University of Innsbruck

Institute of Computer Science

Intelligent and Interactive Systems

Robot Simulation and Motion Planning

Martin Griesser

B.Sc. Thesis

Supervisor: Emre Ugur, PhD
Univ.-Prof. Dr. Justus Piater, PhD
28th September 2014

Abstract

Der erste Teil dieser Arbeit befasst sich mit der realitÃďtsgetreuen Nachbildung der Versuchsanordnung im IIS-Lab in einer Simulationsumgebung und der Implementierung der entsprechenden Steuerungsschnittstelle. AnschlieÃ§end wird die Integration und Konfiguration des Motion-Planning Frameworks MoveIt sowohl fÃijr den Simulator, als auch fÃijr den realen Roboter erlÃďutert. Am Beispiel eines Referenz - 'Pick and Place' Tasks wird anschlieÃ§end die Verwendung der Planungstools erlÃďutert. Messungen (Kollisionen, IK-Tests, Vergleich verschiedener Planungsalgorithmen). MoveIt Konfigurationen fÃijr unterschiedliche Setups kÃűnnen sowohl fÃijr den Simulator, als auch mit dem richtigen Roboter verwendet werden.

Acknowledgements

Emre Ugur (Betreuer, Ideen) Alex Rietzler (Transformationen, Torso Modell, Technische Daten SDH) Simon Hangl (Kukie Schnittstelle) Simon Haller (Technische Hilfestellung, System, Git) All other members of the IIS team that frequently provided very useful feedback.

Contents

List of Figures

List of Tables

Declaration

By my own signature I declare that I produced this work as the sole author, working independently, and that I did not use any sources and aids other than those referenced in the text. All passages borrowed from external sources, verbatim or by content, are explicitly identified as such.

Signed: ..... Date: .....

Chapter 1

Introduction

Robots play an increasingly important role in today's world because they enlighten human work in many different areas. Robots allow the automation of production processes but they are also utilized in other areas like medical surgery or even as household robots (autonomous vacuum cleaner, lawn mower, ...). Depending on the application domain, those robots require the ability to act in differing levels of autonomy. (LaValle, 2006) describes the branch of *robotics* as the area of automating mechanical systems that have sensing, actuation and computation capabilities. Part of the research in that branch is to create control software that allows to autonomously perform high-level tasks like grasping and object manipulation on robot hardware. The design and implementation of such high-level control software for a robot is a cumbersome task. The algorithms need to be tested and debugged during the implementation process, but those tests come with a certain level of risk. Incorrect algorithms can lead to damages on robot components or their environment and entail costly repairs. In the worst case even people can get hurt by uncontrolled robot motions.

One solution to those problems is the usage of a simulator that physically models the robot and its behaviour as accurate as possible. Therefore it has to provide the same control interface, allowing to test and debug each part of the software on the simulator before utilizing it on the real robot. Using a simulator allows to easily evaluate different design approaches and algorithms during the software development process. It can act as replacement for the real robot and facilitates parallelization of testing and debugging tasks, as the robot may be blocked by other persons or unavailable at certain times. It also allows to skip the technical overhead that often comes with working on the real device. Those considerations motivated the first part of the thesis which focuses on the design and implementation of a simulation solution for the robot setup in the IIS-Lab along with the corresponding control interface. This includes to create suitable simulation models of the involved robot components that show physically similar behaviour as their real counterparts. The necessary steps are explained in chapter 2.

The second part of the project focuses on robot motion planning. Control software for an autonomously acting robot needs to be able to plan the motions, the robot has to perform to fulfil a task. Moving the robot's hand for example cannot follow any arbitrary trajectory towards a desired target pose. During that motion it might collide with itself or any other obstacle within its environment. Therefore those trajectories have to be planned and executed carefully to avoid accidental collisions and to generate smooth and well controlled motions with respect to the limits of the robot. The resulting trajectories should be preferably short and may not contain unnecessary motions. Planning such robot motions is the aim of a motion planning framework. A motion planning framework is a software component that provides infrastructure

which can be used by other high-level control software to generate motion plans for given problem descriptions. Therefore the planner requires a detailed description of the physical properties of the robot and its environment. The integration of such a motion planning framework into the IIS-Lab robot setup is explained in chapter 3.

Both parts of the thesis require to determine the kinematic, dynamic and volumetric properties of the involved robot components. This also includes to do an exact measuring of the existing robot setup in the IIS-Lab and determine the placement of the various elements relative to each other. This information can then be used to create models that exactly describe the robot and its environment. The accuracy of those models and the proper functioning of the integrated motion planning framework is shown on the example of a benchmark pick and place task that can be executed on the simulator and the real robot as well. Chapter 4 gives an overview about grasping tasks in general and describes the implemented benchmark task in detail.

1.1 The IIS-Lab Robot setup

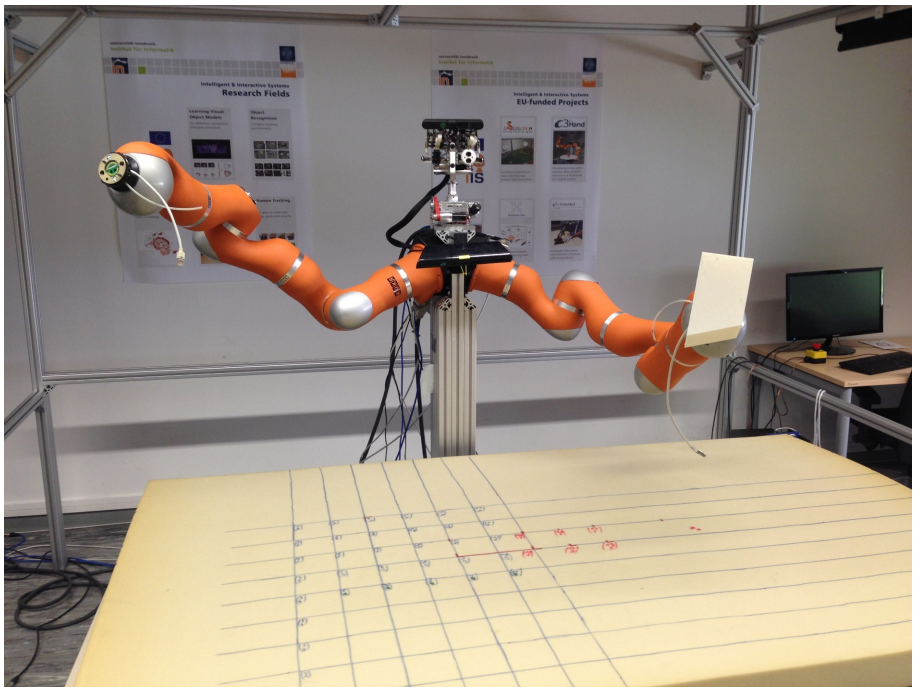

Figure 1.1: Current setup in the IIS-Lab

The structure of the robot setup in the IIS-lab changes frequently, as new robot components are introduced and arrangements are modified. The setting considered within this thesis is a snapshot though the solutions that will be developed should be customizable to reflect alternating settings. The main part of the robot setup consists of an aluminium torso with two mounted 7 DOF[1] Kuka LWR4+ (KUKA, 2012) robot arms, as shown in Figure 1.1. The IIS-Lab also owns two Schunk SDH-2 (Schunk, 2010) robot hands that can be mounted to the robot arms for grasping experiments. The workspace where those experiments usually take place is the table in front of the robot which is covered with a foam mat for security reasons. Additionally

[1] Degrees of freedom - the number of independent variables, necessary to describe a robot's configuration

there is a Kinect camera mounted on the robot torso, between both arms. A Kinect camera contains a vision sensor and a depth sensor. The vision sensor provides RBG images of the task environment. The depth sensor returns depth images where each pixel represents the distance to its corresponding point in the world (Andersen et al., 2012). Control and data exchange with all available components is based on the robot operating system ROS which will be described in the next section.

1.2 The Robot Operating System (ROS)

The implementation of the project requirements is based on ROS. Therefore a brief introduction about the basic concepts[2] shall be given here. The explained terminology will be used throughout this thesis. As stated in Quigley et al. (2009), ROS is not an operating system in the classical sense. It runs on top of a host operating system (usually linux) and can be seen as an additional communication layer, providing various mechanisms for inter-process communication. A ROS system consists of a number of *nodes*. Each node is an independent computation unit that runs in its own process, adding clearly defined functionality to the overall system. For example one node can be responsible for planning, another one for perception and a third one for controlling the hardware. Nodes communicate to each other by passing *messages*, using the ROS communication infrastructure. Messages are strictly typed data structures, defined in a special message composition format[3]. They can be composed of primitive types like float, integer or string, but also of other message types. Therefore it is possible to create arbitrarily complex messages for each use case.

Messages are published to *topics*. A topic is a strongly typed message bus, addressed by its name. Arbitrary nodes can connect to a topic in parallel, as long as they use the correct message type. Each node can publish and subscribe to a number of topics. It is also possible that various nodes publish to the same topic. Topic names are strings, used to identify topics. They can be organized into namespaces to build a tree hierarchy comparable to the directory structure in a file system. This is very important, as for example the simulator should use similar topic names as the real robot. The namespace concept allows both instances to use identical names but each one in its own namespace. The following samples represent valid topic names:

- `/` (this is the root namespace)
- `/component/topic`
- `/simulation/component/topic`
- `/real/component/topic`

The topic names used in the current project follow the IIS lab internal naming conventions. Each topic name has the structure

```
/[namespace]/[component name]/[control type]/[name]
```

The namespace is necessary to distinguish between the topics of simulator and real robot. All simulator related topics reside in the `simulation` namespace. The component name identifies a specific device within that namespace, e.g. `left_arm` or `right_sdh`. The control type groups sets of topics into different categories. Possible values are `joint_control`, `cartesian_control`,

[2] http://wiki.ros.org/ROS/Concepts
[3] http://wiki.ros.org/msg

`sensoring` and `settings`. Finally, the chosen name is used to identify a specific topic within that category. For example the topic `/simulation/left_arm/joint_control/move` states a topic named `move` in the category `joint_control` for a simulated component, named `left_arm`.

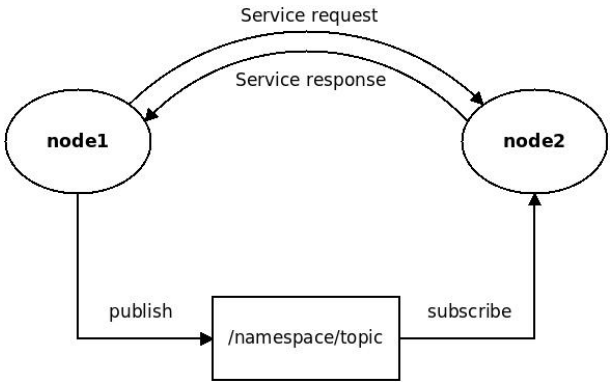

Figure 1.2: ROS nodes, topics and services

The communication via ROS topics is asynchronous - involved nodes may even not be aware of each others existence. Synchronous message exchange between nodes happens via ROS *services*. In contrast to topics, a service with a given name can only be offered by one single node. Services are addressed, using the same naming strategy as topics. The service message is composed of a request and a response part. A client node that sends a service request will block, until the advertising node has handled the request and delivers a response. The concepts of ROS topics and services are shown in Figure 1.2.

The nodes of a ROS system can be distributed over various different machines. One of them has to be the dedicated *ROS master*. The master is responsible to handle topic and service registrations and holds information about the involved ROS nodes. Other machines connect to the master via network. The ROS master also provides a centralized *parameter server*. This is a shared dictionary that can be used to store and retrieve configuration data and other shared parameters. Nodes can access the parameter server at runtime and read or modify its content.

A system usually consists of a large number of nodes that have to be configured and started. This startup process can be automated, using so called *launch files*. Those are simple textfiles, holding startup information and configuration details for one or more nodes in an XML like syntax[4]. Using the *roslaunch* command line tool, a whole system of nodes can be configured and launched at once.

ROS is a modular software system organized into *packages*. Each package adds clearly defined functionality and can be reused in other systems. Custom functionality is added to a ROS system by creating a new package and developing the required piece of software. A package might contain one or more ROS nodes or even only configuration data. Existing packages are usually installed, using a software repository package manager. A very useful ROS package is the visualization tool *RViz*. This tool provides a number of different plugins that allow to display robot setups and configurations, planned trajectories or image data from a vision sensor. RViz gets utilized in the motion planning related part of this project.

[4] http://wiki.ros.org/roslaunch/XML

1.3 Project Targets

The first goal of this project is to create a realistic replication of the IIS-Lab robot setup, using a suitable simulation platform. The solution needs to be able to generate proper sensor data and provide the same ROS control interface as the real robot. Additionally it would be preferable that the solution is able to detect and visualize accidental collisions of robot components with the environment. The necessary steps are explained in chapter 2.

The second objective is to choose and integrate a state of the art motion planning framework into the existing setup. The required functionality includes solving inverse kinematics[5] (IK) problems and planning collision-free trajectories for complex robot motions in joint space and Cartesian space. This integration process is described in chapter 3.

The proper functioning of the planning framework will then be shown by planning and executing a benchmark pick and place task as explained in chapter 4. That chapter starts with an overview about grasping tasks in general and explains the resulting planning problems in detail. After that it will be shown how those planning problems can be solved using the planning framework and how the resulting motion plans are executed on the simulated or real device.

The implementation of those objectives requires to determine the kinematic, dynamic and volumetric properties of the involved robot components. This includes to do an exact measuring of the existing robot setup in the IIS-Lab and determine the placement of the involved components relative to each other to be able to create an internal representation of the world.

[5] Problem of finding possible joint settings for the robot to achieve a desired end effector position and orientation in Cartesian space (Craig, 2005)

Chapter 2

Robot simulation

This chapter focuses on the simulation part of the thesis. The objective is the creation of a simulation model, particularly designed for the IIS lab robot setup. This model needs to reflect the properties and the physically behaviour of the robot arms and the grippers as good as possible. Certainly the simulated components have to provide the same control interface as their real counterparts, allowing to test and optimize control code on the simulator before utilizing it on the real robot. Preferably, the control code sees no difference about on which instance it is executed. The requirements to such a solution can be summarized as follows:

- The simulator needs to be able to generate realistic sensor and feedback data that can be used by the control software. This includes forces and torques that are measured within force sensors, the current state of the various robot joints (position, velocity, effort) and also RGB and depth images, usually produced by Kinect cameras.
- The simulation solution has to provide exactly the same ROS control interface as the real robot. This interface essentially consists of a number of ROS topics, that can be used to send control commands to the various different robot components or to read actual joint states and sensor data.
- The utilized simulation platform has to provide a graphical user interface that allows to visualize the motions of the robot and it's interaction with the environment. Possibly accidental collisions of robot parts have to be registered and should also be visualized.
- In order to be used by as many people as possible, it is very important that the solution is really easy to use and does not require a long lead time. Therefore it has to be put particular focus on usability and documentation.

The following sections explain the design of such a simulation solution step by step. In the beginning stands the process of finding a suitable simulation platform that meets the considered criteria. After that the chosen platform is introduced and general information about the concepts and used terminology is given. Subsequent sections focus on the necessary steps to achieve the final solution, namely modelling required robot components, assembling and configuring the simulation scene and the implementation of the ROS control interface.

2.1 Choosing a suitable simulation platform

The tasks executed on the robot are in most cases variations of so called *pick and place* tasks. An object gets picked up, lifted and placed somewhere else within the robot's workspace. Therefore

joint target positions are sent to the control interfaces of the various robot components and they execute the commanded motions if possible. The question, if the execution is possible at all and in that case in which velocity, is influenced by a number of dynamic parameters. The maximum effort of the motors in the joints is limited. If the force that acts upon a joint is higher than the maximum effort of the motor it will not be able to maintain its current position or to reach the desired target position. This can happen if the picked object is to heavy or if the robot collides with an immovable object in it's environment. The forces that act upon each single joint are influenced by a number of parameters like the position within the kinematic chain of the robot, the summed own weight of the robot components and also the weight of a possibly additional payload.

The required solution should be able to provide a realistic simulation of those dynamic interactions. Therefore the utilized simulation platform has to make use of a physics engine. A physics engine is a software component, that is capable of computing parameters of physical processes and the dynamic properties of the involved objects. Examples for such engines are the Open Dynamics Engine[1] (ODE) and Bullet physics[2].

The candidates that have been taken into account were Gazebo[3], V-Rep[4], MORSE[5] and Openrave[6]. After some investigation only two of them (Gazebo and V-Rep) had been evaluated in greater detail. Criteria for the selection had been:

- Which physics engine is used respectively is it possible to choose among various engines?
- Usability and stability
- Expandability
- Availability of required model components (arm model, gripper,...)
- Quality of the documentation
- Licence issues

Taking into account those criteria it went clear that V-Rep will be the simulation platform of choice. In some initial tests V-Rep seemed to be much more stable than Gazebo and the user interface is very intuitive. Another important point is that V-Rep ships with a fully functional model of the KUKA LWR4+ robot arm.

2.2 The Virtual Robot Experimentation Platform(V-Rep)

This section gives an overview of the utilized simulation platform. V-Rep is a robot simulator, developed and maintained by Coppelia Robotics[7]. The current version (V3.1.2) allows to choose among three configurable physics engines, namely ODE, Bullet and Vortex (only trial version) for simulating dynamic processes. The simulation environment is modelled, using the included scene editor. The distribution contains a large number of predefined models for different robot types. Additional modules provide functionalities related to collision visualization (*collision*

[1] http://www.ode.org
[2] http://bulletphysics.org
[3] http://gazebosim.org/
[4] http://coppeliarobotics.com
[5] http://www.openrobots.org/wiki/morse/
[6] http://openrave.org
[7] http://www.coppeliarobotics.com

detection module) and inverse kinematics calculation (*IK calculation module*). The behaviour of the simulator is highly customizable and its functionality can be extended via plugins. A plugin is a software library that is written in C++ and interacts with the simulator by using the provided programming API. V-Rep also allows to add embedded scripts to specific simulations. Those scripts need to be written in the scripting language LUA[8]. V-Rep is no open source software but it provides a free licence for educational units and can therefore be used for research purposes. Freese et al. gives a detailed insight into the V-Rep simulation platform.

2.3 Dynamic simulations in V-REP

For a better understanding of the modelling process it is necessary to explain a few fundamental concepts about designing simulations in V-Rep. This section covers the aspects that are important for the underlying project along with the used terminology. A detailed explanation can be found in the official V-Rep documentation[9].

Simulations in V-Rep are organized into *scenes*. A scene represents the simulated world which is composed from a number of elements like three dimensional bodies, joints, sensors or cameras that are combined in a tree structure and arranged within the environment. This tree structure forms the scene hierarchy. The elements within that hierarchy are called *scene objects*. The scene hierarchy can be modified in the *scene hierarchy view*, located on the left hand side of the scene editor. V-Rep provides various types of scene objects, but only those, which are important for the implementation will be explained here.

- **Shape**

  Rigid objects like robot links or parts of the furniture are represented as shapes. V-Rep distinguishes between primitive shapes (cylinder, cuboid, sphere, plane and disk) and complex shapes (triangle meshes). Various shapes can be combined to groups and therefore treated as one single object. Shapes can be defined to be *static* or *non-static*. The position of a static object is fixed relative to its parent node within the scene hierarchy and will not change during simulation. Non-static objects underlie gravity and will fall down if they are not constrained by a joint or a rigid connection. It is also necessary to distinguish between *respondable* and *non-respondable* shapes. Respondable shapes are handled by the physics engine and create collision reactions when colliding with other respondable objects during simulation. Therefore their physical attributes like mass, moment of inertia and material settings need to be clearly defined. Non-responable shapes are visible in the scene but they do not create collision reactions. Shapes allow to set additional attributes that specify special properties, used by other modules of the simulator. The *collidable* attribute states that the shape is considered by the collision detection module. The *renderable* attribute marks a shape to be recognized by a vision sensor.

- **Joint**
  A joint is a flexible connection between two rigid parts of a robot. All the joints that were used within this project are *revolute* joints which allow the rotation on a single-axis. Those joints are actuated by a motor. The motor settings include maximum force or torque and velocity limits. V-Rep supports joint control in three different modes:

[8] http://www.lua.org
[9] http://www.coppeliarobotics/helpFiles

  - **Torque/force mode**
    The behaviour of a joint in *torque/force* mode is simulated by the physics engine, based on the specified motor limits. The joint is operated by simulated controller that allows to set a target position. The controller then actuates the joint towards the commanded target position. This is the most realistic control mode as the dynamic parameters of the joint motor and the connected bodies are taken into account.
  - **Inverse kinematics mode**
    A joint in *inverse kinematics mode* is controlled by V-Rep's IK calculation module. The configuration option *hybrid operation* specifies, that the dynamic parameters of the joint are also taken into account.
  - **Dependent mode**
    Operating a joint in *dependent mode* means that it's position depends on the position of another joint within the scene. This dependency is formulated as *dependency equation* which calculates a target position based on the current position of the connected joint.

  The configuration of the angular joint limits is defined based on a minimum position and the maximum possible opening angle.

- **Vision sensor**
  A vision sensor is a simulated image capturing device. Each vision sensor captures images depending on its location within the scene and the configured opening angle and resolution. It is able to produce sequences of RGB and depth images as well.

- **Force sensor**
  A force sensor in V-Rep is used to rigidly connect two non-static, respondable shapes. The force sensor then measures the linear forces and and angular torques that act upon this connection. It is also possible to define a maximum force the connection is able to bear. When this maximum value is exceeded, the connection will break.

- **Dummy**
  Dummies are the simplest scene objects but they provide important functionality for the various modules of V-Rep. Dummies are used to explicitly define the origin of a specific reference frame, e.g. the world reference frame or the base of a robot arm. Other scene objects can then be configured to be child objects of such dummies within the scene hierarchy. Their relative placement can then be adjusted with respect to that reference frame. Two dummies can also be linked as *tip-target pairs* to be used by the IK calculation module. This functionality is explained in section 2.4.4.

Groups of scene objects can be organized in *collections* and treated as one single entity. Collections play an important role in the collision detection module. Let $\mathcal{A}$ and $\mathcal{B}$ be two distinctive collections of collidable shapes. The collision detection module allows to explicitly check collection $\mathcal{A}$ for collisions against collection $\mathcal{B}$. Collections also allow to override the collidable or renderable settings of all contained shapes. The configuration of the collision detection module is explained in section 2.4.3.

Scene objects can be loosely arranged within the scene hierarchy or be part of a *model*. A model is a subtree of elements within this hierarchy that logically belong together, as they form a specific object like a robot arm or a table. The root element of this model tree is called the *model base*. Models can be saved in model files and therefore be reused in various different scenes. Robot models are composed from rigid bodies called *links*, connected by joints

that allow to actuate the flexible parts of the simulated robot component. Each link is usually represented by two different shapes. The first one forms the visual geometry and is usually a non-respondable, complex shape. The second shape forms the collision geometry which has to be a simplified approximation of the visual geometry, composed from groups of primitive shapes. That shape states the respondable part of the robot link as it is seen by the physics engine and needs therefore appropriate configuration of the dynamic parameters. That part is usually invisible but very important as it applies realistic behaviour to the model and allows it to interact with other respondable objects within the scene. Without respondable parts, the model would just move through other bodies as only respondable shapes are able to produce collision reactions.

This separation into visual and respondable part is necessary because V-Rep and the utilized physics engines are not able to use arbitrarily complex shapes for dynamics calculations. Utilized meshes have to fulfil at least the *convex* criteria[10], but the V-Rep documentation recommends to use groups of primitive shapes for the collision geometry to increase the simulation performance. The V-Rep scene editor provides a special *shape edit mode* that allows to edit and simplify meshes and to extract different types of primitive shapes from a selection of vertices. The shape edit mode can be entered by selecting the desired shape and then click the item `Tools -> shape edition` from the menu bar or the corresponding tool bar button.

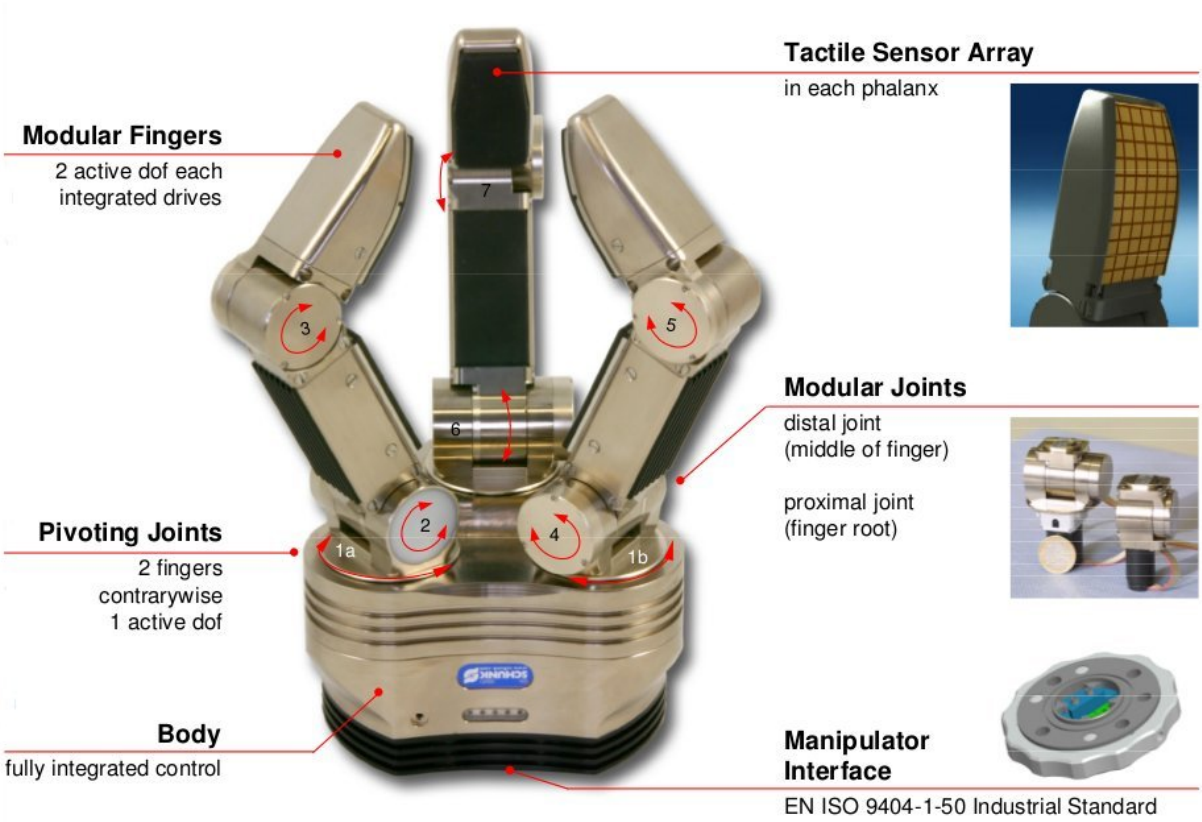

Figure 2.1: Schunk SDH-2 gripper
Image source: Schunk (2010)

[10] A line between any two vertices of the mesh lies completely within the mesh boundaries

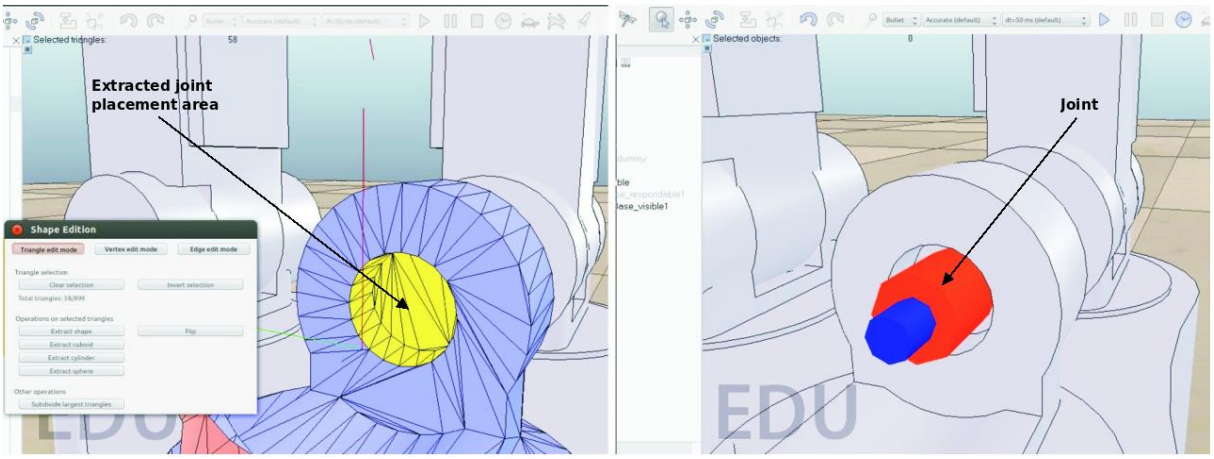

Figure 2.2: Placing a joint within the model

2.4 Designing the simulation scene

This section explains in detail the structure of the simulation scene and the contained models. The setup contains three types of robot component - the Kuka LWR4+ robot arm, the Schunk SDH-2 gripper and the Kinect camera. For each one of them, a simulation model was required in order to be able to build up the scene. V-Rep ships with realistic and fully functional model for the Kuka arm and also for the Kinect camera. But as there is currently no model for the Schunk SDH-2 robot hand available, it had to be created from scratch. The necessary steps of the modelling process are explained in the next section.

2.4.1 Modelling the Schunk SDH-2 gripper

Figure 2.1 shows the Schunk SDH-2 hand. The gripper has 3 fingers, each one containing two modular joints. The joints located closer to the wrist are called the *proximal* finger joints whereas the joints, actuating the finger tips are called the *distal* finger joints. The joints 1a and 1b in Figure 2.1 can be rotated along their vertical axis though their rotation angles are codependent. That means if one finger rotates to the left, the other one is rotated to the right for the same angle, actually adding one additional degree of freedom. Those two joints are called the *pivoting* joints.

The visual part of the hand model basically consists of 4 different shapes - the gripper body, finger knuckles, finger links and finger tips. Suitable meshes were taken from the `schunk_description`[11] ROS package and imported into the V-Rep editor. They have then been arranged according to the technical description in (Schunk, 2010). The next step was to connect those parts by the corresponding joints.

The correct placement of those joints is very important to achieve the desired transformations of the actuated parts when changing the joint angles. Therefore it was necessary to determine the correct position and orientation for each single joint within the model. Each flexible part of the gripper has cylinder shaped holes at the locations where the parts are connected by joints. Those holes also exist in the utilized meshes and they were used to extract the correct joint placement locations. This was achieved by selecting the vertices within the mesh that form the area, where a specific joint had to fit. From that selection a cylinder was extracted, using the

[11] http://wiki.ros.org/schunk_description

corresponding editor functionality. The origin of that cylinder's reference frame which is located in the cylinder center stated the exact place location for the corresponding joint. After creating a new joint on that location, the extracted cylinder could be removed again. Those steps had to be repeated for all 8 gripper joints. This placement process is visualized in Figure2.2. The left image shows the extraction of the target area. On the right image, the joint is already placed on it's appropriate location.

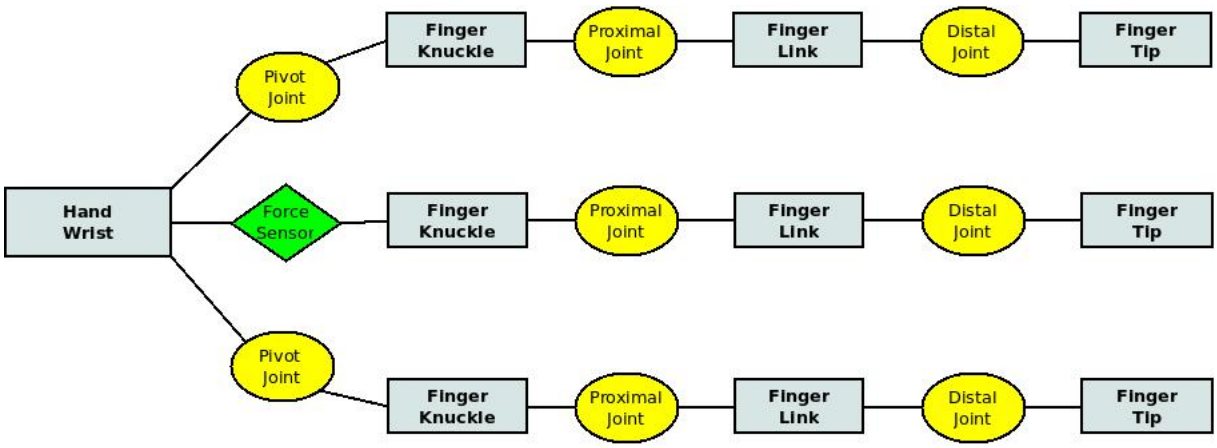

Figure 2.3: Kinematic chain of the Schunk gripper

After placing all the joints and links within the scene, the model tree was adjusted to form the kinematic chain of the hand as can be seen in Figure2.3. The dynamic parameters of the joints were set according to the technical description in (Schunk, 2010). The configured values are listed in table 2.1. As the positions of the pivot joints are connected to each other, the configuration of the second finger's pivoting joint had to be set sightly different. To achieve this mirroring behaviour the joint is operated in the dependent mode and its dependency equation f is defined as $f(x) = -x$ where x is the position of the first finger's pivoting joint.

Joint type	**pos. min.**	**pos. max.**	**max. effort**	**max. velocity**
Pivoting joints	$0°$	$90°$	$5.0Nm$	$210°/s$
Proximal joints	$-90°$	$90°$	$2.1Nm$	$210°/s$
Distal joints	$-90°$	$90°$	$1.4Nm$	$210°/s$

Table 2.1: Joint dynamic parameters

The visual part of the gripper model was formed by the utilized meshes. The next step in the design process was to define the collision geometry that was used to model the underlying physics. We decided to do that by approximating the original shape by groups of primitive shapes. This was achieved by executing the following steps for each part of the model:

- The shape edit mode was used to locate and select parts of the mesh that could be approximated by either a cuboid or a cylinder. The extraction was done by selecting groups of suitable vertices with the mouse while holding the `shift` button.

- Those parts were then extracted as primitive shape by using the corresponding editor functionality. The extraction process is visualized in Figure2.4.

- Those steps had to be repeated until the most important parts of the shape were extracted that way. After that all extracted shapes were grouped into one single shape.
- The respondable flag was activated for the resulting shape and the dynamic parameters were adjusted accordingly. The mass values and the inertial matrix have been provided by Alex Rietzler who had already evaluated those values for his own project. The material settings were left at the default value which is the predefined *highFrictionMaterial*. This material has a friction coefficient of 1.0 which is the maximum possible value but it uses no linear or angular damping.
- For the name of the extracted shape the same value was used as for its visual counterpart but the `_respondable` suffix was appended. The suffix allows to easily identify the respondable model parts within the scene hierarchy.
- The extracted shapes were then removed from the active visibility layer because they are just used for dynamics calculations.

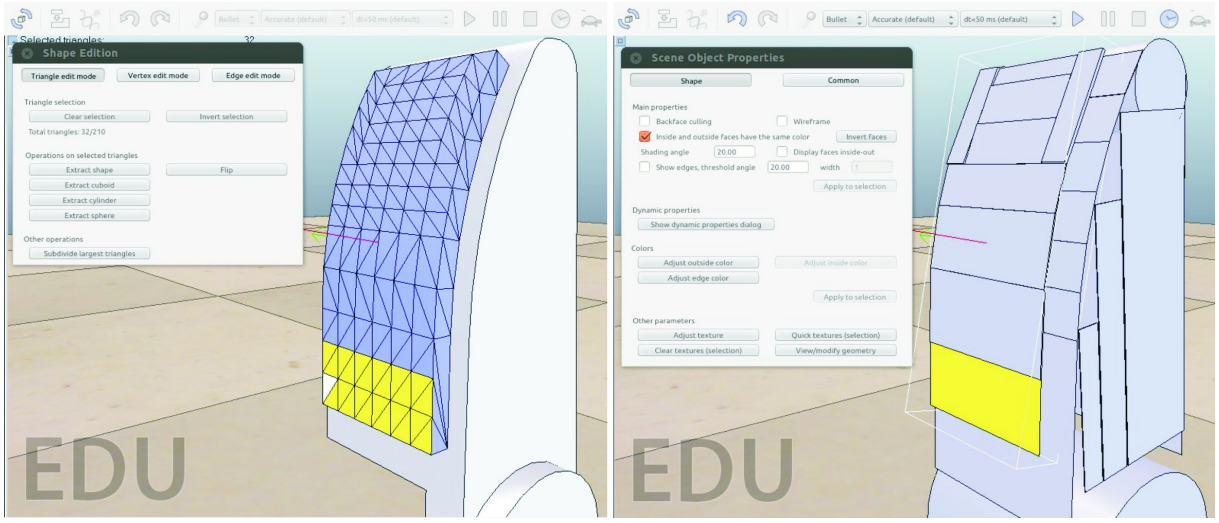

Figure 2.4: Process of approximating the original mesh with pure shapes

The last step of the modelling process was the adjustment of the model tree. This adjustment was done by rearranging the nodes within the scene hierarchy view. Root element of the hand model is the respondable part of the gripper body which was also defined to be the dedicated model base. At this point it is very important to follow the V-Rep guidelines for designing dynamic simulations because if the hierarchy is wrong, the model will simply fall apart when starting the simulation (detailed information about that topic can be found in the corresponding chapter[12] of the V-Rep dodumentation). Each non-static and respondable shape needs to be connected to it's parent by a joint or a force sensor. The visual part of the link is always a child object of it's corresponding respondable. That way, the kinematic chain of the gripper was formed. The resulting model tree is shown in figure 2.5.

2.4.2 Assembling the scene

After finishing the gripper modelling process, each necessary component was available to build up the simulation scene as can be seen in figure 2.6. The final scene consists of a model of

[12] http://www.coppeliarobotics.com/helpFiles/en/designingDynamicSimulations.htm

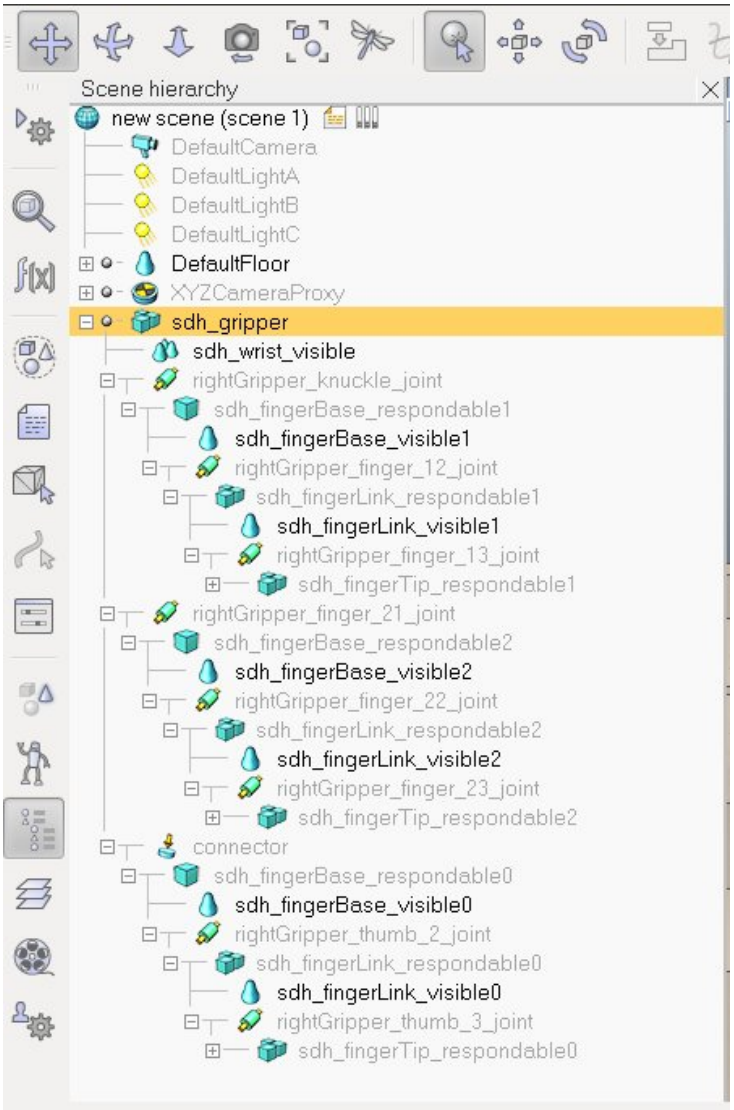

Figure 2.5: Gripper model tree

the robot's torso, two KUKA LWR4+ arm models with attached grippers and a model of the table in front of the robot. The origin of the world reference frame is located on the upper left side of the table, indicated by a slightly green shimmering sphere. A dummy object called `ref_frame_origin` was placed at that location. Each position for the simulation is defined relative to this dummy element. If it is necessary to change the origin of the world reference frame, this can simply be achieved by just adjusting the position and orientation that dummy. The positions and orientations of the torso, the table and the two arms were set relative to the world reference frame. The grippers were placed on the tip of each arm. Within the scene hierarchy they are child elements of the last node in the corresponding arm tree. The correct rotation and offset was measured on the real counterpart and then adjusted accordingly.

The plate and the legs of the table were modelled as group of primitive cuboids. The table shape is respondable to ensure that it will produce a collision reaction if a robot component collides with it. But as it is a static object it's position will not be influenced by such a collision because it is fixed within the scene. The material setting is set to the default *highFrictionMa-*

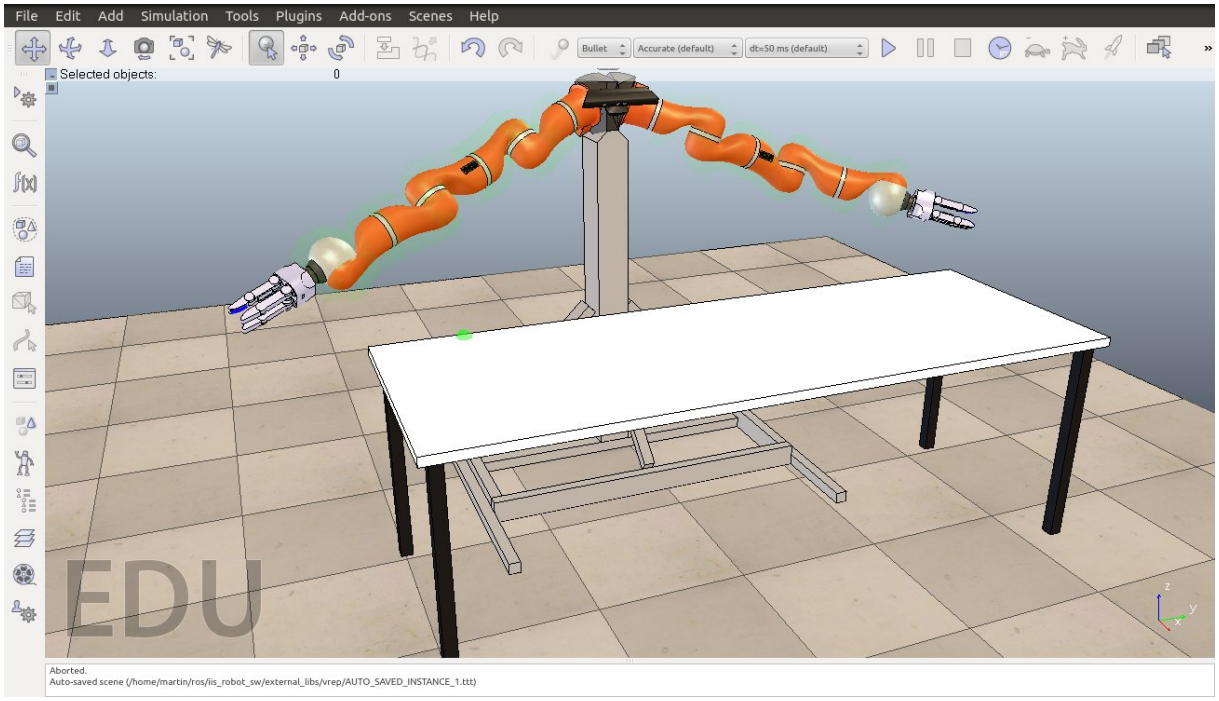

Figure 2.6: Structure of the V-Rep simulation scene

terial.

The utilized Kinect camera model was contained in the V-Rep distribution and just had to be inserted into the scene. Position and orientation of the Kinect camera with respect to the robot base was measured in the real world and adjusted accordingly in the simulator. Figure 2.7 shows a part of the resulting scene tree for the final simulation scene.

2.4.3 Configuring the collision detection module

The final solution needs to be able to detect and visualize collisions of the simulated robot with itself or its environment. This section describes the utilized simulator functionality along with the necessary steps to achieve the desired behaviour.

V-Rep's *collision detection module*[13] is capable of detecting and visualizing collisions between (collections of) shapes. The shapes used for this collision detection can be any shapes within the scene as long as they are defined to be *collidable*. It is important to note that the collision detection module only *detects* collisions by checking for interferences between meshes. Handling *collision reactions* is the responsibility of the physics engine. Configuration of the collision detection module is done by registering one or more *collision objects*. Each collision object consists of a *collider* and a *collidee*. The collider can be for example the robot arm and the collidee a collection of objects, the arm is not allowed to collide with. Both of them can be single shapes or collections of shapes. The big advantage in using collections is that they allow to exactly describe which shapes should be checked against which other ones. It is also possible to configure a collection to overwrite the collidable attributes of the contained shapes. That means a whole collection can be marked to be collidable even though the contained shapes are not. The collidee settings also offer the option *all other collidable objects in the scene*. In

[13] http://www.coppeliarobotics.com/helpFiles/en/collisionDetection.htm

that case, the collider is checked against all collidable shapes in the scene that are not part of the collider itself. Detected collisions are visualized by applying different colouring either to the collider or to the collidee. Each collision object needs to be *handled*, which means the collision detection module has to be triggered to check for collisions and visualize them as defined for the specific collision object. This can happen automatically in each simulation step or it can be triggered manually by using the corresponding API function. The desired behaviour is defined by a configuration option called *Explicit handling* in the collision object properties.

One of the aims of the solution is to allow to check actions in the simulator before executing them on the real robot. Therefore the simulator needs to be able to detect and visualize collisions that possibly happen during those actions. Moreover it would be a convenient feature to have a proximity warning if parts of the robot come dangerously close to an obstacle during movement. To achieve that goal, we decided to create an enlarged version of the robot arm geometry which we called the *collision shield* and do additional collision checking. These considerations lead to the two different types of possible collisions which we called *soft collisions* and *hard collisions*. Soft collisions are collisions, detected on the collision shield of the model. They just indicate a warning that the robot comes very close to an object it is not allowed to touch. *Hard collisions* describe a direct hit of a robot part on another object. This would be also a collision in the real world. The final solution is able to distinguish between those two types as can be seen in figure 2.8. The left image shows a soft collision, the right image indicates a hard collision.
.

The first step during realization was to model the collision shield. Therefore it was necessary to create an additional shape for each single link of the robot arms that is slightly larger than the original one. The images in figure 2.9 show snapshots of this modelling process. For all links within the arm models the following steps had to be performed:

- Create a copy of the shape that forms the visible part of the robot link
- Morph the copied object into a group of convex shapes to reduce complexity. This can be done by using the corresponding shape editor functionality (figure 2.9a).
- Ungroup the resulting group of shapes and merge them into one single shape (figure 2.9b).
- Grow the bounding box of the resulting shape, but only in x and y direction. The resulting mesh should be 10cm larger but keep the same height (figure 2.9c).
- Adjust the outside color to make it green and nearly transparent. The collision shield should be visible but not occlude the original model (figure 2.9d).
- Apply a meaningful name to the newly created shape to be be able to easily identify it within the model hierarchy. Here the name of the original shape with the `_col` suffix was used.
- Place the new shape as child element of the corresponding respondable shape within the model tree to ensure that the shield element correctly moves when actuating the arm model.
- Adjust the shape object properties. Define the new shape to be *static* and *non-respondable* because it is not intended to produce any collision reactions.
- Disable the *collidable* flag on the shape. This behaviour will be overridden in the collection settings later on when configuring the collision detection module.

Collision object	Collider	Collidee
left_arm	leftArm	all other entities
right_arm	rightArm	all other entities
left_armShield	leftArmShield	exLeftArmShield
right_armShield	rightArmShield	exRightArmShield

Table 2.2: Configured collision objects

Collection	Definition
leftArm	$\{ s \mid s \in$ *subtree of left arm* $\}$
leftArmShield	$\{ t \mid t \in$ *element of left collision shield* $\}$
exLeftArmShield	$\{ u \mid u \in$ *scene and* $u \notin$ *subtree of left arm* $\}$
rightArm	$\{ v \mid v \in$ *subtree of left arm* $\}$
rightArmShield	$\{ w \mid w \in$ *element of left collision shield* $\}$
exRightArmShield	$\{ x \mid x \in$ *scene and* $x \notin$ *subtree of right arm* $\}$

Table 2.3: Collection definitions

The next step was to define the required collision objects in the configuration of the collision detection module. The collision objects were created as listed in table 2.2. The collection definitions that were used for the collider and collidee settings can be seen in table 2.3. Each arm requires two collision objects - one for hard collisions and another one for soft collisions. The responsibility of the collision objects named `left_arm` and `right_arm` is to check for hard collisions. The colliders (collections *leftArm* and *rightArm*) are checked for collisions against *all other collidable objects in the scene*. This setting is only possible because the collidable flag is disabled within the scene object settings of the collision shield elements as mentioned above. That means that they are excluded from collision checking by default. The collision objects, named `left_armShield` and `right_armShield` are responsible to check for soft collisions. The colliders (collections `leftArmShield` and `rightArmShield`) contain only the collision shield elements of the corresponding arm models. As those elements were defined without the *collidable* flag, the option *Collection overrides collidable properties* was selected in the collection settings to explicitly enforce collision checking when using those collections. The collections, defining the collidees (`exLeftArmShield` and `exRightArmShield`) include all other objects in the scene *except* those, contained in the left/right arm's subtree. This exclusion is necessary because otherwise those collision objects would also detect collisions between the shield elements and the other arm links.

All collision objects are defined not to be handled explicitly. It is necessary to first check for direct hits before looking for shield hits because a direct hit always implies a shield hit. Therefore the collision objects have to be handled sequentially and for soft collisions is only checked if no hard collision was detected before. Handling those collision objects happens from code and will be explained later on in the section about control interface implementation.

2.4.4 Configuring the IK calculation module

The ROS control interface requires the robot arm to be controllable in joint space and in Cartesian space, depending on the selected control mode. Joint space targets are relatively easy to handle as that only means to set the target position of each single arm joint. Setting targets in Cartesian space requires to solve the *inverse kinematics* (IK) problem which is defined by (Craig, 2005) as the problem of finding a possible joint configuration for the robot to achieve a desired

end effector position and orientation. This section describes the IK functionality provided by the V-Rep simulator along with necessary configuration steps.

The *inverse kinematics calculation module*[14] allows to define and register *IK groups*. An IK group is capable of solving IK problems for a simulated manipulator like our robot arm. It consists of one or more so called *IK elements* and configuration settings for the utilized IK calculation method. IK elements are used to define the kinematic chain, constraints and desired precision settings. The kinematic chain is specified by configuring the dedicated *base* link and the *tip*. The tip is a dummy object, indicating the end effector reference frame. It needs to be linked to a *target* dummy, forming a *IK, tip-target* connection which is done by selecting the appropriate link type within the dummy object settings. The IK goal pose is set by placing that target dummy at the desired location. IK elements can be configured to enforce position constraints and/or orientation constraints for each single axis. The precision settings define the maximum allowed deviation (linear and angular) between desired and achieved end effector position to consider a solution to be correct.

The IK calculation module allows to define various IK groups for each robot component with differing configurations. Along with the IK elements, the IK group settings include the desired calculation method and the maximum amount of calculation iterations to use. Available IK calculation methods are *pseudo inverse* (PI) and *damped least squares* (DLS). As stated by Buss (2004), the PI method is usually faster than DLS. The tradeoff is that PI tends to be unstable in configurations where the target position is unreachable. That fact leads to a jittery behaviour of the manipulator. The DLS method provides higher stability in such situations but requires more calculation time and is therefore slower.

A target pose in Cartesian space is set by adjusting the position and orientation of the IK target dummy. Then it is necessary to *handle* one more of the responsible IK groups which follows the same approach that is used for the collision objects described above. The IK calculation module then tries to adjust the joint configuration of the corresponding manipulator until the tip pose matches the target pose, respecting joint limits and configured constraints. Figure2.10 shows a schematic description of that concept. The joints within the kinematic chain need to be operated in *inverse kinematics mode*, otherwise they can not be controlled by the IK calculation module. This can be specified within the joint settings.

IK group	**Method**	**Iterations**	**Prec. lin/ang**
left_arm	PI	9	0.001 / 0.1
left_arm1	PI	3	0.002 / 0.2
left_arm2	DLS	3	0.002 / 0.1
right_arm	PI	9	0.001 / 0.1
right_arm1	PI	3	0.002 / 0.2
right_arm2	DLS	3	0.002 / 0.1

Table 2.4: IK group definitions

Three IK groups have been created for each arm contained in our simulation scene. The configuration settings are listed in Figure2.4. They were chosen, following the guidelines in the V-Rep documentation. The precision settings specify the tolerance values that have to be

[14] http://www.coppeliarobotics.com/helpFiles/en/inverseKinematicsModule.htm

enforced by the corresponding IK group. The first two groups use the faster PI calculation method. The first one allows a higher amount of maximum iterations while demanding stricter precision settings than the second one. The third one is designed to increase stability especially for positions close to singularities. Therefore the DLS method was chosen with an increased position tolerance value. That configuration is slower because of the DSL calculation method and therefore it is only used if the other groups failed to find a solution. The IK groups are handled sequentially until one of them is able to solve the problem. This process is explained in the ROS control interface section later on.

2.5 Implementing the ROS control interface

In the real world, each type of robot component has its own, clearly defined ROS control interface. Those interfaces are composed from sets of inbound and outbound ROS topics that allow sending commands and to retrieve state data. The simulated components have to provide exactly the same ROS interface as their real counterparts, using similar topic names and message types. The structure of this control flow can be seen in Figure2.11.

The existing ROS interface that is part of the V-Rep distribution is not suitable as it only provides a very general approach for controlling joint values and reading state data. Therefore V-Rep provides various extension points, allowing to add custom functionality. The final solution was implemented as a *simulator plugin*, written in C++ and using V-Rep's *regular API*[15]. This approach states the most flexible solution as this API provides more than 400 functions that can be used to extend the simulator functionality. A plugin is a compiled library file, written in C++ that has to follow some V-Rep specific naming conventions and must reside in the V-Rep working directory. The library file gets automatically loaded on V-Rep startup and runs in the main simulation thread. The source code is located in the *iis_simulation* package that is part of the *iis_robot_sw* repository. The following section describes the design of the plugin architecture and the most important components within the package.

2.5.1 Plugin architecture

A plugin is a compiled library file, written in C++. This library needs to be placed in the V-Rep working directory and follow the V-Rep specific naming conventions. On startup, V-Rep looks for library files, prefixed with `libv\_repExt`. Matching files are automatically loaded. A plugin runs in the main simulation thread - that means it has to be programmed really carefully to avoid performance leaks during simulation. The plugin has to provide a clearly defined interface, consisting of 3 function definitions:

- `unsigned char v_repStart(void* reserved,int reservedInt)`
  This function is called on V-Rep startup and is used to perform necessary initialization steps. A return value of zero indicates that the initialization process failed and the plugin gets unloaded immediately. The current solution depends on a running roscore during startup. If that is not the case it is not able to work.

- `void v_repEnd()`
  Called before shutdown and is used to do some general cleanup and free allocated memory.

- `void* v_repMessage(int msg, int* auxData, void* custData, int* resData)`
  This function is called very often during the whole V-Rep lifecycle and is therefore a very performance critical method. Via this function V-Rep notifies plugins about events like

[15] http://www.coppeliarobotics.com/helpFiles/en/apiOverview.htm

start/end of simulation, simulation step, scene content change, scene switch and more. The plugin code can react to those events accordingly. In the current implementation, those messages are just passed to the software components that are responsible to handle the specific action.

The plugin code is organized as can be seen in the UML diagram in Figure2.12. The major parts of the system are described in the subsequent paragraphs.

SimulationComponent A simulation component is a single, reusable model of a specific robot component that can be utilized in different environments. Each component provides it's own clearly defined control interface. A simulation scene can contain multiple components from various types. The *SimulationComponent* class is the abstract base class for all simulation components. Currently there are existing two concrete implementations – the *LWRArmComponent* and the *SchunkHandComponent*. If the scenario should be extended and new components have to be introduced, it is necessary to create a new subclass of *SimulationComponent* and provide implementations for the abstract methods.

ComponentContainer This class represents the set of all identified simulation components within the current scene. On V-Rep startup an instance of *ComponentContainer* is created. Each time, the content of the current simulation scene changes, the method *actualizeForSceneContent* is triggered. This method then performs the following steps:

- It validates all currently registered *SimulationComponent* instances if they are still valid and present in the scene.
- It traverses the whole scene hierarchy to identify newly created components
- If a new component is identified, a corresponding concrete instance of *SimulationComponent* is created and added to the container.

The process of traversing the scene hierarchy and identifying components is explained in section2.5.2. During a running simulation, the *ComponentContainer* gets notified about each single simulation step. It simply forwards that message to all registered components. Those can then perform all necessary steps like triggering collision checking or handling IK groups.

ROSServer The *ROSServer* is a static class that encapsulates all ROS related functionality. It tries to initialize ROS on plugin startup, forcing a shutdown, if the connection to the master cannot be established. Otherwise it creates and maintains a ROS *NodeHandle* for the 'simulation' namespace. Each *SimulationComponent* can register *ComponentController* instances at the *ROSServer*. On simulation start it initializes all registered controllers with the maintained *NodeHandle*. The *ROSServer* gets also notified about each simulation step and forces the controllers to handle the received commands and publish all the necessary data. On simulation end it triggers the shutdown of all registered controllers.

ComponentController This is the abstract base class for all controllers. A controller actually represents the ROS interface of a specific simulation component, maintaining all the inbound and outbound topics used to control the simulated hardware. It is responsible for delegating commanded values to the underlying component as well as reading and publishing state data. Concrete implementations are the *LWRArmController* and the *SchunkHandController*. A *ComponentController* needs to be registered at the *ROSServer* and gets initialized on simulation

start. Concrete implementations can use the provided NodeHandle to create all the necessary publishers and subscribers. The *update* method is called by the *ROSServer* on each simulation step and forces the controller to publish all the required data. The *shutdown* method is called by the *ROSServer* on simulation end, forcing the controller to shutdown all publishers and subscribers.

2.5.2 Identifying simulation components

The plugin functionality should not be tied to a specific simulation scene but to specific models of robot components. Each time a known component is added to the scene it should be recognized by the plugin and the corresponding instance of *SimulationComponent* has to be instantiated and added to the *ComponentContainer*. As visualized in Figure2.13, each simulation scene is a tree structure, consisting of various different types of scene objects. It is necessary to identify subtrees within this hierarchy that belong to known simulation components and should therefore be handeled by the plugin. If a component is identified, the plugin has to discover each single part of the model. Depending on the model this can be joints, force sensors, reference frame dummies, IK groups or collision objects.

One possible way would be to give each part a clearly defined, unique name and then search for those names within the scene. But this approach would require to hardcode each single object identifier and this is not a preferable solution for this problem. If a user accidentally changes a name inside the model tree then the solution is broken because the plugin looses connection to the underlying object and cannot control it any more. Here V-Rep's custom developer data functionality comes into play. It is possible to put auxiliary data segments to each single object in the scene. Those data segments are serialized together with the object and can be read programatically. Each data segment starts with a header number which is used to uniquely identify the data from a specific developer. A visualization of this concept can be seen in Figure2.14.

As the format of the data can freely be chosen, it was decided to use string representations of key/value pairs, separated by a colon (:). The key is an integer number, used to determine the type of the tagged object. The value segment can be used to provide additional information, e.g. the name of a joint. The left arm's model base for example is tagged with the data segment

```
2497,1:left_arm
```

The number 2497 is the header that identifies the data segment to belong to this plugin. The data segment identifies that element as the model base of a *LWRArmComponent* ($Key = 1$) with the name `left_arm`. The available keys are explained in the sections that correspond to the specific simulation components. When actualizing for scene content change, the *ComponentContainer* traverses the scene hierarchy and looks for objects, that are tagged as known components. On success, it creates the specific instance providing the object ID of the underlying scene object to the constructor. During initialization, the concrete *SimulationComponent* implementation then traverses the rest of the model subtree to extract all the remaining parts that belong to that specific model (joints, dummies, force sensors...), also by looking for tagged objects. The implementations for arm and hand model provide feedback output on the console window about the status of this initialization process and provide meaningful error messages in case that not all necessary parts of a model could be successfully located.

2.5.3 The LWRArmComponent

The *LWRArmComponent* is a concrete subclass of *SimulationComponent* and states the abstraction of a simulated KUKA LWR4+ arm model. It is composed of two parts - an instance of the *LWRArm* class and a corresponding *LWRArmController*, as can be seen in the UML diagram in Figure2.12. Their functionalities are explained in the following paragraphs.

LWRArm This class states the connection to the (simulated) hardware itself, providing methods to set commanded values and access all available state data like joint positions, Cartesian position of the end effector, current collision state and more. On creation, it is passed the base of the model tree as constructor argument. The base is identified by the `LWR_ARM_COMPONENT` tag. The value segment of this tag states the name of the arm which has to be unique within the scene. During initialization it traverses the model tree and extracts all required parts by searching for tagged scene objects as described in the previous section. The parts to identify are the 7 arm joints, the force sensor on the last link of the arm, IK tip and target dummies. The corresponding tags are listed in Table2.5. Additionally the *LWRArm* also requires a con-

Constant	Key	Value	Description
LWR_ARM_COMPONENT	1	Arm name	Identifies KUKA LWR arm model
LWR_ARM_JOINT	12	Joint name	Joint in KUKA LWR arm
LWR_ARM_CONNECTOR	13	-	Force sensor on arm tip
LWR_ARM_TIP	14	-	IK tip dummy
LWR_ARM_TARGET	15	-	IK target dummy

Table 2.5: Tag data items for LWRArmComponent

nection to the configured IK groups and collision objects. As there is no possibility to place custom developer tags on IK groups and collision objects, the extraction is done by using a special naming strategy. The first IK group needs to have the same name as the arm itself and it's existence is mandatory, leading to a configuration error message if no such group can be extracted during initialization. Subsequent groups are optional and must have the same name with consecutive numbering (`[ARM_NAME]1`, `[ARM_NAME]2`...). That allows to reconfigure the IK calculation module and introduce additional IK groups without touching the plugin code. The two required collision objects are also searched, based on the name of the arm (`[ARM_NAME]` and `[ARM_NAME]Shield`). If one or both of them cannot be detected, an error message is stated on the console and the collision detection functionality will not work as expected.

To fullfill the requirements for the control interface, the arm has to be able to operate in *joint control mode* (FK mode) and in *inverse kinematics mode* (IK mode). Initially the arm starts in FK mode, which means the joints are operated in *torque/force* mode and accept target positions to be set. Switching to IK mode is done by changing the joint control mode to *inverse kinematics* mode, which means that they are conntrolled by the IK calculation module further on. Setting a target pose in Cartesian space is done by moving the IK target dummy to the required location and orientation. The known IK groups are then handled sequentially, until one of them is able to solve the problem and set the proper joint target positions. When switching from FK to IK mode, the IK target dummy has to be aligned with the tip dummy to prevent the arm from doing uncontrolled motions.

The current *collision status* is determined by using the configured collision objects. On each simulation step the collision object that is responsible for detecting direct collisions is handled

first. If that one detects a collision, a direct hit is reported and it is not necessary to handle the second object at all, because a direct hit always implies a hit with the shield as well. Only if no direct hit was detected, the second collision object is handled. The outcome can be queried as the current collision state of the arm. The collision state is evaluated on each simulation step.

LWRArmController The *LWRArmController* is a subclass of *ComponentController* that encapsulates the whole ROS interface for the *LWRArmComponnt*. On creation it is passed a reference to the underlying *LWRArm*. The arm controller offers 4 basic control modes:

- **Joint control mode**
  The controller accepts target positions in joint space via the `joint_control/move` topic. The message basically consists of a vector, containing a target angles for each single joint, measured in *radiant*. Incoming target positions are validated not to exceed a specified velocity limit, enforcing the constraint

  $$|c_i - t_i| < \delta, \forall\ i\ \in\ [0, 6] \tag{2.1}$$

  where c_i are current and t_i are target positions for the joints 0 to 6 and δ is the current velocity limit. If the velocity limit is violated, the message is dropped and a corresponding error message is published to the `sensoring/error` topic and written to the console output. The default limit is set to a value of 0.1 and can be adjusted via the `joint_control/set_velocity_limit` topic. Valid target positions are simply passed to the underlying *LWRArm* instance, which is operated in FK mode.

- **Cartesian control mode**
  The controller accepts target positions in Cartesian space via the `cartesian_control/move` topic. Therefore, the *LWRArm* needs to be operated in IK mode. The commanded target pose is also validated before accepting it. The constraint is defined as

  $$|c_x - t_x| < \sigma\ and\ |c_y - t_y| < \sigma\ and\ |c_z - t_z| < \sigma \tag{2.2}$$

  where c is the current and t is the target position in Cartesian space and σ is the current Cartesian velocity limit. The velocity limit can be adjusted, using the `cartesian_control/set_velocity_limit` topic. Invalid messages are dropped and result in an error message as well, published in `sensoring/error` topic. Valid target poses are passed to the underlying *LWRArm*.

- **Follow mode**
  This mode is only on the simulator available and designed to mirror the behaviour of the real robot. The controller reads the joint states of it's real counterpart from the corresponding topic and uses it's current position as target. Therefore, the *LWRArm* is operated in FK mode. This results in an exact copy of the real robot's motions. This mode can be used to test the accuracy of the simulated model, for example by carefully moving the robot close to positions where it collides with the table and check when the collision is reported.

- **Stop mode**
  The arm does not accept any movement commands at all. Ongoing motions will be stopped immediately when switching the arm into this control mode.

The controller has to be registered at the *ROSServer* in order to be able to work. This section only covered the most important control modes and topics. The complete interface description can be found in the documentation, located in Appendix A.

2.5.4 The SchunkHandComponent

The *SchunkHandComponent* is designed, using the same approach as for the *LWRArmComponent*. Therefore only the most important facts will be stated here. The two composing parts are the *SchunkHand* and the *SchunkHandController*.

Constant	Key	Value	Description
SCHUNK_HAND_COMPONENT	2	Hand name	Identifies Schunk hand model
SCHUNK_HAND_JOINT	22	Joint name	Joint in Schunk hand model

Table 2.6: Tag data items for SchunkHandComponent

SchunkHand States the connection to a Schunk hand model. The only additional parts that have to be discovered during initialization are the 7 gripper joints and their corresponding names. Available tags are listed in Table2.6. The *SchunkHand* class provides methods to set joint positions and to retrieve current joint states. Additionally it allows to modify the motor strength for each single joint, based on a percentage of maximum force. This reflects the possibility of adjusting the motor currents in the real hand, which also leads to the effect that the maximum motor strengths of the finger joints can be increased or decreased.

SchunkHandController The *SchunkHandController* provides an implementation of the ROS interface for the Schunk gripper model. The basic topics are quite similar to those of the *LWRArmController*, as there are `joint_control/move` for setting joint target positions or `joint_control/get_state` for retrieving joint states. Additionally the hand is able to perform grasps, based on specific *grasp types* as visualized in Figure2.15. This functionality can be accessed via the `joint_control/gripHand` topic. A grasp is defined by the parameters *grasp type* and *close ratio*. The controller then calculates the corresponding joint positions, based on those parameters and sets the desired target positions. Each grasp can be expressed, using three functions – one for the position of the pivoting joints and the other two for the positions of the distal and proximal finger joints. The utilized functions were taken from the original controller code of the `schunk_sdh` ROS package, located in the `iis_robot_sw` code repository. The definitions are listed in Table2.7. The grasp strength can be adjusted, using the `settings/set_motor_current` topic. The message basically consists of a vector, holding a value for each joint, defining a percentage of the maximum motor strength. The real gripper also provides a topic that allows to read the current motor temperatures. For the sake of consistency, the *SchunkHandController* also provides this topic, but as V-Rep is not able to simulate motor heatings, only constant fake values are published. A complete interface description can be found in Appendix A.

Name	pivoting	proximal	distal
CYLINDRICAL	0	$(-30+30x)\frac{\pi}{180}$	$(30+35x)\frac{\pi}{180}$
PARALLEL	0	$(-75+82x)\frac{\pi}{180}$	$(75-82x)\frac{\pi}{180}$
CENTRICAL	$\frac{\pi}{3}$	$(-75+82x)\frac{\pi}{180}$	$(75-82x)\frac{\pi}{180}$
SPHERICAL	$\frac{\pi}{3}$	$(-40+25x)\frac{\pi}{180}$	$(40+15x)\frac{\pi}{180}$

Table 2.7: Joint position functions based on close ratio x

2.5.5 Publishing Kinect camera data

The Kinect camera model does not contain any flexible parts that have to be controlled by the plugin. But it is necessary to make the images, captured by the simulated vision sensor available via ROS topics. This was achieved by using the corresponding functionality of the V-Rep default ROS interface that allows publishing vision sensor data, using the correct message types. Therefore a LUA script was attached to the base element of the camera model within the scene. This script simply extracts the object ID of the vision sensor and enables publishers for the captured RGB and depth images. The corresponding topics are `kinect1/sensoring/rgb_image` and `kinect1/sensoring/depth_image`.

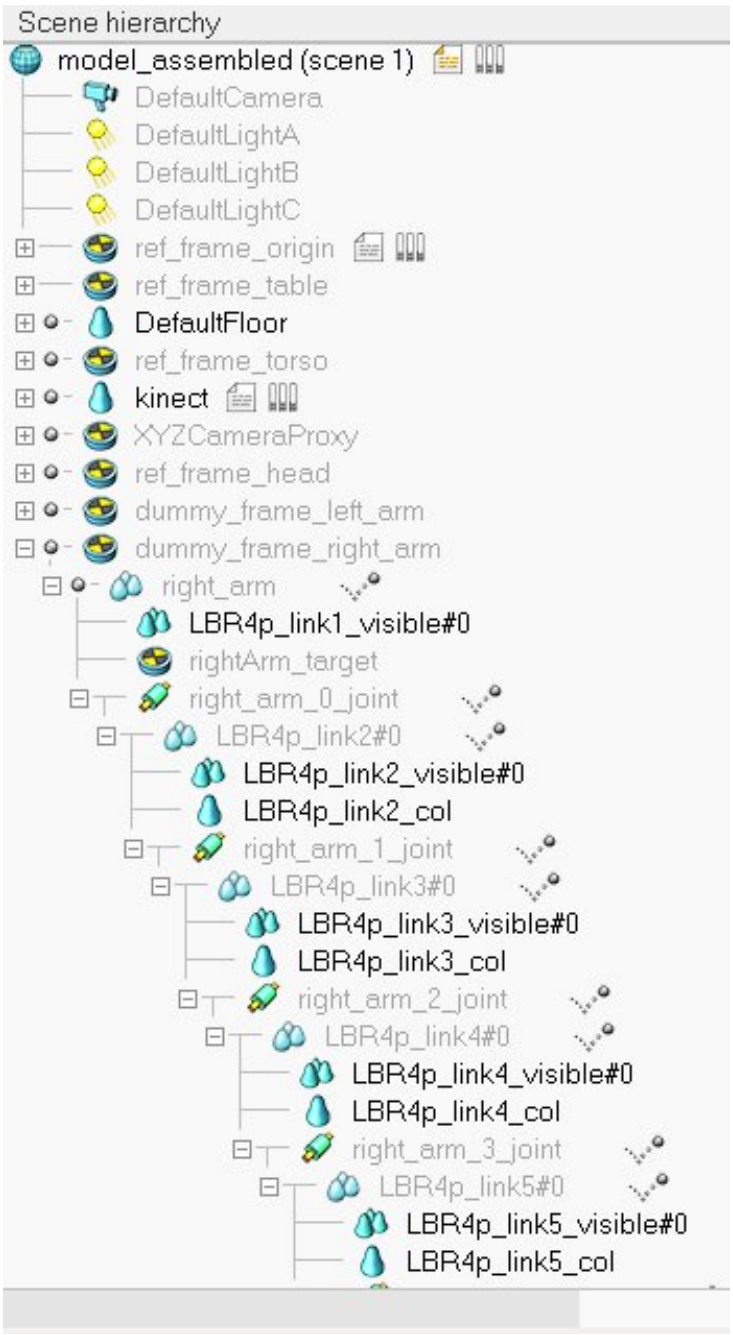

Figure 2.7: Simulation scene hierarchy

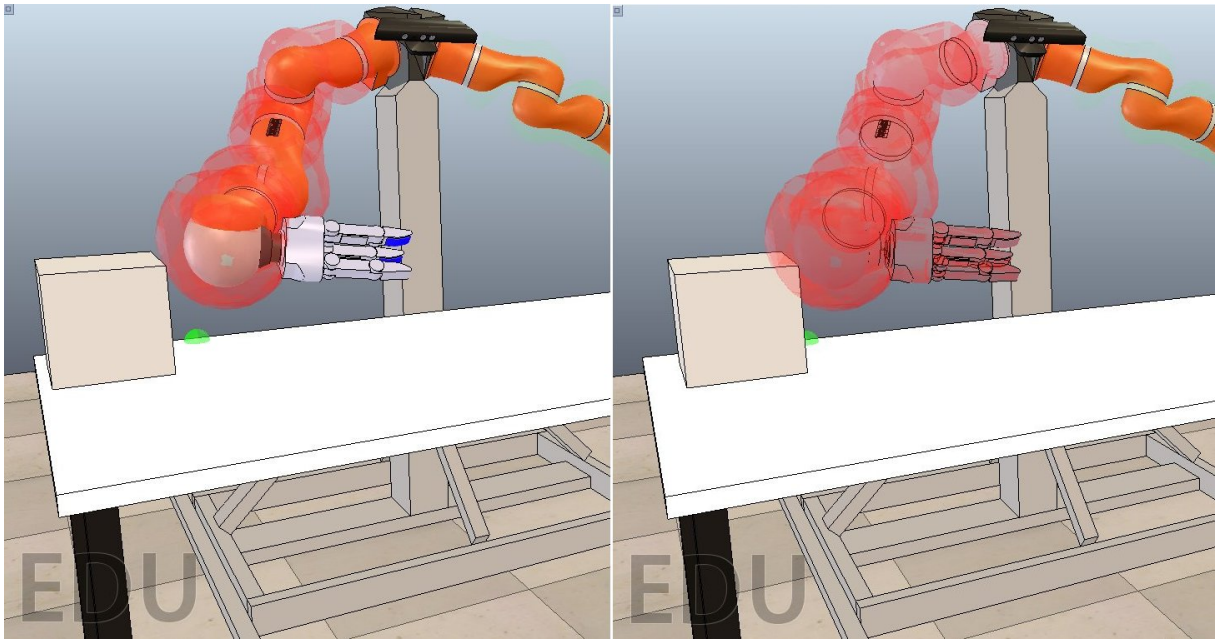

Figure 2.8: Collision detection and visualization

Figure 2.9: Modelling steps for the second arm link

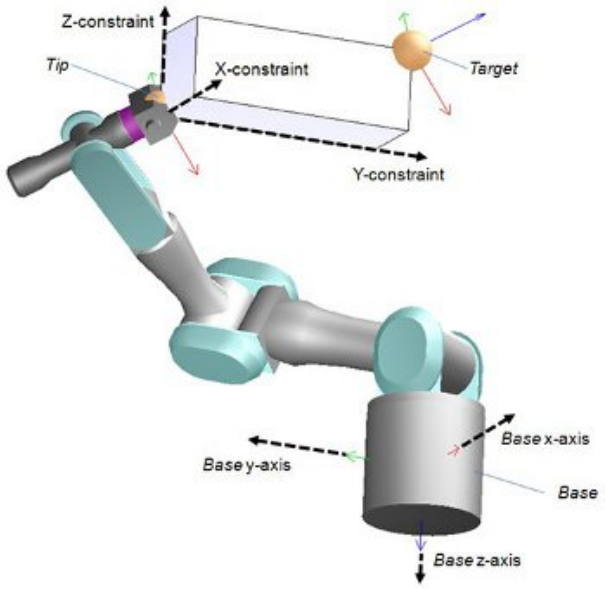

Figure 2.10: IK calculation module concept
Image source: http://www.coppeliarobotics.com/helpFiles/en/solvingIkAndFk.htm

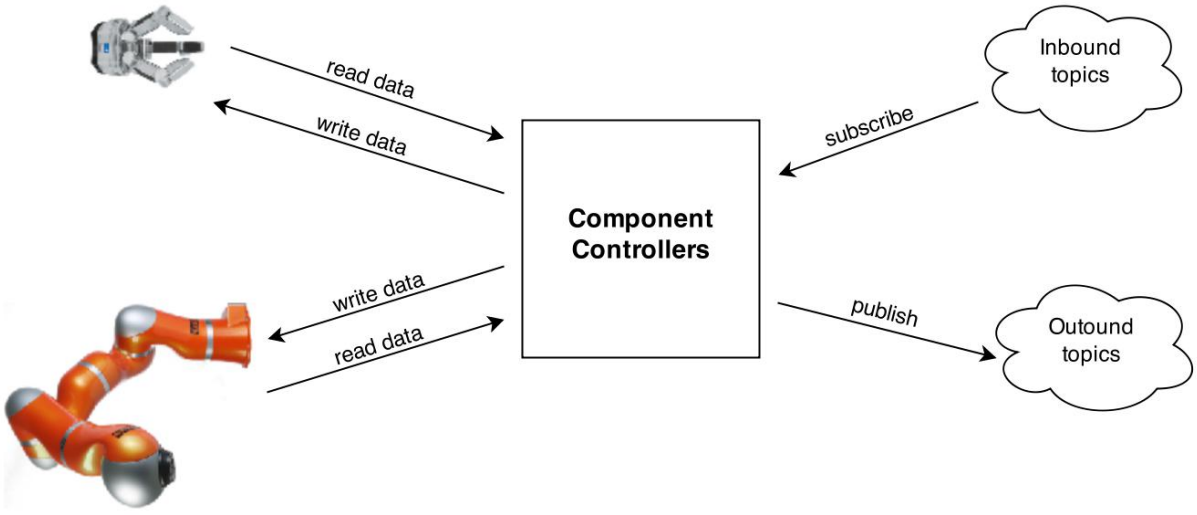

Figure 2.11: Control flow structure

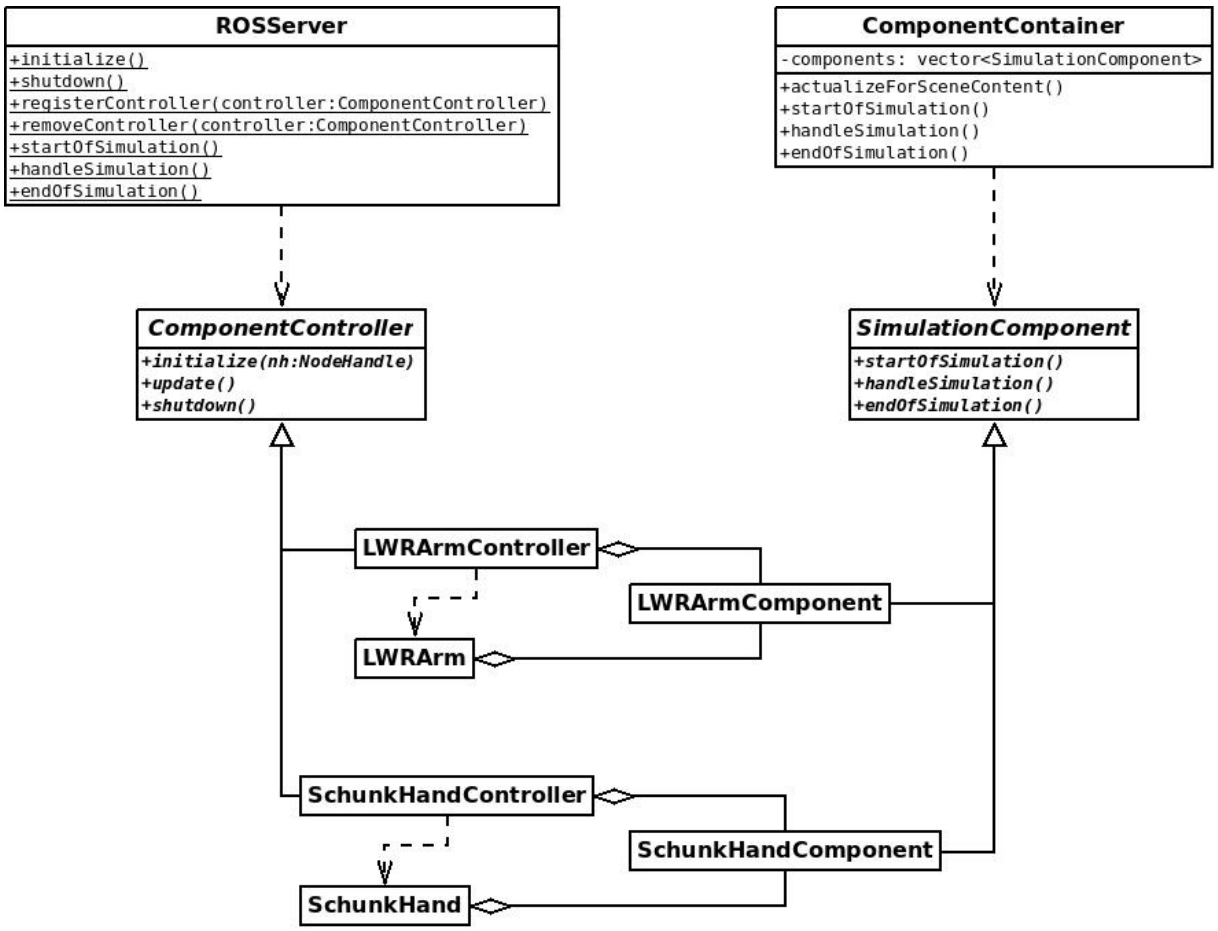

Figure 2.12: Simulator plugin architecture

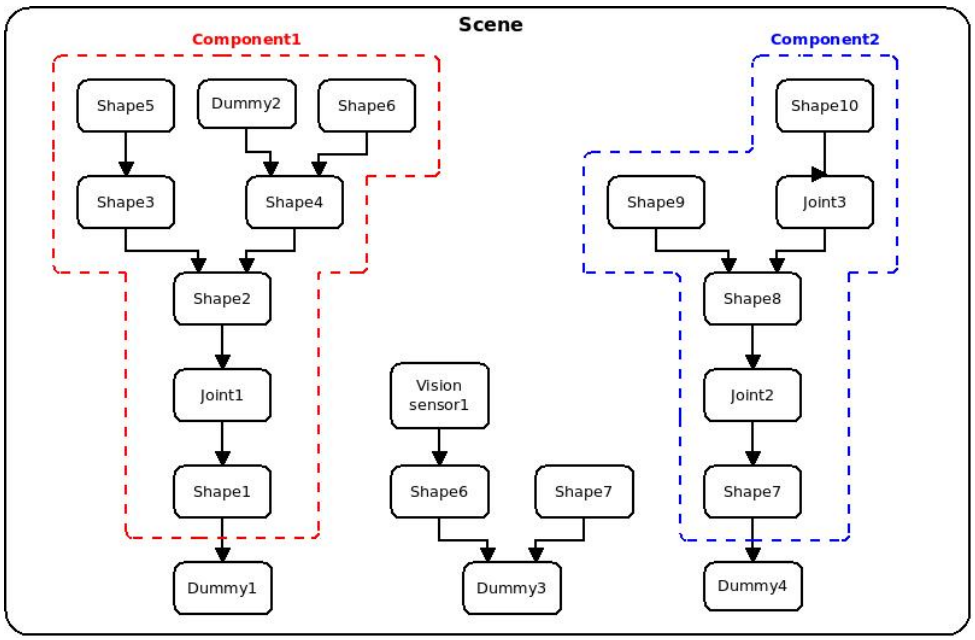

Figure 2.13: Sample scene hierarchy

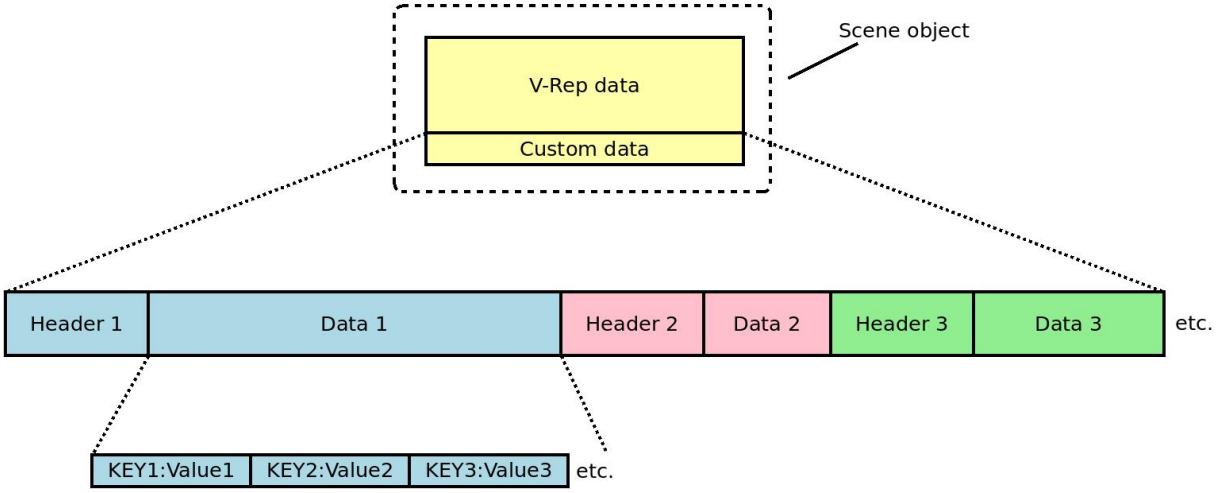

Figure 2.14: Custom developer data segments on scene object

Image source: http://www.coppeliarobotics.com/helpFiles/en/pluginTutorial.htm

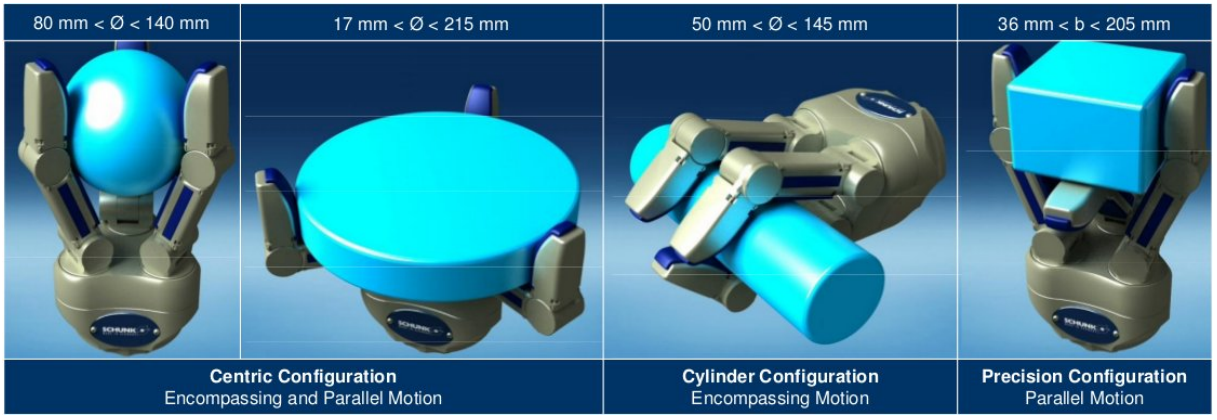

Figure 2.15: Grasp types `SPHERICAL`, `CENTRICAL`, `CYLINDRICAL` and `PARALLEL`

Image source: Schunk (2010)

Chapter 3

Motion planning

This chapter explains the motion planning related part of the thesis. The introduction gives an overview about motion planning problems in general. Further on, the sampling based motion planning approach is described. Subsequent sections describe the integration of the motion planning framework *MoveIt* into the existing robot setup.

3.1 Introduction

(Choset, 2005, p. 1–11) describes motion planning as to be the task of finding a collision free path from one robot *configuration* to another one. The classic path planning problem is the so called *piano mover's problem*, originally mentioned by Schwartz and Sharir (1983). It is assumed to have a piano, which states a three dimensional rigid body and a set of known obstacles. The problem is to find a continuous motion that moves the piano from it's current position to a given target position without touching any of the obstacles. Thereby the piano can freely be moved and rotated in Cartesian space.

A generalized version of the *piano mover's problem* is to find paths for a robot, composed from a set of rigid bodies, linked by joints while enforcing *constraints* during that motions. A *constraint* could be to avoid obstacles or to keep the robot's end effector in an upright position. Therefore it is important to have a representation of a robot's state that allows to determine the location of all robot parts. This representation is called the *configuration* of a robot and the *configuration space* is the set of all possible configurations, the robot is able to acquire. The dimension of the configuration space is the amount of *degrees of freedom* (DOF), which is the number or independent variables that are necessary to describe a configuration. An imaginary free flying piano has six degrees of freedom as it's configuration consists of the position and orientation $(x, y, z, roll, pitch, yaw)$ in Cartesian space. A robot arm with 7 joints has 7 degrees of freedom and it's configuration are the joint positions. The motion planning problem is to find a curve in the configuration space that connects start and goal configuration without violating constraints. This is a very complex problem and there exist various different approaches to find solutions. Examples are among others the *bug algorithms*(Choset, 2005, chapter 2), *potential functions*(Choset, 2005, chapter 4) or *sampling-based methods*(Choset, 2005, chapter 7). As the solution within this project only uses sampling based algorithms, a short overview about this class of methods will be given in the following section.

3.2 Sampling-based motion planning

Based on (omp, chapter 2), sampling-based motion planning can be seen as a powerful concept, capable of handling planning problems efficiently, especially for systems with many degrees of freedom. The general idea is to generate a uniform set of random sample points in the configuration space and then connect start and goal state by connecting the samples via collision free paths, with respect to possible motion constraints. Those methods are usually faster than traditional approaches because it is not necessary to reason about the whole configuration space but only about a finite number of sample configurations. The majority of sampling-based approaches are known to be *probabilistic complete*, which means that the probability of finding an existing solution tends to 1 as the number of sample points increases to infinity. But they are not able to decide if a valid solution exists at all. The following definitions are used throughout this section to describe the concepts of sampling-based motion planning.

- **State space**
  The *state space* $\mathcal{S}$ is equal to the configuration space and consists of all possible robot configurations (states).
- **Free state space**
  The *free state space* $\mathcal{S}_{free}$ is a subset of $\mathcal{S}$, containing only collision free states.
- **Path**
  A *path* is a sequence of states. If each state within the path is contained in $\mathcal{S}_{free}$, it is called a *collision free* path.

The sampling-based motion planners can be categorized into two major types - *probabilistic roadmaps* (PRM) and *tree-based planners*. Common to both methods is that they create uniformly distributed samples within the free state space. As the shape of $\mathcal{S}_{free}$ is not explicitly known, the created sample states are checked for collisions before using them. The following paragraphs give a short overview about both approaches.

Probabilistic roadmaps That approach uses the sampled states to create a "roadmap" of the free state space. Therefore each sample point is connected to an amount of k nearby sample points via collision free paths. This is done by a local planner that simply interpolates between two points in the desired resolution while watching out for collisions. If no collision is detected, a new edge is introduced into the graph that is formed by the roadmap. After completing the graph, a planning query can be reduced to finding the shortest path within that graph that connects the start state and the goal state (Figure 3.1a).

Tree-based planners A lot of different sampling-based planning algorithm are using the tree based approach, as there are for example (RRT, EST, SBL or KPIECE). The difference to PRM is that this method uses a tree data structure of the free state space, which means that the resulting graph contains no cycles. The root of the tree is the start state and the tree is then expanded towards the goal state by creating collision free connections between sample points. If the goal is reached, the solution is found (Figure 3.1b). The different approaches differ in the strategy that is used to expand the tree towards the goal state.

Generally can be said that the tree based approaches are more adequate for *single query planning* because the tree usually does not has to cover the whole free state space. A roadmap could be reused for subsequent queries. As the most methods also require the search of a

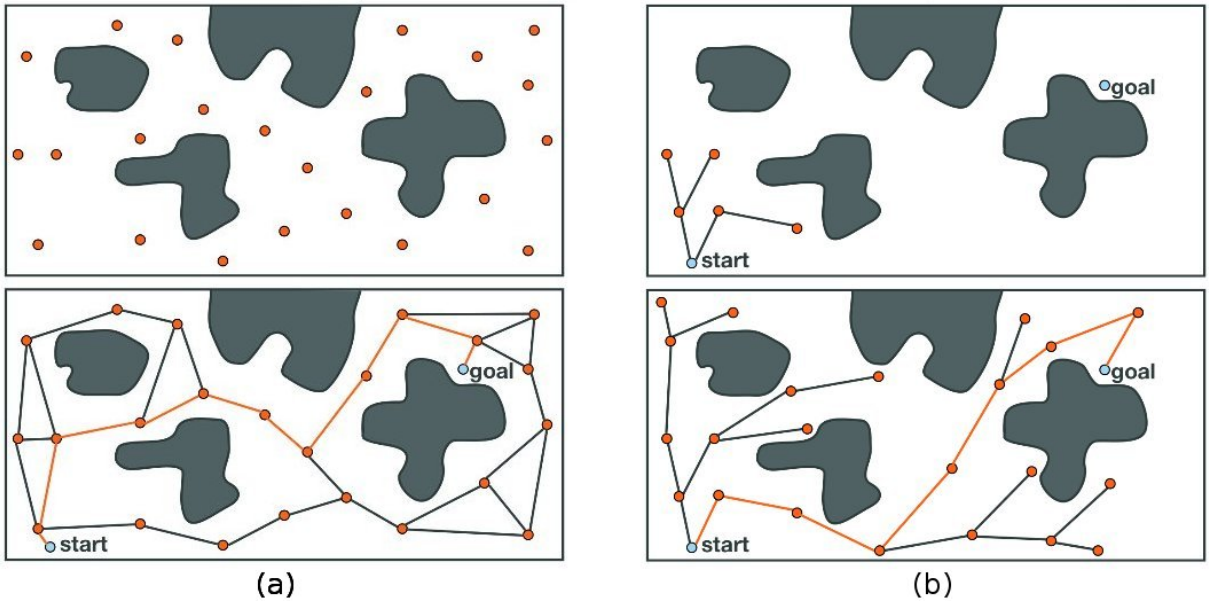

Figure 3.1: Probabilistic roadmap (left) and tree based approach (right)
Image source: (omp)

nearest neighbour, the utilized distance metric is also a crucial part within sampling-based motion planning as it is not always easy to identify the optimal method for finding nearby states in systems with large degrees of freedom.

3.3 The MoveIt! motion planning framework

MoveIt(Sucan and Sachin Chitta) is an open source framework for motion planning. It is the successor of the previous arm navigation stack and therefore fully integrated into ROS. MoveIt was originally developed by Willowgarage[1] but since April 2012 it is maintained by the Open Source Robotics Foundation (OSRF). Figure 3.2 gives an overview about the system architecture. Central part of the framework is the `move_group` node which provides a rich set of ROS topics and services that can be used to solve planning problems and execute calculated motion plans on connected hardware. Configuration of that node happens via the parameter server. It requires a complete kinematic and semantic description of the robot setup and a lot of other configuration parameters which will be described in subsequent sections. Various important parts of MoveIt are implemented as plugins. The default inverse kinematics plugin uses the Kinematics and Dynamics Library[2] (KDL) for solving IK problems. The utilized planning plugin uses the Open Motion Planning Library[3] (OMPL), which is a framework that contains implementations of many different sampling based motion planning algorithms. Therefore it is possible to choose, which one of those algorithms to use during planning requests.

MoveIt maintains a planning scene which is an internal representation of the world, including the robot and it's environment. The base is a description of the robot and it's various planning groups. The kinematic description needs to be available in form of a URDF model. Necessary steps to create that model are described in Section3.4. Based on that URDF model, a semantic

[1] http://www.willowgarage.com
[2] http://www.orocos.org/kdl
[3] http://ompl.kavrakilab.org

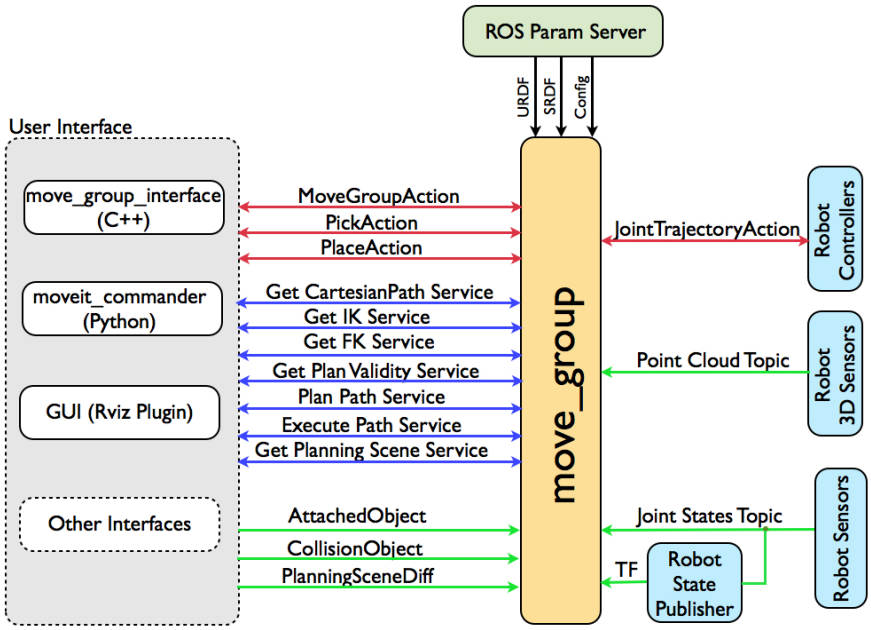

Figure 3.2: MoveIt architecture

Image source: `http://moveit.ros.org/documentation/concepts`

description (SRDF) is created that contains additional information about the setup, as explained in Section3.5. On top of those static descriptions, MoveIt allows to modify the planning scene on runtime via appropriate ROS topics.

To keep track of the current state of the robot MoveIt needs to be informed continuously about actual joint states. This is done via publishing the joint states to a specific topic. If there are additional objects within the robot's workspace they also have to be added to the planning scene. This can be done either by explicitly adding them via the corresponding topic or by integrating sensor information like Kinect camera data. MoveIt can then take those objects into account during motion planning and avoid collisions. But some collisions are intended. For example if an object has to be picked up, the gripper has to get in contact to this object. That means, collision checking for specific objects has to be (temporary) disabled to allow those controlled collisions. Therefore MoveIt maintains an *AllowedCollisionMatrix* to which objects can be added or removed. The connection to the hardware happens via the *FollowJointTrajectory* action interface. Each robot component has to provide this interface if it is intended to be control via MoveIt. This interface consists of a set of ROS topics that provide trajectory execution functionality and also allow monitoring the current execution status. The implementation and usage of a ROS node that provides the necessary interface is described in section 3.6.

3.4 Creating the URDF model of the robot setup

The *Unified Robot Description Format* (URDF) is a markup language, designed to describe robots. The description happens in text files, in a special XML format. The most important elements in the XML specification[4] are:

[4] http://wiki.ros.org/urdf/XML

- **<link>**
  Describes the all necessary properties of a specific robot link. Each link must have a unique name. The visual, inertial and collision details are configured in the corresponding subtags of the link element. The visual part as well as the collision model can either be composed from primitive shapes or from mesh files. If mesh files are used it is important that they are not to complex. Especially for the collision model it is recommended to use a simplified model to avoid a performance loss.

- **<joint>**
  Describes the properties of a joint. A joint is a connection between two links, having exactly one parent and one child link. Each joint states a new reference frame for it's child link and it is positioned relative to it's parent frame. A *fixed* joint is a rigid connection between parent and child link. There are different types of joints available but for the current project only *revolute* joints[5] are of interest. Details like limits, axis orientation and dynamic properties can be configured in the corresponding subtags of the joint element.

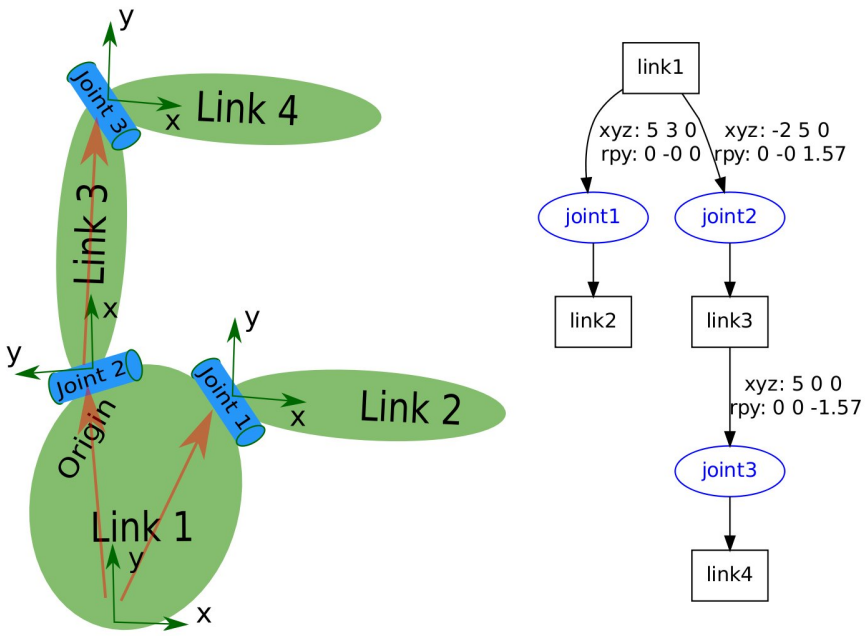

Figure 3.3: URDF graph
Image source: http://wiki.ros.org/urdf/Tutorials/Create your own urdf file

Those elements are used to form the URDF graph that exactly describes the kinematic chain of the robot components and their placement relative to each other as visualized in Figure 3.3. This description can get very large as a lot of different components are involved. So it is possible to organize it into a set of text files, each one describing one part of the whole. For example one file describes the arm itself. Other files can then use that description and insert multiple instances of that arm. The `xacro` ROS package provides the necessary functionality to combine all those text files into one XML string. *XACRO* stands for *XML Macro* and is designed to parse xacro files and combine them into one single XML document, containing the resulting URDF description.

[5] Rotational joint with one degree of freedom

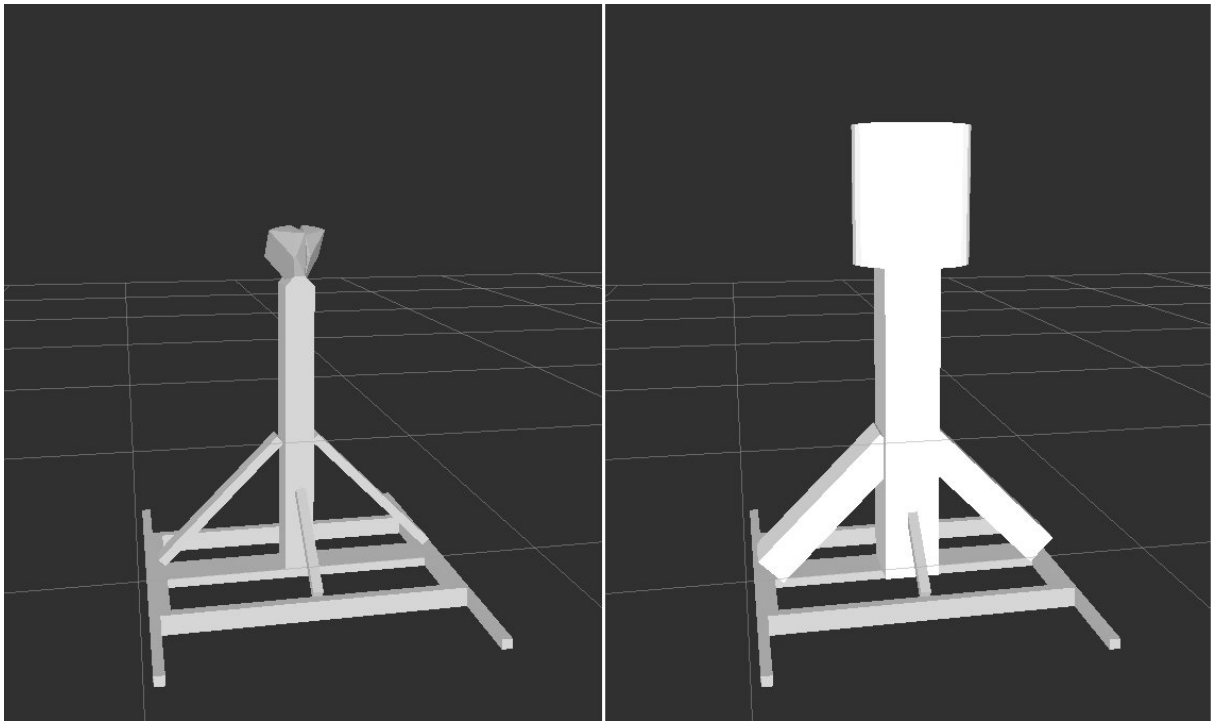

Figure 3.4: Left image shows visual part, right image the collidable part of the torso

The URDF description of the IIS robot setup is spread across multiple packages, located in the `iis_hardware` stack. Arm and gripper descriptions are located in seperate packages (`lwr_description` and `schunk_description`), the `iis_robot` package brings all the components together. The modelling process started with the search for pre-existing URDF descriptions of the required robot components. A suitable description of the Schunk SDH gripper was taken from the `schunk_description` ROS package. A model of the KUKA LWR arm was found in the Github repository[6] of the *Robot Control and Pattern Recognition Group*[7]. The other parts of the model, namely the robot torso and the table had to be created. The descriptions are located in a seperate files within the `uibk_robot` package (`torso.xacro` and `table.xacro`).

The mesh files, used in the description of the robot torso have been exported from the V-Rep simulation scene. For the visual part, the mesh was taken as it is. For the collision model, the width of the original mesh was increased to provide some safety padding. Moreover a cylinder with a diameter of 40cm was added in the head area to ensure that the planner avoids accidental hits in this sensitive region. Figure 3.4 shows the visual and the collidable part of the torso model. The table was modelled from primitive shapes. It was taken care that the size of the table can easily be adjusted, as can bee seen in Listing3.1. The collision model of the table is also slightly larger than the visual part to provide some safety margins.

[6] https://github.com/RCPRG-ros-pkg/lwr_robot/tree/hydro-devel/lwr_defs
[7] University of Warsaw (http://robotyka.ia.pw.edu.pl/twiki/bin/view/Main)

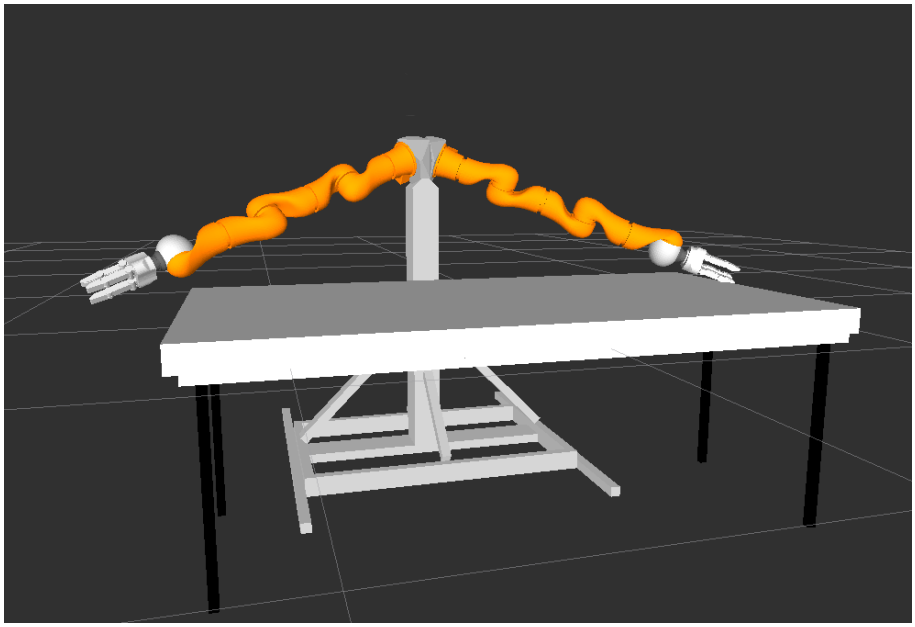

Figure 3.5: URDF description in RViz

Listing 3.1: XML snippet, inserting the table model into the URDF

```
<!-- draw the table relative to the origin -->
<xacro:model_table name="table"
        parent="world"
        length="2.22"
        width="0.8">
  <!-- Place the table relative to the world reference frame -->
  <origin xyz="-0.029 -0.3 0" />

</xacro:model_table>
```

When attaching the two grippers it showed that the offset between gripper wrist and last arm link was not correct. To correct that issue the file `sdh_with_connector.xacro` was created which simply places an additional ring between the last arm link and the gripper.

The file `iis_robot_table.xacro` draws all the pieces together. It describes the whole setup, consisting of the torso, two arms, two grippers and the table. The root element of the model hierarchy is a link called `world_link`. Torso, table and both arms are positioned relative to that root link. Changing the world reference frame could easily be achieved by shifting the root link to a new position. Figure 3.5 shows a visualization of the URDF description.

3.5 Configuring the planning tools

The MoveIt configuration package is created, using the *MoveIt Setup Assistant*. Precondition is an existing URDF description of the robot setup which was created in the previous step. The setup process comprised of the following steps:

- **Computating the self collision matrix**

  The self collision matrix consists of pairs of robot links that can safely be excluded from collision checking. Neighbouring links for example are in permanent collision. Collisions

between other links can never happen because they are simply too far apart. The Setup Assistant calculates a large number of different robot configurations and tracks for link pairs that are mostly always in collision and pairs that are never in collision. The self collision matrix can be adjusted manually if necessary. Excluding a large number of link pairs raises performance during motion planning because collision checking is an expensive process.

- **Defining the planning groups**

  Each planning request in MoveIt is done against one of the defined *planning groups*. A planning group is a group of links and joints within the model that can be seen as one logical component. Planning groups are defined for both arms and grippers. The configuration for the arms also requires the definition of the utilized IK solvers but currently only the default KDL solver is available. An additional planning group, called `both_arms` allows planning requests for both arms simultaneously.

- **Defining the end effectors**

  End effectors are the left and the right gripper. Each end effector has a name and consists of one of the predefined planning groups, a parent group and the parent link which is the last link in the kinematic chain of the parent group.

- **Generate the configuration files**

  After completing all the configuration steps the configuration package can be generated. Therefore a package name has to be specified which was set to `uibk_robot_moveit_config`. This package is a prototype, containing a large number of predefined configuration files. Some of those files have to be extended manually, adding additional information about available robot controllers and sensors.

Completing the last step results in a ROS package, containing necessary configuration and launch files for the given robot setup. The semantic robot description can be found in the `iis_robot.srdf` file. It contains previously defined parameters like the self-collision matrix, planning groups and end effectors. The configuration package can be tested, running the following command on the command line:

```
roslaunch uibk_robot_moveit_config demo.launch
```

This commands launches a `move_group` node in demo mode, using the previously created configuration files and starts an instance of RViz with the motion planning plugin. There it is possible to switch between the planning groups, set start and target configurations, do planning requests and visualize the outcome. The setup can be modified by launching the Setup Assistant again, using the `setup_assistant.launch` file from this package.

3.6 Connecting MoveIt to the existing robot control interface

Now MoveIt is configured and ready to handle planning requests for the robot setup. But execution of the resulting trajectories is still impossible because of the missing connection between MoveIt and the involved robot hardware. Each component that is intended to be controlled by MoveIt needs to provide the *FollowJointTrajectory* action interface, as defined in the `control_msgs` package. This is a special kind of ROS interface that allows to send trajectories to robot components and monitor the execution status. As the existing control interface of the IIS robot components does not fulfil these requirements, an additional node is necessary that is

capable of executing trajectories, using the existing infrastructure.

The planning outcome of MoveIt is a time parametrized trajectory, expressed by the *JointTrajectory* message type. This message consists of a set of waypoints. A waypoint is a joint configuration, described by the tuple (p, v, a, t) where $p \in \mathbb{R}^n$ are the positions, $v \in \mathbb{R}^n$ the velocities and $a \in \mathbb{R}^n$ the accelerations at time t and n is the number of involved joints. Those waypoints mark the important points along the path, the manipulator has to move. The controller needs to be able to translate this trajectory into a sequence of suitable motor commands. Just sending the joint positions within the waypoints to the robot would not suffice as a trajectory is also constrained in terms of *velocities* and *accelerations* over *time*. Therefore the controller needs to be able to interpolate between the subsequent waypoints and calculate intermediate joint positions based on the loop rate of the control cycle. This interpolation is a difficult task and the implementation would be beyond the scope of this project. But the required functionality is already available within the `ros_control` stack[8]. Those packages are created to integrate and control robot hardware in a generalized way, facilitating the usage of existing controllers on different robots.

3.6.1 ROS control stack overview

Figure 3.6 shows an architectural overview of the ROS control stack. The required infrastructure for using generic controllers is mainly provided by two components:

- **RobotHW**

  This is the abstract base class for robot hardware abstraction. It's main purpose is to send commands to the hardware and retrieve state data. The *RobotHW* also maintains a set of *hardware ressource interfaces*. Each generic controller needs a special kind of interface for controlling joints or read their current state. Concrete implementations have to register the required interfaces. For retrieving state data, a *JointStateInterface* is required. Controllers using that interface can query it for a specific joint and ask for the current state. Sending commands to the joints is done via subtypes of the of the abstract *JointCommandInterface*. The *PositionJointInterface* for example allows to send target positions to registered joints. Other implementations are *JointEffortInterface* and *JointVelocityInterface*. Which interfaces are provided depends on the way how the actual hardware is controlled.

- **ControllerManager**

  The *ControllerManager* is responsible to maintain a set of controllers. It provides a ROS interface that allows to load, start and unload controllers and for switching between them. During the control cycle, the *RobotHW* is forced to read current state from the hardware. Then the *ControllerManager* is triggered to update the active controllers based on the current time step. The commanded values are then sent back to the concrete hardware. This control cycle is intended to run in real time when interacting directly with the hardware. Controllers are usually configured on the parameter server and loaded on demand, using the ROS interface of the *ControllerManager*. Custom controllers can be created by inheriting from the abstract *ControllerBase* class.

[8] http://wiki.ros.org/ros_control

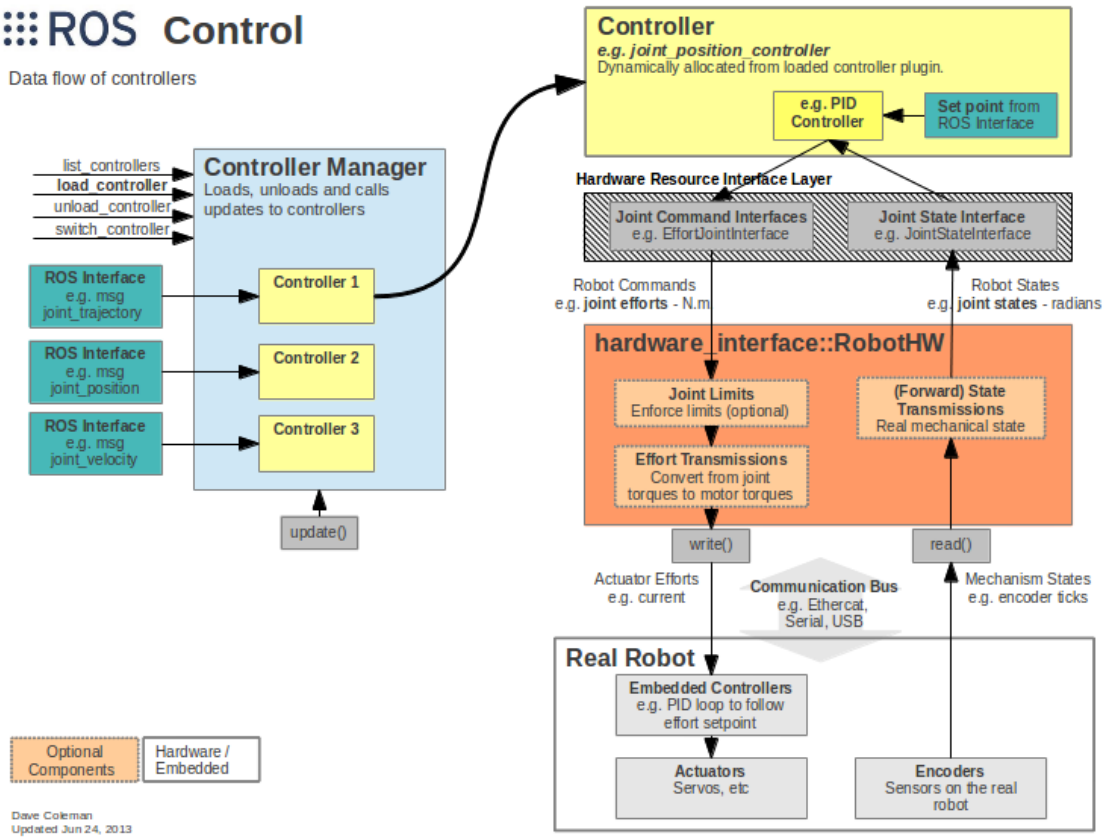

Figure 3.6: ROS control architecture

Image source: http://wiki.ros.org/ros_control

3.6.2 Designing the hardware adapter

The *hardware adapter* is an independent ROS node that acts as connection between MoveIt and the simulated or real hardware. It provides the necessary infrastructure for using generic controllers from the `ros_control` packages. Figure 3.7 gives an overview about the hardware adapter architecture. The *UibkRobotHW* is a subclass of *RobotHW* and represents the connection to the robot. It maintains the complete state of all joints in the connected components. Joints are represented by the *Joint* datatype, consisting of a unique joint name, state parameters and a commanded target position. For controllers, the *UibkRobotHW* class provides a *JointStatesInterface* and a *JointPositionInterface*. Active controllers use those interfaces to access the current state and for commanding target positions.

The *UibkRobotHW* utilizes a set of *JointStateAdapters*, each one representing a connection to one specific robot component. During the control cycle the *JointStateAdapters* read current joint states from the appropriate topics and send commanded values back to the hardware. The *JointStateAdapters* are created during initialization, based on the configuration settings.

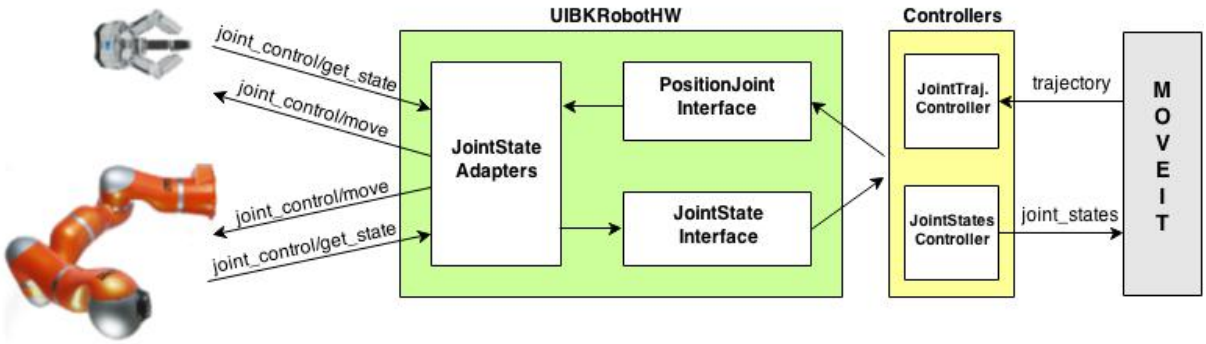

Figure 3.7: Hardware adapter architecture

After creation, each *JointStateAdapter* waits a certain amount of time for an initial joint states message. If it does not receive such a message before timeout, it will automatically shut down for safety reasons and report an error. This is very important as the *JointStateAdapter* imediately begins to send joint positions after initialization. The position to be sent is initially the currently known position, as long as no other values have been commanded by the controllers. Therefore the initial position always has to be known, otherwise dangerous and rapid robot motions could occur on startup. The adapter configuration happens via the parameter server. Available configuration parameters are:

- `adapter_list`

  Contains a list of all *JointStatesAdapters* that have to be created. For each adapter name mentioned in this list a detailed configuration is required. The subsequent parameters have to be configured for each single adapter.

- `joint_state_topic`

  The name of the topic where a specific adapter listens for joint states. The expected message type `sensor_msgs/JointStates`. This parameter is mandatory.

- `readonly`

  This parameter is optional. If true then the adapter will only listen for joint states but not publish commanded values.

- **`joint_command_topic`**

  The name of the topic the adapter should use to publish the commanded values to. The expected message type is `std_msgs/Float64MultiArray`. This parameter is mandatory if the adapter is not configured to be read only.

- **`joints`**

  A list of joint names that should be controlled by the adapter. The order in this list also determines the order of the values in the published message.

- **`joint_name_prefix`**

  This parameter can be used to be able to uniquely identify joints. For example both arms are using the same joint names (`arm_0_joint`, `arm_1_joint`,...). But in a URDF model, joint names have to be globally unique. Therefore the prefix can be used to prepend the original name to achieve uniqueness. A value of `right_` for example will lead to the joint names `right_arm_0_joint`, `right_arm_1_joint`,...).

An example configuration can be found in Listing3.2. As the topic names, used by the *JointStateAdapters* can be configured freely, the hardware adapter is able to interact with the simulator and the real robot as well, as both of them provide exactly the same ROS interface.

Listing 3.2: Adapter configuration for left arm and gripper

```
adapter_list:
  - left_arm
  - left_sdh

left_arm:
    joint_state_topic: "left_arm/joint_control/get_state"
    joint_command_topic: "left_arm/joint_control/move"
    joint_name_prefix: "left_"
    joints:
      - arm_0_joint
      - arm_1_joint
      - arm_2_joint
      - arm_3_joint
      - arm_4_joint
      - arm_5_joint
      - arm_6_joint

left_sdh:
    joint_state_topic: "left_sdh/joint_control/get_state"
    joint_command_topic: "left_sdh/joint_control/move"
    joint_name_prefix: "left_sdh_"
    joints:
        - knuckle_joint
        - finger_12_joint
        - finger_13_joint
        - thumb_2_joint
        - thumb_3_joint
        - finger_22_joint
        - finger_23_joint
```

During hardware adapter startup an instance of the *UibkRobotHW* and *ControllerManager* is created from configuration. As the control loop must not be interrupted by ROS callback functions, it is launched in a separate thread after completing initialization, using a loop rate of 100hz. On each iteration the exact time since the last step is calculated. The *ControllerManager*

then updates all registered controllers based on current state and time since last iteration. After handling the controllers, the *JointStatesAdapters* are forced to send the commanded values to the hardware. The ROS callback functions are handled in the original thread.

After starting the hardware adapter, the required controllers have to be loaded and started. This is usually done, by using the corresponding ROS services, provided by a running *ControllerManager* instance, but the `controller_manager` package contains a tool named `spawner` that simplifies the starting process. The configuration of each controller has to reside on the parameter server. Utilized controllers are the *JointTrajectoryController* and the *JointStateController*. The *JointTrajectoryController* provides the required *FollowJointTrajectory* action interface for a group of joints. For each robot component a separate *JointTrajectoryController* is used. The *JointStateController* publishes the collected states of all robot joints at once to the `joint_states` topic, which is used by MoveIt to monitor the robot configuration.

3.6.3 Launching the hardware adapter

The file `hardware_adapter.launch`, located in the `uibk_moveit_adapter` package was created to handle the necessary configuration parameter upload and launch the hardware adapter node for the simulator and the real robot as well. As can be seen in Listing3.2, the configured topic names for the *JointStateAdapters* are defined, using *relative* graph resource names (i.e. without trailing slash). The required instance is than accessed by simply shifting the node into `simulation` or `real` namespace. This is realized, specifying the `config_name` parameter of the launch file. The command line statement

```
roslaunch uibk_moveit_adapter hardware_adapter.launch config_name:=simulation
```

launches the hardware adapter in `simulation` namespace.

After launching the hardware adapter node, the required controllers are loaded and started. The node provides feedback information about the state and errors during startup process on the console output. It is crucial that the joint state topics of simulator or real robot are available before launching the hardware adapter node, otherwise it will not be able to work because of missing initial joint states. After successful startup, the additional controller topics can be found in the corresponding namespace.

3.7 Adjusting the MoveIt configuration

The prerequisites are now made for connecting MoveIt to the (simulated or real) robot. The last remaining step is to adjust the MoveIt configuration for establishing a connection to the *FollowJointTrajectory* controllers. Therefore the configuration file `controllers.yaml` was created in the `uibk_robot_moveit_config` package. It contains a list, describing the available controllers. Each controller description contains the name of the controller, it's action namespace, the controller type and a list, containing the names of the controlled joints.

For starting MoveIt, the launch file `moveit_planning_execution.launch` was created. This file is intended to launch a `move_group` node either for simulated or real robot, together with the corresponding hardware adapter. The namespace can be selected using a boolean parameter named `simulation`. The statement

```
roslaunch uibk_robot_moveit_config moveit_planning_execution.launch simulation:=false
```

launches a MoveIt configuration, connecting the `move_group` instance to the real robot. The launch file also starts RViz configured with the motion planning plugin. This is used for visualizing the planned trajectories and can also be used to test the connection to the robot by making planning requests and execute the resulting trajectories.

Chapter 4

Pick and place

The implementation of a benchmark pick and place task which is executable on a simulator as well as on the real robot states one of the objectives of this project. The task consists of various stages which have to be planned using the planning framework introduced in the previous chapter. The overview at the beginning of this chapter provides information about general grasping tasks, explains the process step by step and describes the successive stages that have to be executed. The second part focuses on how pick and place tasks are planned and executed using the motion planning framework that was described in the previous chapter. The third section explains the implementation of the benchmark pick and place task in detail and describes the involved message types and action servers. The last section discusses some observations that have been made during the implementation process.

4.1 Overview

A pick and place task is the process of grasping an object, lifting it and dropping it at a target position. Humans can do that without even thinking about it. But directing a robot to perform a pick and place task reveals how difficult and complex it is and how much planning has to be involved to achieve the desired result. A planner would possibly require exact knowledge about the robot and it's environment, including the objects to grasp and the obstacles around. Collisions that could harm the robot have to be avoided but other collisions are necessary when the robot has to get in contact with the world. The gripper definitely collides with the object to pick, but only during grasping and holding. Therefore there has to be a mechanism to explicitly tell the planner that specific collisions are allowed during particular stages of the operation. Moreover, after grasping an object it has to be considered as an additional part of the robot. That means for the planning domain, that the grasped object should be included to the definition of the robot during subsequent planning requests because it possibly increases the size of the end effector that carries the object.

Stationary objects usually stand or lie on a surface, called the *support surface*. During the interaction with an object, possible collisions with the support surface have to be taken into account. It is also possible that the whole process underlies additional constraints, so called *path constraints*. This type of constraint has to be enforced along the whole path which the grasped object takes during the operation. For example, when carrying a glass filled with liquid it has to remain in an upright position, otherwise the liquid is lost. That means, the glass has to be held in a specific orientation during the whole task. This can be described as an so-called *orientation constraint* which is a special type of a path constraint. The planning solution has to provide mechanisms to define and enforce such types of *path constraints*.

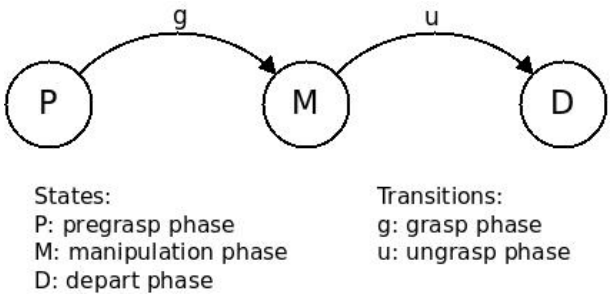

Figure 4.1: State transition diagramm for a pick and place task
Image inspired by (Kang and Ikeuchi, 1994)

(Kang and Ikeuchi, 1994) propose a method for *temporal segmentation* of arbitrary grasping tasks. They divide each task into sequences of *subtasks* that *can be represented as series of states and transitions*. According to their notion, a pick and place operation is divided into 5 phases.

- **Pregrasp phase**

  This phase starts at an arbitrary robot configuration. The hand is moved towards the grasp location and the fingers have to be adjusted to preshape the gripper in a way that allows to enclose the object to grasp (or at least that part that is used to clutch it). The required gripper configuration depends on the size and shape of the object to grasp and the structure of the gripper.

- **Grasp phase**

  This is the stage where the robot gets in contact with the object. The gripper closes around the object and applies as much force as necessary to be able to take and hold it. The grasp phase states the transition between the *pregrasp* and the *manipulation* phase.

- **Manipulation phase**

  The grasped object is enclosed by the gripper and considered to be part of the robot configuration. During *manipulation* phase, the object is translated towards the place location. Possible path constraints have to be enforced whilst that stage.

- **Ungrasp phase**

  The manipulator reached at the goal location and the gripper opens and releases the object. This is the transition between *manipulation* and *depart* phase.

- **Depart phase**

  The object was placed at the target location and relased by the gripper. Now the manipulator retreats from the object. After that the entire task is completed.

The corresponding state transition diagram can be seen in Figure4.1. The task is only considered to be complete if each single stage was successfully executed. Necessary planning parameters like the grasp location and gripper configurations are usually provided by a grasp planner. This is an additional node within the planning pipeline that can be used to identify objects in the environment of the robot, usually based on 3D sensor data and calculate the corresponding grasp parameters. Explaining the functionality of grasp planners is beyond the scope of this project though the section about implementing the reference task discusses the used parameters in greater detail and shows what data would usually be delivered by the grasp planner.

4.2 Pick and place tasks in MoveIt

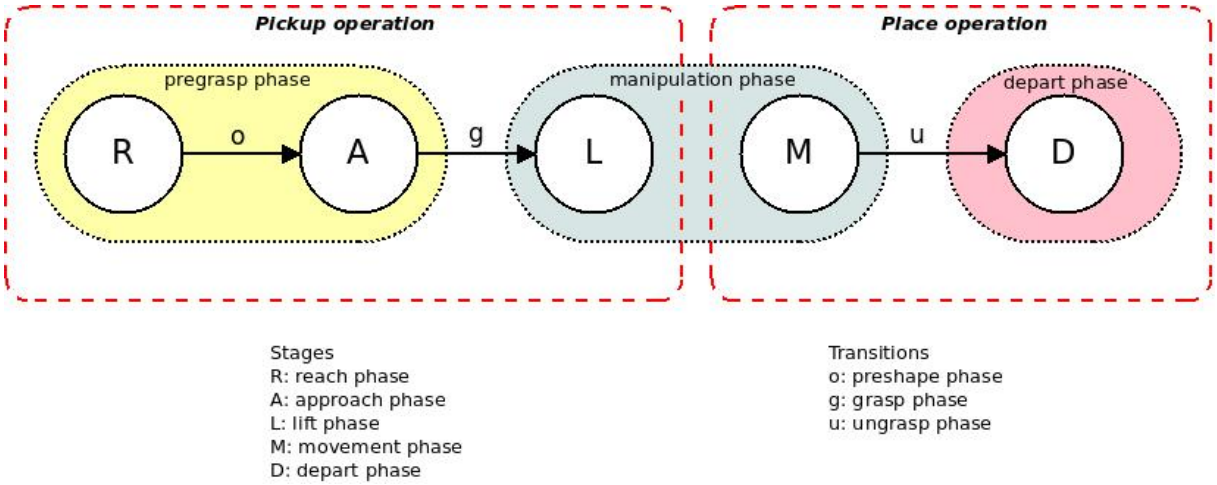

Figure 4.2: Extended state transition diagram

The robot motions during the phases described in the previous section need to be planned. Each single stage requires to plan trajectories that are free of accidental collisions, respect the limits of the robot and enforce possible additional constraints. The task can only be considered executable if a valid motion plan for each single stage exists. Planning pick and place tasks in MoveIt requires to split them into two distinct operations, namely the *pickup* operation and the *place* operation. The explanation of those operations requires to further subdivide the phases described in the previous section. Figure 4.2 shows the resulting state transition diagram.

The pickup operation starts at an arbitrary robot configuration. In the *reach* stage, the manipulator has to be brought into a position close to the object to grasp, but in a distance which allows the gripper to open safely without touching the object. The *preshape* stage sets the gripper into the *pregrasp posture* which means the fingers are brought into a shape that allows to completely enclose the object (or at least that part that is used to clutch it). During the *approach* phase the gripper moves towards the final grasp pose along the defined approach direction. The *grasp* phase moves the gripper fingers into the *grasp posture* - a configuration that encloses the object and applies as much force as necessary to be able to take and hold it. The resulting collisions between the gripper links and the object have to be ignored by the planner. From that point on the grasped object is *attached* to the gripper, which means it is considered to be an additional part of the end effector during subsequent planning steps. The

pickup operation completes lifting the object along the retreat direction (*lift* phase). Figure 4.3 shows the stages of the pickup phase.

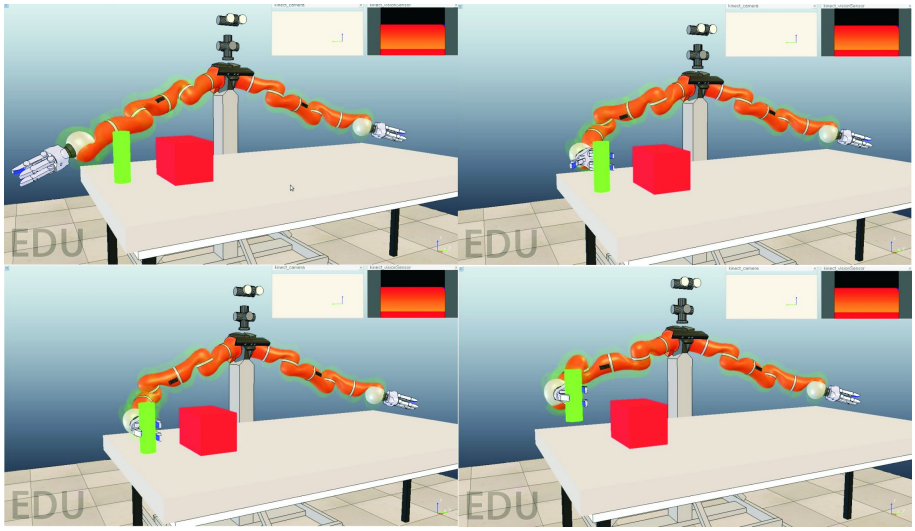

Figure 4.3: Stages of the pickup phase in the simulator

The place operation starts after a successful pickup. The grasped object is enclosed by the gripper and it is considered to be part of the robot. During the *movement* phase, the object is translated towards the final *place location*. The planner has to take into account that the object could possibly get into contact with the support surface again when it's final position is reached. At that stage, the gripper opens (*ungrasp* phase) and releases the object. Now the object has to be treated as an obstacle again and the planner is forced to avoid collisions with it. The place operation completes after the manipulator has moved away from the object (*depart* phase) along the specified retreat direction. The stages of the placement phase can be seen in Figure 4.4.

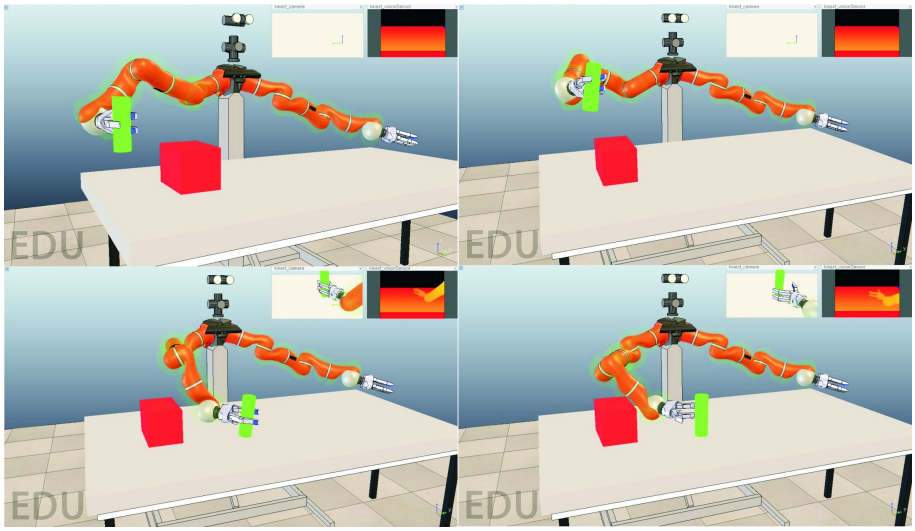

Figure 4.4: Stages of the placement phase in the simulator

The planning of the described operations make use of action servers which are advertised by the `move_group` node, namely the *pickup action server* and the *place action server*. The pickup and placement requests are composed from a large number of parameters that are explained in detail in the next section. Objects contained in the task environment have to be added to

the MoveIt planning scene. This can be done manually by publishing them to corresponding topics or by a MoveIt configuration to monitor the environment through sensor data, e.g. from a Kinect camera. For the sake of simplicity, all involved objects were manually added in the benchmark task. Pickup- and place action servers provide the ability to choose whether to execute motion plans immediately or to just perform the planning part and return the resulting trajectories for later execution. Immediate execution is often preferable because MoveIt executes the trajectories and handles additional requirements like attaching and detaching the grasped object in time. It has the drawback, however, that occasionally overly long and unnatural trajectories are executed immediately because they might be formally valid solutions for the given planning problem, though they are unsuitable for a human observer. Therefore a safety mechanism which can interrupt an execution in face of problems needs to be added. The second, planning-only method allows to visualize the resulting trajectories and then decide whether to execute them or not. But then each single trajectory stage has to be executed manually which also includes attaching or detaching objects to the manipulator. The advantage of this method is the clean separation between planning and execution, allowing maximum control over the execution flow. Therefore this method was favoured during benchmark task implementation.

4.3 Implementation of the benchmark task

This section describes the implementation of the benchmark pick and place task in detail. The source code can be found in the `uibk_moveit_tests` package. The workspace is the table in front of the robot covered with a 9 cm thick foam mat. This mat will be declared as the support surface later on. The object to grasp is a cylinder with 4 cm radius and a height of 25 cm. The cylinder is located on a fixed, known position. A cube with an edge length of 20 cm acts as additional obstacle within the workspace. The goal of the task is to pick the cylinder up, using the right arm of the robot and place it at the goal location without colliding with the obstacle or other parts within the robot's environment. The implementation makes use of the previously described pick and place functionality. The necessary steps are explained in the following subsections.

4.3.1 Creating the environment

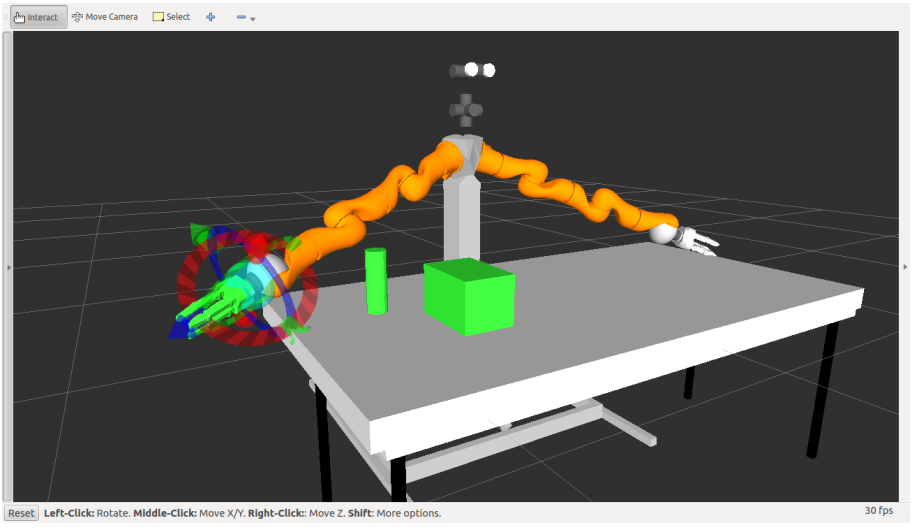

Figure 4.5: Planning scene after inserting the collision objects

As MoveIt is currently not configured to use sensor data, it has to be informed about the task environment. The table and the surface mat (`table_surface_link`) are already part of the URDF description but the cylinder and the obstacle have to be added to the planning scene. This is done utilizing the *PlanningSceneInterface* class that provides functionality to manipulate the current planning scene. The *CollisionObject* type is used to describe those objects. Both of them are primitive shapes. Necessary parameters are the shape type, dimensions and pose. Additionally each *CollisionObject* needs a unique ID which is used to identify the shape within the planning scene. Listing 4.1 shows the necessary code for creating the obstacle, Figure 4.5 shows a visualization of the task environment after adding the *CollisionObjects*.

Listing 4.1: Creating the obstacle

```
1  std::vector<moveit_msgs::CollisionObject> collision_objects;
2  // create an obstacle
3  shape_msgs::SolidPrimitive box;
4  box.type = box.BOX;
5  box.dimensions.resize(3);
6  box.dimensions[0] = 0.2;
7  box.dimensions[1] = 0.2;
8  box.dimensions[2] = 0.2;
9
10 /* A pose for the box (specified relative to frame_id) */
11 geometry_msgs::Pose box_pose;
12 box_pose.orientation.w = 1.0;
13 box_pose.position.x = 0.15;
14 box_pose.position.y = 0.10;
15 box_pose.position.z = 0.1 + SUPPORT_SURFACE_HEIGHT;
16
17 obstacle.primitives.push_back(box);
18 obstacle.primitive_poses.push_back(box_pose);
19 obstacle.operation = obstacle.ADD;
20
21 collision_objects.push_back(obstacle);
22 // Now, let's add the collision object into the world
23 planning_scene_interface_->addCollisionObjects(collision_objects);
```

4.3.2 Generating possible grasps

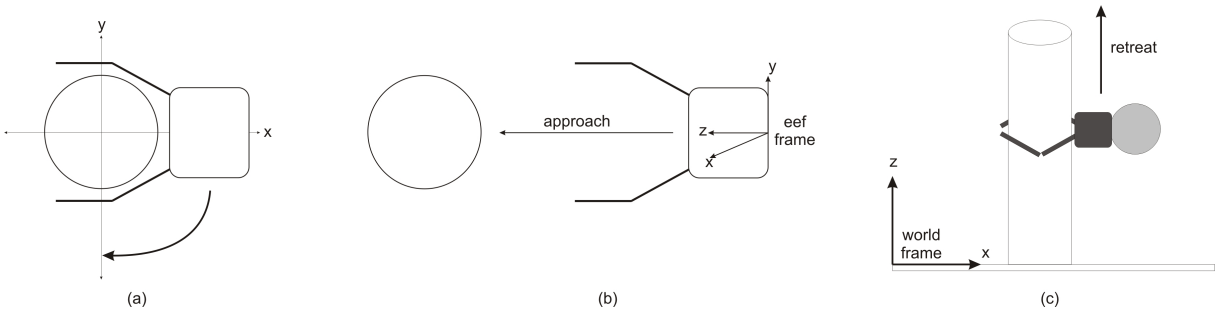

Figure 4.6: Grasp poses, gripper approach and retreat

Picking up an object can always be done in several ways. Depending on the shape of the object there always multiple alternatives how gripper can safely approach and grasp. Providing a number of different grasps raises the probability that the pickup request will be successful. Before calling the pickup action server it is necessary to generate a set of possible grasps for the object to pick. This information is usually provided by a grasp planner. The sample task encapsulates

the grasp planner functionality within the `generateGrasps()` method. This method takes the current pose of the cylinder as input and calculates 10 possible grasp poses along an imaginary circle segment around the cylinder location (Figure 4.6a). Grasps are defined in terms of the *Grasp* data structure. The following paragraphs describe the involved parameters and how they were determined in the benchmark task implementation.

Grasp ID A string that uniquely identifies the grasp. As usually many different grasps are provided for each pickup request, this ID can be used to identify the grasp which is finally picked to solve the planning problem. The grasp IDs are formed according to the pattern `[Grasp]x`, where $x \in [0, 10]$

Pre-grasp/grasp posture Describes the shape of the hand before/after grasping the object. Pre-grasp and grasp postures are joint trajectories for the gripper, containing just one trajectory point stating the target configuration for the gripper in the opened respectively the closed state.

Pre-grasp approach and post-grasp retreat The pre-grasp approach and the post-grasp retreat are defined as *GripperTranslation*, as can be seen in Listing 4.2. This is a special message type that describes the direct gripper movement from one position towards a target. The direction is defined as a three dimensional vector. The length of the translation can be set in a flexible way by specifying a desired distance and a minimum distance. That means that the location where the gripper needs to open is not explicitly set and can be any point along the approach vector between minimum distance and desired distance. Experiments showed that the success rate is higher if the grasp parameters allow some flexibility to the planner at certain points. The approach vector depends on the grasp pose and points along the z-axis of the end effector frame towards the object (Figure 4.6b). The gripper retreat vector points up, along the z-axis of the world (Figure 4.6c).

Listing 4.2: Definition of gripper translations

```
// gripper approach
GripperTranslation gripper_approach;
gripper_approach.desired_distance = 0.2; // cm
gripper_approach.min_distance = 0.1; // cm
gripper_approach.direction.header.frame_id = EE_PARENT_LINK;
gripper_approach.direction.vector.x = 0;
gripper_approach.direction.vector.y = 0;
gripper_approach.direction.vector.z = 1;
new_grasp.pre_grasp_approach = gripper_approach;

// gripper retreat
GripperTranslation gripper_retreat;
gripper_retreat.desired_distance = 0.2; // cm
gripper_retreat.min_distance = 0.1; // cm
gripper_retreat.direction.header.frame_id = BASE_LINK;
gripper_retreat.direction.vector.x = 0;
gripper_retreat.direction.vector.y = 0;
gripper_retreat.direction.vector.z = 1;
new_grasp.post_grasp_retreat = gripper_retreat;
```

Grasp pose The grasp pose specifies the position and orientation of the end effector during the grasp stage as can be seen in Figure 4.7. The function in 4.1 is used to calculate the grasp

pose based on angle θ.

$$f(\theta) = \begin{pmatrix} x_\theta \\ y_\theta \\ z_\theta \\ roll_\theta \\ pitch_\theta \\ yaw_\theta \end{pmatrix} = \begin{pmatrix} -r\sin(\theta) \\ r\cos(\theta) \\ 0 \\ \frac{\pi}{2} \\ \theta \\ -\frac{\pi}{2} \end{pmatrix} \tag{4.1}$$

Parameter $r = 19cm$ specifies the distance between the eef link and the object center. Grasp poses are calculated for 10 different angles θ from the interval $[\pi, \frac{\pi}{2}]$. All poses are computed relative to the cylinder reference frame and then converted to the reference frame of the world.

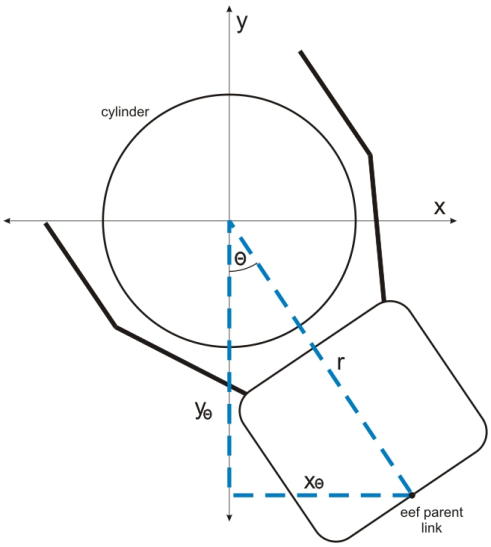

Figure 4.7: Grasp pose determination

Grasp quality This parameter allows to tell the planner, how 'good' a specific grasp is, i.e. how likely it is that the grasp can be successfully executed. MoveIt can then favour grasps with higher quality if several valid solutions are found by the planner. As none of the generated grasps should be favoured among the others, all grasps use the quality parameter value 1.0, which indicates that they are equal in quality.

Allowed touch objects A list, containing the identifiers of the objects, that are allowed to be touched when using that specific grasp. In the benchmark task, this is only the grasped object.

4.3.3 Planning and executing the pickup

After generating the set of grasps, the pickup operation can be planned. The *PickupAction* server handles planning and optionally the execution of all stages of a pickup operation at once. The request is done, using a *PickupAction* client to send *PickupGoal* messages to the server. The required parameters of the request message are described in the subsequent paragraphs.

`target_name` The name of the object to pick which is the ID that was used when adding the object to the planning scene.

group_name The name that identifies the planning group that should be used to perform the pickup, i.e. `right_arm`.

end_effector_name The name of an end effector in the specified planning group. This is necessary because it is possible to define planning groups that contain multiple end effectors.

possible_grasps The set, containing the previously computed grasps. At least one grasp has to be provided, but the success rate rises when providing multiple grasps.

support_surface_name The name of the link that acts a support surface within the planning scene. This is the link that states the surface mat covering the table (`table_surface_link`).

allow_gripper_support_collision That parameter specifies, whether collisions between the gripper links and the support surface are allowed, or not.

path_constraints It is possible to apply a set of path constraints to the planning request. This could be for example an orientation constraint for the end effector. Path constraints are enforced on each single point of the resulting trajectory which drastically raises the complexity of the planning problem. The sample task does not apply any task constraints.

allowed_planning_time The maximum amount of time, the planner is allowed to find a solution. The sample tasks uses a value of 5 seconds.

plan_only Specifies, if resulting trajectories should be immediately executed, or not. This is set to `true` to allow a visual validation of the resulting motion plan.

On success, the pickup action server returns a motion plan, containing trajectories for all stages of the pickup operation. MoveIt automatically publishes planned trajectories to the `move_group/display_planned_path` topic. RViz can be configured to display the trajectories published to that topic. If a visual validation results in a satisfying solution it can be executed on the `execute_kinematic_path` service, which is advertised by the `move_group` node. The service sends given trajectories to the responsible controller and provides feedback information about the execution status in the service response message. The pickup phase completes after successful execution of all trajectory stages.

4.3.4 Planning and executing the placement

The placement phase is planned and executed in a similar manner. The cylinder has to be placed on a specific location within the workspace in an upright position - the rotation around the z-axis doesn't matter. Therefore a set of possible place poses is generated in 20 different orientations around the z-axis (Figure 4.8a), allowing the planner to choose which one to use. The final approach towards the goal location is specified as *GripperTranslation*. The direction vector points down along the z-axis of the world (Figure 4.8b). Desired and minimum distances are set to 20 cm and 10 cm respectively. The same pre-grasp posture configuration is used for the post-place posture. The last required parameter for a place location is the *GripperTranslation* which describes the retreat after releasing the object at the target location. The direction depends on the chosen orientation and points towards the negative z-axis of the gripper reference frame (Figure 4.8c).

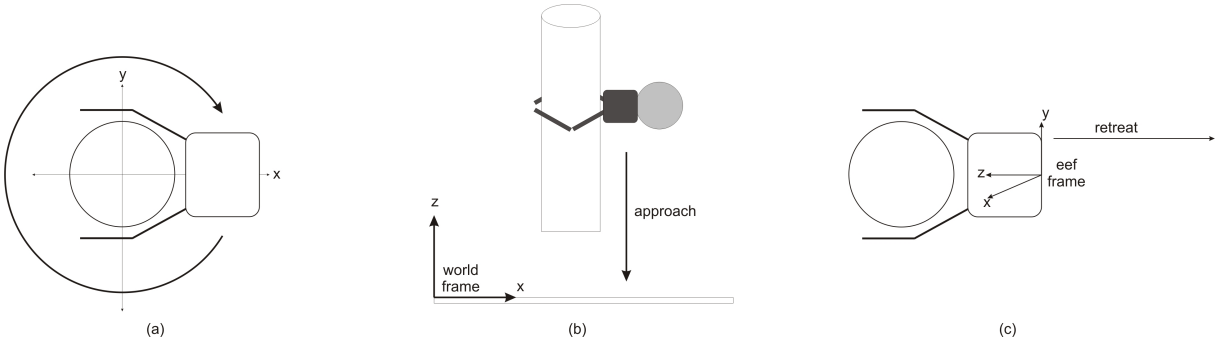

Figure 4.8: Place locations, pre-place approach and post-place retreat

The *PlaceAction* server is responsible for planning the whole placement phase. The request utilizes a *PlaceAction* client and *PlaceGoal* messages. The server call is done in the same way as in the pickup phase. The most important parameters are the ID of the object to place, the name of the planning group, the set of possible place locations and the maximum allowed planning time. On success, again the resulting trajectories are visualized in RViz and can be executed after positive validation. After execution, the task is considered to be completed and the picked object is detached from the gripper and again part of the collision world.

4.4 Running the benchmark task

The benchmark task is designed to run on the simulator and with the real robot as well. It only depends on a running `move_group` instance. Therefore it is necessary to start the simulator or real robot and launch the corresponding MoveIt instance as described in Section 3.6.3. The benchmark task can then be executed in the desired namespace. This is done by setting the `ROS_NAMESPACE` environment variable to either `simulation` or `real` within the terminal that is used to run the task. The following example runs the benchmark task on the simulator.

```
export ROS_NAMESPACE=simulation
rosrun uibk_moveit_tests sample_pick_place
```

After creating the environment and adding the objects to the planning scene the pickup phase gets planned. The outcome can be seen in RViz and the program will ask if the resulting trajectory is ok and should be executed. A negative answer will force the program to replan the pickup and ask again, otherwise the trajectory gets executed. After successful execution, the place phase gets planned. The resulting plan is visualized as well and its execution also needs confirmation. The program exits after successful completing the placement phase. Listing 4.3 shows the main routine of the benchmark task. It was observed that planning sometimes fails due to unknown MoveIt internal issues and subsequent calls with the same parameters are successful. Therefore, the pickup and the place operation are handled within a loop.

Listing 4.3: Benchmark task main routine

```
bool start() {
  // create obstacle and object to move...
  createEnvironment();

  bool picked = false;
  while (!picked && ros::ok()) {
    if (!pick(start_pose_, OBJ_ID)) {
      ROS_ERROR_STREAM_NAMED("pick_place", "Pick failed. Retrying.");
      // ensure that object is detached from gripper
      cleanupACO(OBJ_ID);
    } else {
      ROS_INFO_STREAM_NAMED("pick_place", "Done with pick!");
            picked = true;
    }
  }

  ROS_INFO_STREAM_NAMED("simple_pick_place", "Waiting to put...");
  ros::Duration(5.5).sleep();

  bool placed = false;
  while (!placed && ros::ok()) {
    if (!place(goal_pose_, OBJ_ID)) {
      ROS_ERROR_STREAM_NAMED("pick_place", "Place failed.");
    } else {
      ROS_INFO_STREAM_NAMED("pick_place", "Done with place");
      placed = true;
    }
  }
  return true;
}
```

4.5 Discussion

This section gives an overview of the most important observations that have been made during the implementation of the sample task:

- It is very important to provide some degree of freedom to the planner at various points. This is achieved by providing various grasps and place locations during planning requests. The definitions of gripper translations like pre-grasp approach, post-grasp retreat, pre-place approach and post-place retreat also allow to add additional flexibility by setting the desired and minimum distance values accordingly. Very strictly defined planning requests are very likely to fail whereas requests with a higher degree of flexibility drastically raise the overall success rate.

- Working with path constraints drastically drops the success rate because of the high complexity of the planning problem. Enforcing path constraints results in a drastically increased planning time and a large number of necessary IK requests. Therefore no path constraints were used within the sample task. It is possible, that a faster IK solution than the one that is currently utilized could help to solve the problem but that was not tested in this project.

- It was observed that sometimes planning requests seem to fail due to some MoveIt-internal issues and not because planning was not possible at all. Therefore it is necessary to repeat

failed requests because it is very likely that planning succeeds on subsequent attempts. In the sample task implementation, the planning requests are done within a loop. If a request fails, it gets repeated a couple of times. A successful request breaks the loop and continues the execution flow. This strategy is obviously only feasible in the reference task because its configuration was designed to be plannable and executable.

- Resulting trajectories should always be visualized and validated before confirming the execution. This can be done, using RViz and/or the simulator. The calculated motion plans might be valid in terms of the defined goal constraints but sometimes they are overly long and unnatural and therefore unsuitable. Moreover, the planner can only take into account what it knows about the robot's environment. Especially in the robot lab the environment changes frequently when new equipment is mounted in the area around the robot. Therefore the visual validation is particularly important to avoid damages on the robot and its environment.

Chapter 5

Conclusion

The simulation solution, developed within this project was designed to reflect the IIS lab robot setup at a certain configuration.

During the scope of this bachelor project, a custom simulation solution for the IIS lab robot setup was designed, utilizing the V-Rep simulation platform. It was shown, how the simulation scene was assembled and how dynamic models of required robot components are created. The modelling process was shown on the example of the Schunk SDH2 gripper component. The resulting reference scene reflects the IIS lab robot setup at a certain configuration, but it can easily be adjusted to reflect other settings. The implemented ROS interface is not tied to a specific simulation scene but to tagged simulation models. Therefore it is easy to create additional scenes with different settings. The ROS control interfaces for the models were implemented as a V-Rep simulator plugin. The plugin is loaded on V-Rep startup and identifies known components within the currently active simulation scene. As soon as such a component is identified, it can be controlled by the proper ROS control interface. The plugin design allows to introduce additional components and extend it's functionality.

The second objective of that project was to integrate the motion planning framework MoveIt into the current setup. Therefore a URDF representation of the environment was created that exactly describes the involved robot components and their placement. Based on that URDF description, the planning framework was set up and configured. The integration process also required the implementation of an additional ROS node that allows the proper execution of time-parametrized trajectories. That node was designed to communicate with the existing ROS control interface and can be configured to interact with the simulator and the real robot as well. Finally, the proper functionality of the planning tools and the simulator was shown on a reference *pick and place* task that can be executed on both instances.

Bibliography

Open motion planning library: A primer. `http://ompl.kavrakilab.org/OMPL_Primer.pdf`. Published by Kavraki Lab, Rice University.

Michael Riis Andersen, Thomas Jensen, Pavel Lisouski, Anders Krogh Mortensen, Mikkel Kragh Hansen, Torben Gregersen, and Peter Ahrendt. Kinect depth sensor evaluation for computer vision applications. Technical report, Århus Universitet, 2012.

Samuel R Buss. Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods. *IEEE Journal of Robotics and Automation*, 17:1–19, 2004.

Howie M Choset. *Principles of robot motion: theory, algorithms, and implementation*. MIT press, 2005.

John J Craig. *Introduction to robotics: mechanics and control*. Pearson/Prentice Hall Upper Saddle River, NJ, USA:, 2005.

Marc Freese, Surya Singh, Fumio Ozaki, and Nobuto Matsuhira. Virtual robot experimentation platform v-rep: A versatile 3d robot simulator.

Sing Bing Kang and Katsushi Ikeuchi. Determination of motion breakpoints in a task sequence from human hand motion. In *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*, pages 551–556. IEEE, 1994.

KUKA. Kuka system software 5.6 lr - operating and programming instructions for system integrators, 2012.

Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006.

Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, 2009.

Schunk. Sdh2 documentation overview, 2010.

Jacob T Schwartz and Micha Sharir. On the "piano movers" problem. ii. general techniques for computing topological properties of real algebraic manifolds. *Advances in applied Mathematics*, 4(3):298–351, 1983.

Ioan A. Sucan and [Online] Available: http://moveit.ros.org Sachin Chitta, MoveIt! Archive. `http://moveit.ros.org/`.

Appendix A

Simulator documentation

This chapter provides a documentation for the simulation solution that was developed during this project. Some of the steps, described within this documentation require basic knowledge about using the V-Rep scene editor.

A.1 Installation and startup

The simulator plugin code as well as the scene and model files is located in the `iis_simulation` package which is part of the `iis_robot_sw` repository. The plugin code was written in C++ and needs to be compiled, using the catkin build system which is part of the ROS distribution. Detailed information about how to install and build the source code can be found in the top-level README file within the repository. The compilation process generates a library file, named `libv_RepExtSimulatorPlugin.so`. To be able to work, that library file needs to be copied into the working directory of the V-Rep installation. This happens automatically when using the top-level makefile of the `iis_robot_sw` repository. The plugin library is automatically detected and loaded during V-Rep startup. But as it's functionality depends on ROS it will be unloaded immediately if no running roscore can be detected. The status of recognized plugins can be seen on the console output.

The package contains the reference scene file, called `scenes/model_assembled.ttt` which can be started, using the `robot_scene.launch` file. The command

```
roslaunch iis_simulation robot_scene.launch scene:=MY_CUSTOM_SCENE.ttt
```

starts an instance of V-Rep, loads the scene file provided with the (optional) `scene` parameter and starts the simulation. If no scene file is provided, the default scene is loaded which is the provided reference scene. Scene files have to reside in the `scenes` folder within the package, otherwise the launch file is not able to locate them.

A.2 Modifying the simulation scene

The reference scene contains a fully functional replication of the IIS lab robot setup, composed from two arms, two grippers and a Kinect camera. It is recommended to start with the setup within this file to create alternative scenes and store modified versions in different files. The base of each arm is marked by a dummy element at root level of the scene hierarchy, i.e. `dummy_frame_left_arm` and `dummy_frame_right_arm`. Modifying the mounting of the arms is done by changing the alignment of those dummy elements. Currently they are placed relative

to the origin of the world reference frame. Parts of the scene can safely be deleted without breaking the functionality. To delete a gripper, it's base element has to be located in the scene hierarchy. This is done by expanding the model tree of the corresponding arm until the gripper base can be selected and removed. Each other part contained within the scene can be removed or rearranged in the same way (table, torso, Kinect camera).

The dummy element called `ref_frame_origin` states the origin of the reference frame used by the arm controllers and is indicated by a slightly green shimmering sphere near the top left corner of the table. Cartesian positions received from and sent to the arms are interpreted relative to that reference frame. The origin can be changed easily by just modifying the position and orientation of that dummy element. If it is removed than all positions are interpreted to be relative to the absolute world reference frame (which is currently at the same location).

A.3 Modifying custom developer data tags

Important parts within the scene hierarchy have to be tagged with custom developer data tags, as explained in Section2.5.2.

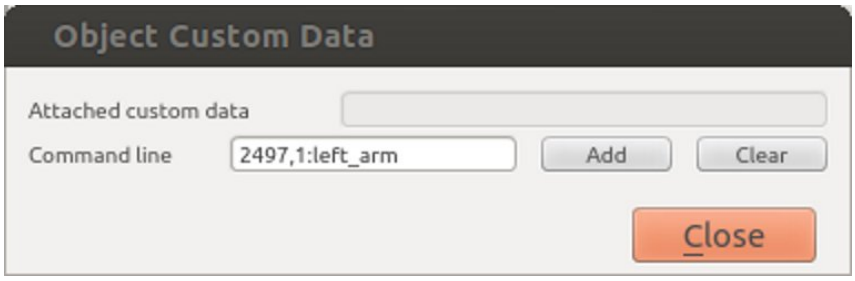

Figure A.1: Edit custom developer data dialog

To edit those data segments, the corresponding scene object needs to be selected in the scene hierarchy. After clicking the option *View/edit custom data* within the `common` section of the *Scene Object Properties* dialog, a new dialog box appears that allows to attach and remove custom data segments. Existing segment have to be removed prior to attaching a new value. This is done by entering the header ID (2497) into the text box and clicking the `clear` button. That causes all data segments, identified by the given header ID to be deleted on the selected scene object. After that, the new data segment can be attached by entering the proper values into the text box and clicking the `add` button. The values, entered in the dialog, shown in FigureA.1 identify a scene object to be the model base of a *LWRArmComponent* with name `left_arm`.

A.4 ROS interface

The ROS interface for the simulated robot components is composed from a set of inbound and outbound ROS topics, that can be used to send commands or to retrieve state data. The used topic names follow the IIS lab internal naming conventions. Each topic name has the structure

`[namespace]/[componentname]/[controltype]/[name]`

The namespace is necessary to distinguish between the topics of simulator and real robot. All simulator related topics reside in the `simulation` namespace. The component name identifies

a specific instance within that namespace, e.g. `left_arm` or `right_sdh`. This is useful as often several devices of the same type are used and all of them are utilizing the same topic names. The control type groups sets of topics into different categories. Possible values are `joint_control`, `cartesian_control`, `sensoring` and `settings`. Finally, the chosen name is used to identify the specific topic within the category. The topic `/simulation/left_arm/joint_control/move` states the topic named `move` in category `joint_control` for a simulated component, named `left_arm`.

The topics are provided by the registered *ComponentControllers* and are dynamically composed, based on the component name. The overall namespace is declared within the `initialize()` method of the *ROSServer* class. The component name has to be provided as value segment of the custom developer data tag that identifies the model base of a specific component within the scene. The control type and name segments of the topics are defined within the `initialize()` methods of the concrete controller instances. Some of the provided topics are only fake implementations as the corresponding functionality cannot be implemented on the simulator. Therefore they are just added for consistency reasons, but only fake data is published. These include all temperature and impedance related topics. When messages are sent to one of those fake topics, a corresponding warning will be displayed on the console.

A.4.1 Arm control topics

The topics, provided by the *LWRArmController* follow the specification of the *KUKIE* control interface that gets utilized on the real robot arms in the IIS lab. The only exception is the `sensoring/get_collision_state` topic that is unique to the simulator and allows to read the current collision status of the underlying arm. A complete list of the topics, available on the simulator can be found in TableA.1. Controller related errors are published to the `sensoring/error` topic but they are displayed on the console output of the simulator as well. Joint names, used by the `joint_control/get_state` topic are defined in the value segments of the custom developer data tags, attached to the arm joints.

Topic	Message type	Direction
cartesian_control/get_impedance	iis_kukie/CartesianImpedance	outbound
cartesian_control/get_pose	geometry_msgs/Pose	outbound
cartesian_control/move	geometry_msgs/Pose	inbound
cartesian_control/set_impedance	iis_kukie/CartesianImpedance	inbound
cartesian_control/set_velocity_limit	std_msgs/Float32	inbound
joint_control/get_impedance	iis_kukie/FriJointImpedance	outbound
joint_control/get_state	sensor_msgs/JointState	outbound
joint_control/move	std_msgs/Float64MultiArray	inbound
joint_control/set_impedance	iis_kukie/FriJointImpedance	inbound
joint_control/set_velocity_limit	std_msgs/Float32	inbound
sensoring/cartesian_wrench	geometry_msgs/Wrench	outbound
sensoring/error	iis_kukie/KukieError	outbound
sensoring/get_collision_state	std_msgs/Int32	outbound
sensoring/state	std_msgs/Int32MultiArray	outbound
sensoring/temperature	std_msgs/Float32MultiArray	outbound
settings/get_command_state	std_msgs/Float64MultiArray	outbound
settings/switch_mode	std_msgs/Int32	inbound

Table A.1: Available topics of *LWRArmController*

A.4.2 Hand control topics

The hand related topics are listed in TableA.2. Joint names, used by the `joint_control/get_state` topic are defined in the value segments of the custom developer data tags, attached to the hand joints. Setting motor currents influences the effort of the joint motors. The provided values are interpreted as percentages, e.g. a value of 0.5 will reduce the effort setting of the corresponding joint motor to the half of the possible maximum value. The currents can be defined for the `move` and `gripHand` topics separately.

Topic	Message type	Direction
`joint_control/get_state`	`sensor_msgs/JointState`	outbound
`joint_control/move`	`std_msgs/Float64MultiArray`	inbound
`joint_control/gripHand`	`iis_schunk_hardware/GripCmd`	inbound
`sensoring/temperature`	`std_msgs/Float64MultiArray`	outbound
`settings/get_motor_current`	`iis_schunk_hardware/MotorCurrentInfo`	outbound
`settings/set_motor_current`	`iis_schunk_hardware/MotorCurrent`	inbound

Table A.2: Available topics of *SchunkHandController*

A.4.3 Kinect camera settings

The Kinect camera model is not directly controlled via the simulator plugin as it contains no moveable parts that have to be controlled from code. The RBB and depth images are published, using the built in ROS functionality of V-Rep. The code listing in FigureA.1 shows the LUA script that is associated to the Kinect camera model within the reference scene.

Listing A.1: LUA script, attached to Kinect model

```
1  if (simGetScriptExecutionCount()==0) then
2    depthCam=simGetObjectHandle('kinect_visionSensor')
3    depthView=simFloatingViewAdd(0.9,0.9,0.2,0.2,0)
4    simAdjustView(depthView,depthCam,64)
5
6    -- publish depth image
7    cmd=simros_strmcmd_get_depth_sensor_data
8    topic='/simulation/kinect1/sensoring/depth_image'
9    simExtROS_enablePublisher(topic,1,cmd,depthCam,-1,'')
10   -- publish rgba image
11   cmd=simros_strmcmd_get_vision_sensor_image
12   topic='/simulation/kinect1/sensoring/rgb_image'
13   simExtROS_enablePublisher(topic,1,cmd,depthCam,-1,'')
14 end
15
16 simHandleChildScript(sim_handle_all_except_explicit)
```

The script determines the object handle of the Kinect vision sensor and enables two ROS publishers for the captured RGB and depth images. Topic names can be changed in that script and the publishers can be dissabled by simply commenting the corresponding lines of code. The script also assigns the sensor data to an additional view within the simulation scene. This is helpful because it visualizes, the camera images during simulation. The correct transformation of the Kinect vision sensor can be obtained by expanding the model tree of the camera model and selecting the vision sensor, called `kinect_visionSensor`. The correct transformation can be read within the *position/orientation* settings of that scene object.