Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

FAKULTÄT
FÜR MATHEMATIK, INFORMATIK
UND NATURWISSENSCHAFTEN

Bachelor Thesis

Multitouch Robot Control

Merlin Steuer

2steuer@informatik.uni-hamburg.de
Studiengang Informatik
Matr.-Nr. 6415125
Fachsemester 10

Erstgutachter: Dr. Norman Hendrich
Zweitgutachter: Dennis Krupke

Abgabe: 23.04.2018

Mein Dank gilt all denen, die mich hierbei unterstützt haben – insbesondere Sven, meinen Eltern und Anna-Lia. Danke. – *Merlin*

Abstract

Controlling robots with a high number of degrees-of-freedom is a big challenge in robotics. Dexterous robot hands give robots the ability to generically grasp objects while potentially being able to grasp a greater variety of things than a human could (e.g. by attaching special sticking fingertips). Having a robot designed like the human hand might give users the ability to control the robot more intuitively. The objective of this thesis is to implement a control interface for a Shadow C5 robot hand with 5 Fingers and 20 degrees-of-freedom and a robotic arm with seven degrees-of-freedom on an ubiquitous tablet computer with a 10 inch screen running the Android® operating system. To communicate with the robot, the *Robot Operating System* (ROS) is used while inverse kinematics are done using *BioIK*[22], an IK solver developed at the TAMS group at the University of Hamburg.

Zusammenfassung

Roboter mit einer hohen Zahl an Freiheitsgraden zu steuern stellt eine große Herausforderung dar. Der menschlichen Hand nachempfundene Roboterhände könnten Robotern die Möglichkeit geben, universell Objekte auch vorab unbekannter Beschaffenheit zu greifen. Hierbei kann die Anzahl der greifbaren Objekte bspw. durch Anbringen geeigneter Instrumente (z.B. klebriger Fingerspitzen) größer sein als die des Menschen. Weiterhin könnte ein Roboter, welcher menschlichen Extremitäten nachempfunden ist durch einen Bediener intuitiver zu nutzen sein. Ziel dieser Bachelorarbeit ist es, eine Multitouch-Schnittstelle zu einem Roboter bestehend aus einer *Shadow C5* Roboterhand mit 20 Freiheitsgraden und einem Roboterarm mit sieben Freiheitsgraden zu entwickeln. Die Entwicklung findet auf einem Android®-Tablet mit einer Bildschirmdiagonale von 10 Zoll statt. Zur Kommunikation mit dem Roboter wird das *Robot Operating System* (ROS) eingesetzt. Um Probleme der inversen Kinematik (IK) zu lösen kommt *BioIK*[22] zum Einsatz, ein Algorithmus, welcher im Arbeitsbreich TAMS der Universität Hamburg entwickelt wurde.

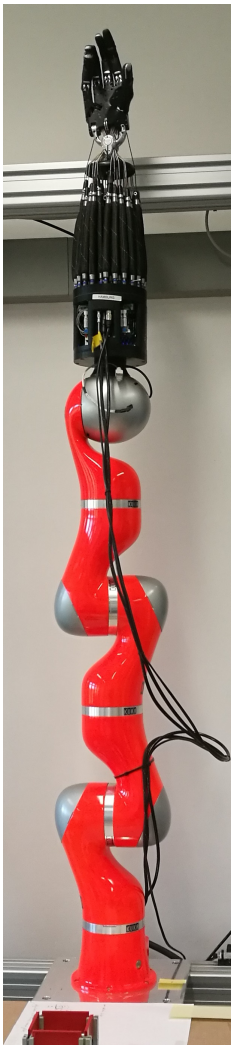
Contents

1 Introduction

1.1 Motivation

Controlling dexterous robot hands is a big challenge of robotics, but their usage has a variety of obvious advantages: The similarity to a human hand enables it to grasp objects in nearly all positions and poses the real human hand could. Especially for complex manipulation tasks, where a simple robotic grasper with just a pair of pliers is not sufficient, the larger amount of degrees-of-freedom comes into action. Also, users might be able to better plan actions when they are controlling a device similar to their own hands, meaning the main task for them is to use a control interface to execute actions they would otherwise execute with their own hands. Additionally, the robotic hand potentially have advantages over the human hand, such as, higher degrees-of-freedom, more strength or special fingertips adding more friction and by that enabling grasping a wider variety (e.g. very sleek) different materials. This would give users the ability to perform tasks as they would with their own hands but with less effort or more effective.

Figure 1.1: *KuKa LWR Arm with Shadow C5 Hand installation*



1.2 Objectives

Within this thesis, a touch-interface for controlling such robotic hands shall be developed, taking advantage of the multi-touch capabilities of modern mobile tablet computers. The user shall be able to control the position of the robotic hand (using the connected robotic arm) and grasp objects with it. All actions shall be mapped to corresponding multi-touch gestures the user can easily understand and learn.

The hardware used within this bachelor thesis is a *Shadow Dexterous Hand C5/C6* by the Shadow Robot Company. It has five fingers controlled by electrical or pneumatic muscles using 20 degrees-of-freedom[6]. The hand is connected to a robotic arm (*KUKA Lightweight Robot*) allowing it to also be moved in space (See Figure 1.1 for an image of the installation).

As a control device an off-the-shelf android tablet will be used, as these devices have become very widespread and - thanks to this - relatively affordable. With screen sizes

of 10 inches and above combined with the capability to record more than 5 independent touch pointers and a number of additional sensors (gyroscope, orientation, ...) and feedback actuators (vibration, sound, ...) they make a good choice for a versatile control device. In this thesis however, only visual feedback will be available to the user.

Specifically, development during this thesis will take place on a *Samsung Galaxy Tab S3*. It has a screen size of 9.7 inches[23] with a resolution of 2048x1536 pixels accompanied by a 2.15GHz Quad-Core processor. These properties give it the ability to also perform some calculation-heavy tasks locally, giving the overall application a better performance. The used Android tablet runs Android 7.0. One goal of this thesis is to make the control application available to a broad variety of devices. Because of this, the application shall run on Android down to Version 4.3 and up to the current 7.0.

A native Android application will be developed and run directly on the tablet. As a programming language Java is chosen, as it is the language natively used on Android. Multiple approaches to the problem will be implemented to give users and researchers the ability to test and evaluate multiple methods against their usability, effectiveness and user-friendliness.

1.3 Outline

After an overview of related and similar works to this thesis is given in Chapter 2, a brief insight in the technology used during the development of this thesis will be described and explained in Chapter 3 to give the reader a basis of knowledge to understand the processes and decisions described later. Chapter 4 depicts the concepts and architectural design process decisions made for later development. The implementation is then described in detail within Chapter 5. Proposals for user studies and usability evaluations are then made in Chapter 6, as they are not part of this thesis.

2 Related Work

A lot of research is and was done in the field of remote-controlling robots with generic devices. This is most probably the case due to them being ubiquitous and well-known by common users. This is important as more and more robots have entered the personal living spaces of users during the last two decades[10] which have to be controlled by untrained persons with the least possible amount of learning. This leads to the need of very easily understandable user interfaces and creates a limit in complexity the possible controls.

One approach to reduce complexity in controlling a robot with a high amount of degrees-of-freedom (DOF) is described in [3]. The researchers conducted a principal component analysis (PCA) on different grasping hand poses. The calculated components can then be used to control a device with e.g. 22 DOF by just 2-3 parameters. As this thesis is partly based on this approach and the given research, more explanation can be found in Chapter 4. Apart from this analytical approach another part of this thesis is based on [26] where researchers developed a method to directly map finger positions on a tablet computer to those on a robotic hand. These are the two main approaches examined in this thesis.

Other ideas to teleoperate robots with generic devices are numerous. A similar approach is to control a mobile robot using an android device by tilting it[1]. The idea here is to simulate a steering wheel of a car to move a car-like robot.

As the fields of virtual reality (VR) and augmented reality (AR) gain more and more attention, different approaches to remotely control robots assisted by such VR systems come up. Hashimoto et al. [15] developed a software called TouchMe, displaying a video image of a robot, allowing to directly alter a robot model perfectly laid over the video image using simple drag and drop actions on a touchscreen. Krupke et al. [16] took the approach one step further by displaying the virtual environment on a head mounted device (HMD, also referred as *VR glasses*). The HMD displays a virtual representation of the manipulated device, allowing the controller to look at the scene from arbitrary angles. Fine control was implemented by putting the controller's hand into a virtual sphere and recording hand movement while mapping it to movements of the controlled robot.

3 Basics

3.1 ROS - The Robot Operating System

The *Robot Operating System* (ROS) is an open-source software framework providing a robust communication layer for distributed robot computing[17]. Despite the name it is not an operating system in the traditional sense, as it does not provide or implement any processing, scheduling or data access functionality. It is a set of programs and libraries enabling developers to develop so-called *nodes* that communicate with each other using the *Publish-Subscribe-Pattern*. This pattern allows multiple loosely-coupled nodes (applications) to exchange messages. This design allows a greater reuse of code since software for robots is written very modular[9]. For example, on a robot with a laser scanner and a motor, one node would decode the laser scanner data, publish the results to a specific topic which is subscribed by a controller node, that processes the data and then publishes motor control messages to another topic, which is again subscribed by the motor controller node. All nodes do not have to know each other. This makes it very easy to reuse the code for either the laser scanner or the motor driver node in other configurations (like multiple different robots) or exchange the controller node that processes the data. Using wireless connections, it is also possible to move specific processing tasks to external (*off-board*) nodes. This comes in handy for example in terms of image processing, which is a task that usually overloads small on-board processing units built into robots.

The communication is organized by a program called *ROS Core*. All nodes connect to this Core and tell it what they'd like to do (e.g. subscribing to topics, publishing to topics etc.). To reduce communication overhead, the actual data exchange is then done in a peer-to-peer manner, meaning the nodes directly exchange data with each other over TCP/IP. This also means that all nodes have to be able to reach each other, which might lead to problems when running ROS in bigger networks.

Sometimes, the publish-subscribe-pattern (and its inherent asynchrony) are not sufficient, as some calculations, which might be too heavy to be executed locally, might still have to be done synchronously. For this case, ROS introduces so-called *services*. These are basically function calls that are offered by a node which may then be called by any other node over the network. These calls are executed synchronously and directly return a result.

Nodes have names seperated in so-called *name spaces*. an example node name is given in Listing 3.1.

Listing 3.1: *An example ROS node name*

```
1 /robot/hand/controller
```

where */robot/hand* is the name space and *controller* the node name. Topics and services do also have a specific name including a name space. This addressing scheme allows it to have multiple equally-called nodes or topics (e.g. for multiple sensors of the same type) by just putting them into different name spaces but preserving their names.

Numerous implementations of the ROS client libraries are available, the most common ones are developed and used in C/C++ and Python[18]. For developing a ROS-enabled Android application, an implementation of the ROS client library in Java is chosen.

3.1.1 Rosjava / Rosandroid

There is an implementation of the ROS client library published on GitHub[1]. It includes support for all needed communication structures within ROS as well as the most common message types exchanged with ROS nodes. *rosjava* is specifically designed to develop ROS-enabled Android applications and is originally developed by Google[11].

The package *rosandroid*[2] is an extension of *rosjava*. It offers functionalities to easily include ROS support into an Android application by offering readily usable *Activities*[3] to connect the application to a ROS core or start an independent core within the application itself. It also includes some basic user controls like a joystick control which we will not make use of within this thesis.

rosandroid is designed for the newest versions of Android, which leads to the fact that a small change has to be made in the code to make it compatible with older versions of Android, too. These changes are described in Chapter 5.1.4.

3.1.2 Using Services in Rosjava

The implementation of how to consume (i.e. call) services is a little different in rosjava than it is in roscpp[4], which is why the main differences will be briefly be elaborated on here. To call services, special ROS messages are exchanged. These so-called *service message types* consist of a request and a response part. While in C++ one object containing both the request and the response is passed to the *service client* which then fills out the response part[21], in other implementations like *rospy* or *rosjava* these messages are separated.

To create a *service client* in rosjava, an instance of

```
org.ros.node.service.ServiceClient
```

[1] `https://github.com/rosjava/rosjava_core`
[2] `https://github.com/rosjava/android_core`
[3] Activities are offering the user interface in Android applications
[4] roscpp is the C++ implementation of ROS

is created within the *onStart* callback of the node, passing the name of the service node and the service message types. Listing 3.2 demonstrates how such a start-up routine could look like.

Listing 3.2: *Example on how to connect to a ROS service in rosjava*

```
@Override
public void onStart(ConnectedNode connectedNode) {
// ....
  try {
    ikService = connectedNode.newServiceClient("/bio_ik/get_bio_ik",
       ↪ bio_ik_msgs.GetIK._TYPE);
  } catch (ServiceNotFoundException e) {
    ikService = null;
    e.printStackTrace();
  }
// ....
}
```

In *rosjava* no such method like *Ros::waitForService()* (in C++) is present[5]. As rosjava is designed in a way that one application can implement multiple ROS nodes, a blocking call to the above method would cause the rest of the application to stop working, which is probably the reason why the developers of rosjava have decided not to implement it. In rosandroid applications, a blocking wait-call would cast the user interface unresponsive and thus unusable. Developers have to make sure that, in the time a node starts, the service it wants to consume is already registered with the *ROS Master*.

Once the *ServiceClient* is created it can be used by creating a new request message. Confusingly, request and response message types are separated in service calls, while the combined message type is passed to the *newServiceClient* method. An example service call is presented in Listing 3.3.

Listing 3.3: *An example rosjava service call*

```
bio_ik_msgs.GetIKRequest greq = ikService.newMessage();
IKRequest req = greq.getIkRequest();
// ... fill in the request parameters into the req object ...

ikService.call(greq, new ServiceResponseListener<GetIKResponse> {
  @Override
  public void onSuccess(GetIKResponse getikResponse) {
    // Handle service response
  }

  @Override
  public void onFailure(RemoteException e) {
```

[5]Albeit requested by multiple users, like in https://github.com/rosjava/rosjava_core/issues/105

```
13      // Handle service error
14    }
15  });
```

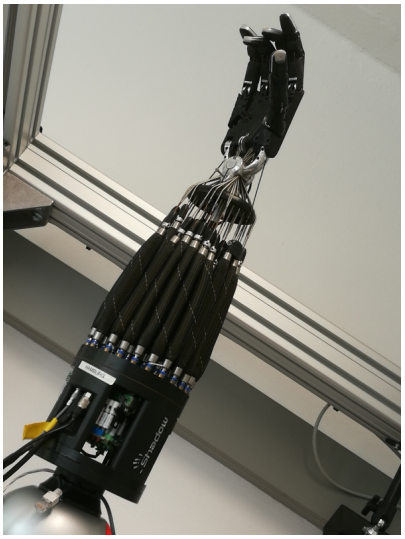
Two things are important to note here. Firstly, the messages exchanged are not *IKRequest* and *IKResponse* as the names would suggest, but *GetIKResponse* and *GetIKRequest*, which are created automatically by the rosjava message generator. The latter service messages then contain the corresponding former message types. This is one main difference in calling ROS services between Java and C++ implementations. Secondly, the service response is processed non-blocking, meaning it is asynchronously passed to the listener object implementing the

```
org.ros.node.service.ServiceResponseListener
```

interface. Busy-waiting service calls are not implemented in rosjava, most probably for the same reasons that busy-waiting for services to come up have not been implemented. This asynchrony is the second main difference developers accustomed to *roscpp* have to get used to when switching over to *rosjava*.

3.2 The Shadow C5 Robotic Hand

Figure 3.1: *The Shadow C5 hand*



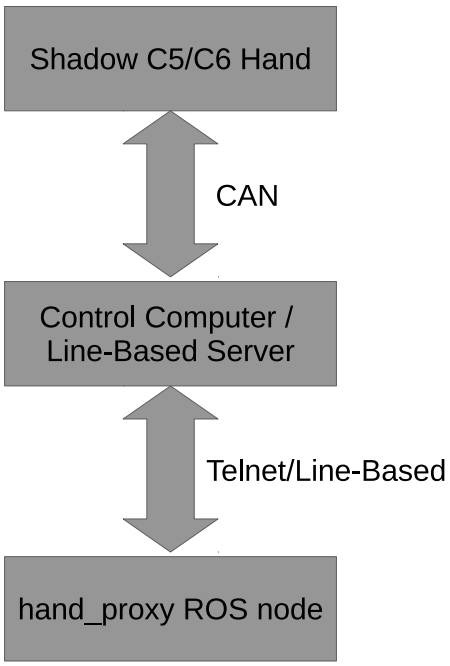
The *Shadow C5 Robotic Hand* was developed by the *Shadow Company*. It is designed to be as similar to a human hand as possible[6] in terms of force output, movement speed and movement sensitivity. It has 24 degrees-of-freedom, all controlled by 48 pneumatic muscles. These muscles, when pressurized, contract a little, applying force to the elements of the mechanical hand over imitated tendons. The developers tried to design the product as close to the average human forearm as they could. It weighs about 4kg and has a maximum movement speed of about half the speed at the joints a human could reach. The pneumatic muscles work with a pressure of 3.5 bar and having a maximum flow of 24 litres per minute, resulting in the need of a relatively powerful air compressor and air pipe system installed near the hand. Joint angles of all joints (controllable as well as non-controllable) are measured by hall-sensors at an accuracy of 0.2 degrees. A similar robotic hand powered by electrical motors instead of pneumatic muscles is also present at the TAMS group at the University of Hamburg. This thesis will, however, mainly work with the pneumatic powered hand.

Table 3.1: *Topics used to send and receive joint states*

/hand/joint_states	The ROS hand proxy publishes the current measured joint states it received from the hand server to this topic.
/hand/joint_goals	The ROS hand proxy receives packages containing joint states sent to this topic and passes it on to the hand server, causing the hand to try to reach the sent joint angles.

3.2.1 Integration into ROS

Figure 3.2: *Schematic overview to the integration of the hand into a ROS environment*



The Shadow Robotic Hand possesses by default a CAN-Bus (Controller Area Network) interface over which it is controlled[6]. The CAN protocol has been implemented using a parallel port on a distinct machine next to the robotic hand. To have the ability to communicate with the hand over network, a server application has been implemented by members of the TAMS group at University of Hamburg. Multiple applications have been developed to control the robotic hand without the integration of ROS. To make use of the features and advantages of a ROS environment, a ROS proxy was implemented. It basically listens to a ROS topic where it receives joint target states and publishes to another ROS topic where it sends the current measured joint angles to. The ROS hand proxy node communicates with the hand server over the line-based protocol and converts all data it receives for the corresponding other side. This set-up makes it easy to integrate the Shadow C6 hand into a ROS environment. See Figure 3.2 for a schematic overview of how the robotic hand is integrated into ROS.

The two important topics used throughout this thesis are denoted in Table 3.1. The message type used for both of these topics is *sensor_msgs/JointState*. These messages consist of the data fields denoted in Table 3.2. A few things are important to be considered while using the data contained in these messages. First, how the data is interpreted is application-specific. While the data fields contain arbitrary data it is important to know that the set-up used in this thesis only has rotating joints, meaning the data in the position field is in radians. For other types of joints (e.g. linear joints) this could possibly deviate. Second, the order of elements is not important, however it is very important to maintain corresponding elements' positions at the same index within the *names* and the *position* fields. This means that e.g. the position for joint *THJ1* must have the same index in the *position* field as the string *THJ1* in the *names* field. Finally it is important to note

Table 3.2: *Contents of the sensor_msgs/JointState message type*

Field	Description
header	Header information including a time-stamp of when the message was sent and a sequence number of the message
names	Array of strings containing the joint names the other fields contain information about
position	Array of floats containing the position of each joint
velocity	Array of floats containing information about the velocity at which each joint is currently or shall be moving
effort	Array of floats containing information about with how much efford (e.g. force) a joint shall be or is moved

that the *effort* and *velocity* fields are currently not used for the set-up. When these rules are followed it is easy to send joint states to the robot and observe its movement.

3.3 The Kuka Lightweight Robot Arm

The robotic arm used in the set-up is a *Lightweight Robot 4+* by the German company *KUKA Roboter Gmbh*. It has 7 degrees-of-freedom, allowing it to operate in a space as big as approximately $1.8m^3$[12]. All joints can be used in ranges of ± 170 or ± 120 degrees. The robot is controlled by a dedicated computer supplied with it. Connected to the computer is an external control interface, which allows basic operations of the robot.

Figure 3.3: *The Kuka LWR robot arm*

3.3.1 Integration into ROS

To control the robot using ROS a special application has to be launched on the control computer of the robot. This application is called *FRI* (*Fast Research Interface*) and is supplied by the KUKA company[13]. When this was successful, a special ROS node has to be started on another computer within the same network. This node is called *ros_fri*. Then messages can be sent to the robot by publishing messages of the type *ros_fri_msgs/RMLPositionInputParameters* to the topic:

```
/lwr/jointPositionGoal
```

The contents of the message type are described in Table 3.3. When such a message is received by the FRI application the robot will immediately start to move to the given position. The arrays in the message all have to have the correct number of entries (i.e. 7, one for each joint). If no value shall be set, a zero value has to be inserted anyway.

Table 3.3: *Contents of the RMLPositionInputParameters message type*

Field	Description
double[] target_position_vector	Desired target positions of all joints in radians.
double[] target_velocity_vector	Desired movement velocities of all joints.
double[] max_acceleration_vector	Maximum allowed acceleration for all joints.
double[] max_velocity_vector	Maximum movement velocity allowed for each joint.

3.4 Inverse Kinematics

Inverse kinematics is one of the challenging fields in many applications like robotics or computer animations[24]. To understand what inverse kinematics is, it is important to look at a robot (or e.g. animated figures in video games) from two different points of view. The normal viewer would describe the position and pose of a robot or effector in his own coordinate system, usually in Cartesian coordinates. This position can be described as an n-dimensional vector X. To describe movement of the robot, the viewer would then tell a difference between the new and the old position vectors $X_{new} - X = \Delta X$. A robot, however, often cannot move in Cartesian space, as its kinematic chain (i.e. the parts of the robot connected by rotational or translational joints) can have m degrees-of-freedom (DOF) with $m > n$. The position in the so-called *joint-space* is referred to as θ. To control such a robot with a high number of DOF, the controller has to be aware of the current position of the robot in Cartesian space X, the desired position change ΔX and the change in joint-space $\Delta\theta$ that has to be applied to the current position in joint space θ. θ is known by the current state of the robot, finding X is done by applying *forward-kinematics* to θ:

$$X = f(\theta)$$

Forward kinematics usually is a straight-forward process of beginning at the base of the robot and iterating through all joints up to the *end-effector* to find its position. The inverse kinematics to find the corresponding position in joint-space to reach the desired position in Cartesian space

$$\theta = f^{-1}(X)$$

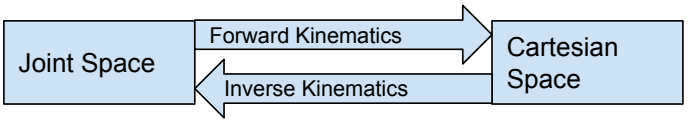
however is not as easy as the forward kinematics as with rising numbers of DOF no analytical solution is possible and multiple (up to an infinite number) valid joint-positions can exist - or even none at all[7].

Many approaches to this problem have come up during the years. Aristidou and Lasenby give a good overview over the existing methods in their technical report[7].

3.4.1 BioIK

BioIK is the name of a newly developed algorithm for inverse kinematics by the TAMS research group at the University of Hamburg[24]. BioIK is a multi-goal evolutionary

Figure 3.4: *Forward and inverse kinematics*



algorithm. This means in particular, that it accepts goals for multiple end-effectors in a kinematic chain whereas most other algorithms only accept one goal for one end-effector. This makes the algorithm especially suitable for highly articulated robots and models like humanoids[25]. Being an evolutionary algorithm means that solutions are created using predecessors and applying random mutation to a solution. Solutions of the algorithm are then classified by a fitness function, while good solutions remain within the so-called genom and bad solutions are not used for further evolutions[22] - similar to the so-called and name-giving real world evolution. Within this thesis BioIK is used to calculate joint angles for given robot poses. The big advantage of BioIK is that it accepts multiple goals, i.e. one goal for every fingertip, and calculates corresponding joint positions based on the given goals.

Integration into ROS

Philipp Ruppel integrated BioIK into ROS during his Master Thesis[22]. He integrated the BioIK solver into *MoveIt!*, which is a motion planning framework integrated into ROS[5]. Using *MoveIt!* it is possible to plan motions and poses of robots from just calculating a pose of a robot to plan full motion trajectories from one pose to another while avoiding obstacles and collisions.

In addition to the functionality directly calling *MoveIt!* interfaces from C++, a ROS service was implemented to get IK solutions from BioIK over a ROS service from arbitrary nodes - especially from non-C++ nodes like ones written in Java (rosjava, rosandroid) or Python (rospy)[6]. Having this BioIK ROS service available makes it relatively easy to get IK solutions within nodes separated from *MoveIt!* which is why it will be used within this thesis to request joint positions for given robot poses within the developed Android application. The process of integrating the service into the application (i.e. requesting joint angles for given robot poses) is described in Chapter 4.2.

[6]The implementation of the ROS BioIK Service can be found at `https://gogs.crossmodal-learning.org/philipp.ruppel/bio_ik_service`

4 Concepts

4.1 User Interface

4.1.1 Desired position of the Android Tablet

The application (and thus the screens within it) will be designed for the tablet to be placed in front of the operator on a table. The person should have clear sight on the controlled robot. It seems sensible to place the tablet on the table in front of the robot while looking at it. Most interactions with the application will be performed by touch gestures using the right hand. For better usability a housing or case can be used to position the tablet at a slight angle to the table.

Figure 4.1: *Example placement of the control device*

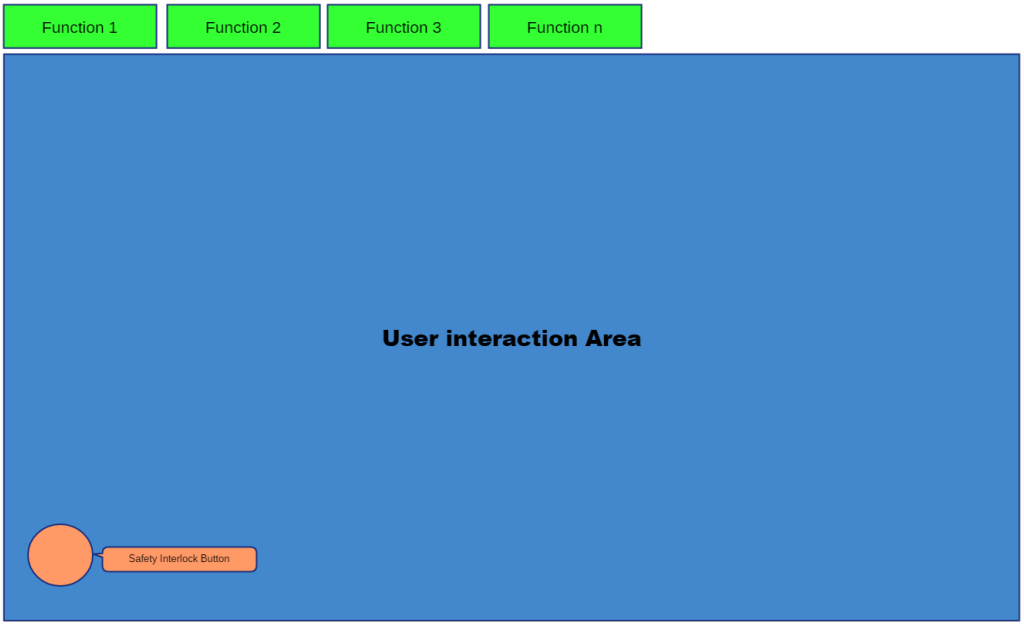
Interaction with the application is done using the commonly known touch gestures like

- Touch (short press on the screen)
- Long press (finger remains on a control for a longer period of time)
- 1-Finger-Movement
- 2-Fingered gestures (*Pinch-Zoom*, Rotation)
- 3-Fingered gestures (Rotation, Movement)

4.1.2 General Screen Layout

As a screen of a diagonal size of 10 inches (25.4cm) is very limited compared to the size of the workspace of the robot, good considerations have to be made according to a well-designed user interface. Since we are mainly operating the robot with touch gestures, significant parts of the screen should be blank, as only few information can be displayed while the user poses his hands above or on the screen. Figure 4.2 gives a first overview of how the portions of the screen shall be distributed. The biggest part of the screen is

Figure 4.2: *A first overview of the screen space distribution*



reserved for touch interactions by the user. Since multiple approaches to control the robot shall be implemented, the method shall be selected and switched using a tabbed layout with the tabs on the top, as they use the least space of the screen this way.

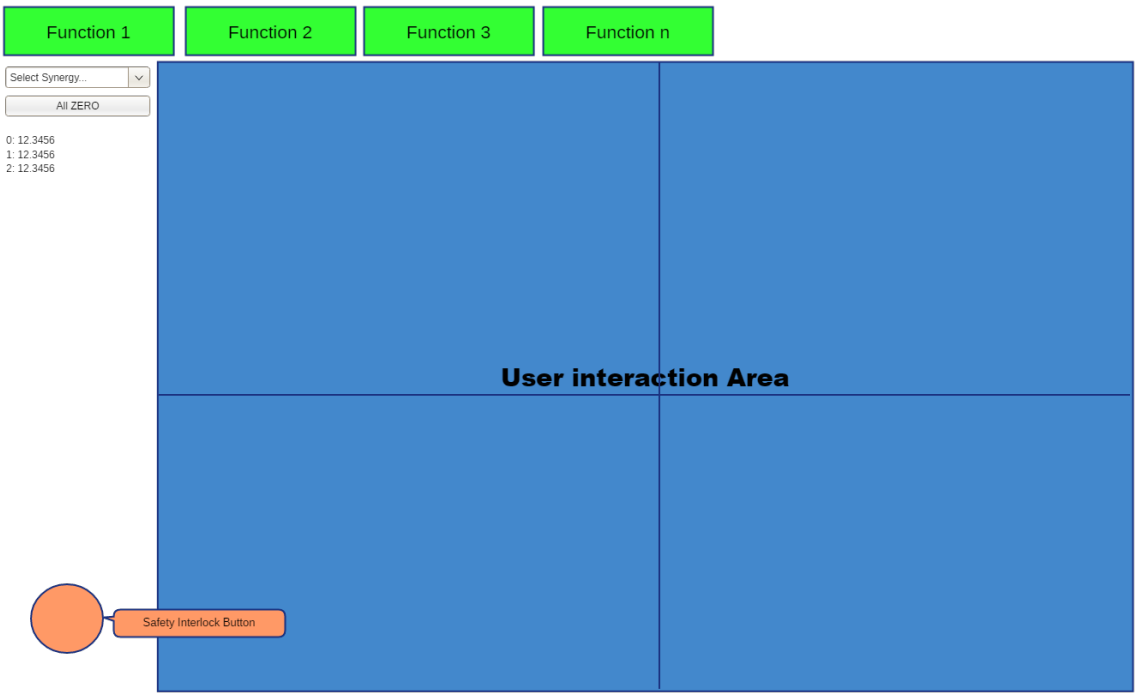
Interlock Button

On all screens where the robot can be remotely operated, a security interlock button shall be displayed. For the actions on the screen to have effect on the robot (i.e. to be sent to the controller) the button shall remain pressed. This implements the functionality of a dead-man-switch, stopping all robot action once released. Although this is only a software measure it should be a good solution against unwanted movements of the robot as the button can easily be released when pressed with a single finger of the left hand. **Of course, this software measure does not replace hardware safety measures like emergency switches, but only supports them.**

4.1.3 Grasp Synergy Screens

As the control of grasp synergies allows multiple different types of synergies to be selected, a drop-down selection of the synergy shall be displayed to the left side of the screen. This keeps the right side of the screen clear for better operability by right-hand users. Additionally, a cross of lines shall be displayed on the screen so the user knows where the middle of the touch interaction area is. As described later, this is particularly important in the approach with absolute synergy control (See Chapter 4.3.2). To give the user some information about the state of a synergy, the values of significant amplitudes applied to the synergy shall also be displayed on the left hand side of the screen. A but-

Figure 4.3: *Synergy control screen*



ton to set the hand into the idle state of the synergy (i.e. all significant amplitudes set to 0) is also sensible to be implemented. A sample of how this screen could look like is demonstrated in Figure 4.3.

4.1.4 Direct Fingertip Mapping Screen

As there are no additional controls required to control the direct fingertip mapping, the control screen looks mostly like the general touch interaction screen seen in Figure 4.2.

4.1.5 Single Axis/Joint Control

An interface shall be implemented to give users the ability to control each joint of the robot individually. This is sensible for a variety of reasons. Firstly, it might sometimes be required to move the robot out of a specific state by moving just one axis and not by applying multiple changes at once. Possible scenarios for this use case could be a state where the robot could harm users or the environment if uncontrolled or unpredictable movements occur. An interface to control joints individually is also very practical for testing purposes, for example if one part of the robot is suspected to be broken.

Figure 4.4: *Axis control widget with different status indicators*

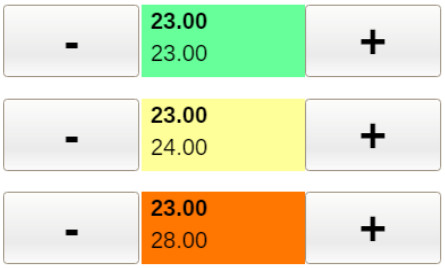
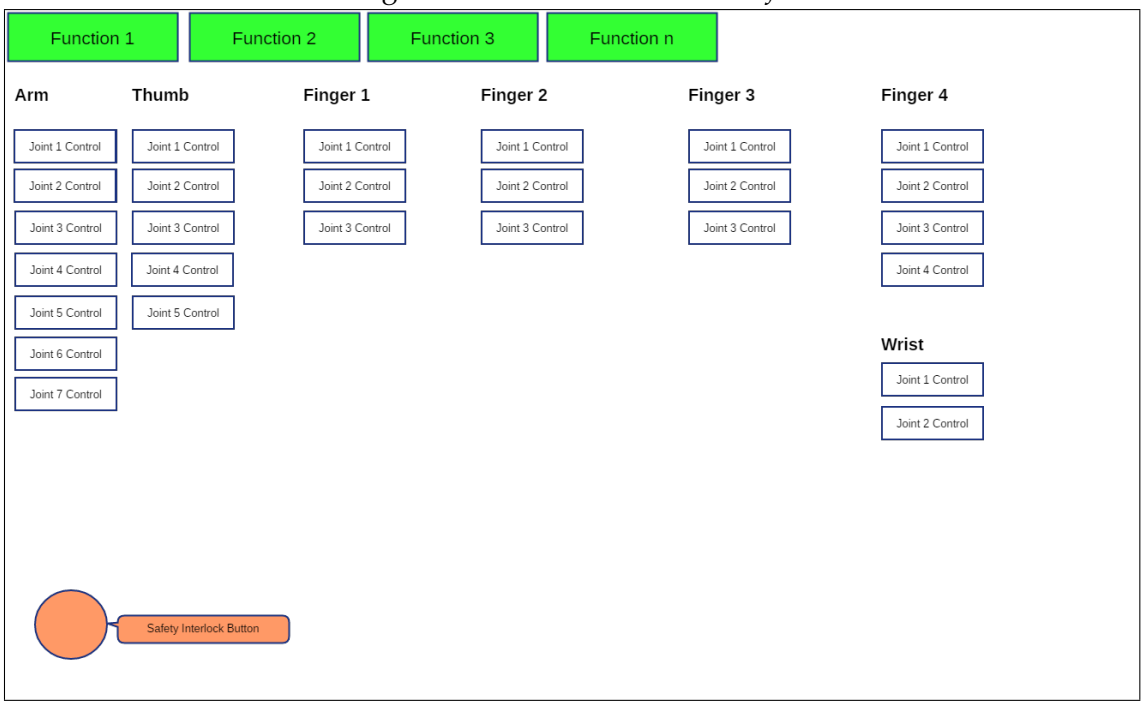


Figure 4.5: *Axis control screen draft*



Within this interface, the currently measured joint angle shall be displayed for each joint, next to the currently set target value (printed in bold). This gives the user a good insight of what state the robot should be in according to the program and what state it actually is in. This shall be supported by a coloured indicator, giving a quick visual feedback on the difference of the target and actual joint angles $\Delta\alpha = |\alpha_{actual} - \alpha_{target}|$. The colours of the visual feedback shall be:

- Green for $\Delta\alpha \leq 0.5°$
- Yellow for $0.5° < \Delta\alpha \leq 2.5°$
- Red for $\Delta\alpha > 2.5°$

The buttons to move the axis shall be displayed right and left of the angle displays. The visual feedback shall be shown as the background of the angle values. With all these requirements put together, an axis control widget for a single joint or axis could look like depicted in Figure 4.4. To each control widget, a heading will be added to unambiguously denote which axis or joint will be controlled when using the corresponding buttons.

Multiple of these widgets shall be added to the axis control screen, one for each controllable joint or axis. This will result in a screen containing 29 of these (22 for the hand, 7 for the robot arm). To get an idea of how the control screen for individual joint control will look like, the reader is referred to Figure 4.5. Please note, however, that the number of hinted joint control widgets does not resemble the actual number of controllable joints for each part of the robot.

4.2 Using the BioIK Service

The BioIK serivce (see Section 3.4.1) is used according Section 3.1.2. The important message types are:

- **GetIK** The combined Request/Response service message type
- **GetIKRequest** The message type containing the request to the service
- **GetIKResponse** The message type containing the response of the service
- **IKRequest** The actual request data
- **IKResponse** The actual response data

In the process of getting joint data from the BioIK service, the message types *IKRequest* and *IKResponse* are most important, because they contain the data needed on both sides. As the used message types contain a large number of members, only those significant for this thesis will be discussed within this section.

The *IKRequest* message contains information about the current robot state and about the desired robot pose (see Table 4.1, pg. 17). The so-called *goals* used here are *PositionGoal*, *OrientationGoal* and *PoseGoal*, which is basically a combination of the first two. All goals contain a link name, which describes the end-effector that shall be brought to the given position or orientation. A *PoseGoal* contains a *Point* message type, which is a vector in 3D space. The *OrientationGoal* consists of a *Quaternion* which encodes the desired orientation of the end-effector in space. Lastly, the *PoseGoal* contains both, a *Point* and a *Quaternion*, defining a distinct position and orientation in space. When provided with the current state of the robot, the IK algorithm begins looking for solutions at this state, possibly speeding up the overall solving process. Different goals for multiple end-effectors (called *links*) can be passed to the BioIK service which will then try to find a solution fulfilling all of the given goals.

When the BioIK service has finished it returns a *IKResponse* (see Table 4.2, pg. 17) to the caller. Within the response, a status (or error) code is given, indicating success or failure of the algorithm. If the status code indicates 0 (meaning success), the *RobotState* field of the message contains joint angles for all joints of the robot, representing a state in which the goals passed to the service in the first place are reached. All angles for the joints (in *IKRequest* as well as *IKResponse*) are given in radians.

4.3 Grasp Synergies

In their work Bernardino et al. [3] describe a way to record hand postures using a data glove and the Shadow C5 robotic hand. The result of their research are data recorded for 8 different grasp postures of the human hand. Using *Principal Component Analysis (PCA)*

Table 4.1: *Important contents of the IKRequest message type*

Field	**Description**
string group_name	The MoveGroup name of the robot. Fixed to *lwr_with_c5hand* here
bool approximate	If true, an approximate solution is returned when no exact solution could be found by the IK solver
duration timeout	The timeout after which the solver stops looking for solutions. If no solution was found by then, an approximate solution is returned if approximate is true, otherwise no solution is returned. Fixed to one second here.
int32 attempts	Number of attempts the solver shall take. Fixed to 1 here.
string[] fixed_joints	Names of the joints the IK solver shall not move while looking for solutions.
bool avoid_collisions	If true, the BioIK solver tries to find solutions that do not collide with the environment (and the robot itself).
RobotState robot_state	The current state of the robot. Contains a JointState message containing the current joint angles of all joints. These values are taken as a starting point while searching for solutions.
PositionGoal[] position_goals	The positions of multiple end-effectors that shall be reached with the IK solution.
PoseGoal[] pose_goals	The poses of multiple end-effectors that shall be reached with the IK solution. A pose goal is similar to a position goal, but extends it by a desired orientation.
OrientationGoal orientation_goals	The orientations multiple end-effectors shall have within the IK solution.

Table 4.2: *Important contents of the IKResponse message type*

field	**Description**
MoveItErrorCodes error_code	An error code stating if a solution was found or not. 0 indicates success, whereas all other values indicate that no solution was found.
RobotState solution	If error_code is 0, this field contains the found solution encoded in a JointState field with an angle for every joint.

the datasets for each posture were parametrized. This process resulted in a matrix

$$S = (s_1, \ldots, s_M)$$

for each grasp posture with $s_m \in \mathbb{R}^N$ being the eigenvectors of the parametrized grasp postures. For each posture $s_0 \in \mathbb{R}^N$ is the mean value of all recorded posture data sets, defining the rest position of the hand within this posture. To get joint angles from these synergy matrices, they have to be multiplied by a vector $\alpha = (\alpha_1, \ldots, \alpha_N)$ containing the so-called amplitudes for each parameter. For a given synergy matrix S, a synergy offset s_0 and amplitude vector $\alpha \in \mathbb{R}^N$, the joint angles $\theta = (\theta_1, \ldots, \theta_N)$ are described as

$$\theta = s_0 + S\alpha \tag{4.1}$$

Since S is sorted in such a way that changes to α_1, α_2 and α_3 already cover approx. 80-90% of the variance in the grasp postures recorded[3], we will only look onto these three amplitudes when implementing the approach to grasping objects in this thesis.

s_0 and S are provided by the above research and θ_n is given in degrees, α_n is $-50 \leq \alpha_n \leq 50$. The goal is to find a good mapping between touch gestures and α_n, so that the grasp can be controlled intuitively.

4.3.1 Touch Gestures

To find solutions to map touch gestures to synergy amplitudes, we first generalize the understanding of touch gestures.

Definition 1 *$p = (p_x, p_y) \in \mathbb{R}^2$ is called a pointer on a touch screen, i.e. the coordinates of a registered finger the user has laid onto the touch surface.*

Definition 2 *A set of pointers $G = \{p_1, \ldots, p_n\}$ with $|G| \geq 1$ is called a **gesture**.*

Although pointer positions on a touch screen are usually given in integer numbers, pointers are defined in real space to make the following definitions possible. After having defined gestures, we have a look at different properties of them. Firstly, the two most basic properties of a gesture are defined, being the **position** and the **size**.

Definition 3 *Let G be a gesture.*

- *The **position** $c(G)$ of G is defined as*

$$c(G) = \frac{1}{|G|} \sum_{i=1}^{|G|} p_i \qquad p_i \in G. \tag{4.2}$$

- *The **size** $s(G)$ of G is*

$$s(G) = \frac{2}{|G|} \sum_{i=1}^{|G|} d(c(G),\, p_i) \tag{4.3}$$

with $p_i \in G$ *and* $d(x,y) = \sqrt{(y_1 - x_1)^2 + (y_2 - x_2)^2}$ *for* $x, y \in \mathbb{R}^2$ *the euclidean distance between two pointers.*

In other words: The position of a gesture is the *center of mass* of all pointers of a gesture. The size is the doubled mean distance of all pointers to the position of a gesture. Before defining the last property of a gesture, the **orientation**, we first have to define what the **thumb pointer** of a gesture is.

Definition 4 *The* ***thumb pointer*** $th(G)$ *of a gesture* G *is defined as*

$$th(G) = \begin{cases} p_1 & |G| = 1 \\ p_n \text{ with } p_{n,y} = max\{p_{i,y} : p_i \in G\} & |G| = 2 \\ p_n \text{ with } d(c(G), p_n) = max\{d(c(G), p_i) : p_i \in G\} & |G| > 2 \end{cases} \tag{4.4}$$

The thumb pointer shall be evaluated and memorized whenever $|G|$ changes, i.e. a pointer is added or removed from a gesture. Note that the y coordinate is rising to the bottom, as it is usual on digital screens. Having this in mind, the thumb pointer is the lowest pointer in a 2-pointer gesture or the one furthest away from the position of a gesture with 3 or more pointers. The last part of the definition is important, as the lowest pointer does not necessarily remain the lowest when the gesture is rotated on the screen, so to have a consistent definition of the orientation, the thumb pointer may only be evaluated when pointers are added or removed from a gesture.

Definition 5 *Let* G *be a gesture,* $b_y = (0, -1) \in \mathbb{R}^2$*. For* $v = c(G) - th(G)$*, the orientation* $o(G)$ *is defined as*

$$o(G) = \text{sign}(\det(b_y\, v)) \cdot \arccos\left(\frac{v \cdot b_y}{|v| \cdot |b_y|}\right). \tag{4.5}$$

In other words the orientation of a gesture is the angle between the vector from the thumb pointer to the position of a gesture and b_y, which is pointing upwards in screen coordinates as, again, y is growing downwards. If the determinant of $(v\, b_y)$ is negative, the angle between b_y to v is counter-clockwise[4]. Using this, the values of $o(G)$ range from $-\pi$ (which is 180°*counterclockwise*) and π.

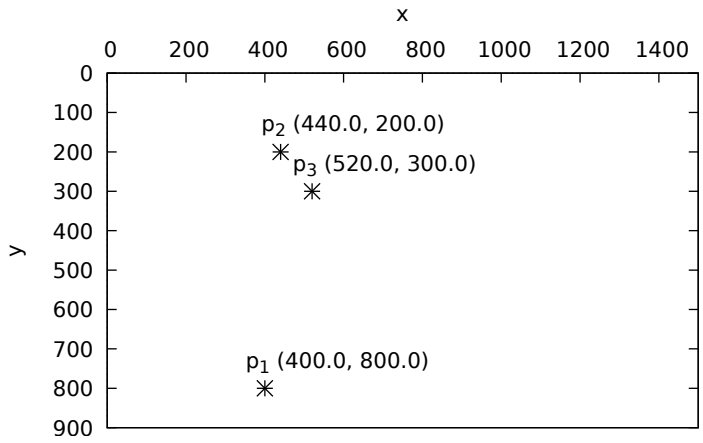
Example

The above definitions will be briefly elaborated with an example of a gesture with 3 pointers. Figure 4.6 shows the example gesture. First, the position of the gesture is calculated:

$$c(G) = \frac{1}{3}\left(\begin{pmatrix}400\\800\end{pmatrix} + \begin{pmatrix}440\\200\end{pmatrix} + \begin{pmatrix}520\\300\end{pmatrix}\right) \approx \begin{pmatrix}453.3\\433.2\end{pmatrix}$$

The size of the gesture is

Figure 4.6: *Example gesture with 3 pointers*



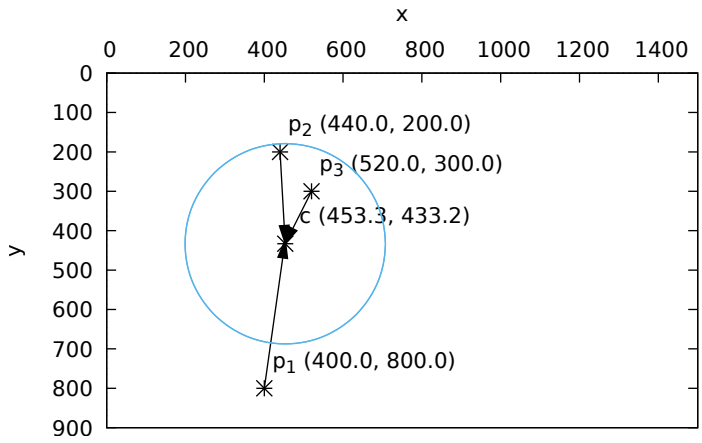
$$
\begin{aligned}
s(G) &= \frac{2}{3}\left(d(c(G), p_1) + d(c(G), p_2) + d(c(G), p_3)\right) \\
&= \frac{2}{3}\left(|c(G) - p_1| + |c(G) - p_2| + |c(G) - p_3|\right) \\
&= \frac{2}{3}\left(\left|\begin{pmatrix} 53.3 \\ -376.8 \end{pmatrix}\right| + \left|\begin{pmatrix} 13.3 \\ 233.2 \end{pmatrix}\right| + \left|\begin{pmatrix} -66.7 \\ 133.2 \end{pmatrix}\right|\right) \\
&\approx \frac{2}{3}\left(380.6 + 233.6 + 147\right) \\
&\approx 508.8
\end{aligned}
$$

Figure 4.7 shows the example gesture extended with the position of the gesture and a circle with the diameter of the gesture size around the position of the gesture. From the above calculation it can be seen that p_1 has the biggest distance to the position of the gesture, that implicates $th(G) = p_1$ as G has more than 2 pointers. With this we can calculate the orientation of the gesture as follows.

$$
\begin{aligned}
o(G) &= \operatorname{sign}\left(\det\begin{pmatrix} 0 & 53.3 \\ -1 & -376.8 \end{pmatrix}\right) \cdot \arccos\left(\frac{\begin{pmatrix} 53.3 \\ -376.8 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ -1 \end{pmatrix}}{380.6 \cdot 1}\right) \\
&= 1 \cdot \arccos\left(\frac{376.8}{380.6}\right) \approx \arccos\left(0.99002\right) \\
&\approx 0.141398 \,\widehat{\approx}\, 8.101^\circ
\end{aligned}
$$

G has an angle relative to the y axis from about 8.1°.

Figure 4.7: *Example gesture with center and size*



4.3.2 Absolute Approach

Within the absolute approach a solution shall be found to map the properties of one or more gestures to the different degrees of freedom of the hand synergy and the arm. To control the hand synergy, it is sensible to control all three significant amplitudes with just one gesture, so the user does not have to change the pointer count of the gesture while operating the hand. This is especially useful as it seems relatively difficult to position the fingers at the exact same place after hacing used another gesture. However, to reproduce a distinct position the pointers have to be placed at the exact same place on the screen.

It is $c : G \to \mathbb{R}^2$, $s : G \to \mathbb{R}$ and $o : G \to \mathbb{R}$, which means if both components of the position are viewed separately it is possible to control as many as four DOF with just one gesture. As, for the hand posture synergies, we only want to control 3 DOF, considerations should be made which properties shall be taken into account for controlling the synergies. It seems best to choose those properties having the biggest range in value:

- The y component of the gesture position has a range of slightly less than the screen height
- The x component of the gesture position has a range of slightly less than the screen width
- The orientation has a range of 2π.
- The gesture size has a range corresponding to the user's finger span, which should usually be around $\frac{2}{3}$ of the screen width

In first tests, it turned out complicated to have the position of a gesture remain at the same place in both axes while rotating it or changing its size, the x component of the position is chosen as one DOF, since it is larger and thus it is possible to have more granular control on the value of the amplitude.

All mappings from gestures' properties to DOF values shall be done linearly. A general approach has to be found to create these linear functions for all DOF of the hand synergy as well as the arm.

Linear DOF Mapping

There are multiple requirements for the function to find. Firstly, the linear equation shall be defined by two points $x, y \in \mathbb{R}^2$ with x_1 and y_1 being in gesture property space and x_2, y_2 being in the corresponding DOF space (amplitude or arm position). Secondly, the output value shall be limited to the possible values of the DOF, for the synergy amplitudes this will be -50 and 50.

A function $lm : (x \in \mathbb{R}^2, y \in \mathbb{R}^2, v_{min} \in \mathbb{R}, v_{max} \in \mathbb{R}, p \in \mathbb{R}) \to \mathbb{R}$ shall be found with the parameters being the two points the linear equation spans in between, the minimum and maximum output value and the value of the gesture property. It outputs the value that can then directly be passed into the DOF value.

Finding the linear equation spanned between two points $x, y \in \mathbb{R}^2$ is relatively easy. Beginning from the general linear equation

$$f(x) = m \cdot x + b$$

it is well known that

$$\begin{aligned} m &= \frac{y_2 - x_2}{y_1 - x_1} \\ b &= x_2 - m \cdot x_1 \\ &= x_2 - \frac{y_2 - x_2}{y_1 - x_1} \cdot x_1 \end{aligned}$$

To clip a value v to a minimum and maximum value v_{min} and v_{max} (with $v_{min} \leq v_{max}$) one *min* and a *max* operation have to be performed[1]:

$$clip(v_{min}, v_{max}, v) = max(v_{min}, min(v, v_{max}))$$

Combination of the linear equation and the $clip$ function results in the function defined below.

Definition 6 *Let*

$$lm : (\mathbb{R}^2, \mathbb{R}^2, \mathbb{R}, \mathbb{R}, \mathbb{R}) \to \mathbb{R}$$

be a function for ***linearly mapping*** *a value p onto a linear function between two points* $x, y \in \mathbb{R}^2$

[1] If it's unknown which limiting value is greater, two *min* and *max* operations have to be performed: $clip(v_1, v_2, v) = max(min(v_1, v_2), min(v, max(v_1, v_2)))$

with the output limited between $v_{min}, v_{max} \in \mathbb{R}$. *Then lm is*

$$lm(x, y, v_{min}, v_{max}, p) = clip\left(v_{min}, v_{max}, \left(p \cdot \frac{y_2 - x_2}{y_1 - x_1} + x_2 - \frac{y_2 - x_2}{y_1 - x_1} \cdot x_1\right)\right). \quad (4.6)$$

Mappings

Let w_{screen} be the width of the screen in pixels, $\alpha \in \mathbb{R}^N$ the vector of amplitudes for a selected synergy S, G a gesture with $|G| = 2$. The proposed mappings for later implementation are:

- $\alpha_1 = lm\left((1200, 50), (300, -50), -50, 50, s(G)\right)$

  As the size of a gesture is the property easiest to manipulate (with 2 pointers by the so-called pinch-zoom-movement) the amplitude with the biggest effect is assigned to it. The values of 1200 and 300 are chosen as 1200 is the size of a gesture with normally spanned fingers, and 300 is the size of a gesture when the pointers are already relatively close. Values lower than 300 render the amplitude value of -50 unreachable, as pointers are perhaps not be able to be placed that close.

- $\alpha_2 = lm\left((w_{screen} \cdot 0.25, 50), (w_{screen} \cdot 0.75, -50), -50, 50, (c(G))_1\right)$

  For the second significant amplitude the x component of the gesture position is chosen. It is mapped between $\frac{1}{4}$ and $\frac{3}{4}$ of the width of the screen as the actual limits cannot be reached with gesture positions due to all pointers having to be at (or beyond) screen borders.

- $\alpha_3 = lm\left(\left(-\frac{\pi}{2}, 50\right), \left(\frac{\pi}{2}, -50\right), -50, 50, o(G)\right)$

  For the third significant amplitude the orientation is mapped from $-90° \mathrel{\widehat{=}} -\frac{\pi}{2}$ to $90° \mathrel{\widehat{=}} \frac{\pi}{2}$. Rotations to values greater or less than these have shown to be only possible when moving the tablet or changing the size and position of a gesture.

α_i, $i = \{1, 2, 3\}$ are evaluated simultaneously whenever the position of a pointer within the gesture changes. Joint angle values are then calculated using (4.1) with S being the currently selected synergy. Once the joint values are calculated they can be published to the ROS nodes of the Shadow C5 hand.

4.3.3 Relative Approach

Within the relative approach, the values of the significant amplitudes shall not depend on the actual properties of a gesture but on the change each property experiences. This makes it necessary to store the current value of the amplitudes and initialize them, e.g. with 0. Whenever a gesture changes, the difference of the properties is evaluated and mapped to a difference of the amplitude which is then applied.

Definition 7 *Let G be a gesture. $G(t)$ describes the state of the gesture (i.e. the position of the pointers within the gesture) at time t. Then*

$$c_i(G) = c(G(t_0 + i \cdot \Delta t)) \,. \tag{4.7}$$

Accordingly, $s_i(G)$ and $o_i(G)$ are defined.

Definition 8 *Let G be a gesture. At the time i it is*

$$\Delta c(G) = c_i(G) - c_{i-1}(G) \,. \tag{4.8}$$

Again, $\Delta s(G)$ and $\Delta o(G)$ are defined accordingly.

With the above definitions we can describe the change of G between two time steps and the change of the gesture's properties between two time steps. The touch pointers are evaluated by the Android operating system periodically, so while a touch pointer is present on the screen, periodic updates are sent to the application. As we do not know the time steps, Δt may be small, in particular it may be 0 without affecting the validity of following calculations.

Next, a rate of change of the output value has to be declared. This rate is given in $\frac{\text{value change}}{\text{property change}}$. If, for example, the value of an amplitude shall change by $c_v = 25$ every $c_p = 1000$ pixels the gesture is moved in one direction, the value would be $\frac{c_v}{c_p} = \frac{25}{1000} = 0.025$. For reasons of simplicity, c_v and c_p shall be passed to the relative change function defined below.

Definition 9 *Let $rm : (\mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R}) \to \mathbb{R}$ be the **relative mapping** function that alters the value v_{old} by the changed parameter Δp at the rate of change $\frac{c_v}{c_p}$ while clipping the output between v_{min} and v_{max} with $v_{min} \leq v_{max}$. Then*

$$rm(c_v, c_p, v_{min}, v_{max}, v_{old}, \Delta p) = clip(v_{min}, v_{max}, v_{old} + \frac{c_v}{c_p} \cdot \Delta p) \,. \tag{4.9}$$

Updating the output value upon a changing gesture takes two steps, for example with amplitude α_1:

$$\begin{aligned} \alpha_{1,new} &= rm(50, 1200, -50, 50, \alpha_1, \Delta s(G)) \\ \alpha_1 &= \alpha_{1,new} \end{aligned}$$

For α_2 and α_3 the calls to rm are accordingly:

$$\begin{aligned} \alpha_{2,new} &= rm(50, 1200, -50, 50, \alpha_2, (\Delta c(G))_1) \\ \alpha_{3,new} &= rm(40, \pi, -50, 50, \alpha_3, \Delta o(G)) \end{aligned}$$

Whether these values are well chosen and usable has to be evaluated during tests. When all amplitudes have been calculated the process of calculating the joint angles and publishing them is the same as in Section 4.3.2.

4.3.4 Arm Control

Controlling the arm with touch gestures is not as easy as controlling the grasp synergies since the arm's joint angles have to be calculated using the BioIK service. The objective is to give the user the opportunity to control the position of the palm of the robotic hand in Cartesian coordinates. To achieve this, it is important to know the orientation of the coordinate system around the arm as well as *safe* coordinates, where the robot arm can move without causing any danger to its environment. Figure 4.8 gives a good insight in the coordinate system used. Potentially, an arbitrary coordinate system could be used, but the BioIK service at the point of writing only accepts coordinates in the so-called *world* coordinate system. It has its origin on the floor below the base of the robot arm. Each square in the illustration represents one meter in the real set-up. The shown model is assumed to be sufficiently exact to use it as a simulation for the real robot with its surroundings. It is important to keep in mind that the y-coordinate is counter-intuitively rising to the back, which means it is falling in the direction to the front. Also, from the perspective of the robot, the x axis is oriented to the left.

Figure 4.8: *The coordinate system relative to the arm*

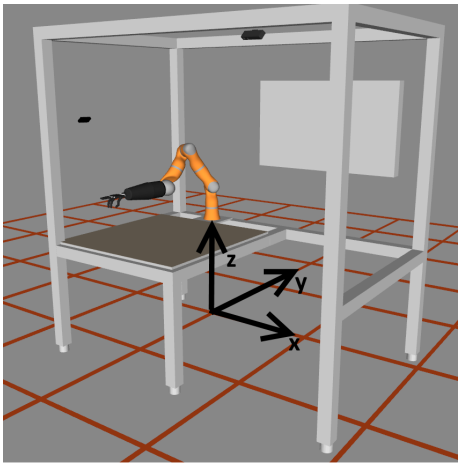


Table 4.3: *Position limits for the palm of the robot's hand*

Axis	Min.	Max.
x	−0.2	0.4
y	−1.2	−0.8
z	1.05	1.17

In order to prevent damages to the robot or its environment, a *bounding box* has to be found where the actions of the robot do not cause any unwanted movements which can occur since the shortest way from position x to position y may be short in cartesian space, but still long in joint coordinates. It is important to keep in mind that for close points, the BioIK service might output extremely different joint positions, as it assumes the new solution more fitting than the old one. This, however, results in a quite small work space for the robot compared to the overall table size. For the first approach, the limits for all 3 axes assumed *safe* are denoted in Table 4.3.

To map touch interactions to positions of the palm of the hand the same functions as in Sections 4.3.2 and 4.3.3 are used with the clipping borders set to the safe axes' limits. As a starting point, the x-axis is mapped to the x-position of a three-pointer gesture, the z-axis to the y-position of a three-pointer gesture. As before, only half of each screen dimension

will be mapped, leaving out a quarter on each end to make it easier to reach all limits. The y-axis will later be mapped to another property of the three-pointer gesture or the position of a four-pointer gesture. Once the desired palm position was calculated it is passed to the BioIK service first, the resulting joint angles returned by the service are then passed to the robot.

4.4 Direct Fingertip Mapping

As Toh et al. [26] state in their work, dexterous grasping and telemanipulation tasks frequently focus on precise control of fingertips in a plane surface. They directly map fingertip positions from the touchscreen to a plane in the working space. As this approach seems interesting it shall also be implemented within this thesis.

Heavy use of the BioIK service is made for this, as not only the position of the palm will be given to the service as a goal, but potentially up to 5 positions for each fingertip will be passed. For the first approach only 3 fingertips shall be usable by the functionality. To find out which positions in three-dimensional space shall be passed for the fingertips, we first have to define a plane the fingertips shall be placed on in three-dimensional space. This is done using the parametrized form to represent a plane

$$\vec{E} = \vec{b} + t \cdot \vec{e_1} + s \cdot \vec{e_2}$$

with $\vec{b}$ being the base vector of the plane and $\vec{e_1}$, $\vec{e_2}$ determining the orientation of the plane from the base. It is $|\vec{e_1}| = |\vec{e_2}| = 1$ and $\vec{e_1} \cdot \vec{e_2} = 0$, which means that on the surface a new two-dimensional coordinate system is *created* with its origin in $\vec{b}$. By assigning the x and y coordinates of a point to t and s, points from a two-dimensional coordinate system can then easily be mapped into three dimensions.

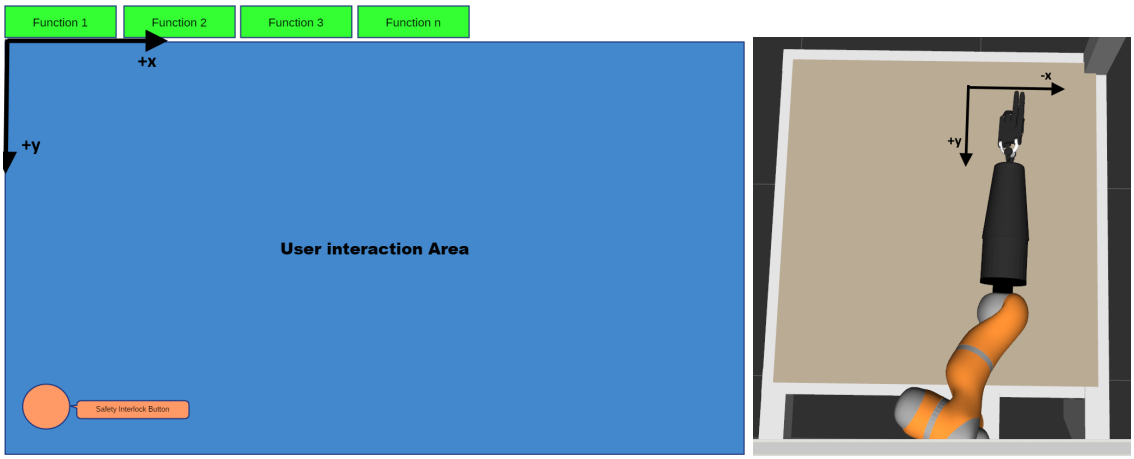
As one unit in the coordinate system of the robot represents one meter in the real world, the coordinates of pointers on the screen shall be calculated in meters of distance to the origin of the screen (which usually is the top-left corner). The position of a pointer is known in pixels and the screen resolution is known in dots per inch (DPI). To transform a pixel value p into meters on a screen with a resolution of r DPI, the calculation

$$p_m = \frac{p}{r} \cdot \frac{2.54}{100} \tag{4.10}$$

has to be made, as an inch (in) is $2.54cm$. The resulting values can then be passed into the equation for the surface to get the corresponding points in the higher dimensional space. These points have to be calculated for each fingertip on the screen, the resulting positions shall then be sent to the BioIK service, resulting in joint angles for every joint on the robot to reach these positions.

If the plane on which the fingertips are mapped shall be changed, only $\vec{e_1}$ and $\vec{e_2}$ have to be changed in a way that they span another plane, e.g. a tilted one. Also, to move the

Figure 4.9: *Comparison of the coordinate systems between which shall be mapped. Left: Tablet, Right: Arm coordinate system*



fingertips in space (e.g. to drag an object in one place and put it to another) no more work has to be done than changing $\vec{b}$.

For first tests, the vectors defining the plane are chosen as

$$\vec{b} = \begin{pmatrix} 0.1 \\ -1.1 \\ 1.2 \end{pmatrix} .$$

This value is relatively straight ahead of the arm and about 15-20cm above the table. This point represents the top-left corner of the touchscreen. Fingertips are mapped using this point as a base in the coordinate system where one unit is one meter. This following two spanning vectors are chosen accordingly.

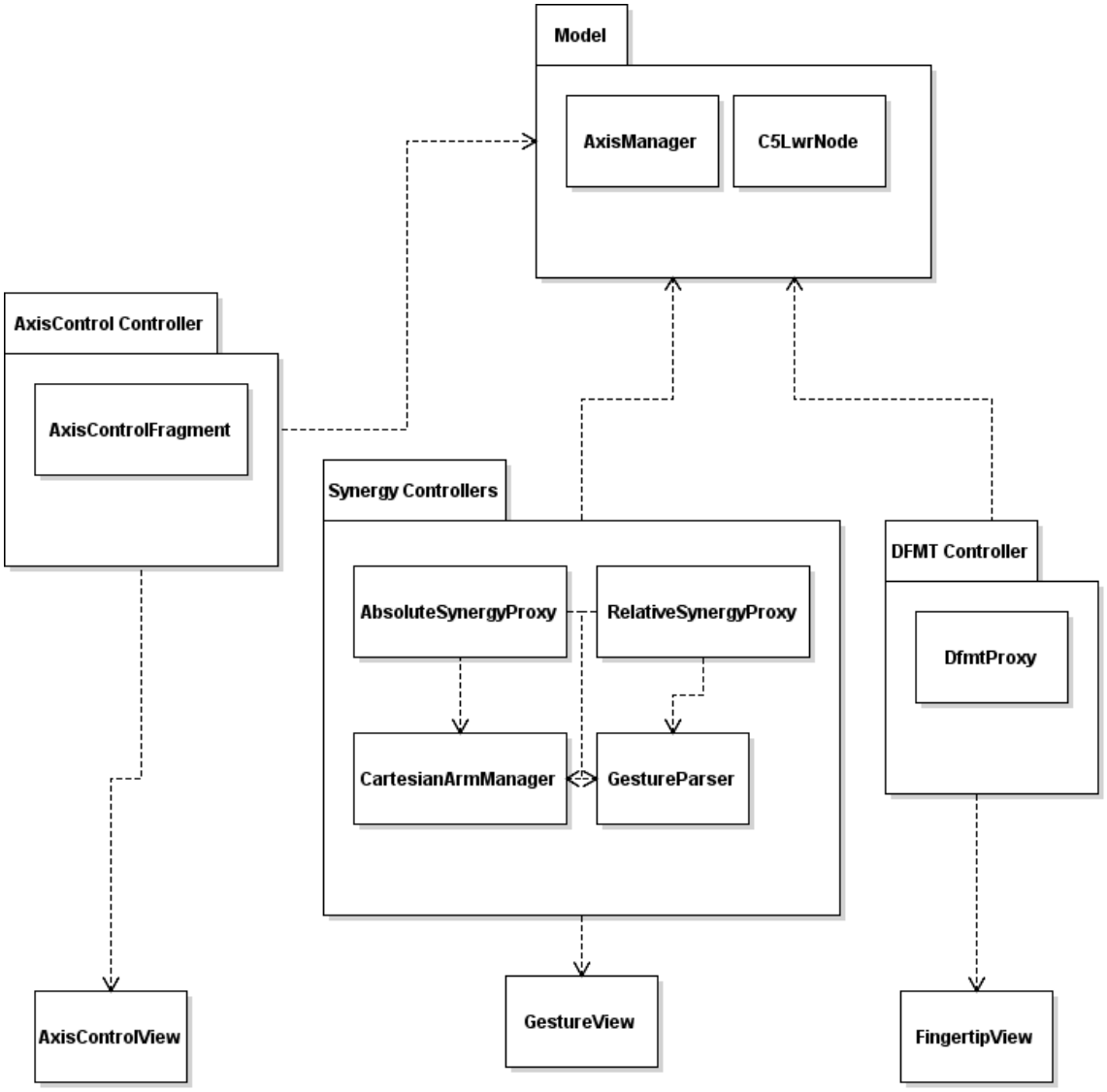
$$\vec{e_1} = \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix} \qquad \vec{e_2} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

The x value rises to the left for the coordinate system of the robot, but to the right for the screen of the tablet, which is why the first component of $\vec{e_1}$ is negative. The y coordinate of the robot's coordinate system rises to the bottom, so does the y coordinate of the touch screen, the corresponding component in $\vec{e_2}$ is therefore positive (See Figure 4.9 for a visual comparison of the coordinate systems).

4.5 Software Architecture

Within this section the software architectural design will be discussed. The goal is to create a software design which is easily extensible by further work, making the software a basis for future research. To accomplish this, well-known design patterns are used. The

Figure 4.10: *General Model-View-Controller separation*



three most commonly used patterns in the software are:

- Model-View-Controller (MVC)
- Observer
- Singleton

The use of all three is described by examples from the overall software design. Eilebrecht and Starke [8] give a good overview over existing patterns and their use-cases and implementation in their book, which is where the information about patterns used in the following sections was mainly taken from.

4.5.1 Model-View-Controller (MVC)

Model-View-Controller is a pattern widely used in applications where a set of data (e.g. data from a database) shall be displayed and modified from within multiple interfaces (so-called views)[8]. To accomplish this, a *Controller* is set in between the data (*Model*) and the interface implementation (*View*). This *separation of concerns* makes it easy to extend or replace one of the three modules without directly affecting the others. A controller gets data from the *Model*, prepares it for display and passes it on to the *View*, where the concrete display of the data is then defined. The Controller gets feedback from the view, which may then be converted into actions affecting the Model again. Within the developed app design, all three modules can be found, although not always a single class can be assigned for one module. Figure 4.10 gives a rough overview of how the different functionalities are distributed between classes.

Model

The Model, which in this case is the data about the robot and its joints, is accessed using the *AxisManager*. The actual communication is done within the *C5LwrNode* class, encapsulating all ROS-specific calls and hiding away the actual communication layer from the rest of the application. Having chosen this design, it is relatively easy to replace either the joint data management or the communication layer to the robot.

The *AxisManager* offers functionality to get or set information for each axis of the robot. It gives interface methods to

- Get or set the current angle
- Enable or disable continuous movement

of each joint. It is lockable, meaning that - when locked - no axis updates are accepted by callers, stopping all movements of the robot independent from other controllers. This functionality is important for the safety interlock button described in Section 4.1.2.

Angle representation within the *AxisManager* is in degrees, whereas the communication with ROS takes place in radians. Conversion between those two has to be made in all actions. To make this conversion functionality exchangable, an interface is declared called *ValueConverter*, allowing to define and assign each joint a different converter. The only implementation used in this application, however, is the *AngleRadianConverter*, converting angles from degrees to radians and vice versa.

The *C5LwrNode* class is a *rosjava* node offering all ROS-specific interfaces to the application. Joint angles are received by the node and passed to the *AxisManager* over an implementation of the *Observer-Pattern* (see Section 4.5.2). It also receives joint angles over the observer pattern (by *AxisManager*), passing them on to the robot over ROS. Additionally the BioIK service calls are implemented within the node. This on the one hand creates the need for controller classes to call more than only the *AxisManager*, but encapsulates all ROS specific calls to one class, which was assumed as the smaller drawback

here. The conglomerate of *AxisManager*, *C5LwrNode* and all helper classes represents the *Model* within the software design.

View

The *View* modules are represented by classes implementing the *View* class from Android. They display data either directly from the *AxisManager*, like the axis control page (*AxisControlFragment*) or states of e.g. touch gestures, like *GestureView*, which is used in the pages implementing the grasp synergy control functionality. The type of display is very different, specific implementations are described in section 5.2 in a more detailed manner.

As the most important feedback to the user should be the action of the operated robot itself, the view modules are not as elaborated and specifically designed as the other two. As seen in Figure 4.10, the view usually consists of a single class, whereas *Model* and *Controller* are deeper partitioned and described.

Controller

Multiple types of controllers exist in the application. There are controllers that take gestures as an input and provide joint angles as an output (*SynergyProxies*, *AbsoluteSynergyProxy* and *RelativeSynergyProxy* for absolute and relative synergy control) and ones that take gestures as an input and output joint angles for the arm, but by using the ROS functionality offered by the *C5LwrNode* (*CartesianArmManager*). As these classes are coupled relatively strong, they can be seen as one controller divided into multiple sub-controllers. Also, the touch parsing functionality used by both synergy approaches is encapsulated into its own class (*GestureParsing*), taking touch input data from the *View* and outputting gesture information.

For the direct fingertip mapping approach, the number of classes is small compared to the other approaches. The *DfmtProxy* takes care of parsing the touch data provided by its corresponding View (*FingertipView*), requesting joint angles from the *C5LwrNode* and passing them on to the *AxisManager*.

4.5.2 Observer

Multiple classes have to exchange data between each other and have to be able to report data changes at arbitrary times. To prevent the occurrence of circular dependencies between classes a variation of the *Observer-Pattern* is used within the application. The *Observer-Pattern* is used to give loosely coupled classes the ability to unidirectionally give notifications to each other about data changes[8].

There are two main uses of the *Observer-Pattern*: *SynergyProxyBase* (The base class for the absolute and relative synergy proxies) implements the *GestureObserver* interface to get notified about gesture changes from the *GestureParser*. Figure 4.11 shows the dependencies between the different classes and interfaces within this pattern as used here.

Figure 4.11: *Observer-Pattern for GestureParser*

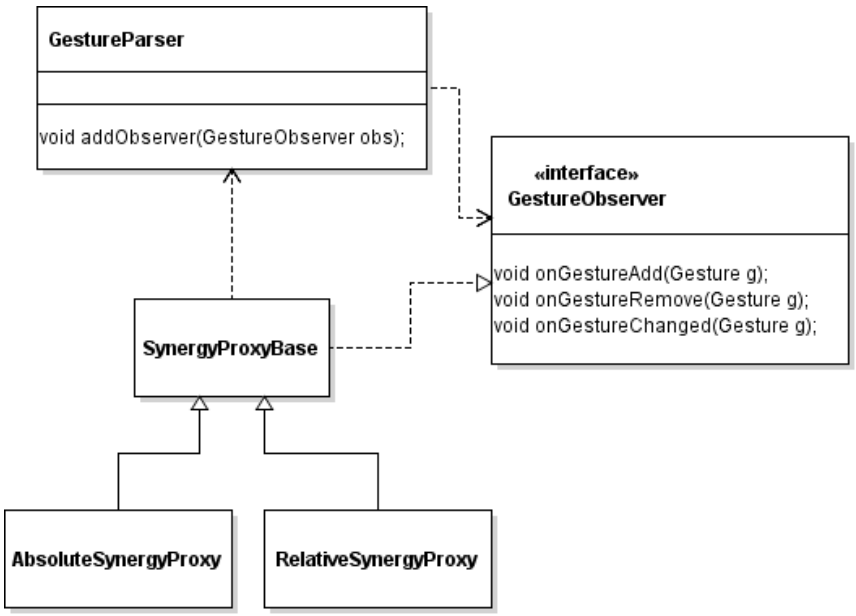
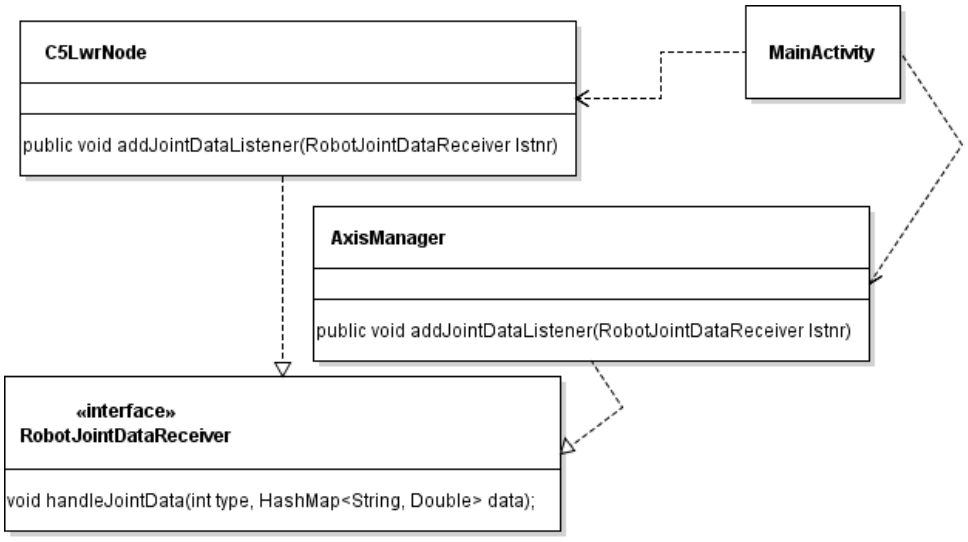


Figure 4.12: *Observer-Pattern for AxisManager and C5LwrNode*



AxisManager and *C5LwrNode* have to exchange joint data between each other in both directions. To accomplish this, both implement the *RobotJointDataReceiver* interface (see Figure 4.12. The design here is relatively far away from the original observer pattern, as both classes do not know of each other. However, the common interface is used to notify each other about changes in the joint angles (either to send them to the robot or as they are received from it). The connection between both classes is made in the entry point of the Android application, *MainActivity*. As *MainActivity* manages the overall application set-up, it is the most suitable point to do so. If one or both of the classes were exchanged in later development, the only place that had to be changed to alter the connection would also be *MainActivity*.

Aside from the original *Observer* pattern described in [8] multiple interfaces are described for observers to directly pass data to them. Following the conservative approach, observers would just be notified about changes, giving them the necessity to fetch them from the observed object on their own.

4.5.3 Singleton

As several classes exist of which only one instance is needed (and sensible to be created) within all of the functionality of the application, the *Singleton-Pattern* is used to ensure that only one instance exists. The singleton pattern is implemented by creating a private constructor which is called from a static method on the desired class. The created instance is saved into a static field and then returned on all subsequent calls. An example implementation of the pattern (as taken from [8]) can be found in Listing 4.1.

The *Observer-Pattern* is implemented by the following classes:

- *AxisManager*, as it is important that all axis actions (and especially the safety interlock) are existent and performed just once over all of the application. Multiple instances of the *AxisManager* may interfere with each other and thus cause unpredictable behaviour of the overall set-up.
- *CartesianArmManager*, as it holds the current position of the arm in Cartesian coordinates. For the same reasons as with *AxisManager*, having multiple instances (and states) of this class available in the application would not be sensible – or even dangerous!

Only one instance exists during the runtime of other classes, too, like the *C5LwrNode*. As for the above classes, having multiple instances available could cause unexpected and unwanted behaviour and thus it has to be ensured only one instance is created. However, with the node implementation ensuring this is not as easy as using the *Singleton-Pattern*, because multiple parameters have to be passed to the class on creation. In this case, *MainActivity*, as the main entry point for the application, takes care of the node's instance, passing it to all classes that need access to it. The same applies for all the user interface specific classes (mainly the *Fragment* classes, offering the different tabbed views). Here,

no singleton pattern can be used by Android's restrictions, so *MainActivity* takes care of these, too.

Listing 4.1: *Example implementation of the Observer-Pattern*

```
class Singleton {
  private static Singleton instance = null;

  public static Singleton getInstance() {
    if(instance == null) {
      instance = new Singleton();
    }

    return instance;
  }

  private Singleton() { }
}
```

5 Implementation

5.1 Preparations

5.1.1 Setting up a Virtual Machine

For all developments within this thesis a virtual machine was used. This makes it easy to reproduce the environment within the labs at the TKRN group as well as having a portable development solution isolated from the rest of the computer's operating system. As the ROS version called *Kinetic* is widely used within the set-ups around the robot, I will also develop the application using this version. This reduces the risk of incompatibility issues during development. ROS *Kinetic* is available as packages for Ubuntu up to version 16.04[19], which is why we install this version of Ubuntu within a new virtual machine. Enough virtual hard disk space and memory is assigned to the virtual machine (200GB HDD, 8 GB RAM) as well as 4 processing cores. This set-up should be sufficient for all purposes during this thesis.

If the virtual machine shall run ROS nodes which have to be accessible by ROS nodes outside the machine itself (i.e. the Android tablet running the control application) the network interface of the virtual machine should be configured as a bridged network connection. This lets the network's DHCP (if present) assign the virtual machine its own IP address reachable from the network. However, this was not possible within the university's network, as Oracle VirtualBox was not able to create a working bridged network adapter using the computer's Wi-Fi connection. During development within the lab another computer directly connected to the university network was used to run *roscore*.

5.1.2 Setting up ROS

Installing ROS

Setting up ROS *Kinetic* within a fresh Ubuntu 16.04 installation is fairly simple. First, the ROS Aptitude-repository has to be added to the packages sources file and the corresponding key has to be added to the key storage to enable downloading the packages. Aptitude is the package and dependency-manager used in Ubuntu. Once this is done, the package *ros-kinetic-desktop-full* can be installed which will download and install all available packages for ROS *Kinetic*.

The commands to install ROS are denoted in Listing 5.1. After these commands have been executed in a terminal window ROS is readily installed.

Listing 5.1: *Commands for installing ROS[19]*

```
1  sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(
       ↪ lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.
       ↪ list'
2  sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --
       ↪ recv-key 421C365BD9FF1F717815A3895523BAEEB01FA116
3  sudo apt-get update
4  sudo apt-get install ros-kinetic-desktop-full
```

ROS is by default installed to */opt/ros/kinetic/*. To make use of all available command line tools provided by ROS it is important to load the file */opt/ros/kinetic/setup.bash* into the currently open (bash)-command-prompt. This is either temporarily done by issuing

Listing 5.2: *Temporarily loading the ROS environment into bash*

```
1 source /opt/ros/kinetic/setup.bash
```

or permanently by adding this line to the file *∼/.bashrc* by executing the following command:

Listing 5.3: *Permamently installing the ROS environment into bash*

```
1 echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
2 source ~/.bashrc
```

When this is done, ROS is completely set up on the development machine.

Setting up a Catkin Workspace

Catkin is a build system and workspace management utility provided with ROS. It supports developers to create, develop and build packages for ROS applications. To create a catkin workspace within the current user's home directory, issue the commands from Listing 5.4 after setting up ROS and sourcing the *setup.bash*-file. The instructions to set up catkin are taken from [20].

Listing 5.4: *Setting up a catkin workspace*

```
1 mkdir -p ~/catkin_ws/src
2 cd ~/catkin_ws/
3 catkin_make
4
5 source devel/setup.bash
```

ROS and catkin are now fully set up and can be used for further development.

5.1.3 Installing Android Studio

Android Studio is used as IDE during development of this thesis and should be installed according to the official documentation[1]. It is sensible to add the *bin* directory within

[1] https://developer.android.com/studio/install.html

Figure 5.1: *Needed Android SDKs*

Android Studio's installation path to the *PATH* environment variable to make Android Studio accessible by just typing *studio.sh* into a terminal window.

After Android Studio was installed successfully, it is important to select and install the correct Android SDK versions as the project will compile with the Android 7 compiler to work with Android 4. To do so, open The SDK Manager (*Tools* > *Android* > *SDK Manager*) and select the SDKs according to Figure 5.1. When this is done, Android Studio is set up to develop and compile the application.

5.1.4 Modifying and Compiling Rosandroid

Since the application developed in this thesis shall work on Android from versions beginning at 4.0.3 we have to modify the rosandroid code on one little detail to make everything work fine. In the created catkin workspace, go to the *src* folder and clone the rosjava and rosandroid repositories there:

Listing 5.5: *Cloning the rosandroid and rosjava repositories*

```
1 git clone https://github.com/rosjava/rosjava_core.git
2 git clone https://github.com/rosjava/android_core.git
3 git clone https://github.com/rosjava/rosjava_messages.git
```

Then line 38 in the file

```
android_core/android_10/src/org/ros/android/RosActivity.java
```

has to be replaced by

Listing 5.6: *Change to make to RosActivity.java*

```
38 public abstract class RosActivity extends android.support.v7.app.
     ↪ AppCompatActivity {
```

This gives us the ability to use the already-built features in rosandroid like the automatically displayed activity to connect to a ROS master and built-in node handling even

in older Android versions. When changes are made, issue a *catkin_make* command in the catkin workspace's root directory. *Rosjava* and *rosandroid* will then be built from source and deployed to a *Maven*[2] repository from where the binaries will be loaded by Android Studio on compile time.

5.1.5 Starting the Environment

To start up the development environment with the ROS master, the BioIK service and the *rviz!* simulation of the robot, first the *tams_cml*[3] and *bio_ik_service*[4] packages has to be cloned into the catkin workspace, as well as the *tams_multitouch* package, which has to be copied into the workspace. After *catkin_make* was executed successfully, the programs are ready to be started. The following commands have to be entered in this order, but within different terminal sessions:

Listing 5.7: *Commands to start up the development environment*

```
1 roscore
2 roslaunch tams_multitouch demo.launch
3 roslaunch tams_f329 4_moveit.launch
4 rosrun bio_ik_service bio_ik_service
```

If interaction with the robot hardware is wanted, the corresponding programs and nodes have to be started according to the file *tam_cms/tams_f329/README.txt* within the catkin workspace.

5.2 User Interface

The user interface of the application was developed according to the considerations made in Chapter 4. Additionally, it turned out during development, that a basic tele-operation screen for the robot arm would be useful, that enables the user to bring the arm into a defined home position as well as doing simple step-wise manipulation to the robotic arm by moving the desired position of the hand palm by single small steps per button-press. The screen's layout and functionality is described in Section 5.2.4.

The safety interlock button on all screens is immplemented using a *FloatingActionButton*, a predefined control by Android which is designed to float in one corner of the screen above the rest of the screen's contents. To have the *FloatingActionButton* work in the expected way, all screen contents have to be embedded into a *CoordinatorLayout* container. The icon of the button has a *Play* symbol in idle state, in activated state is shows a *Pause* symbol until the button is released. The code to make the interlock button is described in Listing 5.8. It has to be inserted into the overridden *onStart()* method in every Fragment

[2]Maven is a dependency and package management system for Java libraries.
[3]https://gogs.crossmodal-learning.org/norman.hendrich/tams_cml
[4]https://gogs.crossmodal-learning.org/philipp.ruppel/bio_ik_service

of the application, in which the functionality shall exist - i.e. every page with controls for the robot.

Listing 5.8: *Code for the interlock button*

```
@Override
public void onStart() {
  super.onStart();

  final FloatingActionButton lockButton = ((FloatingActionButton)
     ↪ getView().findViewById(R.id.lockButton));

  lockButton.setOnTouchListener(new View.OnTouchListener() {
    boolean locked = true;

    @Override
    public boolean onTouch(View view, MotionEvent motionEvent) {
      switch(motionEvent.getAction())
      {
        case MotionEvent.ACTION_DOWN:
                        // Code to unlock robot operations
            lockButton.setBackgroundTintList(ColorStateList.valueOf(
               ↪ getResources().getColor(R.color.posOk)));
            lockButton.setImageResource(android.R.drawable.
               ↪ ic_media_pause);
        break;

        case MotionEvent.ACTION_UP:
          // Code to lock robot operations
          lockButton.setBackgroundTintList(ColorStateList.valueOf(
             ↪ getResources().getColor(R.color.posNOk)));
          lockButton.setImageResource(android.R.drawable.ic_media_play)
             ↪ ;
        break;
      }
      return true;
    }
  });

  // ... more code for onStart()
}
```

F

Figure 5.2: *The synergy control screen*

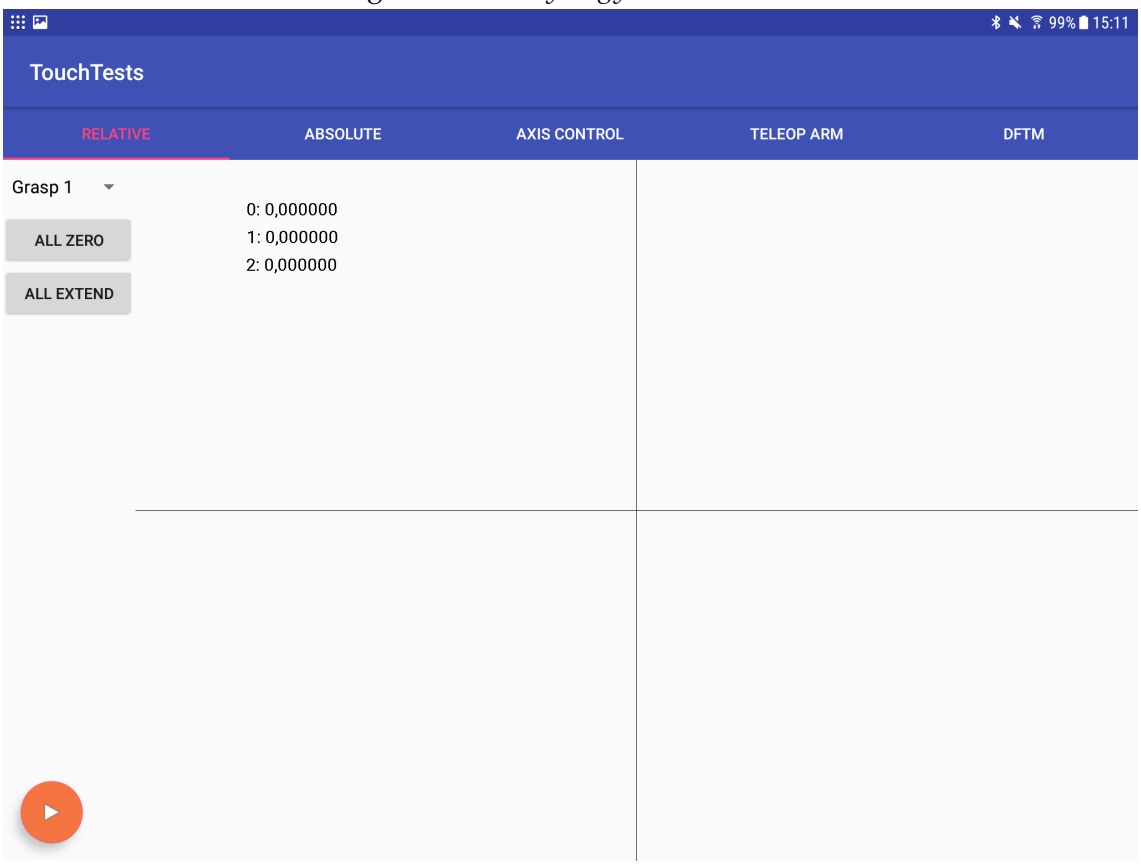
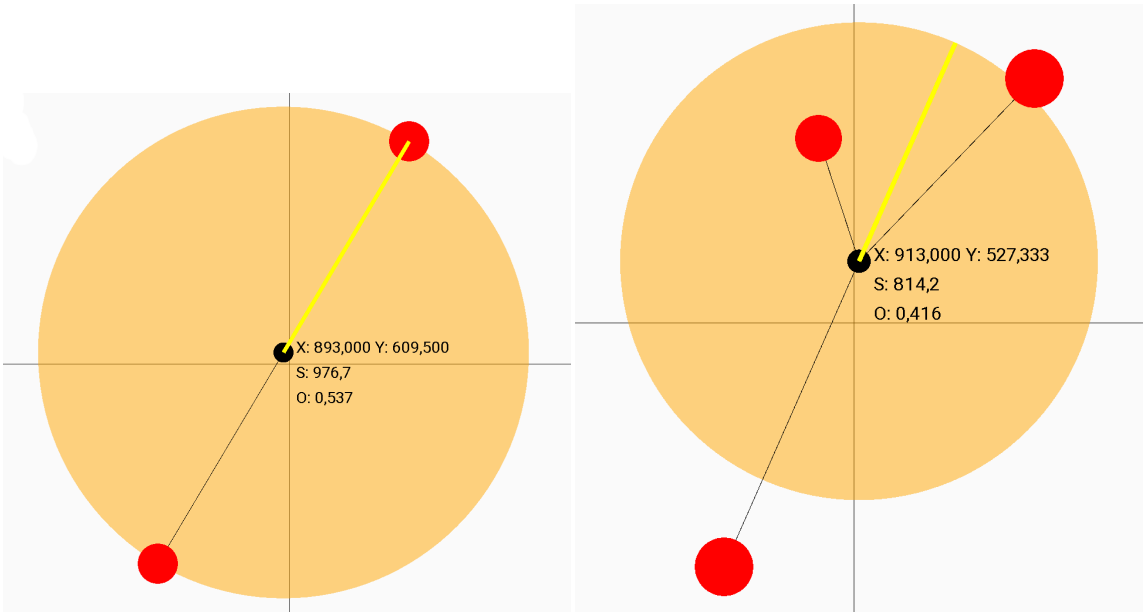


Figure 5.3: *display of a two-pointer and a three-pointer gesture on the synergy screen*



5.2.1 Synergy Pages

The synergy pages are implemented as a mostly blank white page, with a small drop-down control on the left to select the synergy that shall be controlled, as well as two buttons to set all amplitudes to a known state (i.e. either all zero or all 50). The values of all three controllable amplitudes are displayed in the upper left corner of the control area. The screens for absolute and relative control look the same, which mode is active can be seen in the upper bar with the tab controls. Two lines determine the middle of the control area, which is important for the absolute approach, as the absolute placement of a gesture is important. Figure 5.2 gives an impression of how the screen looks on the tablet computer.

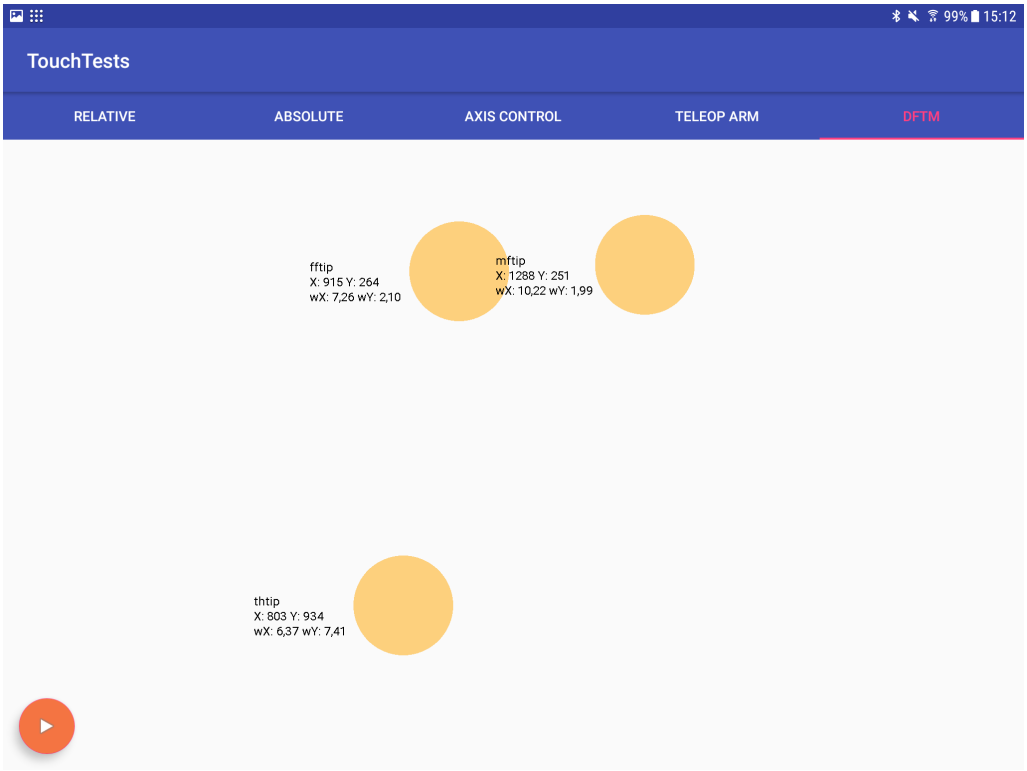
Figure 5.3 shows how a two-pointer and a three-pointer gesture is displayed on the synergy pages. While each pointer is marked by a red circle, the center (i.e. the position) of a gesture is displayed as a small black circle, with all pointers being connected to the center by a black line. The orange circle gives an impression of the calculated size of a gesture while the yellow line within the orange circle points in the direction of the orientation which was calculated for a gesture. The calculated values are also displayed in clear text next to the center of a gesture. This is done mainly for debugging purposes, but may also give an interesting insight into the state of a gesture, for example for training purposes. Note that the orientation is not given in degrees, but in radians.

5.2.2 Direct Fingertip Mapping (DFTM) Page

Figure 5.4 gives an overview of how the DFTM page looks like. It has even less contents than the synergy pages, as no selection has to be made for the current implementation of the DFTM approach. In later iterations it would be sensible to add controls to move the base of the current workspace on which the fingertips are mapped. As this is not implemented within this thesis, no such controls are displayed. In the same figure, an example of how touch pointers are displayed is given for three points. Next to each pointer the name of the link controlled by this pointer is displayed, as well as the coordinates in screen coordinates (i.e. pixels) and world coordinates (i.e. centimeters), both originating in the top-left corner of the white control area. As described more detailed in Section 5.6, the pointers are assigned to the links they control in the order in which they are placed on the screen. If a finger is lifted from the screen (i.e. the touch pointer is de-registered by the Android operating system), one of the following two actions will be performed:

- If a pointer is removed from the screen and another pointer still is on the screen, which was added later, the removed pointer is marked as *not present*. It is still included in IK requests while its position does not change.
- If no such pointer exists, i.e. the lifted pointer was the last present pointer in order of occurrence, it is removed and all pointers laid down after the current one but marked as *not present* are also removed and not included in IK requests anymore.

Figure 5.4: *The DFTM screen with three pointers*



A pointer which is marked as *not present* is displayed as an unfilled circle with a black border (refer to Figure 5.5 for an example). Once the user places a finger within the black circle, it is registered as this pointer again and the pointer is marked as *present*.

5.2.3 Axis Control Page

The axis control page consists of many *single axis controls* (one for each axis or joint available in the robot, see Figure 5.6), which – as defined earlier – possess two buttons, one to increase and one to decrease the position of the axis or joint. Between those two buttons the target value is displayed (on the top in bold), as well as the currently measured value as received from the robot (in the bottom). The colour between the buttons indicates the magnitude of difference between the target value and the currently measured value as described in Section 4.1.5. Axis control widgets for axes that are not controllable (i.e. the first one for every finger except the thumb) have their buttons greyed out and are thus only there to display the current value of the axis.

Two extra buttons are placed on the screen, one labelled *Stop* and one *All Zero*. These buttons are mapped to functionality explained in Section 5.4.6. The former sets all target values to zero, while the latter copies all currently measured values into the target value, causing the robot to stop all movements.

Figure 5.5: *The DFTM screen with three pointers, of which one is currently not laid on the screen*

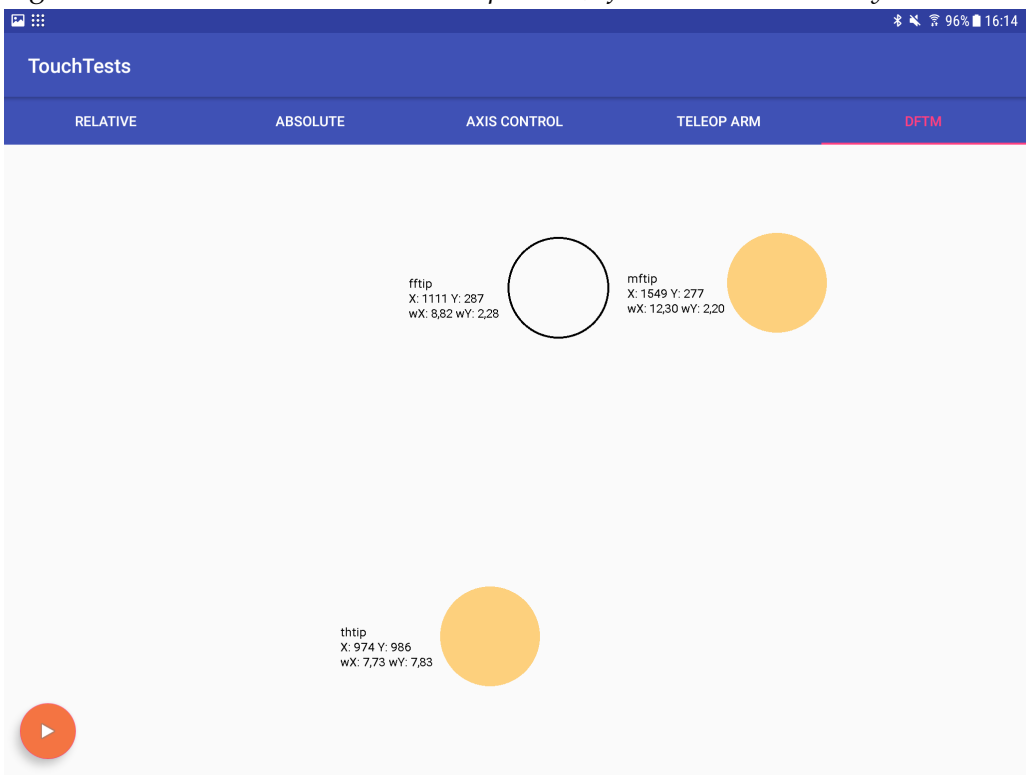
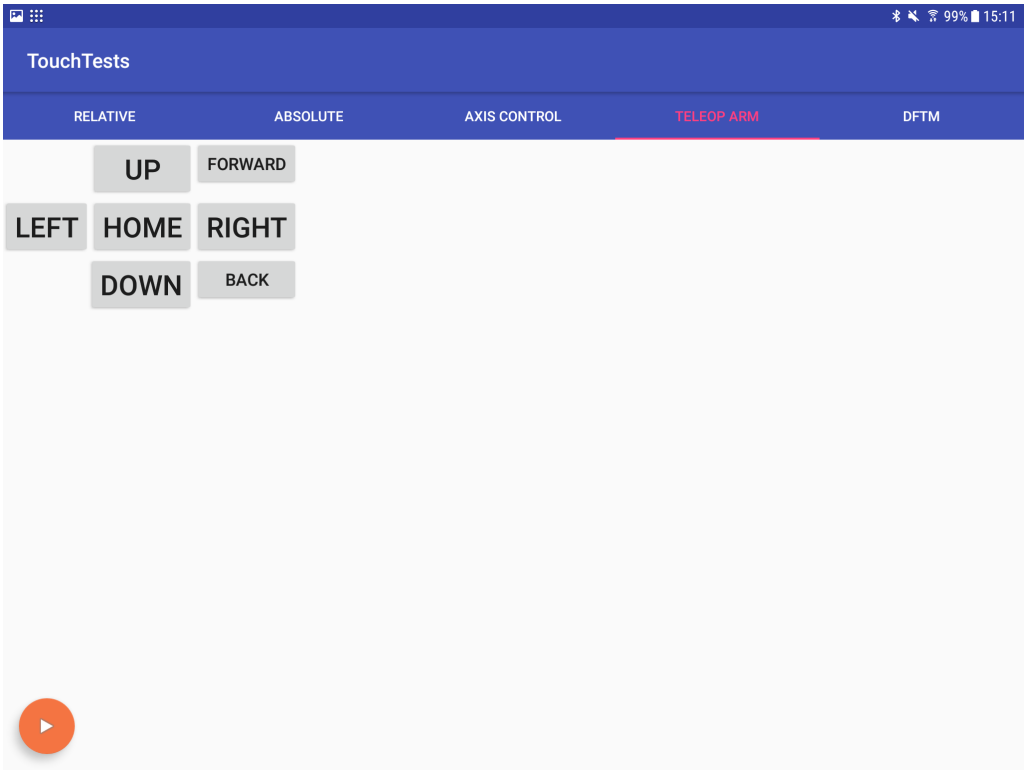


Figure 5.6: *The axis control page*

Figure 5.7: *The arm tele-op page*



5.2.4 Arm Tele-Operation Page

As a last page, the arm tele-op page serves as a remote control to move the arm in Cartesian space. The functionality of the *CartesianArmManager* is described in Section 5.5.2. It is important to note that the wording *left*, *right*, *forward*, *backward* are relative to the operators point of view, as standing in front of the robot. Each button press moves the palm of the robot by one centimeter into the desired direction. A press on the *Home* button brings the robot into a home position, which is located directly in front of the robot at the border of the table approximately 20cm above the plate. This page was implemented for testing reasons, but comes in handy when using the synergy approaches, as the arm should be moved into the *Home* position first before using the gesture control. Doing this using this page is easier than moving every joint on its own using the axis control page. Figure 5.7 gives an impression of the page.

5.3 ROS integration

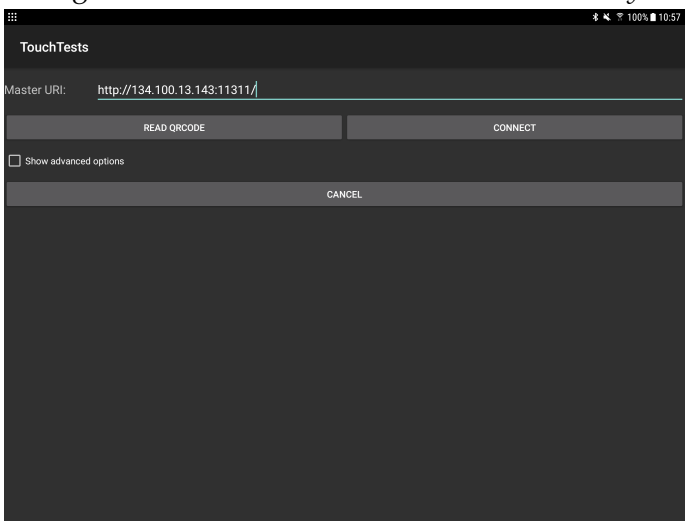
Thanks to *rosandroid* it is easy to integrate ROS into an Android application. When the libraries are included in the project the main thing to change is that the main activity has to inherit from *RosActivity* instead of the plain Android *Activity* class.

Listing 5.9: *Changes to MainActivity*

```
1 public class MainActivity extends RosActivity {
```

```
2   //...
3 }
```

Figure 5.8: *The rosandroid master chooser activity*



After this change was made, the *RosActivity* implementation handles multiple tasks, beginning with showing a *master chooser* activity, in which the user can connect to a ROS master node, to handling all connection lifetime events of ROS parts. The *master chooser* activity (see Figure 5.8) lets the user connect to an existing ROS master node. Additionally, it gives the Android application the opportunity to create a dedicated master node on the device itself. As this could affect the overall performance of the application and a significant amount of functionality has to be run on a more powerful machine, the ROS master is started on a dedicated computer and the Android application connects to this existing master node.

RosActivity is an abstract class. To implement it, the *init()* method has to be overriden by inheriting classes. This method is called once the connection to the ROS master node was established and custom nodes can be initialized and connected. All other initialization steps regarding the implemented ROS nodes should also be done here. As shown in Listing 5.10, first an instance of the *C5LwrNode* is created and then assigned to all instances that consume its functionality[5] (*AxisManager*, *CartesianArmManager*, *DfmtProxy*). After all this is done, the node is registered with the ROS master node. Details on the initialization of the *C5LwrNode* node can be found below. In theory, multiple nodes can be started and registered with the master within one application.

Listing 5.10: *Initialization of the ROS connection*

```
1 @Override
2 protected void init(NodeMainExecutor nodeMainExecutor) {
3   axisManager = AxisManager.getInstance();
4
5   node = new C5LwrNode("/joint_states", "/hand/joint_goals", "/lwr/
      ↪ jointPositionGoal");
6   node.addJointDataListener(axisManager);
7
```

[5]Information on how to initialize ROS applications was taken from an official rosandroid example found at `https://github.com/rosjava/android_core/blob/kinetic/android_tutorial_pubsub/src/org/ros/android/android_tutorial_pubsub/MainActivity.java`

```
8   axisManager.setRobotNode(node);
9   CartesianArmManager.getInstance().setNode(node);
10  DftmProxy.getInstance().setNode(node);
11
12  NodeConfiguration cfg = NodeConfiguration.newPublic(getRosHostname(),
        ↪ getMasterUri());
13  nodeMainExecutor.execute(node, cfg);
14 }
```

5.3.1 The C5LwrNode Class

The class *C5LwrNode* inherits from *AbstractNodeMain*, which already offers the very basic lifetime functionality of a ROS node. Some methods have to be implemented by the developer, like *getDefaultNodeName()*, which determines the name of the ROS node as it is registered with the ROS master. In the *onStart()* method, all start-up procedures are implemented, like registering topic subscriptions as well as creating publishers and *service clients*. Within the *onShutdown()* method, all resources created before (subscribers, publishers, service clients) shall be closed and deleted to ensure a clean de-registration from the ROS master and a clean shut-down of the application. The start-up code for the ROS node can be seen in Listing 5.11.

Listing 5.11: *Startup of the C5LwrNode*

```
1  @Override
2  public GraphName getDefaultNodeName() {
3    return GraphName.of("ba_android/c5lwrnode");
4  }
5
6  @Override
7  public void onStart(ConnectedNode connectedNode) {
8    cNode = connectedNode;
9    handJointStatePub = connectedNode.newPublisher(handPublishTopic,
         ↪ JointState._TYPE);
10   armJointStatePub = connectedNode.newPublisher(armPublishTopic,
         ↪ RMLPositionInputParameters._TYPE);
11
12   jointStateSubsc = connectedNode.newSubscriber(subscribeTopic,
         ↪ JointState._TYPE);
13   jointStateSubsc.addMessageListener(/* ... */);
14
15   try {
16     ikService = connectedNode.newServiceClient("/bio_ik/get_bio_ik",
           ↪ bio_ik_msgs.GetIK._TYPE);
17   } catch (ServiceNotFoundException e) {
18     ikService = null;
19     e.printStackTrace();
```

```
20   }
21 }
```

The methods that are used to offer the functionality of the *C5LwrNode* are denoted in Listing 5.12. Because arm and hand joints have to be published to different topics, the *handleJointData()* method calls either *publishHand()* oder *publishArm()*, depending on the parameter *jointType* which is passed by the caller indicating the type of the joint data given. The two methods to request IK solutions from the BioIK service are relatively similar, as both initialize the request with the current robot state passed as a parameter and add a *MinimumDisplacementGoal* to communicate to the IK solver that a solution is desirable where the least possible movement in all axes is done. The number of attempts is set to 1, the time-outs to find a solution are set to 10*ms* for the palm position and 500*ms* for the fingertip positions. While for the palm only one *PoseGoal* is added, containing the desired pose of the palm constructed by the x, y, z position and *Quaternion* rotation as passed by the caller, in the method to get a solution for multiple fingertips one *PositionGoal* is added for each fingertip as well as an *OrientationGoal* to have the palm of the robotic hand always point in the same direction. In *GetIKJointsFingertips()*, the *fingertips* parameter contains a map with the link names (e.g. *fftip*, *thtip*...) as key values and the desired 3-dimensional position of the link.

Listing 5.12: *C5LwrNode interface*

```
1  public class C5LwrNode extends org.ros.node.AbstractNodeMain implements
       ↪  RobotJointDataReceiver {
2    private void publishHand(HashMap<String, Double> data);
3    private void publishArm(HashMap<String, Double> data);
4
5    @Override
6    public void handleJointData(int jointType, HashMap<String, Double>
       ↪ data);
7
8    public void GetIkJointsPalm(Map<String, Double> currentState,
9      String[] lockedAxes,
10     double x, double y, double z,
11     double rotx, double roty, double rotz, double rotw,
12     ServiceResponseListener<GetIKResponse> hdl);
13
14   public void GetIKJointsFingertips(Map<String, Double> currentState,
15     Map<String, PointInSpace> fingertips,
16     ServiceResponseListener<GetIKResponse> hdl);
17 }
```

5.4 The AxisManager

The most important and most central functionality of the overall application is offered by the *AxisManager* class. It is responsible for holding the current joint angles for all joints in memory, as well as the current target values along multiple other bits of information about each axis or joint. Joints are more generically called *axis* within the *AxisManager*, so this wording will be adopted in the rest of this section.

5.4.1 AxisInformation

All information about an axis is stored in a *AxisInformationImpl* object. This class implements the *AxisInformation* interface, which is returned when axis information shall be given to callers in a read-only manner. The *AxisInformation* interface is given in Listing 5.13. All the information stored about an axis is accessible here. In particular, this is:

- The identifier of an axis, i.e. a string literal containing the name at which The axis or joint is known to the ROS nodes.
- The maximum speed the axis may move at.
- The target value to which the axis shall be currently moved.
- The value representing the axis' current position.
- The value representing the axis' *current target value* (see Section 5.4.4 for details).
- The minimum and maximum values the axis may have as position value.
- flags indicating whether the axis is enabled and moving.
- An integer representing the type of an axis. Allowed types are *JointType.ARM* and *JointType.HAND*.

Listing 5.13: *The AxisInformation interface*

```
1  public interface AxisInformation {
2    String getIdentifier();
3
4    double getMaxSpeed();
5    double getTargetValue();
6
7    double getMaxValue();
8    double getMinValue();
9    double getCurrentTargetValue();
10   double getCurrentValue();
11   boolean isMoving();
12   double getSpeed();
```

```
13   boolean isEnabled();
14   int getJointType();
15 }
```

All the information is held within the *AxisManager*, referenced by the axis identifier. Manipulation of the data is only done through calls to the *AxisManager*, to give it full control about what happens with all axes. The difference between *AxisInformation* and the concrete implementation *AxisInformationImpl* is, that the implementation has a setter function for every property. All calculations are done within the *AxisManager* itself.

5.4.2 AxisManager Timer Tick

The *AxisManager* is designed to work fully asynchronous. All information about axes' target values is stored in the according *AxisInformation* object, but only processed from within the main timer event used in *AxisManager*. The timer is set to a frequency of $f_{am} = 10Hz$. This value can easily be changed by altering the static constant field *UPDATE_FREQ* in the *AxisManager* task. An implementation of a timer is used which gives the ability to schedule an event at a fixed rate. *java.util.Timer* is able to ensure the desired frequency is reached in the long run by slightly alternating the delays between two executions[27]. This is important to have axis movements and publishing done at the correct speed and frequency. To use this functionality, the method *scheduleAtFixedRate()* on the timer has to be used.

All calculations regarding axis movements (Section 5.4.4) are done within the timer tick only. After all calculations have been done the current joint angles are all sent to the *C5LwrNode* to be published over ROS (see Section 5.4.5).

5.4.3 Initialization

When the application starts or is resumed from a sleeping device (i.e. the screen went off), the *AxisManager* is initialized. This means that it blocks all actions until it has received a specific number of joint states from the *C5LwrNode*. This measure was implemented to prevent the application from sending joint data to a non-existent robot and to initialize the joint data within memory with the current state of the robot. After 20 samples (*JointState*s) have been received, the values are copied into the target values for each axis. Initializing all joint target values with 0 is obviously not a good choice, as the robot would then go to this position out of any state it is currently in, causing unwanted movements and behaviour. During initialization a modal dialog is shown, blocking all user interaction with the graphical user interface.

5.4.4 Axis Movements

The main task of the *AxisManager* is managing movement of all axes in a safe manner. To accomplish this, it restricts the movement for each axis to the maximum speed stored

within the *AxisInformation*. Two main modes are available for movement. The first one is by enabling a constant movement of an axis at a specified speed. The second is by setting target values, which the axis will then be moved to at a maximum speed defined on a per-axis basis in the *AxisInformation* object.

Constant Movement

To set an axis to constant movement at a constant speed, the method

```
1 public boolean startMoving(String identifier, double speed);
```

on *AxisManager* can be called. The parameter *identifier* is filled with the string literal identifying an axis, while *speed* indicates the speed at which the axis shall move. Since only rotational joints are present it the used set-up, the speed (as well as the maximum speed defined in *AxisInformation*) is denoted in $\frac{\text{degrees}}{s}$. The movement speed can be given either positive or negative, depending on the direction the axis shall move in. With this function call, only the information that the axis shall move is stored in *AxisInformation*, actual movement takes place in the timer event, in which the movement speed in clipped to the maximum speed defined for an axis and then divided by the frequency of the timer tick f_{am}. In each timer tick event, the position of an axis moving at speed v is altered by $\frac{v}{f_{am}}$. When a constantly moving axis reaches its limit value, the *moving*-flag is not reset, but as the position for an axis is clipped to its maximum and minimum values, no actual movement is done any further. To cancel a constant movement of an axis, simply

```
1 public boolean stopMoving(String identifier);
```

has to be called with the string literal identifying the axis which shall be stopped.

Setting Target Values

The most common use-case in the application is that different parts of the program can set target values for every axis. Instead of simply sending the values received by other parts of the program to the robot over ROS, the axis manager implements a safety feature limiting the movement speeds of an axis at a maximum speed. To accomplish this another variable is introduced in *AxisInformation*, the *current target value*. While the *target value* of an axis determines the desired position where the axis shall be at the end, the *current target value* is the value which is actually sent to the robot. The *current target value* is modified in the timer tick.

To change the target value of an axis, *setTargetValue()* has to be called. The signature of the method is

Listing 5.14: *Signature of setTargetValue()*

```
1 public boolean setTargetValue(
2   String identifier,
3   double value,
```

```
4   boolean force,
5   boolean notifyObservers
6 );
```

A call to this method sets the target value p_t of the axis identified by *identifier* to *value*. If *force* is *true*, the smooth movement mechanism described below is overridden and the current target value p_c is directly set to p_t as well. Setting *notifyObservers* to *true* results in the observers of *AxisManager* being notified about the new target value. As the notification often has UI updates as a consequence, it is sensible to notify observers on setting the last value only, not on changing of every value. For convenience, multiple overloads of this method exist, setting either *notifyObservers* to a default of *true*, or *notifyObservers* to *true* and *force* to *false*.

With p_t being the target value as set in *AxisInformation*, p_c the current target value, v_{max} the maximum speed of an axis and f_{am} the frequency of the main timer tick in *AxisManager*, the procedure to move an axis within the main timer tick is as follows:

First, Δp is calculated, which is the maximum value change within one timer event, thus

$$\Delta p = \frac{v_{max}}{f_{am}}.$$

Second, the *current target value* is updated as follows:

$$p_{c,new} = \begin{cases} p_t & |p_t - p_c| < \Delta p \\ p_c + \Delta p & |p_t - p_c| > \Delta p \wedge p_t > p_c \\ p_c - \Delta p & |p_t - p_c| > \Delta p \wedge p_t < p_c \end{cases}$$

The calculated value $p_{c,new}$ is then stored into the *AxisInformation* for each axis.

5.4.5 Passing Axis Data to ROS

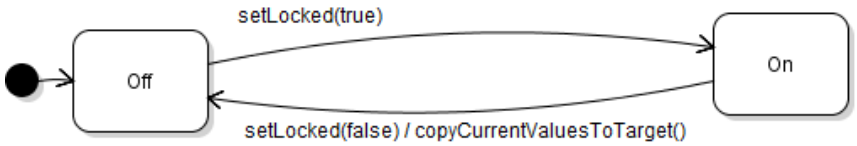
After each execution of the movement processes described in Section 5.4.4, the newly calculated values are published to the ROS nodes controlling the robot arm and hand. As angles for arm joints and hand joints have to be published to different ROS topics, two calls have to be made. The interface *RobotJointDataReceiver*, which is implemented by *C5LwrNode*, contains a method

```
1 void handleJointData(int type, HashMap<String, Double> data);
```

accepting the joint type as the first parameter. The ROS node implementation will choose the topic to publish the data to by the given type identifier.

At the end of the timer event in *AxisManager*, first all *current target values* for arm joints are taken from the list of *AxisInformation*. The angle values are converted to radians and put into a Map, which has the axis identifier as the key and the angle of the corresponding axis as value. *handleJointData()* is then called with *JointType.ARM* and the created map. The same procedure is then executed for all *AxisInformation*s with the type *JointType.HAND*.

Figure 5.9: *Enabling and disabling the AxisManager*



5.4.6 Stopping Movement and Setting All Axes to Zero

Two extra functions are implemented in *AxisManager*. The first,

```
1 public void copyCurrentValuesToTarget();
```

takes all values currently stored as *current value* in *AxisInformation* and copies them into the *target value* and *current target value* fields. This mainly takes the current robot state and copies it into the target state, which causes all movements to stop. This is especially useful when the robot cannot reach a position defined by the target position. This is for example the case when all joints of the hand shall be set to 0°, as some joints are not able to completely reach this value. Copying the currently measured value into the target value stops the robot from trying to reach the actual desired value and, by this, prevents the hardware from being damaged by trying to reach a non-reachable position for too long.

The second special method is

```
1 public boolean setAllZero(boolean force);
```

which sets all axis *target values* to 0. If *force* is *true*, the *current target value* is also set to 0. **Extreme caution has to be used when calling this function!** The robot will move all joints to a position of 0 degrees either immediately (*force* is *true*) or smoothly. The shortest way in joint space from the current state of the robot to $\vec{0}$ may cause damage to the robot or its environment. This function was implemented to bring the robot to a known state using the axis control page. If the corresponding button is pressed, it is called only after the user has stated that he is aware of the possible dangers.

5.4.7 Enabling and Disabling

The *AxisManager* can be enabled or disabled. In the disabled state, all calls to *setTarget-Value()* are discarded. Upon disabling the *AxisManager*, all current measured values of all joints are copied to the *current target value*, causing the robot to stop all movements (see Figure 5.9). The function used to enable and disable the *AxisManager* is

```
1 public void setLocked(boolean locked);
```

This function is designed to be called upon touch actions on the safety interlock button on the user interface pages (see Section 4.1.2).

5.5 Grasp Synergies

The grasp synergy approach is implemented according to the concepts described in Section 4.3. The approaches are implemented using a gesture page for both relative and absolute method. Relative and absolute approach are differentiable by the title of the tabbed page selector, apart from that, the pages look the same.

5.5.1 Gesture Parsing

Gesture parsing is separated into multiple classes. The *GestureParser* class accepts touch events redirected to it by the *GestureView*s on the pages for gesture control. It has a method

```
1 public void handleTouchEvent(MotionEvent e);
```

which is called from within the *onTouchEvent()* handler of the user interface element. The *GestureParser* is otherwise independent from the user interface element it is invoked by. When a new pointer is encountered, it is either added to a already present gesture gesture it fits or – if the pointer is too far away from any existent gesture – creates a new gesture. If a pointer is removed from the screen, it is also deleted from the assigned gesture. If no pointers are left within the gesture, it is also removed. If an event is received stating a pointer has moved, the pointer location is updated in memory. Upon all of the described actions, the observers of the *GestureParser* are notified. These observers implement the *GestureObserver* interface, which is shown in Listing 5.15. It gives the *GestureParser* the ability to notify observers about the addition or removal of a gesture, as well as the case in which a pointer of the gesture has changed its position.

Listing 5.15: *The GestureObserver interface*

```
1 public interface GestureObserver {
2   void onGestureAdd(Gesture g);
3   void onGestureRemove(Gesture g);
4   void onGestureChanged(Gesture g);
5 }
```

If the pointer count of a gesture changes, the *GestureParser* calls the *onGestureRemove()* method of observers, and then the *onGestureAdd()* method, both with the same *Gesture* object.

When a gesture is added or its pointer count changes, it is marked as *locked* by the *GestureParser* for the duration of one second. This indicates to dependent classes that the gesture is new and the data should not yet be used for any control purposes, as the user might still adjust finger positions. This also takes care of the fact that, to add a multi-pointer gesture, the android system calls different events for each pointer subsequently: The software first recognizes a one-pointer gesture, then a two-pointer gesture and finally a three-pointer gesture, if three fingers were laid on the touchscreen. By ignoring gesture

input for the first second of a new gesture, the user should have put all fingers onto the screen and can then control the application. Having input by an unwanted gesture may cause unexpected behaviour.

In the following the functionality of the three main material classes *Location*, *Pointer* and *Gesture* is explained.

The Location Class

The *Location* class represents a two-dimensional vector within the program. It has two coordinates, x and y, which can be accessed by *getter methods*. It also offers basic functionality to work with vectors, including addition, subtraction, multiplication with scalars and the scalar-product. All functionality is implemented as expected by common sense. When a mathematical operation is performed using two *Location* objects, the result is returned as a new one, as a *Location* is immutable once it is created.

Listing 5.16: *The public interface of the Location class*

```
1  public class Location {
2    public Location(float x, float y);
3
4    public float getX();
5    public float getY();
6
7    public Location add(Location loc);
8    public Location substract(Location loc);
9    public Location multiply(float c);
10   public Location divide(float c);
11
12   public double scalarProduct(Location loc);
13
14   public double getVectorLength();
15   public double distanceTo(Location loc);
16   public double getAngleTo(Location loc);
17   public Location getTurned(double angleRad);
18   public boolean isSame(Location l2);
19 }
```

As visible in Listing 5.16, multiple advanced operations are also available on *Locations*. *getVectorLength()* returns the length of the vector calculated using the Pythagorean theorem. *getDistanceTo()* is implemented very similar, as it basically calculates the length of the differential vector between the *Location* it is invoked on and the passed second *Location*. Although this could be done by one *substract()* operation and then performing *getVectorLength()* on the result, the calculation is directly implemented here for performance reasons.

Listing 5.17: *Implementation of getVectorLength() and getDistanceTo()*

```
1 public double getVectorLength() {
2   return Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2));
3 }
4
5 public double distanceTo(Location loc) {
6   return (float)Math.sqrt(Math.pow(loc.x - this.x, 2) + Math.pow(loc.y
      ↪ - this.y, 2));
7 }
```

getAngleTo() returns the angle between the *Location* it is invoked on and the one passed as a parameter (Please note that this represents the calculation as defined in Equation 4.5 implemented for arbitrary vectors). The output of this method ranges from $-\pi$ to π, with positive angles meaning a clockwise rotation from the invoked *Location* to the one passed as a parameter. The reader is referred to Listing 5.18 for the implementation of *getAngleTo()*. In Line 4 the determinant of the two combined vectors is calculated and the result is multiplied with -1 if the determinant is negative.

Listing 5.18: *Implementation of getAngleTo()*

```
1 public double getAngleTo(Location loc) {
2   double val = Math.acos(scalarProduct(loc) / (getVectorLength() * loc.
      ↪ getVectorLength()));
3
4   if(x * loc.getY() - y * loc.getX() < 0) {
5     val *= -1;
6   }
7
8   return val;
9 }
```

Lastly, *getTurned()* returns the current *Location* rotated by an angle of *angleRad*. The angle may range from $-\pi$ to π, with positive angles meaning a clockwise rotation. *isSame()* is a numerical comparison of the two vectors. Note that floating point numbers are compared here, on which equality comparisons are problematic. This function is used to check for exactly the same values, meaning probably the same *Location* objects.

The Pointer Class

Pointers are the contents of *Gestures*. They contain of a *Location*, representing their coordinates on the touch-screen and an id, which is their touch-pointer-id as assigned by the Android operating system. This class was basically introduced to semantically separate the pointer id from the location. The id is needed to identify a pointer within the motion event raised by the operating system. A method is provided to update the location of a pointer, in which a new *Location* object is created.

Listing 5.19: *The Pointer class*

```
1 public class Pointer {
2   public Pointer(int id, float x, float y);
3
4   public int getId();
5
6   public Location getLocation();
7   public void setLocation(float x, float y);
8 }
```

The Gesture Class

The *Gesture* class represents a gesture as defined in Section 4.3.1. It is basically a set of *Pointer*s with a set of properties. Its public interface is denoted in Listing 5.20.

Listing 5.20: *Public interface of the Gesture class*

```
1 public class Gesture {
2   public boolean isLocked();
3   public void setLocked(boolean locked);
4
5   public void addPointer(Pointer p);
6   public void removePointer(Pointer p);
7   public int getPointerCount();
8
9   public boolean catchesPointer(Pointer p);
10  public float getCatchRadius();
11  public float getDistanceToCenter(Pointer p);
12
13  public Location getCenter();
14  public float getSize();
15  public double getOrientation();
16 }
```

The first two methods set and query the *locked* state of a gesture, followed by three methods to add and remove pointers, as well as querying the number of currently available pointers in a gesture. Whenever a pointer is added to or removed from a gesture, the *thumb pointer* is evaluated according to the rules defined in Section 4.3.1. *catchesPointer()* determines whether a new *Pointer* can be added to the gesture. This is done by checking whether the *Pointer* is within a distance of 2.5x the size of the gesture around its center. If only one pointer is present in a gesture no size is available. In that case, a size of 1200 is taken as the *catch radius*. *getCenter()*, *getSize()* and *getOrientation()* represent $c(G)$, $s(G)$ and $o(G)$ as defined in Section 4.3.1.

5.5.2 Arm Control

The PointInSpace Class

The *PointInSpace* class implements functionality as a vector in three dimensions, offering basic operations like adding other *PointInSpace* instances and multiplying with scalars. It is not as elaborated as the *Location* class, but extending the functionality according to *Location* could easily be done if needed.

Listing 5.21: *The public interface of PointInSpace*

```
1  public class PointInSpace {
2    public PointInSpace(double x, double y, double z);
3
4    public double getX();
5    public double getY();
6    public double getZ();
7
8    public PointInSpace add(PointInSpace pw);
9    public PointInSpace multiply(double v);
10 }
```

The CartesianArmManager Class

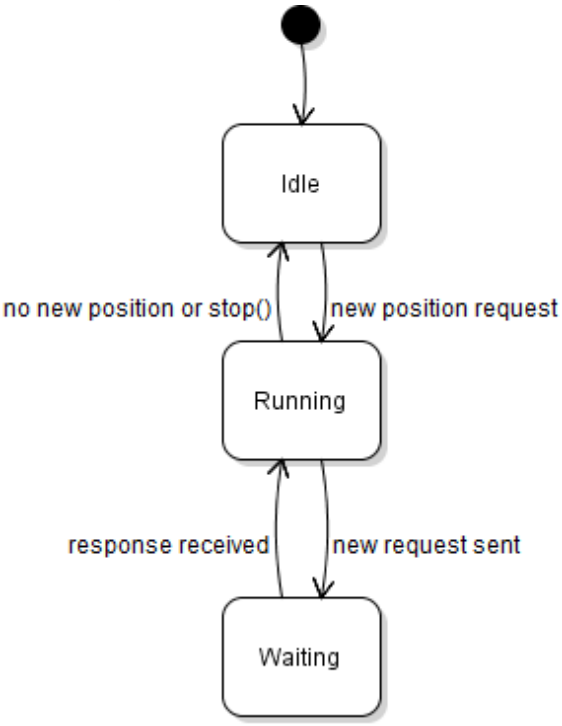
The functionality to move the arm (or better: the palm of the robotic hand) in Cartesian space is provided by the *CartesianArmManager*. It accepts positions for the palm and manages the querying of the BioIK service asynchronously in the background. Once it has received a result from the IK service it is forwarded to the *AxisManager* instance. The *CartesianArmManager* holds a list of all joints that shall not be affected by it, i.e. all joints of the Shadow C5 hand, thus placement of the palm is only managed by moving joints belonging to the Kuka robot arm. It also only updates joints in the *AxisManager* that it shall affect, leaving control of all the other joints to different parts of the software.

Listing 5.22: *The public interface of CartesianArmManager*

```
1  public class CartesianArmManager implements ServiceResponseListener<
       ↪ bio_ik_msgs.GetIKResponse> {
2    public static final double Y_MIN = -1.2;
3    public static final double Y_MAX = -0.8;
4    public static final double X_MIN = -0.2;
5    public static final double X_MAX = 0.4;
6    public static final double Z_MIN = 1.05;
7    public static final double Z_MAX = 1.17;
8
9    public static final int MAX_AXIS_CHANGE = 15;
10
11   public static CartesianArmManager getInstance();
```

```
12
13    public void setNode(C5LwrNode node);
14
15
16    public boolean goHome();
17    public boolean movePalm(PointInSpace offset);
18    public boolean movePalmTo(PointInSpace position);
19
20    public PointInSpace getPosition();
21
22    public void stop();
23  }
```

Figure 5.10: *State diagram for the CartesianArmManager update loop*



Whenever a new target position for the palm is set using *goHome()*, *movePalm()* or *movePalmTo()*, the IK-loop within the *CartesianArmManager* is started. It tries to get solutions from the BioIK service as fast as it can, initiating a new request as soon as one result was received until either no new position was requested or *stop()* was called. Once one of these two cases occur, the loop is stopped and restarted only when a new position is set. Whenever the loop starts a new request, it uses the last set value as target value, values set in between two requests are omitted. Figure 5.10 gives an overview about this process in form of a state diagram.

When a result is returned by the BioIK service it has to go through multiple checks before it is sent to the robot. First, if the *error_code* field has another value than 0, the BioIK solver could not find any solution for the queried robot pose. In this case, the *CartesianArmManager* sends the same request to the BioIK service again so it tries to find another solution again. If the *error_code* field is 0, the joint angles from the solution are compared to the currently measured state of the robot. When one joint is changed by more than 15 degrees, the solution is rejected as unsafe and a new solution for the same request is queried at the BioIK service. This is done to prevent big movements in joint space, possibly causing unwanted and uncontrollable movements of the robot, potentially causing damage to the robot or its environment. The check for high distances in joint space can be bypassed, which is used when letting the robot go into home position, as this specific position most probably has a bigger dis-

tance to the state before than allowed by the check. This means that **when going to the home position, the user has to use caution and observe the robot throughout the whole movement.** Whenever unwanted movement is observed, the user can lift the finger off the safety interlock button to immediately stop any robot movement. As an additional precautionary measure, a hardware emergency switch should be within reach of the operator. When the above checks pass, the joint angles of the arm returned by the BioIK service are written to the *AxisManager* which then sends them to the robot over ROS.

The functionality of the three methods to set new target value is in particular:

- *goHome()* sets the target position of the palm to a fixed position which is known to be safely reachable. In this case, it's $p_{home} = \begin{pmatrix} 0 \\ y_{min} \\ z_{max} \end{pmatrix}$, which is a position located directly in front of the robot at the border of the table.
- *movePalm()* moves the current target position of the palm by *offset* (by adding it to the position returned by *getPosition()*).
- *movePalmTo()* overwrites the target position by *position*.

5.5.3 Grasp Synergies

An implementation of the grasp synergies as well as the recorded data matrices representing the matrix of eigenvectors for each of the different grasp synergies is provided by the work of Bernardino et al.[3]. The functionality is bundled within the *GraspSynergy* class. The data files which can be read by the class methods are included into the application as text file resources. A dataset for one grasp consists of two files, which is one file containing the *mean* value or the offset of a grasp (s_0 in Section 4.3) and one file containing the matrix with the matrix data (S in Section 4.3). Mean files are called *g1mean.txt*, *g2mean.txt* and so on, while the matrix files are called *g1vecs.txt*, *g2vecs.txt* up to *g8vecs.txt*. Originally, these files were named differently when provided by Dr. Norman Hendrich, but due to restrictions of the Android operating system regarding names of resources, they had to be renamed. *GraspSynergy* objects are handled and maintained within the *AbsoluteSynergyTouchFragment* and *RelativeSynergyTouchFragment* classes, as they load all grasp synergies, make them selectable within the drop-down list and assign the currently selected synergy object to either the *RelativeSynergyProxy* or the *AbsoluteSynergyProxy*. Due to the implementation of *GraspSynergy*, loading synergy data from application resources is done in only a few lines as shown in Listing 5.23. The variables *mean_res* and *vec_res* are the Android resource IDs for the *mean* and the *vecs* resource file.

Listing 5.23: *Loading GraspSynergy data*

```
1 GraspSynergy synergy = new GraspSynergy(21);
2 synergy.parseMatlabSynergyMean(getResources().openRawResource(mean_res)
    ↪ );
```

Table 5.1: *The mapping from array index to joint name*

0	1	2	3	4	5	6	7	8	9	10
FFJ1	FFJ2	FFJ3	FFJ4	MFJ1	MFJ2	MFJ3	MFJ4	RFJ1	RFJ2	RFJ3
11	12	13	14	15	16	17	18	19	20	
RFJ4	LFJ1	LFJ2	LFJ3	LFJ4	THJ1	THJ2	THJ3	THJ4	THJ5	

```
3 synergy.parseMatlabSynergyVecs(getResources().openRawResource(vec_res))
    ↪ ;
```

Within the above named fragment classes, loading of synergies is surrounded with error handling for the case loading fails. Additionally, a loaded grasp synergy is added to the drop-down list. All this is encapsulated in a method *loadSynergy()* accepting the synergy name (as displayed) and the corresponding resource IDs. Loading of a synergy is then done in one line as shown in Listing 5.24.

Listing 5.24: *Call to loadSynergy()*

```
1 loadSynergy("Grasp 1", R.raw.g1mean_n, R.raw.g1vecs_n);
```

Once loaded, a grasp synergy is used by passing an array of double values to the *toJoints()* method on the *GraspSynergy* object. The return value of this method is again an array of double values, representing the calculated joint angles in degrees. The mapping from the indexes of the returned array to the corresponding joint names is shown in Table 5.1, as extracted from code.

The output of *toJoints()* is then passed to *toSafeAbduction()*, which modifies the joint angles in a way that no collisions between fingers are present and returns the modified array of joint angles. These joint angles can then directly be passed to the *AxisManager* which will forward it to the ROS nodes.

Listing 5.25: *Example call of toJoints() and toSafeAbduction()*

```
1 double[] jointData =
2   _currentSynergy.toSafeAbduction(
3     _currentSynergy.toJoints(_amplitudes)
4   );
```

As this functionality is the same for both the relative and the absolute synergy approach, it is bundled within the abstract class *SynergyProxyBase*, which handles the gesture events received by a *GestureParser* and handles the calculation of joint angles using the synergy assigned by the containing fragment class. It analyses the gesture events for changes in size, orientation and position for each gesture and then performs specific calls to abstract methods which shall be implemented by the inheriting classes *AbsoluteSynergyProxy* and *RelativeSynergyProxy* as reactions to changes of a gesture are the only thing unique to the different approaches.

```
1 protected abstract void handleSizeChange(GestureState oldState, Gesture
    ↪  gesture);
```

```
2 protected abstract void handleLocationChanged(GestureState oldState,
     ↪ Gesture gesture);
3 protected abstract void handleOrientationChanges(GestureState oldState,
     ↪  Gesture gesture);
```

All three methods take the current state of the gesture as well as its state from the call before as parameters. The *old state* of active gestures is maintained by the *SynergyProxyBase* implementation. In fact, only *RelativeSynergyProxy* will make use of it, as the absolute synergy control fully relies on the current state of a gesture.

5.5.4 Absolute Control

The *AbsoluteSynergyProxy* is contained within the *AbsoluteSynergyTouchFragment*, which passes touch events on to the *GestureParser* and registers the *AbsoluteSynergyProxy* with the *GestureParser* to receive events. It also maintains the list of loaded synergies and assigns the one currently selected to the *AbsoluteSynergyProxy*. Within the three abstract methods of *SynergyProxyBase* only the current states of gestures are handled. According to their pointer count different actions are made, as two-pointer gestures are used to control the hand synergies, while three-pointer-gestures are used to control the robotic arm. Only the X and Z position of the robot hand palm are used when controlling the arm at the time of writing. The *CartesianArmManager* described in Section 5.5.2 is used to control the position of the palm.

As values of properties of a gesture shall be mapped linearly to amplitude or position values, a new class is introduced, the *LinearEquation* class. It offers functionality to initialize the parameters of the linear equation by passing it two points between which values shall be mapped as well as the minimum and maximum values that are allowed for the output. Another method available is *calculateClipped()* which calculates the output of the linear equation and clips it to the previously passed minimum and maximum values. The public interface of the *LinearEquation* class can be reviewed in Listing 5.26. Parameters of the linear equation can either be set directly from the constructor or calculated later. This is used when reacting to the screen size, as this value might change at runtime (by changing orientation), so the *calculateParameters()* method is called when the screen size changes and new parameters are calculated.

Listing 5.26: *Public interface of the LinearEquation class*

```
1 public class LinearEquation {
2   public LinearEquation(double x1, double y1, double x2, double y2,
       ↪ double min, double max);
3
4   public void calculateParameters(double x1, double y1, double x2,
       ↪ double y2);
5   public void setLimits(double min, double max);
6
```

```
7   public double calculateClipped(double x);
8 }
```

To react to screen size changes, *AbsoluteSynergyProxy* implements the method *setCanvasSize()* which is called by the containing fragment class when the screen size changes. Listing 5.27 shows how the *LinearEquation* objects are initialized first and updated from the *setCanvasSize()* method. The parameters are chosen according to Section 4.3.1.

Listing 5.27: *Initizlization of LinearEquations for hand and arm*

```
1  private LinearEquation[] _eqsHand = new LinearEquation[] {
2    new LinearEquation(1200, 50, 300, -50, -50, 50),
3    new LinearEquation(0, 50, 1000, -50, -50, 50),
4    new LinearEquation(-(Math.PI / 2.0), 50, Math.PI / 2.0, -50, -50, 50)
5  };
6
7  private LinearEquation[] _eqsArm = new LinearEquation[] {
8    new LinearEquation(300, CartesianArmManager.X_MIN, 1200,
       ↪ CartesianArmManager.X_MAX),
9    new LinearEquation(0, CartesianArmManager.Z_MIN, 1000,
       ↪ CartesianArmManager.Z_MAX)
10 };
11
12 public void setCanvasSize(float width, float height) {
13   LinearEquation leq = _eqsHand[XPOS_AMPLITUDE];
14   leq.calculateParameters(width * 0.25, 50, width * 0.75, -50);
15
16   _eqsArm[ARM_X_EQ].calculateParameters(width * 0.25,
       ↪ CartesianArmManager.X_MIN, width * 0.75, CartesianArmManager.
       ↪ X_MAX);
17   _eqsArm[ARM_X_EQ].setLimits(CartesianArmManager.X_MIN,
       ↪ CartesianArmManager.X_MAX);
18
19   _eqsArm[ARM_Z_EQ].calculateParameters(height * 0.75,
       ↪ CartesianArmManager.Z_MIN, height * 0.25, CartesianArmManager.
       ↪ Z_MAX);
20   _eqsArm[ARM_Z_EQ].setLimits(CartesianArmManager.Z_MIN,
       ↪ CartesianArmManager.Z_MAX);
21 }
```

As an example, the usage of the above functionality is shown with the *handleLocationChanged()* event for a gesture. First it is checked whether the pointer count corresponds to that of an arm or a hand control gesture. The coordinates of the gesture are then passed to the corresponding *LinearEquation* objects and the results are written either to an amplitude or the *CartesianArmManager* to update the position of the palm of the hand. The Y coordinate of the hand is maintained the same, as only X and Z coordinates are changed.

Listing 5.28: *Example usage of LinearEquation*

```
protected void handleLocationChanged(GestureState oldState, Gesture
    ↪ gesture) {
  if(gesture.getPointerCount() == HAND_GEST_POINTER_COUNT) {
    setAmplitude(XPOS_AMPLITUDE, _eqsHand[XPOS_AMPLITUDE].
        ↪ calculateClipped(gesture.getCenter().getX()));
  }
  else if(gesture.getPointerCount() == ARM_GEST_POINTER_COUNT) {
    Location p = gesture.getCenter();
    double x = _eqsArm[ARM_X_EQ].calculateClipped(p.getX());
    double z = _eqsArm[ARM_Z_EQ].calculateClipped(p.getY());

    PointInSpace pos = arm.getPosition();
    PointInSpace newPos = new PointInSpace(x, pos.getY(), z);
    arm.movePalmTo(newPos);
  }
}
```

5.5.5 Relative Control

The implementation of *RelativeSynergyProxy* is similarly embedded into the environment of *RelativeSynergyTouchFragment*, *GestureParser* and *CartesianArmManager* as the *AbsoluteSynergyProxy*. Within the methods that are called when a gesture changes, the *RelativeChanger* class is used to change amplitudes or the position of the arm relatively to their current position. The parameter *oldState* within all gesture events comes into action here, as only the difference between the old and the current state matters for the calculations.

The *RelativeChanger* class is initialized with a rate of change and a maximum and minimum value, to which the output shall be clipped. The rate of change is given in two numbers, the one representing the change in the output value corresponding to the change in input value, which is given as a second parameter. This is implemented according to Definition 9 on page 24. Additionally it gives the opportunity to invert the changes made by *RelativeChanger* using the *invert* flag during the initialization. Listing 5.29 depicts the public interface of the *RelativeChanger* class while Listing 5.30 gives an example of its usage according to the one given in Section 5.5.4 to show the main differences between the two approaches. In this approach uses the *movePalm()* method of *CartesianArmManager*, which already changes the position relatively and clips input to the allowed boundaries. Because of this, a *LinearEquation* is used instead of *RelativeChanger* to only get the desired change to the output value, which is then passed to the *CartesianArmManager*.

Listing 5.29: *Public interface of the RelativeChanger class*

```
public class RelativeChanger {
  public RelativeChanger(double valueChange, double inputChange,
      ↪ boolean invert, double min, double max);
```

```
3   public void setRateBySpan(double valueChange, double inputChange,
        ↪ boolean invert, double min, double max);
4   public double getChangedClipped(double oldValue, double inputChange);
5 }
```

Listing 5.30: *Example usage of RelativeChanger in RelativeSynergyProxy*

```
1  RelativeChanger[] _changers = new RelativeChanger[] {
2    new RelativeChanger(50, 1200, false, -50, 50),
3    new RelativeChanger(50, 1200, false, -50, 50),
4    new RelativeChanger(40, Math.PI, false, -50, 50)
5  };
6
7  LinearEquation armXEq = new LinearEquation(0, 0, 2200,
       ↪ CartesianArmManager.X_MAX - CartesianArmManager.X_MIN);
8  LinearEquation armZEq = new LinearEquation(0, 0, -1000,
       ↪ CartesianArmManager.Z_MAX - CartesianArmManager.Z_MIN);
9
10 protected void handleLocationChanged(GestureState oldState, Gesture
       ↪ gesture) {
11   if(gesture.getPointerCount() == HAND_GEST_POINTER_COUNT) {
12     double xChange = gesture.getCenter().getX() - oldState.getCenter().
           ↪ getX();
13
14     setAmplitude(XPOS_AMPLITUDE, _changers[XPOS_AMPLITUDE].
           ↪ getChangedClipped(getAmplitude(XPOS_AMPLITUDE), xChange));
15   }
16   else if(gesture.getPointerCount() == ARM_GEST_POINTER_COUNT) {
17     Location old = oldState.getCenter();
18     Location newLoc = gesture.getCenter();
19     Location offset = newLoc.substract(old);
20
21     PointInSpace armoffset = new PointInSpace(
22       armXEq.calculate(offset.getX()),
23       0,
24       armZEq.calculate(offset.getY())
25     );
26
27     CartesianArmManager.getInstance().movePalm(armoffset);
28   }
29 }
```

5.6 Direct Fingertip Mapping (DFTM)

The *direct fingertip mapping* (DFTM) is mainly implemented using the *DfmtProxy* and *FingertipPointer* classes, in which most of the functionality is contained. The *DftmProxy* class

implements the *Singleton* pattern. Its instance is used from the *FingertipFragment* class which offers the user interface for this approach and forwards all touch events of the user interface to the *DftmProxy*. For BioIK service interactions, the *DftmProxy* class directly depends on the *C5LwrNode* which offers the functionality needed to request joint angle solutions for a set of given fingertip positions.

The *FingertipPointer* class stores information about one fingertip that is laid onto the touch screen, which is its position in screen coordinates, the name of the effector (also called *link*) which is controlled by this fingertip and the coordinates of the pointer in meters. The latter is stored within this class for convenience reasons and is calculated whenever the screen coordinates of the pointer change. Having the world coordinates available reduces the computational load when writing the information to the screen on the *FingertipFragment*. Both coordinates originate in the top-left corner of the screen. In addition to the position of the pointer a flag is stored whether the pointer is currently available on the screen, i.e. a finger is currently positioned on the screen controlling this pointer.

For the calculation of the world coordinates in centimeters to be correct, the screen metrics, namely the width, height and number of dots per inch (DPI) have to be set once they are known to the user interface (*FingertipFragment*). The coordinates are calculated using Equation 4.10 from Page 26 with r being the dots per inch as set by the user interface class.

The effector controlled by a *FingertipPointer* is determined by the order in which the fingertips are laid down onto the screen. At the moment, at most three fingertips may be controlled using this approach. The first finger put down onto the screen controls the thumb (*thtip*), the second controls the first finger (*fftip*) and the third the second or middle finger *mftip*. This should usually map the finger controlled to the actual finger of the controller's hand. Fingers can be lifted during operation. If the lifted finger was the last finger laid down onto the screen, the pointer is removed from the list of pointers sent to the BioIK service. If the finger was not the last one laid down onto the screen, the corresponding *FingertipPointer* is marked as *not present*, which means that no finger is currently controlling its position but it is still included in BioIK service calls. Once a finger is put onto the approximate position where it was lifted, the pointer is again marked as *present* and its position is updated periodically. The public interface of the *FingertipPointer* class is shown in Listing 5.31.

Listing 5.31: *Public interface of the FingertipPointer class*

```
1 public class FingertipPointer {
2   public FingertipPointer(String effectorName, Location screenLoc, int
       ↪ id);
3
4   public String getEffectorName():
5
6   public int getPointerId();
```

```
7   public void setPointerId(int pointerId);
8
9
10  public Location getScreenLocation();
11  public void setScreenLocation(Location screenLocation);
12
13  public Location getWorldLocation();
14  public void setWorldLocation(Location worldLocation);
15
16  public boolean isPresent();
17  public void setPresent(boolean present);
18 }
```

Once a new position for a fingertip is found, the *update loop* is started. The update loop for this approach is very similar to the one of the *CartesianArmManager* depicted in Figure 5.10 on Page 57. The *DftmProxy* requests new solutions for the current pointers' locations until either no pointers are registered anymore or the *DftmProxy* is disabled using the *setEnabled()* method. Once a new solution is returned to *DftmProxy*, it is also checked against a maximum movement of 15 degrees on a per-joint basis, solutions with a too big displacement are discarded and a new solution is requested.

To request a solution the desired locations of the fingertips in three-dimensional space have to be calculated first. As the position of the fingertips is known in meters from the top-left corner of the screen, the simple calculations described in Section 4.4 can be done using no more conversions. The *PointInSpace* class offers all functionality needed for these calculations. The base vector and the two spanning vectors are currently defined as fixed values and shown in Listing 5.32. Listing 5.33 shows how the mapping from two-dimensional screen coordinates into three-dimensional coordinates is performed.

Listing 5.32: *Vectors used for planar mapping*

```
1 PointInSpace surfaceBase = new PointInSpace(0.0, -1.25, 1.2);
2 PointInSpace surfaceYBaseVect = new PointInSpace(0, 1, 0);
3 PointInSpace surfaceXBaseVect = new PointInSpace(-1, 0, 0);
```

Listing 5.33: *Mapping of points from two into three dimensions*

```
1 FingertipPointer p = _pointers[i];
2
3 PointInSpace wloc = surfaceBase
4   .add(surfaceXBaseVect.multiply(p.getWorldLocation().getX()))
5   .add(surfaceYBaseVect.multiply(p.getWorldLocation().getY()));
```

Once this calculation has been done for all pointers, the request is passed to the BioIK service. In addition to the fingertip locations, the *C5LwrNode* class adds a *MinimalDisplacementGoal* to the request to indicate to the service that a solution with minimal displacement of all joints is desirable. Additionally, a *OrientationGoal* is added, forcing the

palm of the robotic hand into a top-down orientation. Otherwise, the IK solver could find solutions where the fingertips are at the correct positions but were placed there from the bottom, which is correct in principle, but neither a expected nor a desired solution.

The safety interlock button on the screen sets the enabled flag of the *DftmProxy* which is redirected to the *AxisManager*. This means that once the safety interlock button is released, no movement is done by the robot. If an IK request is still pending and the result is returned after the button has been released, the solution is discarded since the *AxisManager* does not accept new joint angles. At the time of writing it is not possible to move the base vector of the plane fingertips are mapped to. Thus actions are limited to a very small workspace. Functionality to *scroll* the workspace can be implemented by changing the base vector, but it has to made sure that the movement is safe and no unexpected joint angle changes occur.

6 Evaluation

6.1 Usability

First tests and trials have shown that the different approaches yield different results regarding simplicity, manoeuvrability and usability. No formal tests and studies have been conducted by now, the following experiences are subjective and by far not representative.

Controlling the hand grasps using the grasp synergy approaches works decently good. The absolute version seems to give the user a little better and more direct control about what is actually happening with the hand and thus more dexterity in the grasp operations than the relative approach. Controlling the arm in both approaches, however, is fairly inaccurate. In the absolute approach, inaccuracies derive from the small screen size which is mapped to a relatively large workspace. In the relative approach the arm can be controlled more precisely, but the activation time of one second for a new gesture is significantly affecting the workflow. Whenever the gesture is lifted and moved to the other end of the screen to perform another relative movement of the arm, the operator has to wait one second. In both approaches, however, the arm does not remain at one stable position when the control gesture is stationary. This results in a noticeable jitter of the arm. The reason for this seems to be the BioIK solver returning different solutions for the same pose. As long as the gesture is active on the screen, new solutions are queried by the application in an endless loop, since the position of the pointers change very little. These changes are in the second to fourth decimal digit of the value of the amplitude, however the BioIK solver returns a new – and different – solution for the same position. This effect was dramatically reduced once the *MinimalDisplacementGoals* were added to the request messages, but are still noticeable. Additionally, finding solutions with the *MinimalDisplacementGoal* active in the request takes significantly longer (about factor two, see Section 6.2.2). These small movements made precise control and positioning of the hand difficult. One possible solution could be to monitor the movement of a gesture and only calculate new values when movements above a certain threshold (e.g. 10 pixels) occur.

In the direct fingertip mapping, these jitters were even more significant, as the position of the fingertips were to be mapped to positions in space with high dexterity. However, because of the jitter occurring within the BioIK solutions, it is very hard to precisely grasp objects. Again, adding the *MinimumDisplacementGoal* reduced this effect significantly, but slowed down the (already slow) solution finding of BioIK even more.

6.2 Performance

As performance issues occurred during first tests, a small look into where these issues arise shall be given.

6.2.1 Application

The overall performance of the application seems sufficient. No significant lags in response times of the user interface were noticeable. All off-line calculations like calculating joint angles from grasp synergies or fingertip positions in three dimensions do not take any significant processing power. While using the application without any inverse kinematic involved, the CPU load was measured well below 5%. Nonetheless, the device gets significantly warm after a period of usage and the battery power drains perceptibly faster compared to everyday use.

No further effort was done to find out the reason for the warmth and battery drain as it did not directly affect development, but a good assumption might be that the ROS data exchange induces a constant load onto the WiFi chipset of the tablet computer. Joint states are received at about 100 Hz, updates are sent at 10 Hz. Additionally, every BioIK service call issues some wireless communication. This leads to a lot of small packets being sent over the wireless network, resulting in a constant workload preventing the chipset from being put into power saving modes.

6.2.2 BioIK Service

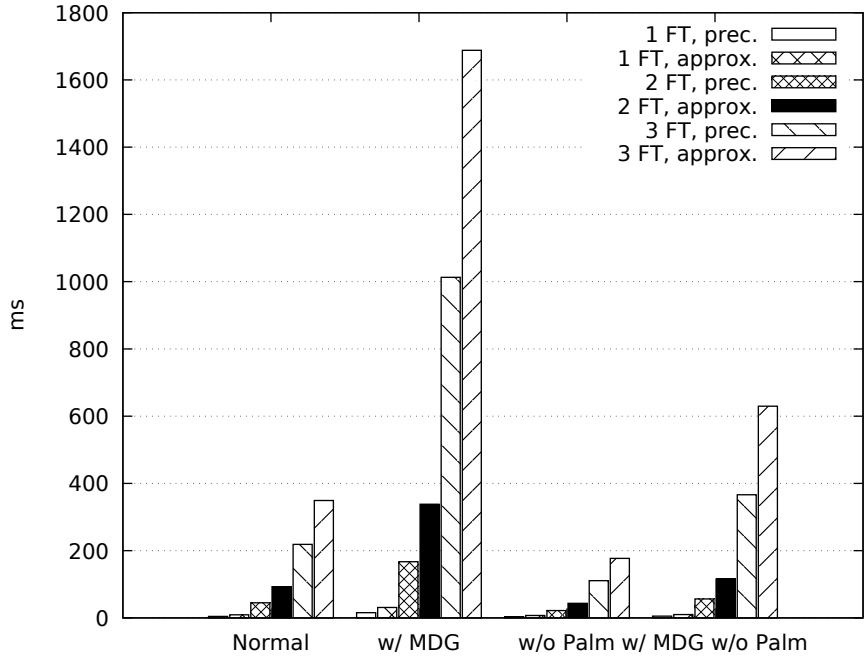
It was observed during tests that getting a result from the BioIK service takes significant amounts of time during arm movements controlled by touch gestures and even more time in the direct fingertip mapping mode. In the grasp synergy approaches, an IK request took about 200-300 ms from sending the request until the *onSuccess()* callback was called in the application. This is a frequency of about 5Hz, which is lower than expected but still enough for a relatively precise positioning of the arm.

In the direct fingertip mapping approach however, response times varied depending on the number of fingers that were currently mapped, from approx. 600ms when using one finger to about 1.6 seconds with three fingers on the screen. Update frequencies of significantly less than 1 Hz are very disturbing and prevent the application from being used in the desired manner.

Additionally it is observable that when the BioIK service is called, the user interface freezes until a response is processed, resulting in a very juddery user experience. Multiple factors can take effect here:

- The rosjava/rosandroid implementation of service calls is faulty. The freezing user interface is an indication for the service calls not being implemented asynchronously as one would think, since a call is initiated with a service response handler which is called when the result was received.

Figure 6.1: *Measurements for 1000ms*



- The BioIK service is very slow or has unexpected behaviours.

Since debugging of foreign code can be very time-consuming, especially in projects of the size of rosjava and rosandroid, first measurements were done concerning the performance of BioIK. A pose of the robot was chosen which was once returned by the BioIK service, so it' iss known that a solution exists. Then, the following cases were measured:

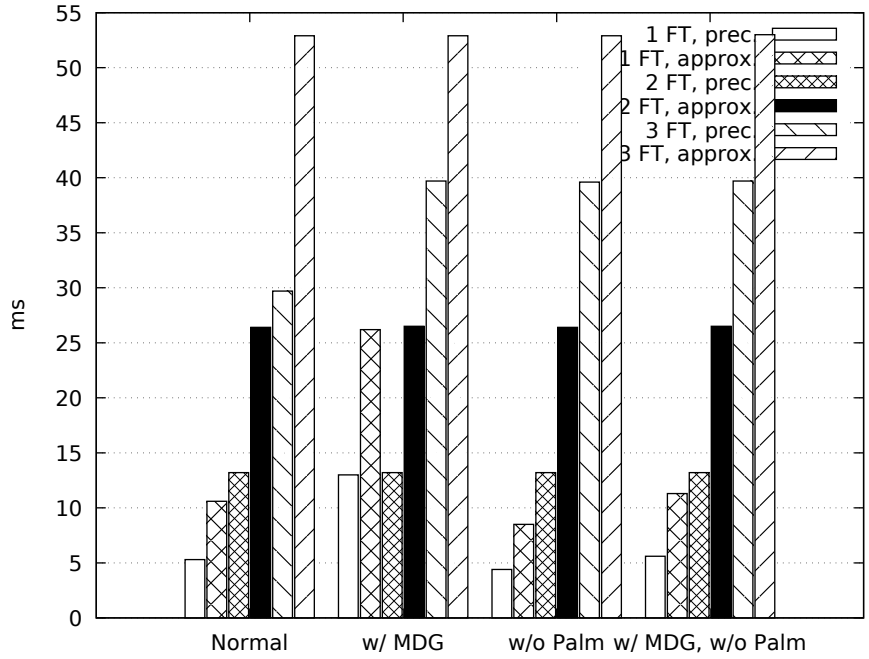
- 1, 2 and 3 fingertips with *approximate = false*.
- 1, 2 and 3 fingertips with *approximate = true*.

for the following cases:

- With *OrientationGoal* for the palm, without *MinimumDisplacementGoal*.
- Without *OrientationGoal* for the palm, without *MinimumDisplacementGoal*.
- With *OrientationGoal* for the palm, with *MinimumDisplacementGoal*.
- Without *OrientationGoal* for the palm, with *MinimumDisplacementGoal*.

Every test request was executed 200 times, as a result the mean execution time of all calls was taken. The aim was to find out what affects execution time of the BioIK service the most, the suspected properties were the *MinimumDisplacementGoal*, the additional *OrientationGoal* for the palm and the *approximate* property of the IK request. Measurements were initially done with a timeout of one second passed to the BioIK solver. A plot

Figure 6.2: *Measurements for 10ms*



of the measurements can be reviewed in Figure 6.1. Three interesting aspects are visible on the first sight:

- If the request is marked with *approximate = true*, the solver takes about twice the time as if a precise solution is requested.
- The *MinimumDisplacementGoal* seems to have a huge effect on execution time, multiplying the time by a factor of about 4-5.
- The time needed by the solver is highly dependent on the number of fingertips included in the request.

All tests were repeated setting the *timeout* for the request to 10 ms. The test results are represented in Figure 6.2. Surprisingly this rendered all measurements completely different. Execution times are nearly constant for every fingertip configuration, the dependency seems to be exponential. The only real difference visible is the measurement of one fingertip with *OrientationGoal* for the Palm and with *MinimumDisplacementGoal*, which is about factor three bigger than without the *MinimumDisplacementGoal*. Most importantly, the maximum times of about 55ms were about two orders of magnitude smaller than the maximum values measured with a timeout of one second. The execution time of the BioIK solver seems to be dependent on the passed timeout, using more time when the timeout is bigger. This assumption was confirmed by Ruppel[22] who states that the BioIK solver **can** return a result before the timeout is exceeded, but does not necessarily do so.

As a consequence, a time-out of 10ms was set for the BioIK service calls within the Android application. The results, again, were surprising, as no significant improvement was observed. The only difference then was that the BioIK solver returned an error code indicating no solution was found, increasing the timeout up to 500ms rendered the DFTM approach usable again, but still with response times of 1.2 to 1.6 seconds. At the time of writing open questions remain. It is yet to be found out why the service calls take such a long time in rosjava/rosandroid, rendering the user interface frozen with the response time not significantly impacted by the timeout set in the request. As the measurements suggest, however, the main issue should probably be searched for within the rosjava and rosandroid implementations, as solving times were found reasonable when calling the BioIK service isolated.

6.3 Possible User Studies

With user studies, one could find out how well the different approaches implemented here work with untrained and trained test persons. Data can be recorded either objectively by measuring times and judging how successful the execution of a task has been or subjectively, by asking the user about how difficult the task was. A famous, standardized questionnaire is the NASA TLX (Task Load Index) test[14]. It yields a good insight of how stressful a task was for the user. Apart from these standardized tests, a domain-specific test should be developed to get data about the actual environment that shall be evaluated.

Each test person should perform multiple tasks with a rising level of difficulty. For example:

- Grasp an object and release it.
- Grasp an object, move it to another place and release it.
- Grasp an object, put it into a box placed nearby.

These tasks can then be performed for both multiple objects (balls, cylinders, cuboids) of different materials (sponge, wood, rubber) and the three different approaches. Each task shall be completed multiple times, while the time needed to complete the task is measured. This gives an insight in how fast a specific task can be trained. Additionally, after the task was executed multiple times by one user, he shall be queried for how difficult he thinks the task was, how much help he needed and how intuitive he thinks the control was. The test supervisor shall also note how much help he had to give to the test person to give an insight on how different perception was.

From the data raised during these studies a conclusion can be made which approach is probably the best to grasp a variety of objects, which is the easiest to learn, which one is the most intuitive and which one causes the least stress on the operator. These

results could then be used in further development and improvement of tele-manipulation methods using a dexterous robotic hand and a robot arm, combined to a robot system with a high number of degrees-of-freedom.

7 Conclusion

The application that was developed within this thesis shows the principle possibilities to control a robot with many DOF using a generic end-user multitouch device running the Android operating system. Having implemented multiple approaches gave an insight into different possibilities to perform tele-operation of a robot and tele-manipulating its environment. While the gesture parsing functionality developed gives a generic method to explore the properties of a simple multi-pointer gesture, which may be used for other purposes as well, the direct fingertip mapping approach showed that mapping of pointers in two dimensions into a three-dimensional space is a task that can be performed with basic maths operations.

First tests were done using the given set-up. Besides some hardware-related issues (like missing air pressure) the functionality mostly worked as expected, only the jitters of the arm in all approaches were a significant drawback, which is what future work should probably put some effort into. Of course, more tests and studies have to be conducted and parameters of the different approaches (especially the relative grasp synergy approach) have to be optimized.

7.1 Outlook

Future work could concentrate on several things. First, the occurred performance and jittering issues should be reviewed. The performance issues can probably be resolved by looking into the rosjava and rosandroid implementations, where at least a factor of two is situated. After that, a look into the jittering of BioIK solutions would be an interesting thing to look into, as precise grasping actions depend on a stable and dexterous positioning of effectors. If these problems are solved, statistically significant user studies should be conducted to deeply evaluate the usability of the different approaches and give suggestions on improving them.

In the end, the application is designed in a way that should make it easy to implement more methods of controlling a robot as they come up and are researched, using the existing framework of *AxisManager* and *C5LwrNode* and the given user interface structure.

Bibliography

[1] Charan Ram Akupati. "Control of ROS Compatible Mobile Robot with Android Platform using ROSJAVA". Diploma. Universität Siegen, 2017. URL: https://www.eti.uni-siegen.de/ezls/lehre/studiendiplomarbeiten/controlofmobilerobotwithandroidplatform/control_of_mobile_robot_with_android_platform.pdf (visited on 2018-04-01).

[2] Andreas Aristidou and Joan Lasenby. *Inverse Kinematics: a review of existing techniques and introduction of a new fast iterative solver*. Tech. rep. CUED/F-INFENG/TR-632. Cambridge University Engineering Department, 2009. URL: http://www.researchgate.net/publication/273166356_Inverse_Kinematics_a_review_of_existing_techniques_and_introduction_of_a_new_fast_iterative_solver.

[3] Alexandre Bernardino, Marco Henriques, Norman Hendrich, and Jianwei Zhang. "Precision grasp synergies for dexterous robotic hands". In: *2013 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. IEEE, 2013. DOI: 10.1109/robio.2013.6739436.

[4] Bronstein, Semendjajew, Musiol, and Mühlig. *Taschenbuch der Mathematik*. Harri Deutsch, 2012. ISBN: 978-3-8171-2008-6.

[5] Dave Coleman. *MoveIt! Strenghts, Weaknesses and Developer Insights*. 2015. URL: https://roscon.ros.org/2015/presentations/ROSCon-%20MoveIt!%20Developer%20Insights.pdf.

[6] Shadow Robot Company. *Shadow Dexterous Hand C5 Technical Specification*. 2008. URL: http://www.shadowrobot.com/downloads/shadow_dextrous_hand_technical_specification_C5.pdf (visited on 2018-03-05).

[7] A. D'Souza, S. Vijayakumar, and S. Schaal. "Learning inverse kinematics". In: *Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. Expanding the Societal Role of Robotics in the the Next Millennium (Cat. No.01CH37180)*. IEEE, 2001. DOI: 10.1109/iros.2001.973374.

[8] Karl Eilebrecht and Gernot Starke. *Patterns kompakt: Entwurfsmuster für effektive Software-Entwicklung*. Springer Vieweg, 2013. ISBN: 9783642347177.

[9] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. "The many faces of publish/subscribe". In: *ACM Computing Surveys* 35.2 (2003), pp. 114–131. DOI: 10.1145/857076.857078.

[10] Jodi Forlizzi and Carl DiSalvo. “Service robots in the domestic environment”. In: *Proceeding of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction - HRI*. ACM Press, 2006. DOI: 10.1145/1121241.1121286.

[11] Rosjava GitHub. *Rosjava Readme*. 2018. URL: https://github.com/rosjava/rosjava_core (visited on 2018-03-05).

[12] KUKA Roboter GmbH. *Lightweight Robot 4+. Specification.* 2010-07-06.

[13] KUKA Roboter GmbH. *Preliminary Documentation. Fast Research Interface (FRI).* 2010-08-29.

[14] Sandra G. Hart and Lowell E. Staveland. “Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research”. In: *Advances in Psychology*. Elsevier, 1988, pp. 139–183. DOI: 10.1016/s0166-4115(08)62386-9.

[15] Sunao Hashimoto, Akihiko Ishida, Masahiko Inami, Takeo Igarashi, and and. “TouchMe: An Augmented Reality Interface for Remote Robot Control”. In: *Journal of Robotics and Mechatronics* 25.3 (2013), pp. 529–537. DOI: 10.20965/jrm.2013.p0529.

[16] Dennis Krupke, Paul Lubos, Gerd Bruder, Frank Steinicke, and Jianwei Zhang. “Natural 3D Interaction Techniques for Locomotion with Modular Robots”. In: *Mensch und Computer 2015: Gemeinsam Arbeit Erleben*. 2015. URL: http://basilic.informatik.uni-hamburg.de/Publications/2015/KLBSZ15.

[17] Morgan Quigley, Ken Conley, Brian P. 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*. 2009.

[18] ROS.org. *ROS: Client Libraries*. 2016-05-23. URL: http://wiki.ros.org/Client%20Libraries (visited on 2018-03-05).

[19] ROS.org. *ROS: Installation of ROS Kinetic on Ubuntu*. 2017-07-25. URL: http://wiki.ros.org/kinetic/Installation/Ubuntu (visited on 2018-03-08).

[20] ROS.org. *ROS: Setting up a catkin workspace*. 2017-05-11. URL: http://wiki.ros.org/catkin/Tutorials/create_a_workspace (visited on 2018-03-08).

[21] ROS.org. *Writing a simple Service and Client*. 2016-09-29. URL: http://wiki.ros.org/roscpp_tutorials/Tutorials/WritingServiceClient (visited on 2018-04-06).

[22] Philipp Sebastian Ruppel. “Performance optimization and implementation of evolutionary inverse kinematics in ROS”. MA thesis. Universität Hamburg, 2017. URL: https://tams.informatik.uni-hamburg.de/publications/2017/MSc_Philipp_Ruppel.pdf.

[23] Samsung.com. *Samsung Galaxy Tab S3 Datasheet*. 2017. URL: http://www.samsung.com/de/tablets/galaxy-tab-s3-9-7-t820/SM-T820NZKADBT/ (visited on 2018-03-05).

[24] Sebastian Starke, Norman Hendrich, and Jianwei Zhang. "A memetic evolutionary algorithm for real-time articulated kinematic motion". In: *2017 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2017. DOI: 10.1109/cec.2017.7969605.

[25] Sebastian Starke, Norman Hendrich, Dennis Krupke, and Jianwei Zhang. "Evolutionary multi-objective inverse kinematics on highly articulated and humanoid robots". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017. DOI: 10.1109/iros.2017.8206620.

[26] Yue Peng Toh, Shan Huang, Joy Lin, Maria Bajzek, Garth Zeglin, and Nancy S. Pollard. "Dexterous telemanipulation with a multi-touch interface." In: *Humanoids*. IEEE, 2012, pp. 270–277. URL: http://dblp.uni-trier.de/db/conf/humanoids/humanoids2012.html#TohHLBZP12.

[27] developer.android.com. *Timer*. 2018. URL: https://developer.android.com/reference/java/util/Timer.html (visited on 2018-04-13).

List of Figures

List of Tables

Listings

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Bachelorstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.
Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

Reinfeld, den _____ Unterschrift: _____