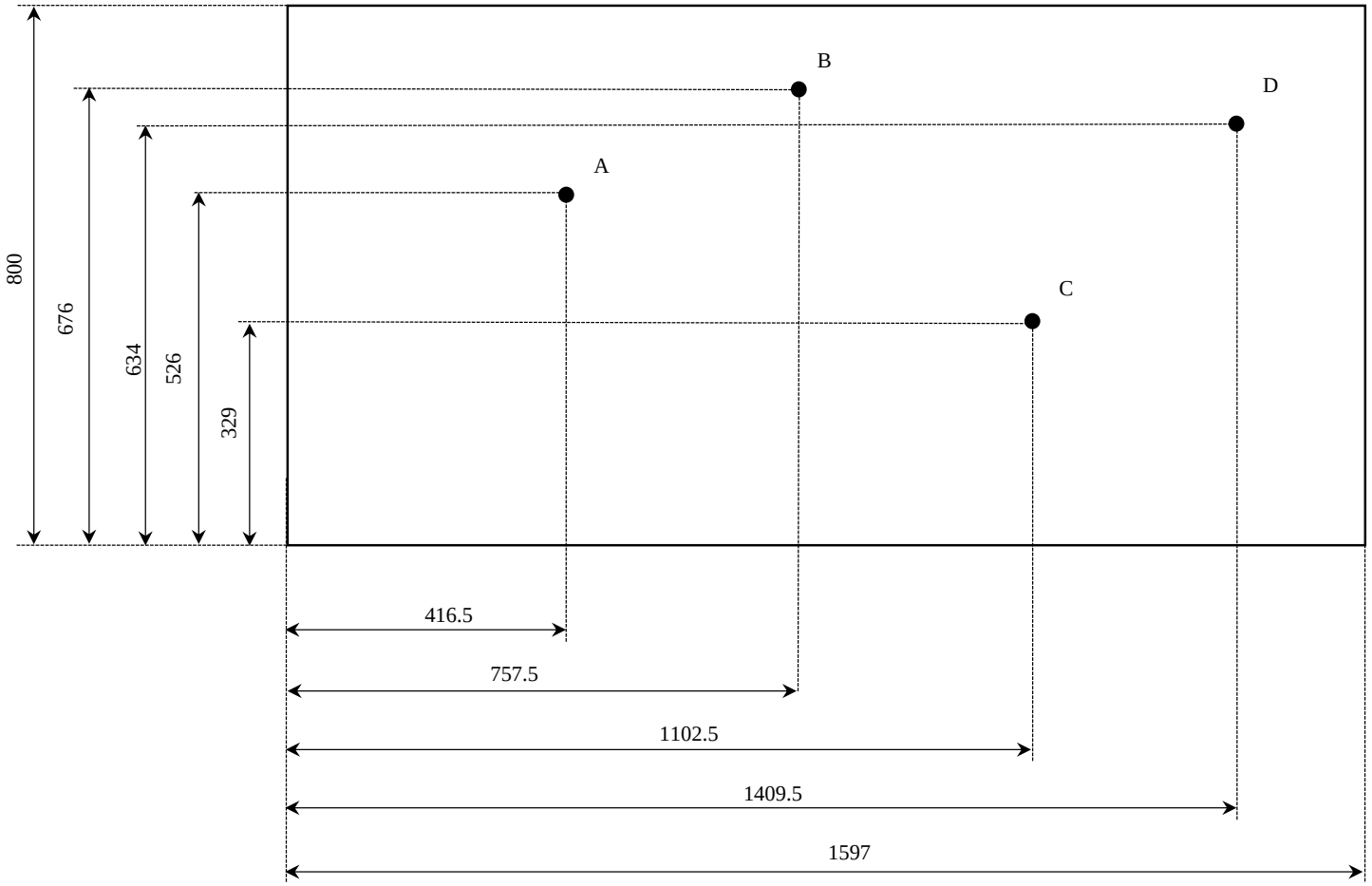


Height of the table: 712

A: Center of the robot base.
B: Center of the tip of the short slide.
C: Center of the tip of the long slide.

Both slides are aligned with their center-lines in parallel to the long sides of the table.
All measurements are given in millimeters.

Demo Application: Bulbs and Slides for Two Robot Configuration



All measurements are given in [mm]. Please be aware that the unit length used in Gazebo is given in [m].
All heights are 1020.
A: The center of the base of modrob1
B: The right corner of the small tip of the short slide
C: The left corner of the small tip of the long slide
D: The center of the base of modrob0