
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
30th August 2014

# Abstract

This document explains how to write IIS theses using the LaTeX $2_\varepsilon$ 'iisthesis' class[1], which is itself based on the LaTeX $2_\varepsilon$ 'book' class and is intended for compiling with pdflatex. This document does not tell you how to structure your thesis.

[1] version 00.06 , released on 2013/06/13.

# Acknowledgements

Thank you to the friendly members of the IIS Team.

# 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

Operating a robot and developing high level control code for it is a cumbersome task. Very often various groups of developers are working simultaneously on different parts of the software. But in most cases there is only one device that can be used for testing and so only one can run his/her code at a time. The others have to wait until they get access to the robot. Moreover, tests on real robots always come with a high level of risk. Incorrect algorithms can lead very fast 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.

The solution to those problems is the usage of a simulator that mimics the robot components and their behavior as accurate as possible. It has to provide the same control interface so that each part of the software can get tested on the simulator before it gets utilized on the real robot. Those considerations motivated the first part of this thesis – the realistic replication of the IIS-Lab robot setup on a suitable simulation platform. This involves the creation of an exact model of the robot setup, containing the various robot components and their environment. The necessary steps are explained in Chapter 2.

The second part of the thesis is about motion planning. Moving the robot hand cannot follow any arbitrary trajectory towards a target pose. During that motion it might collide with itself or any other obstacle within it's environment. That means those trajectories have to be planned carefully to avoid accidental collisions and to generate smooth and well controlled robot motions. This also involves to create and maintain an internal representation of the robot and it's environment, a step that is common to the simulation part of the thesis. Chapter 3 shows the configuration and integration of the motion planning framework MoveIt into the IIS-Lab robot setup along with some usage examples.

Chapter 4 focuses on MoveIt's grasping functionality. It shows, how a reference 'Pick and Place' task can be planned with the planning tools and executed on the simulator and the real robot as well.

In Chapter 5, some test results will be presented, analyzing the quality of the various planning algorithms and approaches.

## 1.1 IIS-Lab Robot Setup

The robot setting, that was considered in this thesis can be seen in Figure 1.1 The main part of the robot setup in the IIS-Lab consists of an aluminium torso with two mounted KUKA light weight robot arms. Those arms have 7 degrees of freedom (DOF) each one can carry up to 7kg of payload. Additionally a Schunk SDH gripper can be mounted on each arm.

- **KUKA LWR 4+ arm**

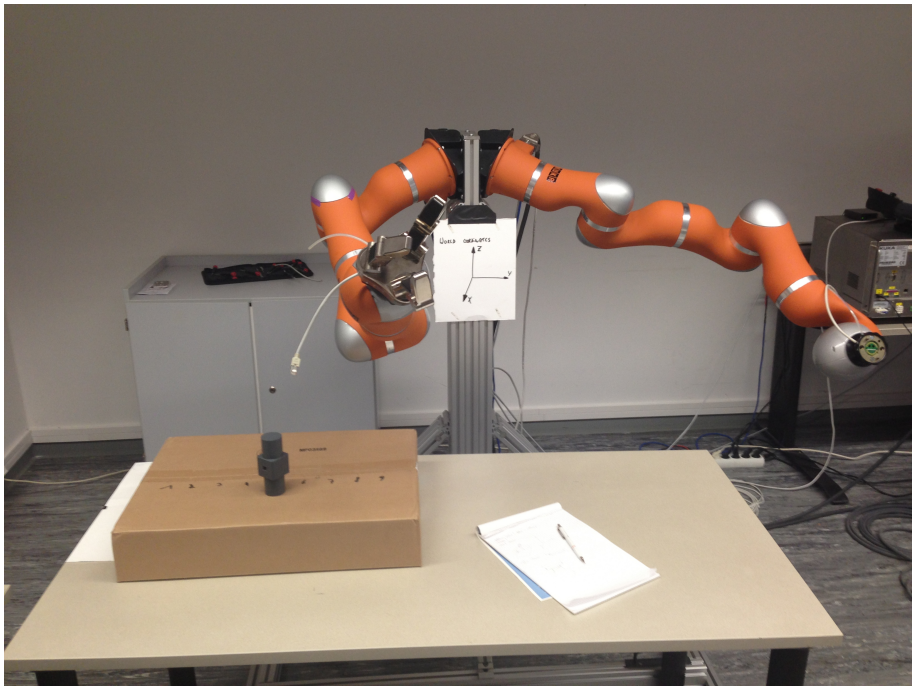

Figure 1.1: Current setup in the IIS-Lab TODO: provide more recent image

Description of the arm(7 DOF, max. 7kg payload, 16kg weight, very flexible,...

- **Schunk SDH gripper**
  Description of the gripper(3-finger-gripper, different grasp types, as powerful as human hand, very sensitive)

- **Kinect camera**
  Description of Kinect(RGB camera, Depth sensor, 3d data under any light conditions)

## 1.2 The Robot Operating System(ROS)

As the control of the robot components is based on ROS, a brief introduction shall be given here. As stated in Quigley et al. (2009), ROS is not an operating system in the classical sense. It can be seen as a 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 and runs in it's own process, adding some clearly defined functionality to the overall system. For example one node can be responsible for planning, another one for perception or controlling the hardware. Nodes communicate with each other by passing messages, using the ROS communication infrastructure. Messages are strictly typed data structures that can be defined in a special message composition language. They can be composed of primitive types like Float, Integer or Strings and also of other message types. That makes it possible to define custom messages for each use case. Messages can be published to topics. A topic can be seen as some kind of address, consisting of a string. Topics can be organized in namespaces. If two instances provide the same interface, each one can be operated in it's own namespace. This is important, as for example the simulator should provide exactly the same interface as the real robot. Without namespaces it would not be possible to operate the simulator and the real robot in parallel. So Each node can publish and also subscribe to a number of topics. It is also possible

that more than one node publishes to the same topic. The organization of the nodes of a system can be visualized as a graph.

A node can also advertise a service. This is a special type of topic that allows a synchronized message exchange. In contrast to topics, a service with a given name can only be offered by one node. The service message is composed of a request and a response part. When a node sends a service request to another node it will block, until the advertising node has handled the request and delivers a response. The nodes that make up a ROS system can be located on different machines. One machine has to be the dedicated master that runs the roscore. Other machines can connect to the master via network. ROS also provides a centralized parameter server that can be used to store configuration data for the various nodes. A system usually consists of a large number of nodes that have to be configured and started. This can be done in launch files. Those are simple textfiles, holding startup information and configuration details for one or more nodes in an XML syntax. Using the roslaunch command, a set of nodes can be configured and launched at once. It is important to understand this terminology because especially the terms node, topic and message are heavily used throughout this thesis. A more detailed documentation can be found on the ROS website.

## 1.3 Project Targets

Both objectives

- **Simulation**
  Realistic replication of the robot setup on a suitable simulation platform Generation of realistic sensor data Visualization of collisions Implementation of a ROS interface, corresponding to that one of the real robot

- **Motion Planning**
  Configuration and integration of a motion planning framework Implementation of a benchmark pick and place task, executable on simulator and real robot Provide an easy to use interface to the planner Analyse the quality of the various planning algorithms

# 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. The model has to reflect the properties and behaviour of the contained robot components 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 recommendations on 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), but also RGB and depth images, usually produced by Kinect cameras and vision sensors.
- 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.

The following sections explain in detail, how this goal was reached. At the beginning stands the process of finding a suitable simulation platform that meets the requirements and the considered criteria. After that, the chosen simulation platform V-Rep is introduced and an overview about how to design dynamic simulations is given, explaining some of the necessary terminology. Subsequent sections focus on the necessary steps to achieve the final solution, namely finding and modelling required robot components, assembling and configuring the final 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 take use of a powerful 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]. Some of the evaluated simulation platforms even provide a number of different physics engines to choose.

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)

V-Rep is a powerful robot simulation platform, developed by Coppelia Robotics. The current version (V3.1.2) provides the ability to choose from three configurable physics engines (ODE, Bullet, Vortex – only trial version) for simulating dynamic processes. It also contains a very comfortable editor for modelling robot components and simulation scenes. In the *shape edit mode* it is possible to edit and simplify meshes. This is very important because for simulating dynamic processes only simple shapes with a low amount of vertices, edges and faces should be

[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

used to reduce complexity. The contained model browser provides a rich set of different robot models, static objects and various sensors, ready to use. A powerful feature are V-Rep's so called *calculation modules*. They can be configured to provide additional calculation functionalities on groups of scene objects. The *collision detection module* is capable of detecting and visualizing all kinds of collisions within the simulation scene. The *inverse kinematics calculation module* allows to solve inverse kinematics problems for robot components. The behaviour of the simulator is highly customizable via a rich programming API for C++ as well as the scripting language LUA[7]. V-Rep is no open source software but it provides a free licence for educational units and can therefore be used for research purposes. A more detailed introduction to V-Rep can be found in Freese et al..

## 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 dynamic simulations in V-Rep. This section just covers those aspects that are important for the underlying project. A more detailed explanation can be found in the official V-Rep documentation[8].

Each simulation scene in V-Rep is composed from a number of models that are arranged within the environment. A model consists of various scene object, combined in a tree like structure to mimic the kinematic chain of a robot component. Each model has a dedicated model base and constitutes a sub-tree of the scene hierarchy. The model base is the root element of the model tree. There are existing various types of scene objects within V-Rep, but only those, which are important for the implementation will be explained here.

- **Shape**
  A shape is a 3 dimensional body. Shapes represent the visual parts as well as the dynamically enabled parts of the scene. It is necessary to distinguish between primitive shapes (Cylinder, Cuboid, Sphere, Plane and Disk) and complex shapes (triangle meshes). Primitive shapes are much easier to handle for the physics engine as there can happen a lot of optimization during dynamics calculations. Complex shapes usually look better and therefore they are used mainly as the visual part of the model but also by the collision detection module. Various shapes can be combined to groups and therefore treated as one single object. Shapes can be defined as *static* or *non-static* objects. The position of a static object is fixed relative to it's 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 dynamically enabled joint or a force sensor connection. It is also necessary to distinguish between *respondable* and *non-respondable* shapes. Respondable shapes have a clearly defined mass and moment of inertia and therefore they create collision reactions when colliding with other respondable objects during simulation. Only respondable shapes are considered during dynamics calculation. Usually a model in V-Rep is composed of a visual part, consisting of complex shapes and a hidden part, consisting of groups of primitive shapes that are configured to be used for the dynamics calculations. This issue will be covered again when explaining the creation of the hand model. Each shape also has some additional flags, defining special attributes used by the *calculation modules*. The *collidable* attribute states that a shape has to be considered during collision

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

detection. The *renderable* flag marks a shape to be recognized by a *vision sensor*. There are more flags available but only those two were used within this project.

- **Joint**
  A joint is a flexible connection between two rigid parts of a robot. It has to be distinguished between revolute or prismatic joints with one degree of freedom and spherical joints with three degrees of freedom. In the arm and hand model only revolute joints are used. Joints can be passive or actuated by a motor. The motor settings include maximum force or torque and velocity limits. If the control loop is enabled then a desired target position for the joint can be set. The controller will then try to reach this position, based on it's PID settings while respecting the motor limits. It would also be possible to program a custom control loop for each joint but for the current project this is not necessary. Joints can be operated in various different modes. A joint in *torque/force mode* is simulated, using the physics engine. This is the most realistic control mode. A joint in *inverse kinematics mode* is controlled by the inverse kinematics calculation module and if additionally the option *hybrid operation* is selected then the dynamic parameters of the joint are also taken into account. Operating a joint in *dependent mode* means that it's position depends on the position of another joint within the scene. How this dependency looks like can be configured by setting a *dependency equation*.

- **Vision sensor**
  States a simulated image capturing device. A vision sensor can capture images of the simulation scene, depending on it's configuration. It can deliver RGB image sequences as well as depth images. Vision sensors are used in the Kinect camera model.

- **Force sensor**
  A force sensor in V-Rep is a rigid connection between two dynamically simulated objects. The calculated forces and torques can be measured and visualized. 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 at all but they provide some important functionality, especially for the various calculation modules of V-Rep. They can be understood as the origin of a named reference frame with a configurable position and orientation within the simulation scene. Two dummies can be linked as *tip-target pairs* to be used by the IK calculation module. The importance becomes more clear when the construction of the simulation scene will be explained, particularly when configuring the inverse kinematics calculation module.

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.

## 2.4 Designing the simulation scene

This section explains in detail the structure of the simulation scene and the contained model components. 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 gripper 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

A model within a V-Rep simulation scene basically consists of three major parts. First there are the shapes that form the visual part of the model. Those shapes are triangle meshes for each single link of the robot component. Preferably they are not to complex, i.e. the amount of faces is not to high. The second part is formed by the joints that connect those shapes and allow to actuate the flexible parts of the model. The third part consists of the dynamically enabled parts of the model. Theese shapes form the body of the model, how it is seen by the physics engine. Those respondable parts are usually an approximation of the original meshes, composed from groups of primitive shapes. Those parts are very important as they make the model realistic and allow to interact with other respondable objects within the scene. Without them, the model would just move through other bodies as only respondable shapes are able to produce collision reactions. The steps, described within this section enclose the modelling of that three parts for the gripper model.

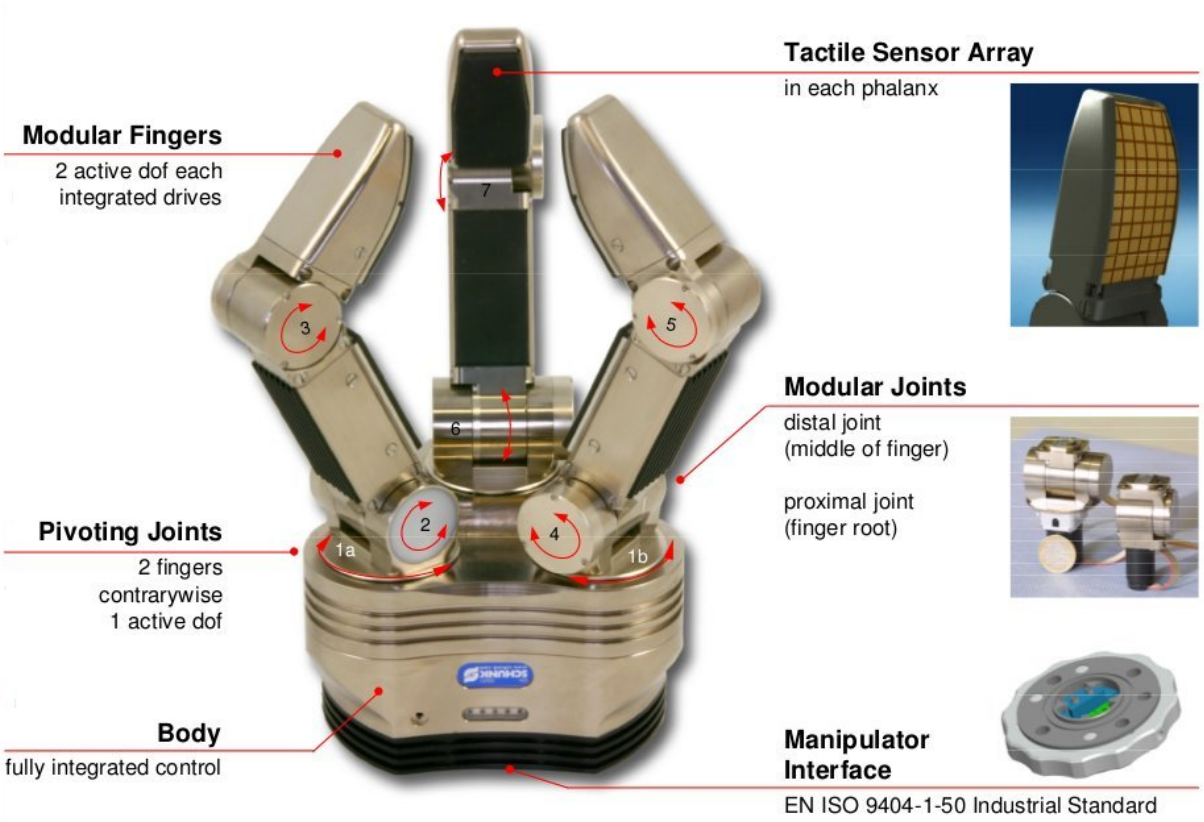


Figure 2.1: Schunk SDH-2 gripper

Figure2.1 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. Two of the fingers can be rotated along their vertical axis but they are connected contrariwise. That means if one finger rotates to the left, the other one is rotated to the right for the same angle, actually adding

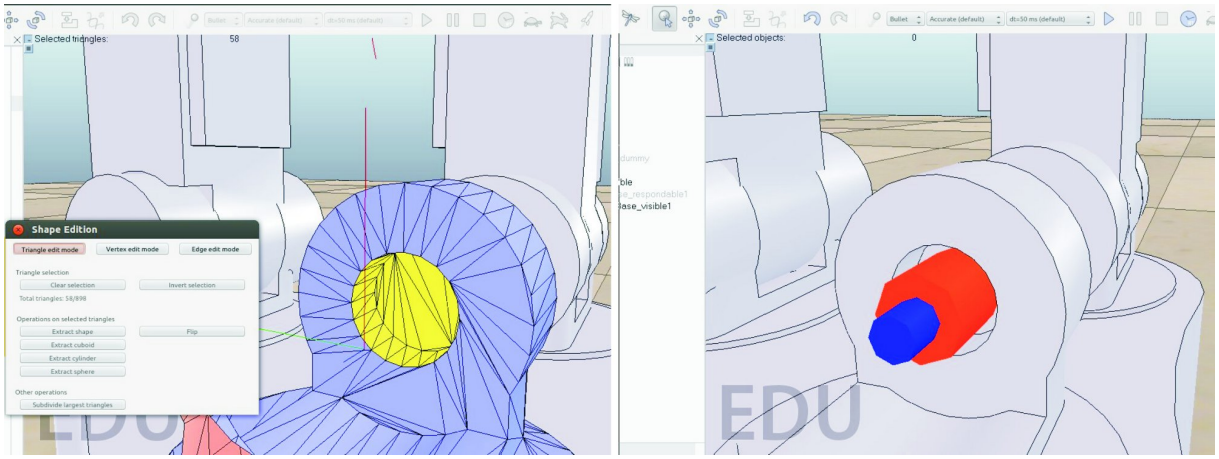


Figure 2.2: Placing a joint within the model

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 wrist, finger knuckles, finger links and finger tips. Suitable meshes were taken from the schunk_description[9] ROS package and imported into the V-Rep robot editor. They have then been arranged according to the technical description. The next step was to insert and arrange the gripper joints on their appropriate locations.

Therefore it was important to determine the correct position and orientation for each single joint within the model to allow the correct movement of the fingers. This was achieved by using the *shape edit mode* and select the cylinder shaped area within the mesh, where the joint has to fit. From that selection a cylinder was extracted and the joint was then centred within this newly created cylinder. Those steps had to be repeated for all 8 joints. The placement process can be seen in Figure2.2.

The left image shows the extraction on the target area. On the right image, the joint is already placed on it's appropriate location.

After placing all the joints and links within the scene, the model tree was adjusted to form

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

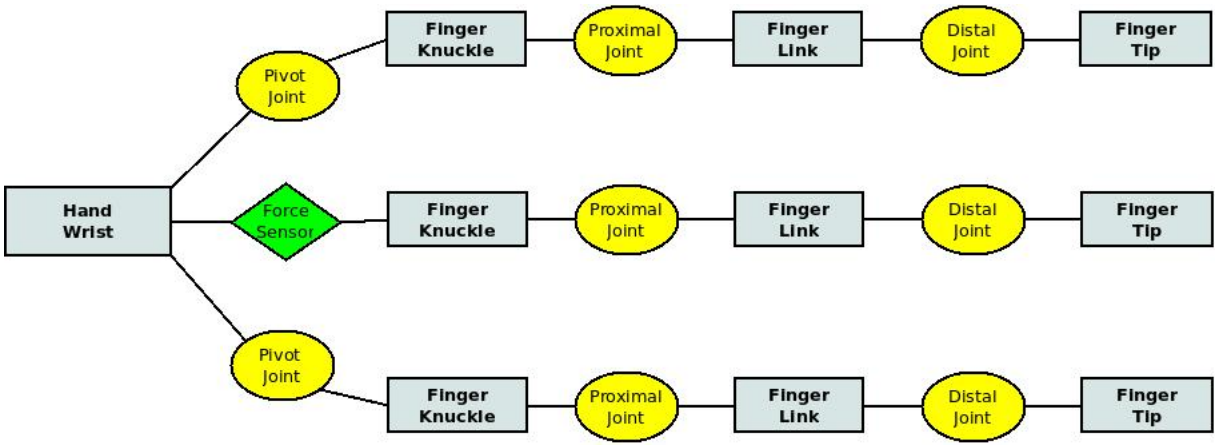


Figure 2.3: Kinematic chain of the Schunk gripper

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. Those parameters include the joint limits, maximum velocity and maximum effort. As the positions of the pivot joints are connected to each other, the configuration of the second finger's root joint looks sightly different. To achieve this mirroring behaviour the joint is operated in the *dependent mode*, which means it's position *depends* on the position of a connected joint and that dependency is expressed as *dependency equation*. The configured equation just copies the actual position but the joint is operated in opposite rotational direction.

The meshes only form the visual part of the model, but they are to complex to be used for dynamics calculations. So the shape of each link had to be approximated by groups of primitive shapes. This was achieved by executing the following steps for each single part of the model:

- Within shape edit mode locate parts of the mesh that could be approximated by a primitive shape (cuboid, cylinder), by selecting suitable groups of vertices
- Extract the primitive shape by using the corresponding editor functionality
- Repeat those steps until the most important parts of the link are approximated that way
- Group those primitive shapes to treat them as one single object
- Adjust the dynamic parameters (mass, material settings, inertial matrix)
- Adjust the local respondable mask
- Give the group the same name as the corresponding mesh, but with the *_res* suffix
- Remove the extracted shapes from the current visibility layer because they are just used for dynamics calculations

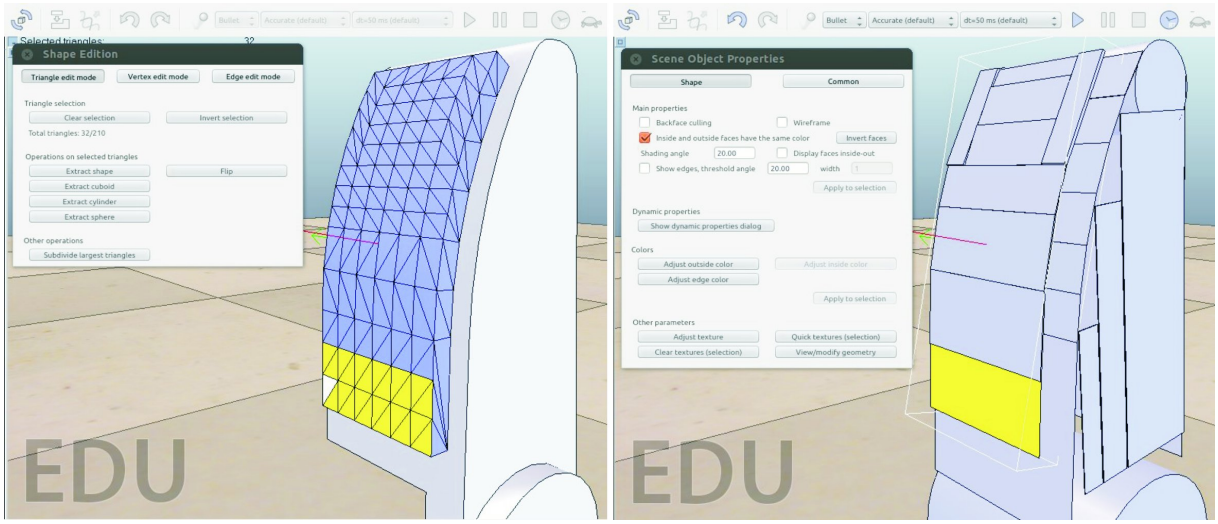


Figure 2.4: Process of approximating the original mesh with pure shapes

The extraction process is visualized in Figure2.4. Dynamic parameters like mass and inertial matrix have been provided by Alex Rietzler. The predefined *highFrictionMaterial* setting was used for each single part of the finger because that showed better results when picking up objects

later on.

The last step of the modelling process was the adjustment of the model hierarchy. Root element of the hand model is the respondable part of the wrist which is also the dedicated model base. 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 can be found in the corresponding chapter[10] of the V-Rep dodumentation). Each non-static and respondable shape has 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 is formed.

### 2.4.2 Assembling the scene

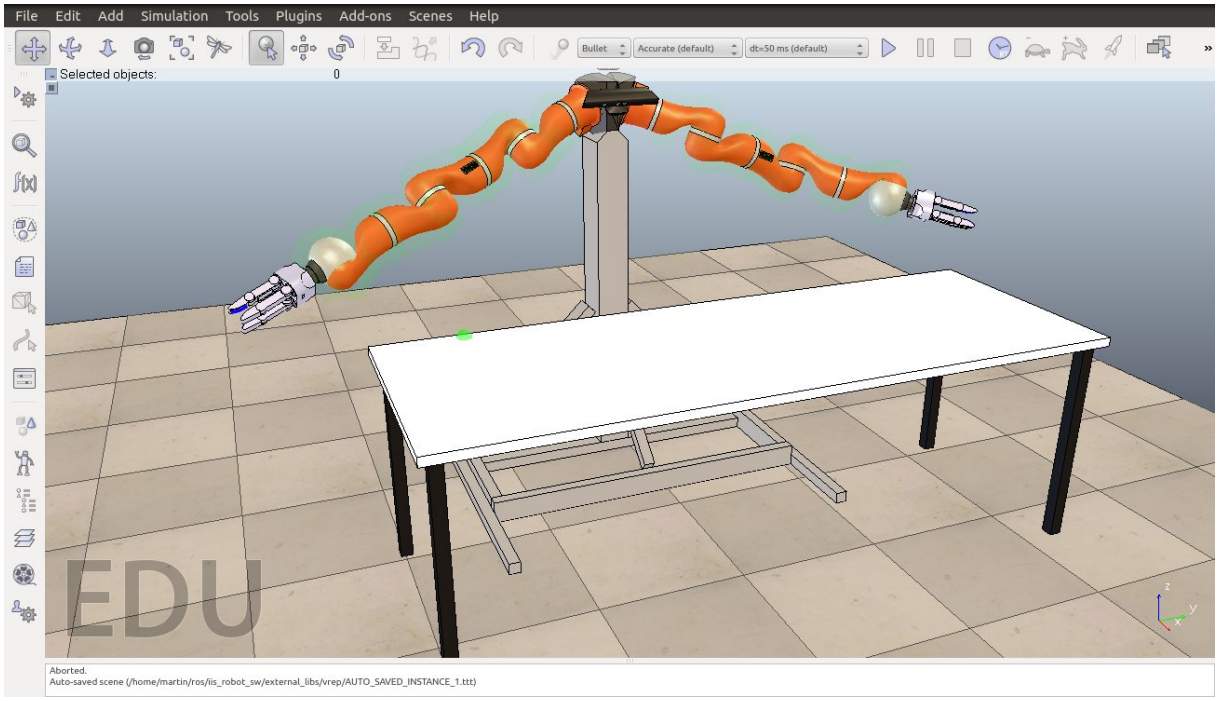


Figure 2.5: Structure of the V-Rep simulation scene

After finishing the gripper modelling process, each necessary component was available to build up the simulation scene as can be seen in Figure2.5. The final scene consists of a model of 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 calculation later on will happen relative to that dummy element. If it is necessary to move the origin to another location within the workspace, this can simply be achieved by just moving that dummy to the required location. The position and orientation of the torso, the table and the two arms are set relative to the world reference frame. (TODO: transformation was provided - how was it achieved?). 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

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

real counterpart and then adjusted accordingly.

The plate and the legs of the table were modelled as group of primitive cuboids. The table is defined as 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 *highFrictionMaterial*.

The Kinect camera model was also taken from the model browser. As it's position and orientation in the real world is not fixed and might change from time to time, it's position within the simulation scene is just an approximation to reflect the real world setting as good as possible.

### 2.4.3 Configuring the collision detection module

One of the requirements to the final solution is the ability to detect and visualize possible accidental collisions of the simulated robot with itself or it's environment. Moreover it would be a convenient feature to have some kind of warning if some of the robot's parts come dangerously close to an obstacle during movement. This could be achieved by creating a *collision shield*, which means an enlarged version of the robot and do additional collision checking. These considerations lead to two different types of possible 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* mean that a robot component directly hits another collidable object. This would be also a collision in the real world. The solution should be able to distinguish cleanly between those two types.

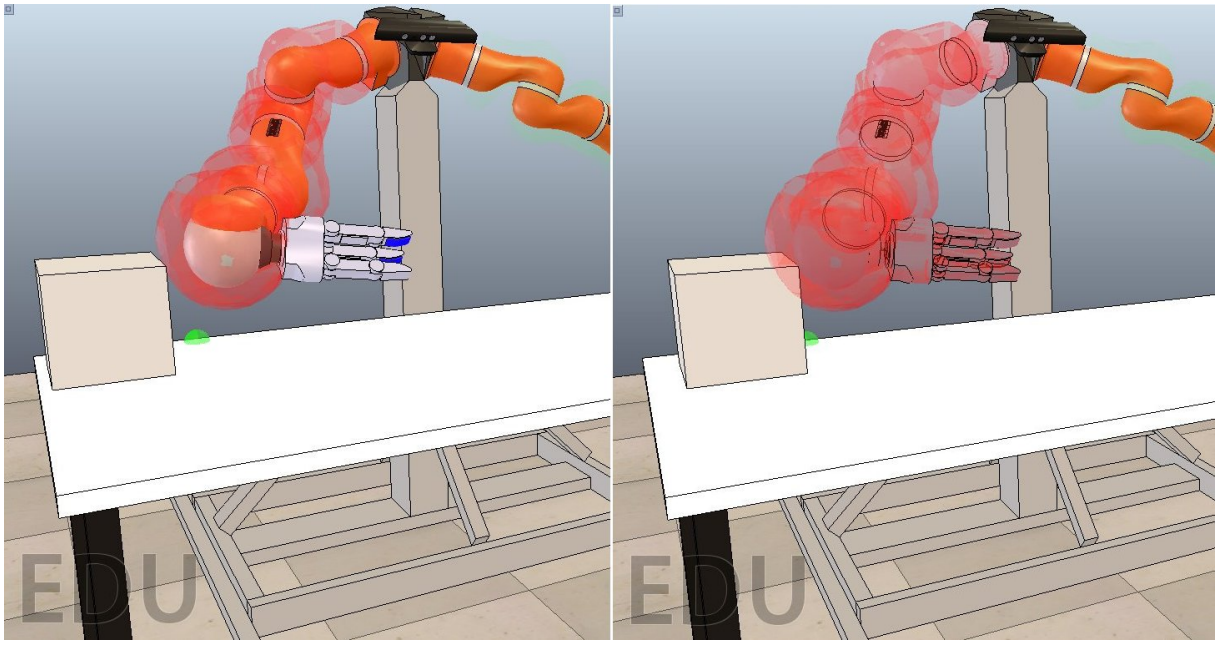


Figure 2.6: Collision detection and visualizing. The left image shows a hit on the collision shield, the right image indicates a direct hit.

That goal was achieved by using V-Rep's *collision detection module*[11]. This calculation

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

module is capable of detecting and visualizing collisions within the simulation scene by checking for interferences between *collidable* shapes. It is important to understand that the collision detection module only *detects* collisions. Producing *collision reactions* is the responsibility of the *dynamics module* and the chosen physics engine. Usually the shapes that form the visual part of the model are used for collision checking. The configuration is done by registering one or more *collision objects*. Each collision object consists of a *collider* and a *collidee*. Both of them can be single shapes or collections of shapes. The collidee settings also offer the option *all other collidable objects in the scene*. In that case, the collider is checked against all elements in $S = \{\ s \mid s \in scene\ and\ s \notin collider\ \}$. The big advantage in using collections is that they allow to exactly describe which shapes should be checked against which other ones. Detected collisions are visualized by applying different colouring either to the collider or to the collidee as can be seen inf Figure2.6.

The first step during realization was to model the collision shield. Therefore it was necessary to create an additional shape for each link that is slightly larger than the original one. This modelling process started at the second link of each robot arm and for all subsequent links the following steps were 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.
- Ungroup the resulting group of shapes and merge them into one single shape
- Grow the resulting shape, but only in x and y direction of it's own reference frame. The resulting mesh should be approximately 10cm larger but keep the same height.
- Adjust the outside color to make it green and nearly transparent. The collision shield should be visible but not occlude the original model.
- 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.
- Make the new shape a sibling of the original one within the model hierarchy.
- Adjust the shape object properties. Define the new shape to be *static* and *non-respondable*.
- Disable the *collidable* flag on the shape. This behaviour will be overridden in the collection settings later on when configuring the collision detection module.

This modelling process is visualized in Fig???.

The next step was to adjust the configuration of the *collision detection module*. Each arm requires two *collision objects*, defined as stated in Figure2.7. The first two collision objects are designed to detect direct hits on the corresponding arm. Therefore 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, which means that they are excluded from collision checking by default. The collision objects, named with the *Shield* suffix are responsible to check for hits solely on the collision shield elements. The colliders (collections *leftArmShield* and *rightArmShield*)

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

Figure 2.7: 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* $\}$ |

Figure 2.8: Collection definitions

consist only of the collision shield elements. As those elements were defined without the *collidable* flag, the option *Collection overrides collidable properties* is selected in the corresponding settings to explicitly enforce collision checking when using those collections. The collections, defining the collidees (*exLeftArmShield* and *exRightArmShield*) include all other objects except those, contained in the left/right arm's subtree. This exclusion is necessary because otherwise those collision objects would detect collisions between the shield elements and the other arm links. The collection definitions can be seen in Figure2.8. All collision objects are defined not to be handled explicitly. This means that V-Rep does not check them automatically on each simulation pass. It has to be done manually 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 needs the ability to set arm target positions in *joint space* or 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 joint. Setting targets in Cartesian space requires to solve the inverse kinematics problem for the corresponding robot component. Therefore V-Rep offers the *inverse kinematics calculation module* [12]. This calculation module allows to define and register various *IK groups*. Each IK group has to contain at least one *IK element*, defining the kinematic chain, constraints and desired precision settings (linear and angular). IK elements can be configured to enforce position constraints and/or orientation constraints for each single axis. The kinematic chain is specified by selecting the dedicated base link and the tip. The tip is a dummy object, indicating the end effector reference frame. The IK target is defined by another dummy object. Tip and target have to be linked, forming a *IK, tip-target* connection by selecting the appropriate link type within the dummy object settings. Figure2.9 shows a schematic description of that concept. The target pose is set by placing the target dummy at the desired location. The IK calculation model will then adjust the joint positions until the tip pose matches the target pose, respecting joint limits and configured constraints and tolerance values. The joints within the specified kinematic chain need to be operated in *inverse kinematics mode*, otherwise the module is not able to control them.

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

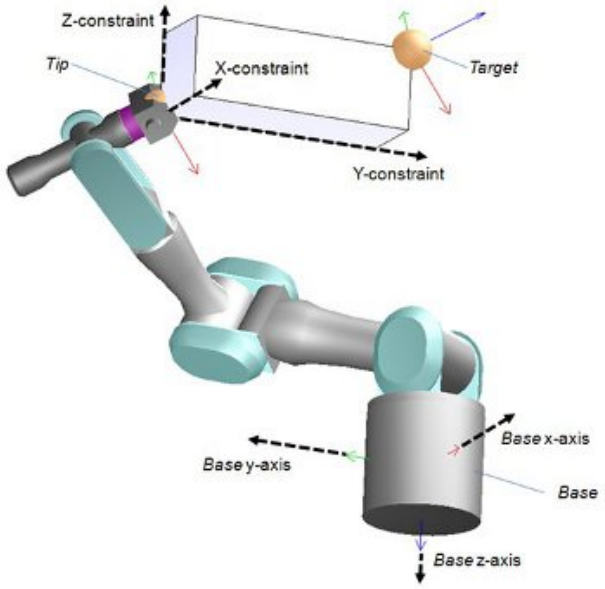


Figure 2.9: IK calculation module concept

The IK calculation module allows to define various IK groups for each robot component with differing configurations. Along the collection of 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.

| **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 |

Figure 2.10: IK group definitions

Three IK groups have been created for each arm. The configuration settings are listed in Figure2.10. They were chosen, following the guidelines from the V-Rep documentation. 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 less performant 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 called 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 Implementation of the ROS control interface

### 2.5.1 Overview

After finishing the modelling process it was necessary to find a proper way to control the robot components via a ROS control interface. The arm as well as the hand is controlled via a set of inbound and outbound ROS topics that allow to send commands to the underlying component or to receive state data from the component(joint states, sensor data,...). Each simulated component should provide exactly the same interface as it's real counterpart.

One problem is how each type of component can be clearly identified within the current simulation scene. A scene is a hierarchy of various scene objects, organized in a tree structure. As already explained, those objects can be shapes, joints, sensors or even only dummies. The scene content can be modified by the user. Maybe a gripper gets replaced by another component, an arm gets removed or an additional Kinect camera gets installed. The required solution should be able to react to changes in the current scene. Parts of the model hierarchy should be clearly identifiable as a specific simulation component. Each single part of a component should be identifiable (joints, sensors, dummies, IK groups, collision objects). Luckily V-Rep provides various extension points for programmers and is therefore highly customizable.

After some investigation about the possibilities the decision was made to create a simulator plugin, using the V-Rep regular API. This approach states the most flexible solution as this API provides more than 400 functions. 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. This means that 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 functions:

- v_RepStart - called on startup and can handle some initialization - v_RepEnd - called before shutdown and can do some cleanup - v_RepMessage - called very often during the whole V-Rep lifecycle and is therefore a very performance critical method. Via this function V-Rep notifies the plugins about events like start/end of simulation, simulation step, scene content change, scene switch, .... The plugin code can react to those events accordingly.

### 2.5.2 Plugin Architecture

The plugin code is organized as can be seen in Figure2.11.

- SimulationComponent A simulation component is a single, reusable part of the robot that can be used in different environments. Each component provides it's own clearly defined control interface. A simulation scene can contain various types of components. 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 are introduced it is necessary to create a subclass of SimulationComponent and provide implementations for the abstract methods. Each SimulationComponent consists of two parts. The first one is a class that provides access to a concrete simulation component instance. The LWRArm class for example stands for a KUKA LWR4+ arm model in the scene. This class provides full access to the functionality of the underlying arm model.

The second part is a controller class for the component instance. This controller encapsulates the whole ROS interface to the simulation component and has to be a subclass of the abstract ComponentController class. On each simulation pass the controller publishes all the necessary state data to it's various topics and sends incoming commands to the simulation component. The names of the various provided topics are composed from the overall namespace, the defined

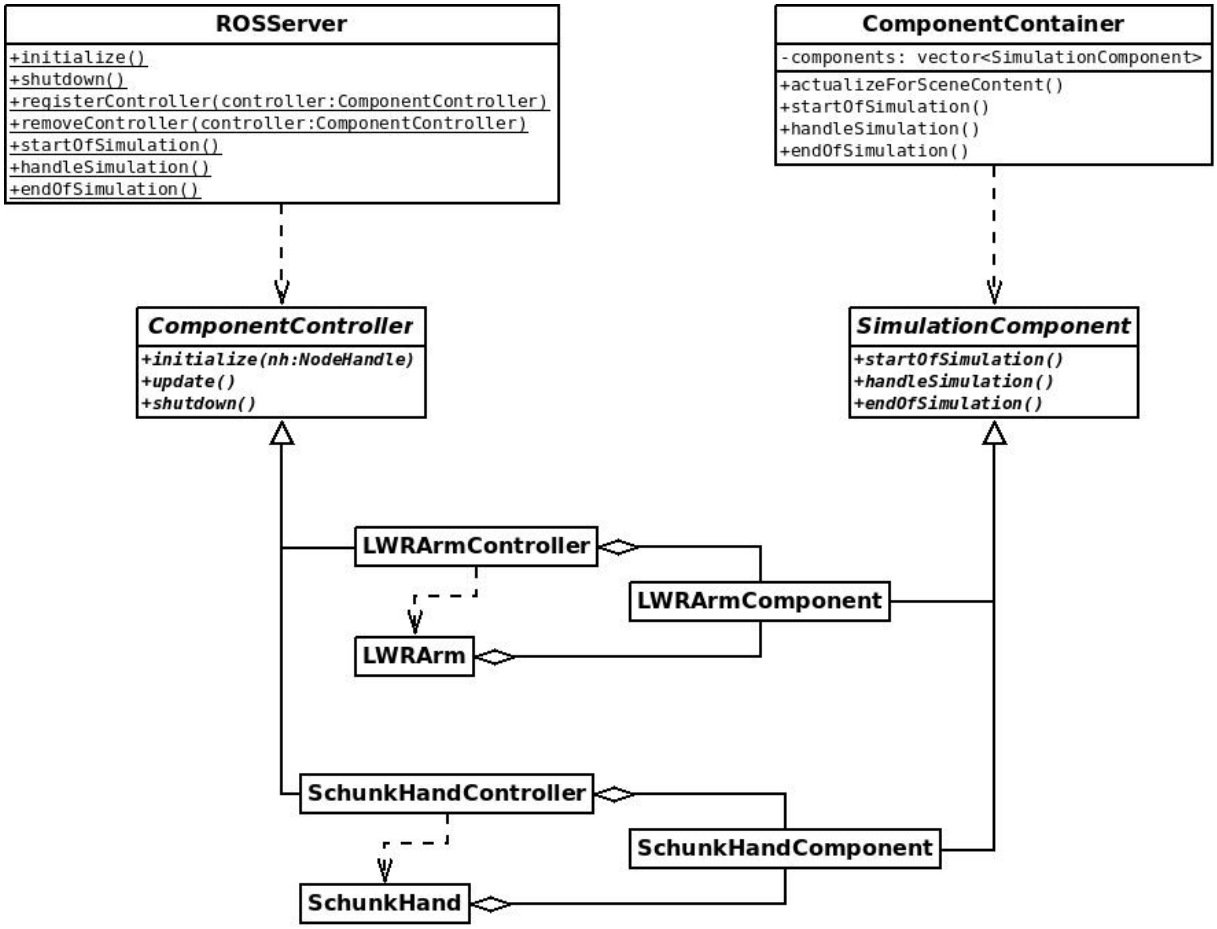


Figure 2.11: Simulator plugin architecture

unique name of the component and the actual topic name. For example the joint control topic of the right robot arm evaluates to '/simulation/right_arm/joint_control/move'

On simulation start each SimulationComponent registers it's ComponentController at the ROSServer and unregisters it on simulation end.

- ComponentContainer This class represents the set of all identified simulation components in the current simulation scene. On V-Rep startup an instance of ComponentContainer is created. Each time, the content of the current simulation scene changes, the method 'ComponentContainer::actualizeForSceneContent' is triggered. This method performs the following steps: - It validates each currently registered SimulationComponent instance if it is 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. To identify a component it has to be marked by using V-Rep's custom developer data functionality. Details are explained further on.

The ComponentContainer gets notified about each simulation step. It then simply forwards that message to all registered components. Those can then perform all necessary steps like triggering collision checking or the IK calculation module.

- ROSServer The ROSServer is a static class that encapsulates all ROS related functionality. It tries to initialize ROS on startup. If the connection to the master can be established it creates a ROS NodeHandle for the 'simulation' namespace. Otherwise it forces the plugin to unload, because it is not able to work without a running roscore. Each SimulationComponent registers

it's ComponentController instance at the ROSServer. On simulation start it initializes the 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 forces the registered controllers to shutdown their publishers and subscribers.

- ComponentController This is the abstract base class for all controllers. A ComponentController gets initialized by the ROSServer 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. On simulation end the shutdown method is called by the ROSServer, forcing the controller to shutdown all publishers and subscribers.

- LWRArm This class provides access to a correctly configured LWR arm model within the simulation scene. To fullfill the requirements for the control interface, the arm has to be able to operate in joint control mode and in inverse kinematics mode. Initially the arm starts in joint control mode. All the joints are switched to torque/force mode and accept target positions to be set. The simulated PID controllers will try to move the joints to their designated target positions.

Switching to IK mode means to operate all the joints in IK mode and use the previously configured IK calculation module. Setting a target pose in Cartesian space means to bring the IK target dummy into the required pose. The IK calculation module tries then to bring the linked IK tip dummy into the same pose by commanding the robot arm's joints accordingly and satisfying the configured constraints and precision settings. At the moment, three IK groups are configured for each arm with different settings to achieve performant and stable solutions. Those groups are sequentially called until one of them is successful. The configuration of the first group focuses on performance but provides less stability. The last group uses a configuration that is slower but provides more stability in positions closed to singularities. If none of the groups was successful it will result in an error message on the console, otherwise the arm will start or continue to move towards it's target pose. When initializing the LWRArmComponent, it will search for IK groups that are named the same as the arm and with consecutive numbering (left_arm, left_arm1, left_arm2,...). That allows to reconfigure the IK calculation module and introduce additional IK groups without touching the plugin code. It is expected that at least one IK group is configured, otherwise an error message is written to the console.

The collision status of the arm is determined by using the configured collision detection module. On each simulation step the collision group 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 group at all, because a direct hit always implies a hit with the shield as well. Only if no direct hit was detected, the second group is handled. The outcome can be queried as the current collision state of the arm. During initialization it is searched for a collision group that has the same name as the arm, responsible for direct hit detection and a group that is named with the naming pattern [arm_nameShield], responsible for shield hit detection. If one or both of the required groups cannot be detected, an error message on the console will be stated and the collision detection functionality will not work as expected. Collision state is evaluated on each simulation time step.

Additional items that have to be identified within the model tree are the 7 joints, the end effector tip dummy, the IK target dummy and the force sensor on the last link of the arm. The origin of the reference frame can be defined by creating a dummy object within the scene with the special name 'ref_frame_origin'. If such a scene object is found, all Cartesian poses are taken with respect to the reference frame of that object, otherwise the poses are interpreted absolute within the world reference frame. All the necessary data that is published by the controller can

be accessed via this class.

- LWRArmController The LWRArmController provides the implementation of the Kukie interface. This interface is created by Simon Hangl especially for the KUKA arms in the IIS Lab.

### 2.5.3 Identifying simulation components in the scene

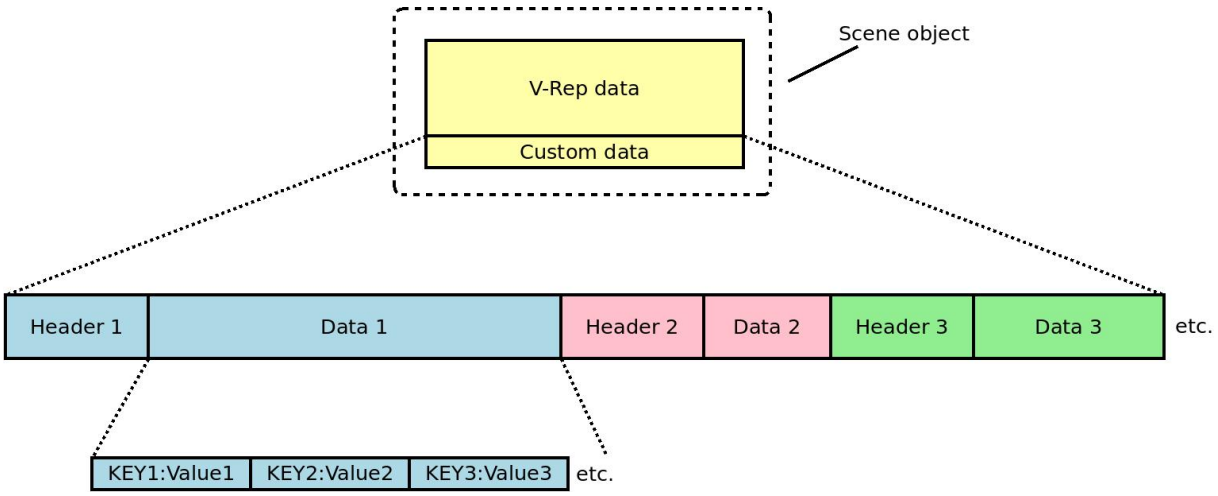


Figure 2.12: Custom developer data segments on scene object

As each scene is a hierarchy of various types of scene objects there had to be found a way how to identify subtrees within this hierarchy that belong to known simulation components and should therefore be handeled by the plugin. One way would be to give each part a clearly defined, unique name and then search for those names within the scene. But than it would be necessary to hardcode the name of each single joint name and this is not a preferable solution for this problem. If somebody accidentally changes a name in 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.(TODO: show image of data segments) This data gets 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. The second element is an integer value that holds the length of the data segment and then comes the data itself. The format of the data can freely be chosen by the developer. It was decided to use string representations of key/value pairs, seperated by a colon (:). The key uniquely identifies the type of component (arm joint, hand joint, IK target dummy,...). The value segment can be used to provide additional data like for example the name of a joint. The left arm's model base for example is tagged with '2497,1:left_arm'. This tag data item identifies that element as the model base of a LWRArmComponent with the name 'left_arm'. 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 with the object handle of the underlying scene object. During the initialization, the concrete SimulationComponent instance searches then the model subtree for all the necessary parts (joints, dummies, force sensors...). The implementations for the arm and the hand model provide feedback output on the console window about the success of this initialization process.

### 2.5.4 Creating startup launch file

Describe startup script, environmental variable (VREP_PACKAGE_PATH), launch file, how to use different simulation scenes.

### 2.5.5 Documentation and usage examples

Detailed documentation and implementation details should go into the Appendix

# Chapter 3

# Pick and place

The implementation of a benchmark pick and place task, executable on simulator and real robot as well, states one of the objective targets of this project. The implementation heavily uses major parts of MoveIt's planning functionality. The overview at the beginning of this chapter gives some general information about pick and place tasks, explaining the process step by step and describing the single stages that have to be performed. The second part focuses on the pick and place functionality of MoveIt and discusses the involved action servers, topics and messages. The third section explains the implementation of the benchmark pick and place task in greater detail and demonstrates how that functionality of MoveIt was used. The last section describes some observations that have been made during the implementation process.

## 3.1 Overview

A pick and place task is the process of grasping an object, lifting it and dropping it somewhere else. Humans do that permanently, without even think about it. But when teaching a robot to perform a pick and place action, it shows how difficult and complex this task is and how much planning is involved to achieve the desired result. The planner requires exact knowledge about the robot and it's environment, including the objects to grasp and possible obstacles. Accidental collisions have to be avoided but other collisions are mandatory 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 during subsequent planning requests. That means that it must be attached to the manipulator temporarily and removed, after releasing the object. As long as the object is part of the robot, it possibly increases the size of the end effector.

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, the grasped object takes during the operation. For example when carrying a glass, filled with liquid it has to stay 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 *orientation constraint* which is a special type of path constraint. The planning solution has to provide mechanisms to define end enforce such types of *path constraints*.

Pick and place tasks can be split into two independent phases, each one composed from a number of trajectory stages:

- **Pickup phase**

  This phase starts at an arbitrary robot configuration. In the first stage, the manipulator has to be brought into a position closed to the object to grasp, but in a distance that allows the gripper to open safely without touching the object. This position is called the *pre-grasp pose*. The next stage is to set the gripper into *pre-grasp posture*. That means, bring it's fingers into an open configuration that allows to completely enclose the object (or at least that part that is used to clutch it) after approaching towards the final *pose*. How this configuration looks like depends on the shape of the object to grasp and the structure of the gripper. Further on, the gripper has to be moved towards the *grasp pose* using the correct approach direction. This is the place where the robot gets in contact with the object. The gripper moves it's 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. At that point the object has to be attached to the gripper and further on treated like an additional link of the robot although still being in collision with the support surface. The pickup phase completes after lifting the object along the retreat direction. The object is now part of the robot and no collisions should be tolerated any more. Figure 3.1 shows the stages of the pickup phase.

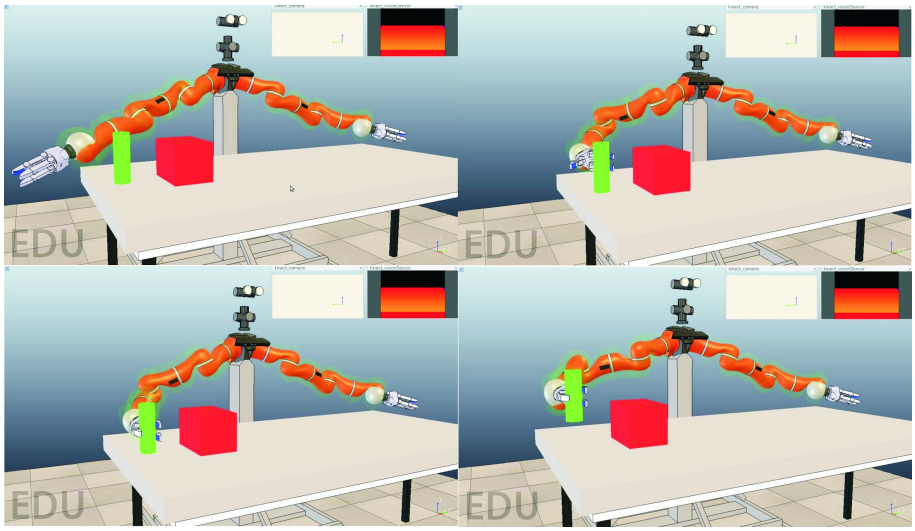


Figure 3.1: Stages of the pickup phase in the simulator

- **Placement phase**

  The placement phase starts after a successful pickup. The grasped object is enclosed by the gripper and considered to be part of the robot. Now the manipulator moves towards the *pre-place location* - the place where the final approach towards the goal starts. The easy way is to just drop the object at that point. In that case, the placement phase completes after bringing the gripper fingers into the *post-place posture* (opening it) and detaching the object from the robot.

If the object should to be placed carefully, the gripper has to approach from the *pre-place location* towards the final *place location* along the specified approach direction. Here has to be considered that the object will get into contact with the support surface again when it's final position is reached. At that stage the gripper can open and the object has to be detached from the robot. From that point, the object needs to be treated as obstacle again which means the planner is forced to avoid collisions with it. The placement phase completes after the manipulator has moved away from the object along the specified retreat direction. The stages of the placement phase can be seen in Figure 3.2.

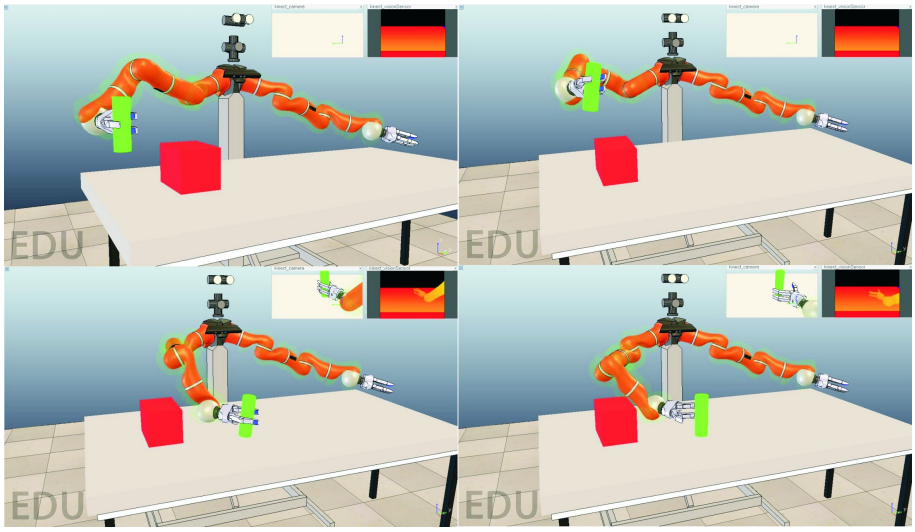

Figure 3.2: Stages of the placement phase in the simulator

Both phases of the task are only considered to be complete if each single stage has successfully been executed. The necessary planning parameters like pre-grasp and grasp pose, gripper postures and approach and retreat directions are usually provided by a *grasp planner*. This is an additional node within the planning pipeline that identifies objects in the environment of the robot, usually based on 3D sensor data and calculates the necessary grasp parameters. Explaining the functionality of grasp planners is far beyond the scope of this project though the section about implementing the reference task discusses the used parameters in greater detail and shows what would be usually delivered by the grasp planner.

## 3.2 Pick and place tasks in MoveIt

During the various stages of the pickup and placement phases a lot of motion planning is required. Each single stage requires to plan a trajectory that is free of accidental collisions while respecting the limits of the robot and enforcing possible additional constraints. Each phase can only be considered executable if a valid motion plan for each single stage exists. Planning all the subsequent stages one after the other would be a cumbersome task. Therefore MoveIt provides a set of messages and action servers that greatly simplify those planning tasks.

The *PickupAction* server handles the planning and optionally execution of all required stages during the pickup phase at once. Pickup requests are done, using a *PickupAction* client to send *PickupGoal* messages to the server. The request message is composed of all the parameters that are necessary to completely describe the planning problem. Picking up an object can always be done in several ways. Depending on the shape of the object there hardly always exist a lot

of possible poses where the gripper can safely approach and grasp. Therefore MoveIt allows to provide a set of possible grasp definitions for a pickup request. There is also a quality parameter that can be used to tell the planner, how 'good' a specific grasp is compared to other ones. MoveIt can then favour grasps with higher quality if several valid solutions are found by the planner. Providing a number of different grasps increases the probability for the request to be successful. The grasp definition is composed of the pre- and post grasp postures for the gripper, the final grasp pose and the approach and retreat vectors. As MoveIt needs to know which object should be picked within the planning scene, it is also necessary to include the ID of the object to grasp.

The *PlaceAction* server is responsible for planning the whole placement phase. Therefore same concept is used as above - a set of possible place locations can be provided to increase the probability of a successful planning attempt. The parameters that are used to describe a place location include the post-place posture of the gripper along with it's target pose, and vectors, describing the pre-place approach and the post-place retreat.

Pickup and placement requests as well require a number of additional parameters that are explained in greater detail within the section about benchmark task implementation.

Objects contained in the task environment have to be brought to MoveIt's attention. This can be done either by manually adding them to the planning scene, using the corresponding topics or by configuring MoveIt to be aware of sensor data. As the integration of depth information from the Kinect camera did not work stable during the evaluation phase, that possibility is not covered within this thesis and all involved objects were added manually.

Pickup- and place action servers provide the ability to choose whether to execute motion plans immediately or to do just the planning and return the outcome. In that case the execution has to be handled manually later on. The first method is much more comfortable as MoveIt executes the trajectories and also handles additional requirements like attaching and detaching the grasped object in time. The drawback is that also possibly weird trajectories get executed immediately as they might be valid solutions for the given planning problem though they are obviously unsuitable. So an additional safety mechanism needs to be introduced that allows to interrupt the execution when facing any problems. The second 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.

## 3.3 Implementation of the benchmark task

This section describes the implementation of the benchmark pick and place task in greater detail. The source code can be found within the 'uibk_moveit_tests' package. The workspace is the table in front of the robot covered with a 9cm thick foam mat. This mat will be declared as the support surface later on. The object to grasp is a cylinder with 4cm radius and a height of 25cm. This corresponds to the size of a usual SIGG bottle which can be used to run the benchmark task in the real world. The cylinder is located on a fixed, known position. There is also a box shaped object located within the workspace that acts as an additional obstacle. 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 MoveIt's pick and place functionality. The necessary steps

are explained in the following subsections.

### 3.3.1 Creating the environment

As MoveIt is currently not configured to use sensor data, it has to be notified about the task environment. The table and the surface mat are already part of the URDF description but the cylinder and the obstacle have to be added to the task environment. This is done utilizing the `planning_interface::PlanningSceneInterface` class that provides functionality to manipulate the current planning scene. The same effects can be achieved by manually publishing to the '/collision_object' topic but using the convenience class saves a lot of boilerplate code. The *CollisionObject* type is used to describe those objects. Both of them are primitive shapes. Necessary parameters are the shape type, dimensions and the target pose. Additionally each *CollisionObject* needs a unique ID which is used to identify the shape within the planning scene. After adding the *CollisionObjects* to the collision world, they are visualized in RViz. The image in Figure3.3 shows the task environment after adding the *CollisionObjects*.

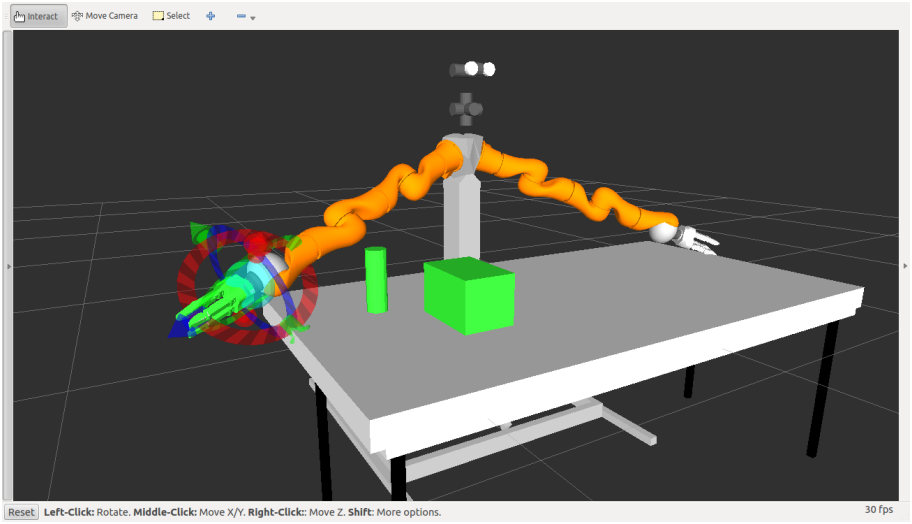


Figure 3.3: Planning scene after inserting the collision objects

### 3.3.2 Generating possible grasps

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 parameter and calculates 10 possible grasp poses along a semi circle around the cylinder location. 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. The pre-grasp approach and the post-grasp retreat are defined as *GripperTranslations*. 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. Experiments showed that the success rate is much better if the grasp parameters allow some flexibility to the planners. The approach vector depends on the grasp pose and points along the z-axis of the end effector frame towards the object. The desired distance between pre-grasp and grasp pose is set to 20cm while the minimum

distance is 10cm. The gripper retreat vector points up, along the z-axis of the world. Desired and maximum distances are also set to 20cm respectively 10cm. As no one of the generated grasps should be favoured among the others, the grasp_quality parameter of each grasp is set to 1. This means that they are equal in quality. The last necessary parameter is a unique identifier for each grasp. As usually a lot of different grasps are provided for each pickup request, this ID can be used to identify, which one was finally used to solve the planning problem.

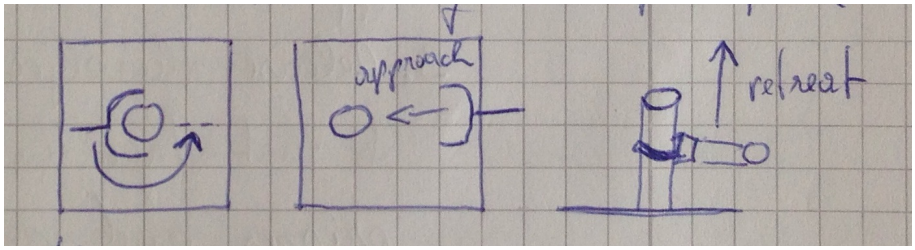


Figure 3.4: Grasp poses, gripper approach and retreat

### 3.3.3 Planning and executing the pickup

After generating the set of grasps, the planning request can be sent to the pickup action server. As there is a lot of boilerplate code necessary, a reusable helper class was created that can be used to simplify all kinds of planning requests. The utility class is called *PlanningHelper* and can be found within the 'uibk_planning_node' package. A call to the pickup action server is done by using the 'plan_pick' method of the helper class. This method takes a set of possible grasps and the ID of the object to pick as parameters. The outcome is a pointer to an instance of *PlanningResult*, a structure that contains all the necessary information about a planning attempt. Success or failure is indicated by the 'status' parameter. On success, the parameter 'trajectory_stages' holds a vector, containing the resulting trajectory stages. The actual call to the pickup action server is done by defining the *PickupGoal*, using the predefined grasps. Additional parameters are the ID of the object to pick, the name of the chosen planning group and the name of the link within the robot model that acts as the support surface. Optional parameters are among others the ID of the planner to use and the maximum allowed planning time. The parameter 'plan_only' within the planning options is set to true to avoid the immediate execution of the planned trajectories. This allows a visual verification of the planning outcome before execution. The resulting robot path is shown in RViz. If the solution is satisfying it can be executed, passing the PlanningResult to the corresponding method of the PlanningHelper class. This method uses the '/execute_kinematic_path' service provided by the 'move_group' node. The service sends a given trajectory to the responsible controller and provides feedback information about the execution status. The 'PlanningHelper' also takes care to attach the picked object to the gripper after the grasp stage. The pickup phase completes after successful execution of all trajectory stages.

### 3.3.4 Planning and executing the placement

The placement phase is planned and executed in a somehow similar manor. 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. This again gives some freedom to the planner as it can choose which one to use (TODO: provide image that shows the principle). The final approach towards the goal location is specified as *GripperTranslation*. The direction vector points down

along the z-axis of the world. Desired and minimum distances are set to 20cm respectively 10cm, allowing some flexibility during that stage. As post-place posture for the gripper, the same configuration is used as for the pre-grasp posture. The last required parameter for a place location is the *GripperTranslation* that describes the retreat after releasing the object at the goal location. The direction depends on the chosen orientation and points towards the negative z-axis of the gripper reference frame.

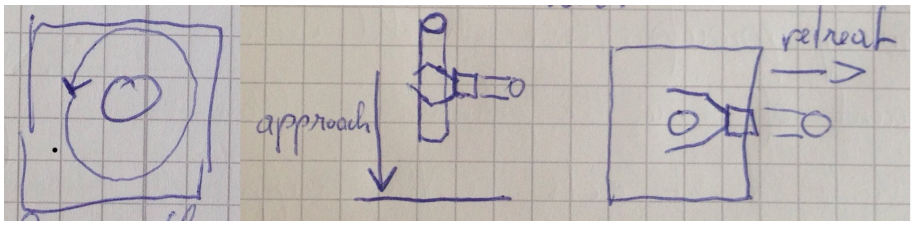


Figure 3.5: Place locations, pre-place approach and post-place retreat

The call to the place action server is again handled by the formerly mentioned planning helper class. The 'plan_place' method takes a vector of the predefined place locations and the ID of the object to place as input parameters. The actual call to the place action server is done, defining a *PlaceGoal* message. The most important parameters are the ID of the object to place, the name of the planning group and the set of possible place locations. Optionally can be specified which planner to choose and also the maximum allowed planning time. The method returns the result of the planning request. On success, the resulting trajectories are again visualized in RViz and can then be executed the same way as during pickup phase. After the execution, the task is considered to be complete and the picked object is detached from the gripper and again part of the collision world.

## 3.4 Executing the benchmark task

The benchmark task is designed to run on the simulator and the real robot as well. The sample program only depends on a running 'move_group' instance. The easiest way to run the sample is to start MoveIt in demo mode (demo.launch) and then execute the sample code, using.

```
rosrun uibk_moveit_tests sample_pick_place
```

This demonstrates the functionality but only executes the trajectories on the fake controllers provided by MoveIt. If the sample should be executed on the simulator or the real robot, the corresponding namespace has to be specified before running the code. This can be done by setting the 'ROS_NAMESPACE' environment variable to either 'simulation' or 'real', within the terminal. For example

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

runs the benchmark task on the simulator. Of course it is necessary to start the simulator and launch the corresponding MoveIt instance before doing that. 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 also is the visualized as well and execution needs confirmation again. The program exits after successful completing the placement phase.

## 3.5 Observations

This section gives an overview about 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. Major points are the amount of different grasps or place locations and the allowed range within the defined gripper translations. 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 the high complexity of the planning problem. Enforcing path constraints requires much more allowed planning time and a very fast IK solver because of the large number of necessary IK requests. Maybe a faster IK solution than the one available could help to solve the problem but that is not guaranteed. Therefore there are no path constraints used within the sample task.
- Planning requests often fail due some MoveIt-internal issues and not because planning is 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 successful request breaks the loop and continues the execution flow.
- Resulting trajectories should always be visualized in RViz before confirming the execution. The calculated motion plans might be valid but sometimes they are a bit confused and therefore unsuitable. Moreover, the planner can only take into account what it knows about the robot's environment. Especially in the robot lab there is equipment mounted in the area above the robot, that is not taken into account during planning. Therefore the visual validation is necessary to avoid damages on the robot and it's environment.

# Bibliography

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.

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

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.