# SensorClouds

A Framework for Real-Time Processing of Multi-Modal Sensor Data for Human-Robot-Collaboration

Alexander Poeppel

Dissertation
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)

Universität Augsburg
University

Fakultät für Angewandte Informatik

14. August 2023

**SensorClouds**

| | |
|---|---|
| Erstgutachter: | Prof. Dr. Wolfgang Reif |
| Zweitgutachter: | Prof. Dr. Jörg Hähner |
| Tag der mündlichen Prüfung: | 23. Oktober 2023 |

# Abstract

With the advent of industry 4.0 with its goals to make production more flexible and products more individual, the need for robots which can collaborate with humans to perform manufacturing is growing. In healthcare, the aging populations of western countries and the growing labor shortage increase the need for robotic assistants capable of relieving workers of menial and strenuous tasks. Both these fields of application require robots to be able to perceive their environment in order to safely interact with humans and perform their tasks correctly. This work presents *SensorClouds*, a modular framework for the processing of multi-modal sensor data in realtime for applications involving human-robot-collaboration. The framework is competitive in performance with similar approaches, yet far more flexible since it is not limited to binary occupancy in its environment model but instead allows the dynamic specification of arbitrary modalities in order to enable more complex sensor data processing and a more informed representation of the robot's surroundings. The architecture aids developers of modules in the creation of massively parallel algorithms by taking over the parallelization aspect and requiring only the implementation of processing kernels for single data points. Application developers can use these modules to quickly solve complex sensor fusion tasks. Module interoperability is guaranteed through the enforcement of data access contracts. This work also includes methods for reconstructing three-dimensional data from sensors which do not inherently provide it so that this data can then also be included in the environment model alongside natively three-dimensional data.

# Contents

**Chapter Summary**. The introductory chapter will first detail the motivations of this work and then state the main contributions. Finally, a brief overview of the structure is given. …

# 1 Introduction

## 1.1 Motivation and Goals of this Work

Robots have been becoming more and more capable and intelligent over the past decades. The most prominent example of this development are the products developed by Boston Dynamics [19], which have also garnered the attention of mass media outlets countless times. Ranging from Spot, the robot modeled after a household dog, up to the humanoid Atlas robot, such systems have been becoming increasingly capable regarding their locomotive abilities and will become even more powerful tools with increasing autonomy through the integration of artificial intelligence towards independent operation and decision making. The Gartner Hype Cycle for artificial intelligence [131], however, estimates that smart robots will reach their so called *plateau of productivity* in the next five to ten years from now while currently classifying them as being in their technological infancy.

In the production industry, robots have been heavily used in assembly lines for the past decades and have been a driving factor in the expansion of industries such as automobile manufacturing [12][129]. Such production lines were governed by a stringent separation between humans and robots and employed complicated safety measures in order to guarantee that no human could approach a running robot. While such production lines were required to conform to the safety standards set forth in ISO 10218 [1], the industrial applications of robots have been evolving over the last years. Collaborative robots are changing the way manufacturing processes are being performed. Robots are generally very proficient at executing tasks with high precision and repeatability, and can do so in perpetuity. In many cases, though, robots are not capable of independently solving tasks and require either fixed processes or operator assistance. Hence, human-robot-collaboration in manufacturing aims to combine the strengths of humans and robots in manufacturing processes. The worldwide market for collaborative robots is projected to grow sevenfold by the year 2030 [120]. KUKA, for example, showcases applications for Volkswagen [63] and BMW [61] which employ collaborative robots. Repetitive stress injuries caused by continuously performing the same task can be alleviated through the use of robotic assistants, which perform tasks with injury potential for humans. They can also assist in the transport of heavy loads or tasks that would necessitate awkward positions of a human worker. Operator safety is especially of concern when humans and robots operate in close proximity to each other, but requires entirely new safety concepts and technologies. This is why the standards bodies are currently developing new guidelines for these types of applications. The technical specification ISO TS 15066 [3] is under ongoing development and is supposed to become the standard for safety in human-robot-collaboration once the emerging technologies required become viable for certification.

Demographic changes in the industrialized nations and the continuing shortage of elderly care workers have led to an increased effort to develop robots which can assist healthcare workers. Similar to industrial applications, the risk of injury is great for healthcare workers when performing strenuous tasks such as lifting patients repeatedly. In particular, Japan is at the forefront of the development of assistive robot technology for healthcare applications, since the problems of aging population and labor shortage

are most intense there compared to other countries [130]. The expectations of such robotic assistants in terms of functionality, reliability and ease-of-use are far greater than in industrial settings, due to the delicate nature of their tasks in healthcare applications. The moral debate is still ongoing on whether such systems should be employed and to what extent they may be allowed to act autonomously in the care of elderly and disabled people, who may not have full command of their mental faculties and thus would be at the mercy of a malfunctioning robot system. However, the acceptance rate of assistive robot systems which can be regarded as mere tools for healthcare workers, patients and family members alike, is far higher. Such systems would only assist healthcare workers on command, for example, to retrieve items necessary for the current procedure being performed. The use of such systems could also free up more time for the healthcare professional to spend time interacting with the patients if such menial tasks like fetching a glass of water could be performed by an automated system. (cf. [17])

All autonomous robot systems face the challenge of needing to perceive their environment and everything contained within it correctly in order to be able to carry out their tasks. Especially for systems employed in the homes of elderly people a robot must be able to reliably navigate through unstructured environments with furniture and other objects not always located in known positions, as is the case in industrial workplaces. Operator safety is a critical issue in industrial settings but even more so when vulnerable people are involved. In consequence, environmental perception is the most profound and important task in any application involving human-robot-collaboration. Taking inspiration from nature, as is so often the case in facing technical challenges, the solution to these problems could be found in the way animals and humans perceive their environment. All animals are equipped with multiple, diverse senses which are integrated into a combined understanding of their surroundings, their location and potential threats or sources of food. Missing information or the loss of a sense can be compensated by the others. (cf. [39])

Modeled after this principle of nature, this work presents a novel framework for sensor data processing in human-robot-collaboration under realtime conditions. The goal is to provide a unified model of the robot's surroundings with information from all available sources integrated into it. Apart from providing a more complete picture of the surroundings, this also allows the deduction of new information from combined sensor data and the compensation of missing inputs.

## 1.2 Main Contributions

The *SensorClouds* framework was developed over the course of this work and represents a dynamic, flexible and realtime-capable framework for the processing of multi-modal sensor data specifically tailored to the field of human-robot-collaboration, yet also applicable in other areas.

This architecture is built entirely upon **modular processing kernels**. The dynamic definition of the global data model utilized as central storage for all computations allows great flexibility, while the just-in-time compilation of all kernels enables the generation of concrete and efficient instructions for sensor data processing. In this manner, the

*SensorClouds* framework combines the advantages of both dynamic and highly targeted programs.

*SensorClouds* differentiates between **two distinct kinds of developers**, whose interaction is governed by clearly defined **software contracts**. **Module developers** provide the algorithmic building blocks out which **application developers** can fashion their specific sensor processing application. The framework enforces the contents of the contracts on all parts of the application and thus ensures correct execution of all employed modules and the overall application.

Sensors which do not measure three-dimensional information can be integrated the global data model of the *SensorClouds* architecture with the appropriate pre-processing. The methodology developed in this work to **reconstruct three-dimensional data** from lesser dimensional measurements can be applied to any sensor type whose sensor characteristics can be determined by physics simulations.

## 1.3 Structure of this Work

This work is structured as follows: Chapter 2 gives a brief overview of relevant technologies and sensor measurement principles relevant in the scope of this work. The application examples that informed the development of the architecture presented in this work, as well as the requirements imposed on it are described in Chapter 3. Chapter 4 elaborates on the architecture of *SensorClouds*, the internal processes in the creation of concrete application instances and the programming interface for developers building applications upon this architecture. Capacitive sensors require various signal processing steps in order to convert the analog capacitance measurement into data which can be utilized for the proximity detection of humans in a robot's workspace. Moreover, the one-dimensional value is not suitable for direct use in a three-dimensional environment model. These concepts and their integration into the *SensorClouds* architecture are presented in Chapter 5. Some additional software concepts which provide convenience to developers using the presented architecture are introduced in Chapter 6. Chapter 7 presents selected details of the reference implementation of the *SensorClouds* architecture targeted at graphics processing units (GPUs) and also specifies the programming interface for developers of modules. The evaluation of the presented framework is presented in Chapter 8. Chapter 9 concludes this work and gives an outlook on future developments.

**Chapter Summary**. This chapter gives a brief overview of technologies relevant in the context of this work.

# 2 Technology overview

## 2.1 Sensor Types and Measurement Principles

This section will give a cursory overview over some relevant sensors and their measurement principles.

### 2.1.1 Capacitive Sensors

Capacitive sensors are based on the principles of electrical capacitors. Various physical characteristics, which will further be explained in Section 5.3, can be exploited in order to measure changes directly or detect differences in correlated physical effects.

The industrial applications of capacitive sensors are extremely diverse. They range from the detection of near proximity and position over tilt to humidity sensing, among others. Depending on the specific application, the electrode configurations may vary greatly from one another. Most capacitive sensor applications do not use electrodes directly opposite of each other, but rather project a non-uniform electric field into the space surrounding the sensor in order to remotely register the desired phenomena.


**Figure 2.1.** An industrial capacitive proximity sensor from SICK [109].

Industrial capacitive sensors ,such as the one shown in Figure 2.1, output a binary signal signifying the detection of an object in proximity. The switching threshold is typically adjusted by a potentiometer on the sensor itself and calibrated for the binary detection of passing or approaching objects in industrial production lines. Since the sensor is mounted in a fixed location and the objects to be detected follow fixed paths, this adjustment can be made extremely precisely. While optical or magnetic sensors could perform similar tasks in detecting passing objects, the strength of capacitive sensors lies in their ability to detect conductive as well as non-conductive objects (unlike magnetic sensors) and are able to detect reflective objects and even objects behind a limited amount of non-conductive casing (unlike optical sensors). Such capacitive sensors are typically constructed in a tubular design with the capacitance between two ring electrodes on the outward facing end of the tube being measured. (cf. [15, p. 69f])

Another industrial application is the measurement of the inclination angle or tilt of an object. Today, such sensors are typically fabricated as Micro-Electro-Mechanical

Systems (MEMS), with movable electrodes implemented in microstructures inside an integrated circuit (IC). When the sensor is moved, the spacing between the electrodes changes, which in turn can be measured in terms of the capacitance between them. Such devices are also found in modern consumer electronics devices, such as smartphones or game controllers in the form of inertial measurement units (IMUs), which provide data on three axes of translation (in form of acceleration) as well as three rotational axes (in form of angles) allowing the device to determine how it is moving through space. (cf. [5, p. 239ff])

Capacitive sensors are also commonplace as input devices in modern consumer electronics. Their uses range from simple contactless input buttons in touch panels to fully sized touchscreens found in devices as small as smart watches and phones and as large as laptop and desktop computers. While these implementations are intentionally limited to very close proximity to the input device (within the order of a few $\mathrm{mm}$), the detection range of capacitive sensors can be increased up to distances of around $35\,\mathrm{cm}$ [54, 80] enabling their use as far proximity sensors.

Such sensors, however, do not directly measure distances but merely the change in capacitance of the system, although a direct correlation exists between the measured capacitance and the distance to an object entering the measurement field. In HRC applications, the human hand is typically chosen as a calibration reference, since it is the smallest relevant part of the human body and most likely to interact with the robot during operation. Any other part of the human body has a greater influence on the electrical field projected by the capacitive sensor thus increasing the capacitance and in turn decreasing the perceived distance to it. This is an advantageous circumstance in safety critical systems because the reaction of the robot in such a case would simply occur earlier than compared to the reaction to hands. Further details on the physical background of capacitive sensors as well as their use in applications involving close interaction with humans are provided in Chapter 5.

### 2.1.2 Cameras

This section will give a brief overview over various camera technologies relevant to this work.

#### RGB Cameras

Digital imaging sensors are commonplace nowadays and can be found in many devices, including smartphones, laptops and even home appliances such as fridges [66]. The most common implementation of imaging sensors today is executed in *complementary metal-oxide-semiconductor* (CMOS) technology, which replaced charge-couple device (CCD) sensors as the main technology in digital cameras towards the end of the 1990s [64, p. 83]. Light sensitive CMOS components are packed into a two-dimensional grid layout and exposed to light through the camera's lens system. Each of the photosensitive elements is called a *pixel*. For color reproduction, the most common technique is the use of on chip color filters which are placed atop the monochrome photosensitive elements. The configuration of the different color filters for red, blue and green light, which are

the base constituents of the additive mixture of colors for light, was patented by Bryce E. Bayer in 1976 [16] and is simply named *Bayer Filter* after its inventor. It is based on the distribution of color receptors in the human eye which are more sensitive to green light than red and blue. The filter arrangement according to Bayer is depicted in Figure 2.2. For every actual pixel in the image, which only possesses one channel of the color information, the other channels are interpolated using information from the surrounding pixels in order to create a full scale image with all color information. This process is known as de-mosaicking or de-bayering [49].

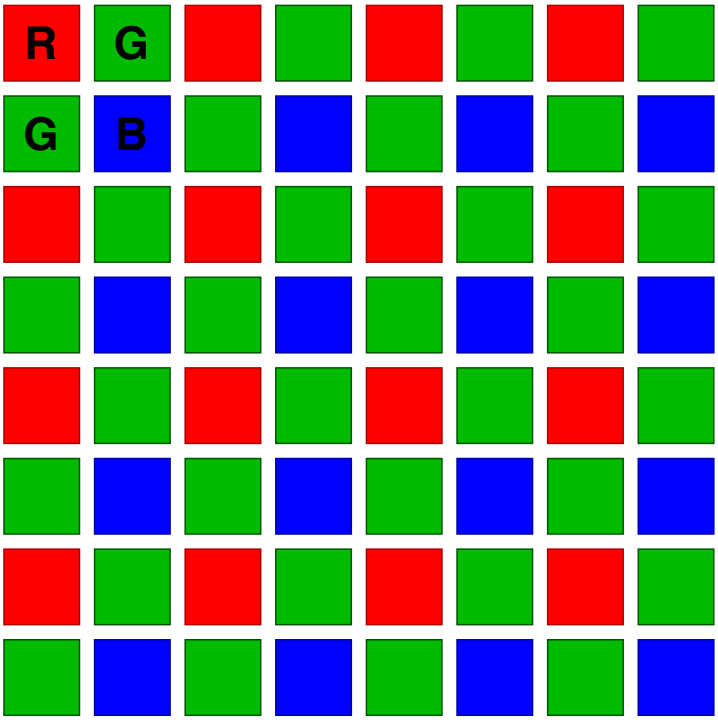



**Figure 2.2.** Bayer filter arrangement in image sensors.

In order to reliably use image information produced by a camera system in computer Vision applications, the camera needs to be calibrated in order to determine the intrinsic camera parameters, such as the focal length and the projection matrix, and any distortions the lens system introduces into the final image. To this end various calibration methods can be employed, although the process involving a planar, black and white checkerboard pattern with known dimensions and recording multiple images of it is the most commonly used, as it is very simple to perform. Figure 2.3 shows the approximate model typically used in the calibration of cameras, called the *pinhole model* [135] since it simplifies the actual physical projection characteristics of optical systems to an approximated system in which all light passes perfectly through a discrete point in space, ergo a pinhole. While this is only an approximate model, its accuracy is sufficient for most tasks in computer vision. It should be noted at this point that although the intrinsic calibration argues over a projection matrix from three-dimensional space to

two-dimensional image sensor coordinates, the reconstruction of three-dimensional information from a single camera image is not possible unless the scale of the perceived object is known, hence the scale of the checkerboard must be known. Since the image projection is identical for a large object at a large distance and a small object at a smaller distance and no depth information is available, the actual size of an object in a camera image cannot be reliably determined mathematically. (cf. [133])

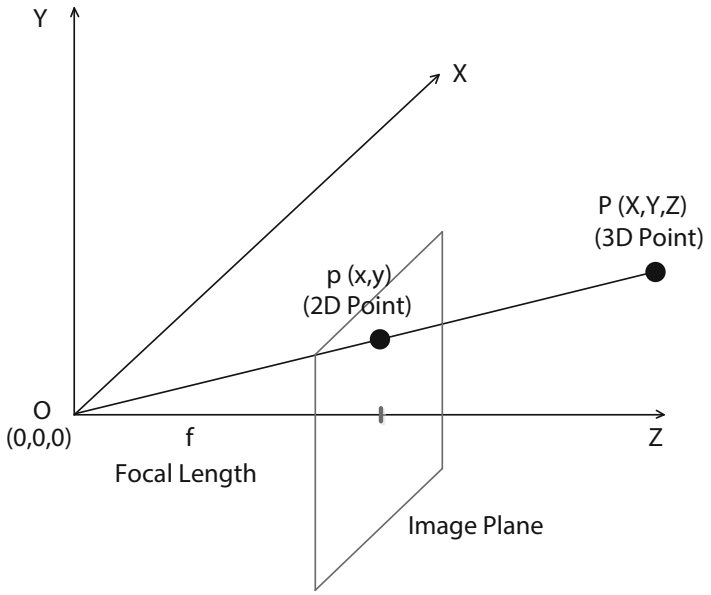



**Figure 2.3.** The pinhole model used for the estimation of camera parameters. Adapted from [56, p. 2]

The extrinsic calibration of a camera determines its cartesian position as well as its rotation in space relative to a known reference point. In robotics it is therefore also commonly referred to as *hand-eye-calibration* due to the fact that the robot base is used as the known reference point. By moving the robot with a checkerboard attached to multiple distinct positions in space with unique rotations the relative position of the camera system can be triangulated from the known robot positions and the known checkerboard dimensions. This procedure allows the determination of the camera position relative to the robot flange at any moment, since the kinematics of the robot, i.e., the transformation from the robot base to its flange, are also known and can be calculated for every axis configuration of the robot at runtime. (cf. [134])

### Stereoscopic Cameras

Stereoscopic cameras are fashioned after the principle of human vision in that they infer three-dimensional information from a pair of images. In order to enable the computational system to reconstruct three-dimensional information from the incoming stereo image streams it must first determine which sections of the images correspond to each other. Based on these correspondences in the two images, the known and fixed

distance between the two cameras as well as their calibration matrices (cf. Section 2.1.2), the three-dimensional coordinates of all found correspondences can be triangulated as depicted in Figure 2.4. An example of such a system is the stereoscopic sensor produced by roboception [99].

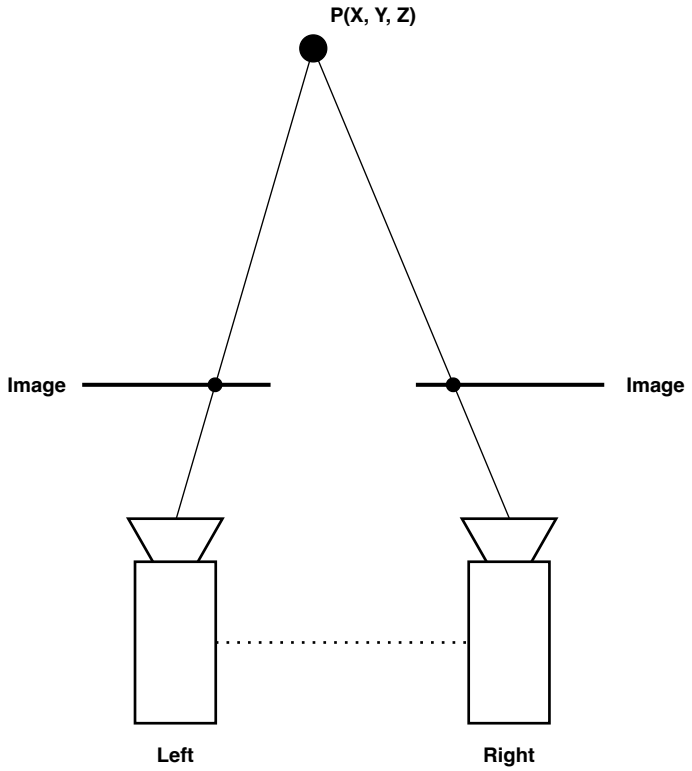



**Figure 2.4.** Triangulation of three-dimensional coordinates through the projected image coordinates of corresponding features. $P$ denotes the three-dimensional coordinates of one found correspondence between the two individual camera images.

### Thermal Cameras

Thermal cameras are specifically targeted at the detection of infrared radiation or heat emanating from objects into the surrounding space. While they used specialized imaging sensors for this purpose, the format they provide their output data in is identical to that of cameras. The meaning of the individual values in the monochrome output image, however, differs from regular RGB camera images in that it denotes the temperature value measured at that specific pixel. The similarity to typical camera sensors also allows thermal cameras to be calibrated in much the same manner, except that a detection pattern must be employed, which reflects or emits thermal radiation to such a degree that the difference in temperature relative to the surroundings can be picked up by the thermal camera. An example of such a thermal imaging system is the A700 camera from Teledyne FLIR [114], which produces thermal images at a resolution of $640 * 480$ pixels at $30\,\mathrm{Hz}$ And can differentiate temperature differences as small as $40\,\mathrm{mK}$.

### 2.1.3 Time-of-Flight Sensors

The basic principle of time-of-flight cameras is fairly simple. Such sensors contain transmitters and receivers for specific light pulses. In each detection cycle the transmitter sends out such a pulse and the sensor records the time of transmission. The light pulse is then reflected off objects in the field of view of the sensor, and the reflected pulse is then detected by the receiver. The time of reception is also recorded by the sensor. With the roundtrip time $t$ (divided by two, since we wish to calculate only the distance to the object, not of the roundtrip) and the known speed of light in air $c$ the distance $d$ can be determined with the formula:

$$d = \frac{ct}{2} \tag{2.1}$$

Time-of flight cameras, such as the Basler tof640 [14] employ infrared strobes that are pulsed in a specific pattern which can then be interpreted by the infrared imaging sensor. These sensors are comparable to classic imaging sensors in their basic structural makeup, except that they only detect a single range of wavelengths, which lies within the infrared spectrum. The optical filters applied to the sensor filter everything but this specific range of light.

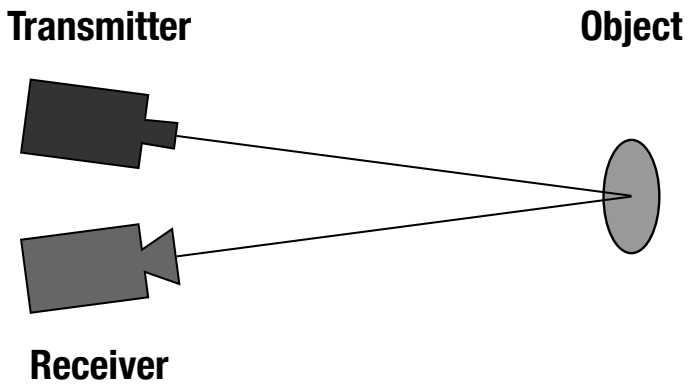



**Figure 2.5.** General principle of time-of-flight sensors.

#### LiDAR

The name for the sensing method called **Li**ght **D**etection and **R**anging (LiDAR) is derived from the acronym RADAR, which stands for **R**adio **D**etection and **R**anging. The similarity between the two technologies lies within the principle of rotating the measurement assembly continuously while performing time-of-flight measurements to determine the distance of objects detected by the sensor. (cf. [52, p. 341ff])

A staple of conventional operator safety systems, two-dimensional scanning LiDAR sensors can be found in many existing industrial robot applications today. An example

of such a sensor is the microScan3 from SICK [110], which can detect objects within a range of $270°$ and at distances of up to $5\,\mathrm{m}$.

Three-dimensional LiDAR systems combine multiple measurement assemblies into a single rotating measurement device and consequently can measure the distances to objects at every point of a revolution on multiple independent levels and thus produce three-dimensional output data. A popular example of such a system often employed in autonomous vehicles is the Ultra Puck from Velodyne [122], which offers an impressive field of view of $360°$ horizontal and $40°$ vertical.

All LiDARs produce point clouds as output, which is a data format describing discrete measurements in three-dimensional space, which may be annotated with additional data, such as the light intensity recorded by the sensor upon reception.

#### Ultrasonic Sensors

Ultrasonic sensors also operate on the principle of time-of-flight measurements, however they emit ultrasonic pulses and measure the time elapsed until the pulse is recorded back at the receiver. Ultrasonic refers to sound waves that possess frequencies beyond the normal hearing range of humans. In contrast to the previously described sensors which use various types of electromagnetic radiation (LiDAR: light, RADAR: radio waves), ultrasonic waves are pressure waves. The consequence of this fact is that ultrasonic waves require a medium such as air to be transmitted through, whereas electromagnetic radiation can travel through a vacuum without obstruction. It also means the speed of sound waves is heavily dependent on the medium it travels through. Using the known speed of sound in air, since this is where distance measurements are typically performed, the distance to an object can be calculated using the same formula as for light-based time-of-flight sensors, except with the speed of light substituted with the appropriate speed of sound through air: (cf. [52, p. 213ff])

$$d = \frac{vt}{2} \tag{2.2}$$

where

$d$ : The resulting distance to an object

$v$ : The speed of sound through air which is $343.2\,\mathrm{m/s}$

$t$ : The roundtrip time measured by the ultrasonic sensor

## 2.2 The Robot Operating System (ROS)

Unlike its name might suggest, the *Robot Operating System* (ROS) is not actually an operating system, but rather an open source robot middleware. The development of ROS began with the open source organization Willow Garage as a control system for their own PR2 personal robot assistant. Since then maintenance and further development of ROS have been taken over the Open Source Robotics Foundation and a steadily growing open source community. Even commercial companies offer ROS drivers for

their hardware products within this community. Since the first version of ROS was not seen as a suitable basis for commercial products, however, development of the second major version of ROS began around 2015, redesigning the architecture from the ground up and thereby focusing on issues of stability, realtime performance and security. (cf. [115])

Basically, ROS is a network-based communication middleware which passes messages between distributed services. Every ROS node can offer or call services and subscribe to topics which are the named communication channels in the ROS infrastructure. Upon this underlying communication infrastructure, many tools and applications have been built. ROS itself contains tools for the visualization of sensor data and robot motions, the motion control and various development tools. The ROS ecosystem has many available drivers for robotics hardware including robot arms, grippers and various sensors. The control software enables applications ranging from motion planning and control of robot arms to autonomous navigation of mobile robots. (cf. [84])

**Chapter Summary**. This chapter gives an overview of two application examples which guided the development of the *SensorClouds* architecture.

# 3 Application Examples

## 3.1 Healthcare Robots (Project: SINA)

During the course of the research project SINA [21] - funded by the German Federal Ministry of Education and Research (BMBF) - a mobile assistance platform (seen in Figure 3.1) for elderly care applications was developed. The design of the mobile manipulator was guided by three main use cases:


**Figure 3.1.** The SINA prototype of an assistive mobile robot platform.

- **Pick up objects from the ground**
  Elderly people suffer from an increasingly constrained range of motion. This especially affects bending over and picking up objects that may have fallen to the ground, such as e.g., items of clothing or keys. Hence, the mobile robot was designed to be able to reach the ground, pick up items and subsequently hand them over to a person.
- **Collaborative transport of objects**
  Transporting heavy items also becomes progressively more difficult for the elderly, yet common household chores such as washing clothes can already involve weights far surpassing their capabilities. To this end, the mobile robot was fitted with a cargo area reachable by the manipulator arm. Items such as a clothing basket can thus be lifted to a comfortable height for the person and stowed on the cargo area for transport. This relieves the strain of carrying heavy objects by transferring the majority of the weight to the robot leaving the person only to stabilize the object in transit.
- **Autonomous retrieval and transport of objects**
  Since getting up from a chair or bed becomes increasingly arduous for elderly

people it can be beneficial to offer assistance in the retrieval of items such as a bottle of water. Additionally, the robot can also retrieve and provide supplies to nurses and caregivers while performing treatments on their patients in order to reduce the number of menial retrieval tasks performed by the staff and allowing them to focus on personal interaction with the patients.

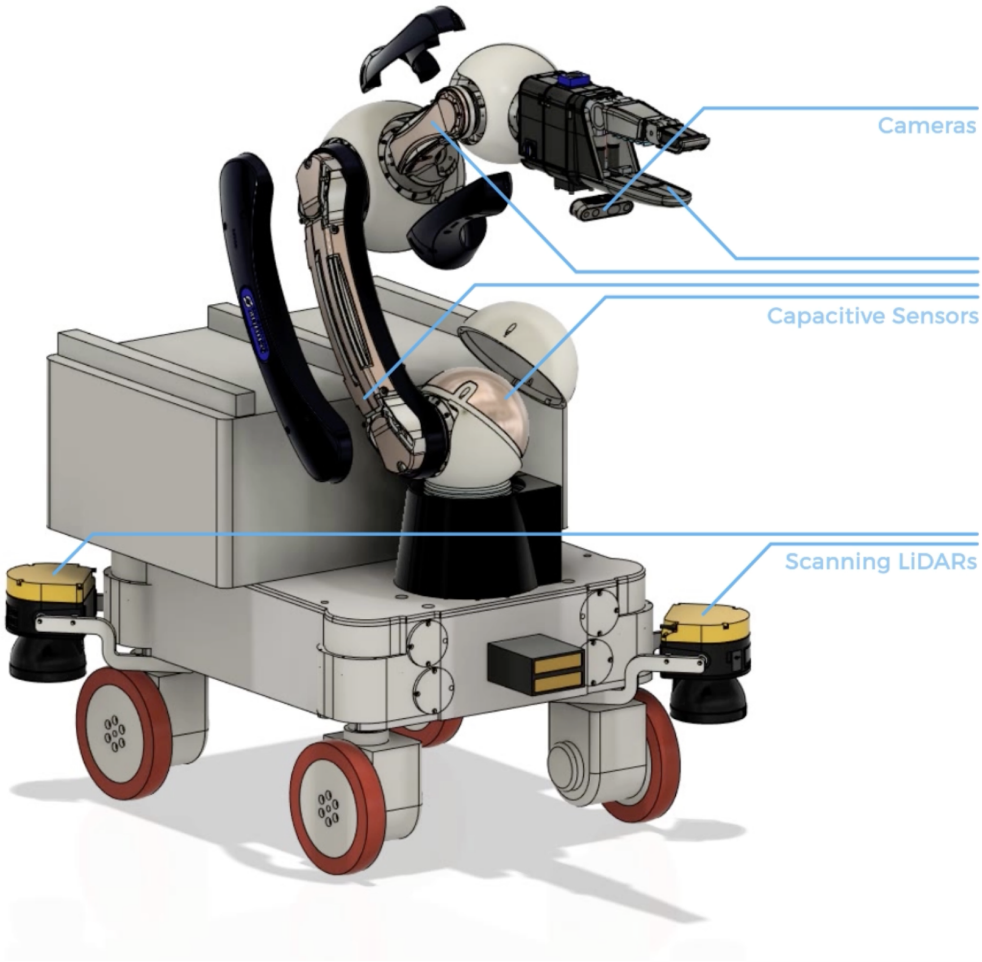



**Figure 3.2.** Overview of sensors installed on the SINA platform.

The mobile platform for the SINA prototype is based upon the MPO-700 platform [83] developed by neobotix [82] which is a spin-off company of the Fraunhofer Institute for Manufacturing Engineering and Automation (IPA) [46]. The robot mounted on the platform is an LWA 4P (Light Weight Arm) [106] developed by SCHUNK [107] and is fully integrated, meaning no external control cabinet is required to control or power the robot axes. It has since been discontinued by the manufacturer. The additional structure at the back of the platform is a custom component housing additional safety components as well as batteries and doubles as the heightened cargo area for transport of heavy items. The overall mechanical setup of the prototype offers a good compromise

between the various constraints of the previously introduced use-cases. Specifically, it allows the robot arm mounted on the platform to reach the floor, the top of tables and cabinets as well as to pass objects to humans in sitting or standing positions.

The construction of this mobile robot prototype was an integral part of the project as it produced a highly customizable testbed for the algorithms and hardware components developed as part of the main research goals:

- **Basic skill: Handover of Objects**
  The theme of the research grant concerned the development of basic robot skills for assistive robots. Project SINA focused on the skill of handing over objects to people. The remaining goals were set in order to realize this basic skill.
- **Multimodal Perception of Surroundings**
  Initially the idea was to create a simple geometric model within which to store sensor correlations, i.e., annotate at which points multiple sensors detected an object. Over time this idea was elaborated upon which ultimately led to further research and finally culminated in this work.
- **Readable and Expected Robot Behavior**
  Especially for untrained operators and specifically elderly people, having to adapt to the idiosyncrasies of such a complex assistive robot system is highly undesirable. To mitigate this circumstance the goal was to develop a motion planning suite capable of controlling the robot in a manner that is easily predictable by outsiders and conforms to typically acceptable movement, especially in close proximity to humans. This includes constraints such as not passing a person closely outside of its field of vision and passing objects at comfortable positions along trajectories similar to those typically employed by other humans.
- **Capacitive and Tactile Sensors**
  In situations involving close interaction between robots and humans most sensor systems reach their limits. Since the distance of these interactions is typically below the minimum detection range of cameras and LiDAR systems, a different type of sensor is required. In order to better monitor and safeguard close interactions in the vicinity of humans combined capacitive and tactile sensors were identified as suitable to this task. Therefore, the final project goal was the further development of these sensors and their integration with the other systems.

Overall the sensors attached to the platform for various detection tasks are comprised of the following (see Figure 3.2):

- **Capacitive sensors on the robot arm**
  The capacitive sensors mounted to the robot structure underneath its protective plastic housing were stipulated for the detection of persons in direct proximity to the robot arm. Since a camera system's ability to reliably detect humans in the robot's surroundings can always be impeded by objects or even the manipulator itself, capacitive sensors are vital to a dependable safety system for applications in HRI.

- **Capacitive sensors in the gripper**
  The gripper must naturally also be secured against colliding with or actually gripping a human hand. To this end, the gripper was also fitted with capacitive sensors albeit with additional tactile and material sensing capabilities in contrast to those deployed on the robot structure. This enables the gripper controller to distinguish between the object which is to be gripped and a human hand, even if a person were to position their hand between one of the gripper's jaws and the object.
- **RGB-D Camera**
  An additional RGB-D camera, specifically the Intel RealSense D435i [57], was mounted on the bottom jaw of the gripper. This camera was intended to enable the exact localization of objects for gripping. As long as no object is in the gripper, thus blocking the camera's view to a large extent, the additional sensor data can be fed into the environment model of the mobile robot's surroundings as well.
- **Scanning LiDAR**
  Two scanning LiDARs positioned at two opposite corners of the platform and with a detection angle of 270° are sufficient for the use of simultaneous localization and mapping (SLAM) algorithms. With the help of the LiDAR sensors and SLAM algorithms the platform can navigate autonomously even in unstructured environments such as nursing homes and domiciles of elderly people. For this purpose the model microScan3 from SICK [110] was chosen, since it not only provides functional safety measures for the safety of humans in the vicinity of the platform, but also supports simultaneously transmitting the raw sensor data to a connected computer for further processing.
- **Time-of-flight camera**
  In order to get a broader overview of the room an additional depth camera based on the time-of-flight principle was included in the overall setup in order to ease navigational planning and the retrieval of objects, which otherwise would have to be painstakingly searched for by the mobile robot. The specific camera deployed for this purpose was a Basler tof640 [14].

## 3.2 Industrial Robotics (Project: KoARob)

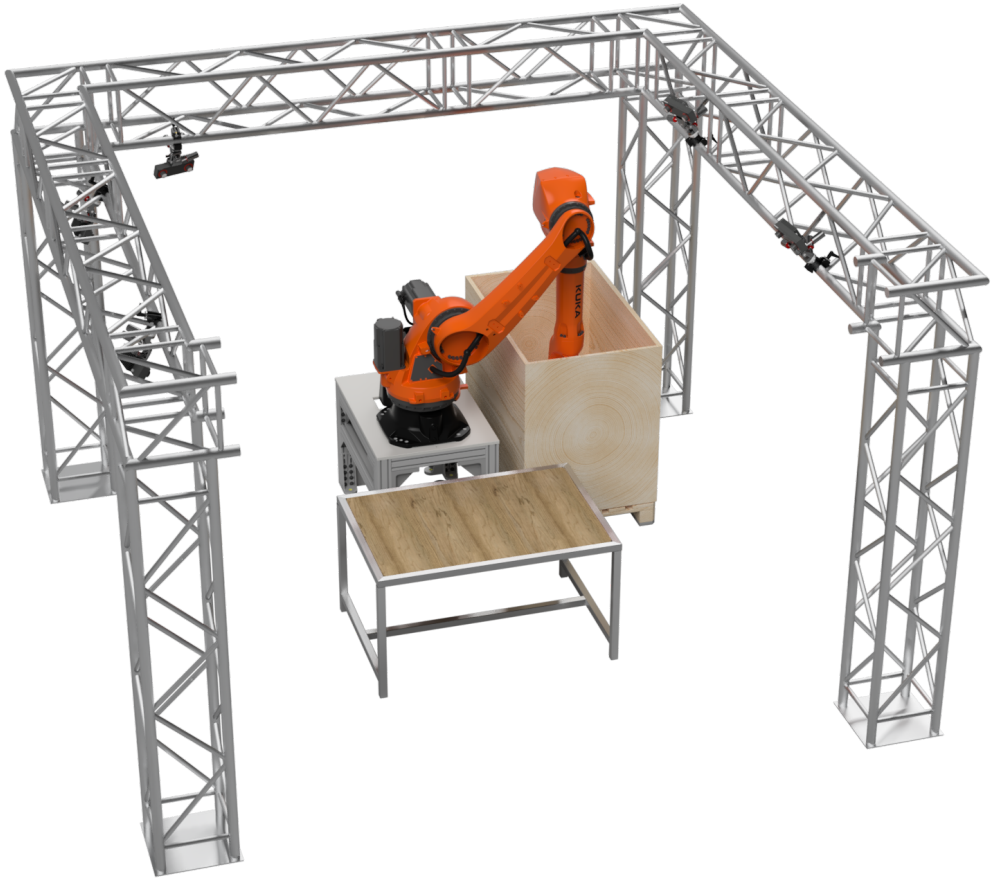


**Figure 3.3.** Overview of the robot cell developed in KoARob.

The project KoARob aims at developing a redundant safety system by combining the data from multiple cheap off-the-shelf sensor systems. The motivating scenario this system is being developed for concerns the partial automation of returns processing in a logistics warehouse. Packages received from customers returning items must be opened and their contents inspected for damage by a human inspector, but the process of depalletizing the incoming packages from large containers is suitable for automation. When space is constrained, especially in smaller businesses, complex and voluminous conveyer belts as well as cordoned off robot systems are impractical solutions. However, increased throughput and prevention of repetitive stress injuries (RSI) to workers are also important considerations when designing the process of returns inspection.

In order to demonstrate the research results in a practical setting, an exemplary robot cell for such returns depalletizing tasks was developed (see Figure 3.3). At the center of the cell is an industrial robot which performs the task of transferring packages individually from the pallet to the table where the human operator performs his task of inspecting their contents and sorting them for further processing. The packages are delivered by

the robot to a fixed set of predetermined locations from which the worker can retrieve them for their subsequent inspection task.

The system architecture of KoARob hinges on a newly developed safety controller, capable of processing multiple input data streams in a common format and verifying the correctness of the spatial occupancy by comparing all input data with each other. The main sensor system consists of five rc_visard stereoscopic cameras from the company roboception GmbH [99] arranged along the truss surrounding the work area of the KUKA KR16-2 robot [62]. Additional sensors monitoring the work area include a 2D-LiDAR from SICK [110], a FLIR thermal camera [114] and capacitive sensors integrated into the custom vacuum gripper assembly mounted to the robot flange.

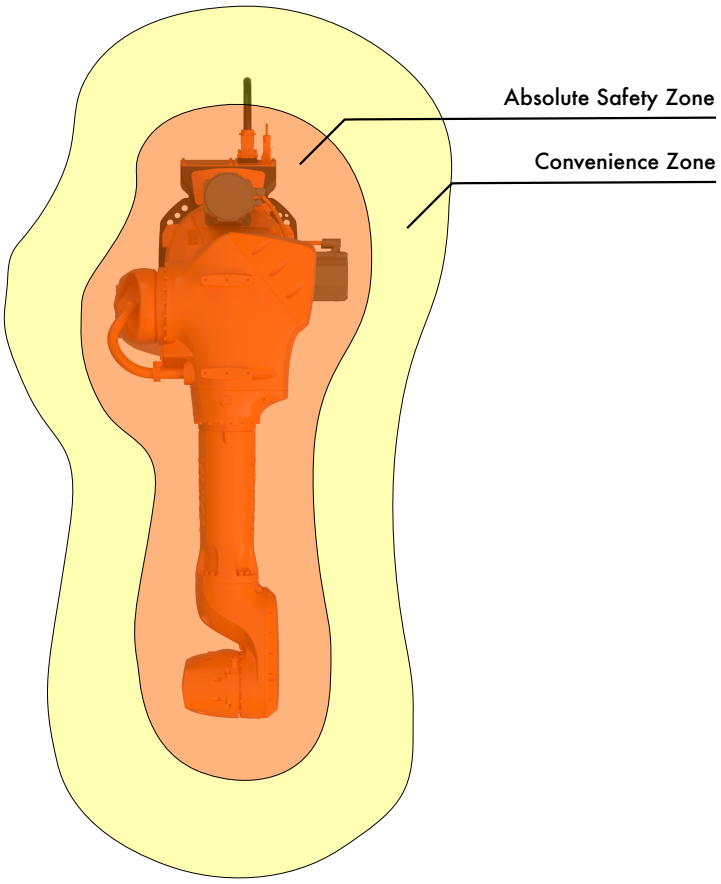



**Figure 3.4.** Zones of the two-staged safety concept employed in KoARob.

In order to balance the usability of the resulting robot system with the required safety guarantees a *two-staged safety* concept is proposed (cf. Figure 3.4). The first stage (*absolute safety stage*) behaves similar to typical safety systems utilized in industrial applications today in that it performs an emergency stop of the robot once an object is detected in close proximity to the robot in an effort to prevent impending collisions.

This, however, leads to downtimes of the robot and consequently productivity, especially if the robot cannot autonomously recover from the stopped condition thus requiring a manual restart operation. To alleviate the need for frequent emergency stops, the second stage (*convenience stage*) attempts to plan the robot's trajectories around possible areas of collision such that the absolute safety stage ideally never needs to be activated.

Such prescient planning of robot motions requires detailed knowledge or, if possible, even a limited form of precognition with regards to the motion of humans in the workspace of the robot. To this end, various machine learning approaches are employed to detect humans and their movements in the workspace of the robot:

- **Human Pose Recognition**
  The first step is to localize all humans present in the work area. This information can already be used to augment spatial data of the robot's surrounding for the purpose of distinguishing areas occupied by humans from those taken up by inanimate objects. The former can then be circumnavigated at greater distances in order to ensure greater operator safety.
- **Human Pose Prediction**
  Human pose prediction or forecasting concerns itself with projecting a human's current pose and trajectory forward in time with the aim of predicting which areas will soon be inaccessible to the robot and planning future trajectories around and away from these regions. Human pose recognition is a prerequisite for this process.
- **Human Activity Recognition**
  Based on the current and future human poses and a semantic definition of previously determined activity classes, human activity recognition can be employed for the sake of attempting to estimate the current task or action a human operator is performing. This can be beneficial for determining general areas where humans are most likely to be further ahead into the future than simple pose forecasting would allow estimating.

These methods can be utilized in the KoARob scenario to create a more intuitive collaboration experience for the worker and thus shifting the perception from a mere tool to more of a robotic colleague. By predicting the worker's future actions, the robot's choice of placement position can be influenced to prefer locations which are less likely to interfere with the human and thus cause potential collision situations. Moreover, attempting to minimize the required simultaneous access to the same workspace areas by humans and robots, arguably leads to a greatly reduced need for the absolute safety system. This also leads to the robot system being less disruptive to the entire workflow since it is not physically in the way of the human worker and it does not need to be restarted as often.

Since current machine learning algorithms are not trustworthy enough to employ them in safety critical systems as single source of information, KoARob utilizes multiple such algorithms simultaneously and compares their respective outputs with each other and also conducts feasibility checks based on the bare geometric information available to the system. This redundancy falls in the same category as typical functional safety

equipment used in industrial applications today in that no single source of information or communication channel is ever trusted alone, but only ever in combination with at least one other.

**Chapter Summary**. This chapter gives an overview of the *SensorClouds* architecture for processing multi-modal point clouds. Based on the requirements formulated in this chapter, the basic concepts enabling the dynamically configurable, yet real-time capable architecture are explained. A brief example of the programming interface for application developers using the *SensorClouds* architecture is given to illustrate its simplicity in commanding powerful functionalities.

# 4 SensorClouds

## 4.1 Requirements

Before the architecture of *SensorClouds* is presented in detail, this section will list the requirements imposed on the architecture and reference implementation (cf. Section 7.3).

### 4.1.1 Functional Requirements

**FR 1 Integration of all sensor data into a global model**
All sensor data must be combined in a global, three-dimensional model such that global reaction strategies and thresholds can be defined. This must also include data that is not naturally three-dimensional, but which can be transformed into its three-dimensional context.

**FR 2 Real-time execution**
Quick reaction times are paramount in HRC applications, since humans in the workspace must be accurately detected before collisions can occur. The exact cycle times are naturally dependent on the specific amount of processed sensor data and the algorithmic modules employed and must therefore be evaluated separately for each concrete application. Overall, the framework should be capable of targeting overall execution times per cycle of under $30\,\mathrm{ms}$.

**FR 3 Dynamic fusion of various sensor modalities**
In order to increase the robustness of the HRC application, the architecture shall facilitate simple fusion of various sensor modalities into a global environment model and enable modality-specific as well as modality-agnostic processing of all data present in the global model.

**FR 4 Modular and reusable architecture**
Applications of sensor fusion are often comprised of similar algorithmic building blocks adapted to the specific context in which they are to be employed. Consequently, the capacity for reuse of such applications is severely limited. The architecture of *SensorClouds* shall therefore enable the generalized implementation of processing algorithms in form of modules, which can then automatically be adapted to the concrete sensor data in a specific application by the framework.

**FR 5 Scalable Parallel Execution**
Computer vision tasks are inherently optimal candidates for highly parallelized execution of processing algorithms, since each individual data point can typically be processed independently. Consequently, the framework must support the parallel execution of modules on varying sizes of input data sets and scale dynamically to fill the available processing resources. This can also significantly aid in the fulfillment of **FR 2**.

**FR 6 Compatibility with established software ecosystems**
Building upon existing infrastructure and established ecosystems ensures the immediate availability of many more pre-existing resources and features than would otherwise be the case for a newly developed framework. Especially the Point Cloud Library (PCL) and the Robot Operating System (ROS) are highly

relevant ecosystems pertaining to the processing of point clouds and control systems for robots as well as integration of sensor data into robotic applications, respectively. *SensorClouds* shall therefore ensure full compatibility with the data types of interfaces of these two frameworks.

### 4.1.2 Non-functional requirements

Beyond these functional requirements, a number of non-functional requirements were defined for the *SensorClouds* architecture:

**NFR 1 Abstraction from parallel computing**
Implementing algorithms for massively parallel computing entails a steep initial learning curve, since many concepts differ from traditional single and even multi-threaded programming. To alleviate this circumstance, the framework shall abstract as much of the added overhead and allow developers to focus on their concrete implementation.

**NFR 2 Programming paradigm**
The programming paradigm for application developers shall be shifted from the typical focus on direct processing of individual data sets to the more abstract specification of connected devices and processing modules.

**NFR 3 Keep with industry standards**
The framework shall refrain from employing exotic methods and complicated custom tooling.

**NFR 4 IDE support**
The framework shall be natively usable in any IDE and support all features such as code completion without the need for custom plugins.

## 4.2 Architecture

The following section gives an overview over the main architectural building blocks of the *SensorClouds* architecture and essential algorithms and concepts required to enable the architecture to function correctly and provide certain guarantees on modularity and interoperability. Throughout the remainder of this work, the term *data point* will be used to denote a collection of values ascribed to a specific point in space, such as an object containing both $x$, $y$ and $z$ cartesian coordinates and a color value in RGB. A *data field*, on the other hand, is a single one of the previously mentioned entries so $x$, for example, is a data field belonging to a specific data point. The *SensorClouds* architecture clearly differentiates between two distinct views of a sensor processing application: the *application developer*, who creates programs performing specific tasks with the help of the *SensorClouds* framework and the *module developer*, whose task it is to create the modular building blocks from which the application developer can draw from when solving concrete sensor processing tasks. The interface for application developers will be presented in more detail at the end of this chapter (cf. Section 4.3) after the in-depth explanation of the architectural concepts, while the interface for module developers will be introduced at a later point along with details of the reference implementation of the *SensorClouds* architecture (cf. Section 7.2).

### 4.2.1 Main components

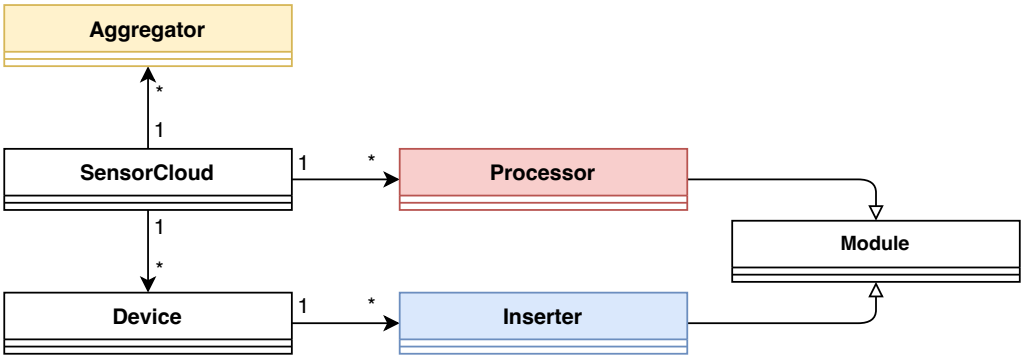



**Figure 4.1.** Basic architecture components of *SensorClouds*.

To realize the previously specified requirements of the architecture, *SensorClouds* defines the following basic components:

***SensorCloud***
The *SensorCloud* class is the main component of the *SensorClouds* architecture and as such responsible for managing all stages of the application lifecycle. Additionally, it conceptually acts as the interface to the global data model (see Section 4.2.5). Although many concrete operations are handled by various other classes internally, this class exposes all the necessary interface options for quickly and easily building applications with *SensorClouds* and delegates tasks to the appropriate subcomponents where required.

***Device***
Device drivers handle all operations related to concrete sensor devices. This includes the actual communication with the devices for the retrieval of configuration and sensor data as well as the provision of the acquired sensor data to the *SensorCloud* in order to be further processed by other modules. An example of this would be an RGB-D camera whose input data stream, cartesian position and calibration data as well as any other parameters describing properties of the camera are all managed by the device driver. This serves to shift the programming paradigm from raw data processing to device-centric applications as per **NFR 2**.

***Inserter***
Insertion modules are tasked with receiving input data from device drivers and inserting it into the global environment model. This can be as simple as transferring the values directly, but may include much more involved processes to determine where in the global model the data needs to be inserted. Data from RGB-D cameras can, for example, directly be inserted, whereas an RGB camera without depth information must first be projected into three-dimensional space, which is a more involved process and requires three-dimensional information to be present in the model before the image can subsequently be projected onto the same.

***Processor***
A *Processor* operates solely on the global environment model and can be employed to perform processing tasks on the contained data after all input data has been inserted into the model. This can include operations such as filtering data points with insufficient data or correlation or cutting known objects from the model in order to ease the calculation of impending collisions.

***Aggregator***
The actual fusion of data points within the global model is performed by *Aggregators*. Every write operation by other modules is channelled through these. The processing method of an *Aggregator* receives as input the old value of the data field alongside the new value to be written as well as any intermediate variables it has defined. The resulting output of the *Aggregator* is then committed to the global model. Unless otherwise implemented by custom *Modules*, voxels always contain only the result of the aggregation calculation and none of the constituent data points individually.

### 4.2.2 Type System

*SensorClouds* maintains two separate type systems for data fields (cf. Figure 4.2). The internal definition of types which is directly based on that of point cloud messages in ROS [100] is a simple enumeration of the supported types (*DataTypeEnum*). In contrast, the definition of kernel functions for modules must be able to process the built-in types of the programming language (*DataType*) with any accompanying qualifiers (e.g., constant, pointer types, etc.).

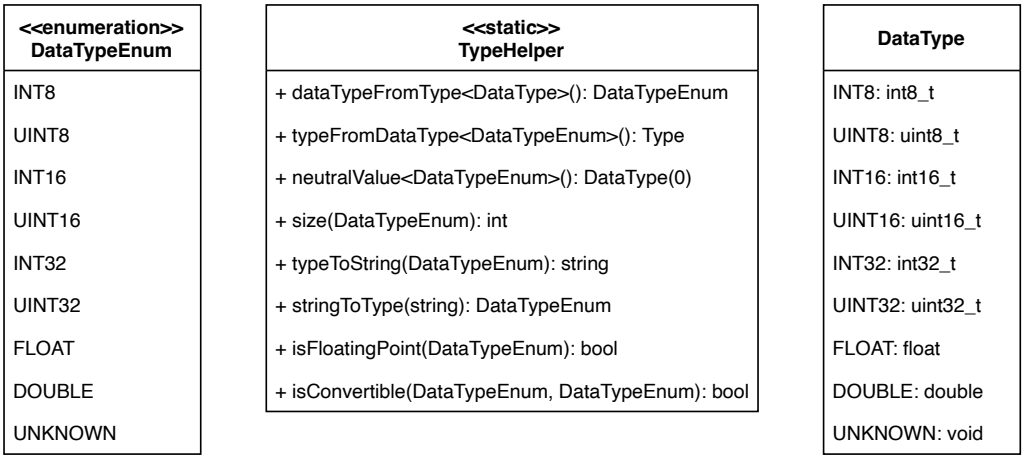



**Figure 4.2.** Type system in *SensorClouds*.

The *TypeHelper* class contains a collection of convenience methods for handling the two type systems. Aside from string conversions and informative attribute functions (e.g., `isFloatingPoint()`) it also contains methods for converting types between the two definitions as well as checking whether the value of a certain type can be contained within another.

### 4.2.3 Modules

*Inserters* and *Processors* are independent modules containing atomic processing kernels (enabling the fulfillment of **FR 5**) in the *SensorClouds* architecture. In order to guarantee the interoperability of an arbitrary combination of modules, the design-by-contract [75] paradigm is enforced throughout the entire *SensorClouds* architecture. This guarantee is also of paramount importance to the fulfillment of **FR 4**, since each individual module would be required to explicitly assure interoperability with other, possibly unknown modules were this responsibility not assumed by the overarching architecture. Design-by-contract aims at improving the reliability of software, especially in object-oriented languages, by likening the construction of software and its constituent functions to a sequence of contracts between a client and a contractor in construction. Each side of such a contract has obligations as well as benefits, which are explicitly spelled out for the mutual assurance of both parties involved in the contract. Applying this analogy to software design, a contract is entered between a subroutine and the calling routine, which specifies the preconditions of the called subroutine, e.g., a data object passed to it must to exist before the subroutine can be called, as well as the post-conditions, e.g., the subroutine guarantees that its results are stored into the previously passed data object after it has completed. Following this paradigm allows stronger focus of each individual subroutine on the task it is intended to perform, since the definition of preconditions shifts the responsibility for error-checking to the calling routine. Preconditions and postconditions together form the contract between callee and caller and moreover provide rudimentary user documentation in the process. Further

details on the enforcement of specific aspects of the design-by-contract paradigm will be described at the relevant points over the course of the following sections. An overview of the modules architecture with all directly involved classes is depicted in Figure 4.3. The diagram is split into two different sections to denote the responsibility for creating and configuring the respective objects.

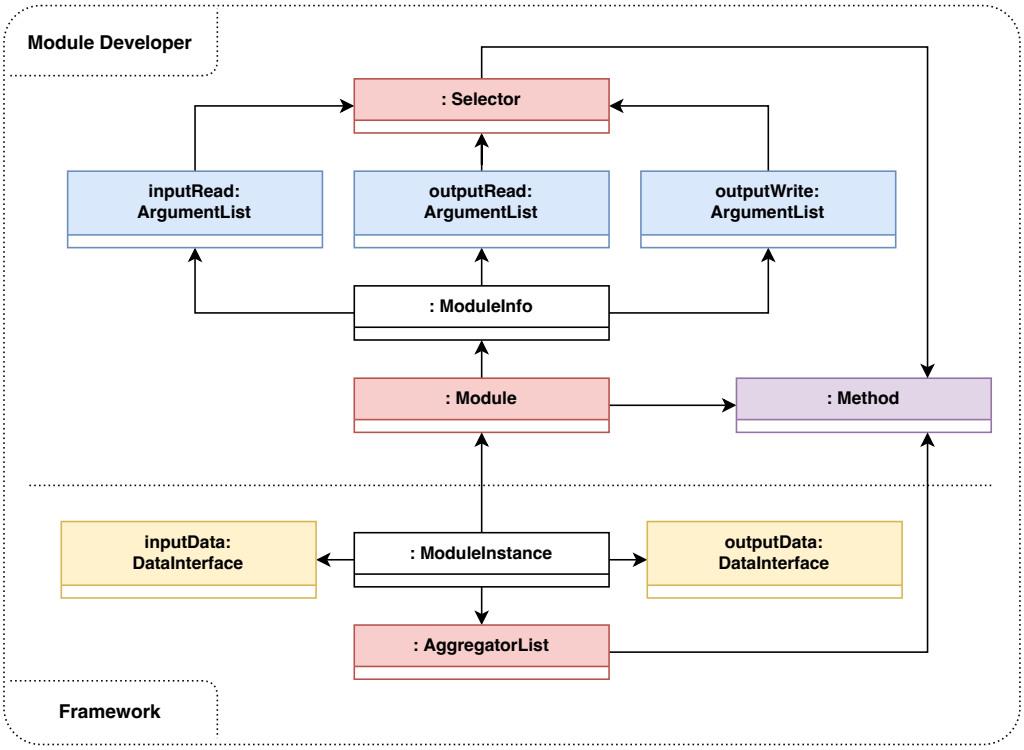



**Figure 4.3.** Module architecture in *SensorClouds*.

### *Module*

*Module* is the basic meta-class which describes all relevant information for the integration and use of a module. Both *Inserters* and *Processors* are derived from *Module* and thus follow the same definitions and lifecycles. This class is later used by the framework to create a concrete *ModuleInstance* according to the application configuration and the current context in which the module is utilized.

### *ModuleInfo*

All relevant information needed to execute a module's kernel function is stored in this class. It contains the definition of the required parameters, the provided result data as well as the pipeline stage (cf. Section 4.2.7) the module can be executed in.

### *ArgumentList*

*ArgumentLists* are employed for the configuration of input and output parameters to

modules. At most a module can have three of these lists, which each have a distinct semantic significance:

- ***Data Input***
  The data fields to be read from the input data stream.
- ***Model Input***
  Data fields which are required from the global model in order for the module to be able to execute correctly.
- ***Model Output***
  The output data written to the global model by the module.

For a single data point, the *ArgumentList* contains a number of data fields and their required types. If multiple data points are to be selected as input or output, the *ArgumentList* can also contain a *DataSet* object. This object contains the same definition of data fields, but can be used to iterate over the resultant number of data points within the processing kernel of a module. The *SensorClouds* Query Language specification described in Section 6.2 can be utilized for the selection of arbitrary data points from the global model. *ArgumentLists* are integral to the enforcement of the design-by-contract paradigm and constitute the pre- and postconditions of a *Module* respectively. Together, the three *ArgumentLists* form the contract of a module, which guarantees the correct flow and processing of sensor data. The simplest possible configuration of these lists would be the definition of cartesian coordinates for both *Data Input* and *Model Output* as is the case for a simple coordinate *Inserter* employed for the insertion of data from a depth sensor.

***Selector***
In case data points shall not be iterated over blindly but specific data points are to be selected depending on the input data or values calculated within a module itself, *Selectors* can be utilized to select the specific data points required. In keeping with the design-by-contract paradigm, this selector and the accompanying *DataSet* specification again form a clear specification of the data accessed by a module and consequently ensure seamless operation with other *Modules*. An example of this is the selection of relevant voxels through raytracing for the projection of RGB images onto three-dimensional data residing in the global data model. *Selectors* are also utilized to select the desired data for output to other applications. A more detailed description of this functionality is given in Section 6.2.

***Method***
Every module defines an executable kernel function which shall be called by the framework at runtime on each data point. The *Method* object is responsible for maintaining a reference to the executable function, the object instance it shall be called upon as well as the parameter definition of the method for later validation against the definitions of data fields in the input and output data of the module.

***ModuleInstance***
After all the definitions of the input and output data are available (e.g., an input point cloud and the generated global model) a *Module* can be instantiated with this specific information. The details of *Module* instantiation are further described in Section 4.2.4.

### 4.2.4 Module Instantiation

A *ModuleInstance* holds references to the original *Module* it was instantiated from, the input and output data interface definitions as well as the list of relevant aggregators to be applied to the written output data. This class is internal to the framework and never directly created by a developer. *ModuleInstances* represent the individual items of the global pipeline ready for execution by the pipeline manager.

A concrete example of a fully initialized *ModuleInstance* can be found in Figure 4.4. In this example a simple *Inserter* for integrating three-dimensional positions without any additional data into the global model (e.g., for a LiDAR; only $x$, $y$ and $z$ values) is depicted.

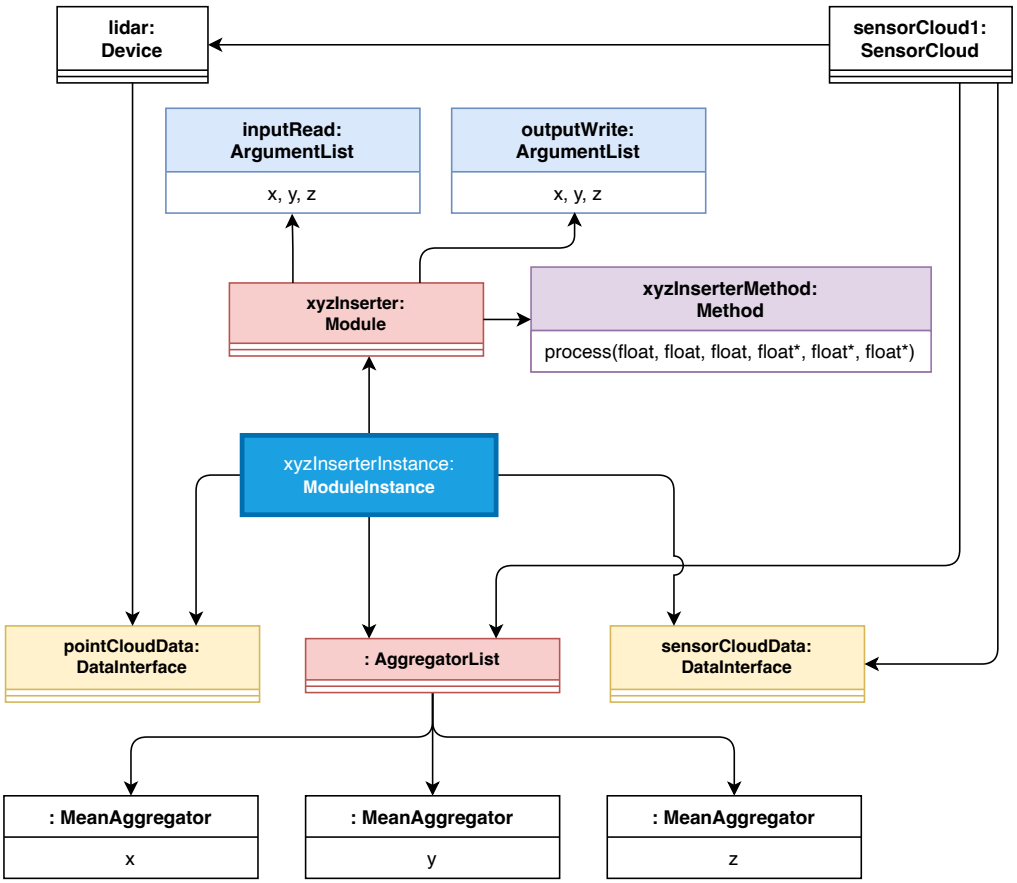



**Figure 4.4.** Object diagram of a concrete *Module* instantiation for a simple *Inserter*.

The instance is configured with two *ArgumentLists* one for the fields it reads from the input data and one for the data it provides to the global model. Each of these are configured with the fields x, y and z, since this *Inserter* only takes the values from the input stream and inserts them into the appropriate voxel of the global model. Hence, the `process()` method of the module takes six arguments, three values and three pointers. The input data (`pointCloudData`) is provided by the device driver for a LiDAR in this example. The output data is, as is always the case for *Inserters*, provided by the *SensorCloud* object (`sensorCloudData`). This also provides a list of *Aggregators* which have previously been configured by the application developer. In this instance, the list contains three *Aggregators* one for each coordinate field, in the form of a *MeanAggregator* which calculates the mean of all values for its respective field upon writing a new value.

Consequently, a *ModuleInstance* possesses all the required data to be executed later as part of the global pipeline. This further satisfies **FR 4** even on the level of executable *ModuleInstances*.

### 4.2.5 Global data model

The global data model in the *SensorClouds* architecture is stored in the form of a *voxel map*. This voxel map is defined by two parameters (cf. Figure 4.5): the first parameter defines the number of voxels in each axis and the second the side length of each individual voxel and thus the resolution of the voxel map.

The reason for this design choice in the overall concept of *SensorClouds* is twofold. The discretization of all sensor data into equidistant voxels allows a significant reduction of the number of data points which must be processed all the while preserving the required level of detail for the intended application by configuring the resolution of the voxel map appropriately. Furthermore, this concept provides a simple and computationally diminished method for fusing data points from multiple sensors which coincide within the same geometric volume. The specific resolution should be chosen according to the requirements of the application being developed. In particular the minimum distance from and the minimum size of objects must be taken into consideration. A larger resolution naturally always incurs more processing time, since more data points must be examined, yet choosing an inappropriately large voxel resolution can cause earlier than desired robot reactions or overzealous restrictions to the robot's range of motion.

An example of the impact of the voxel resolution when attempting to detect a human hand can be seen in Figure 4.6. In most applications, the detection of individual fingers of a human hand will be unnecessary, hence selecting a voxel resolution of $6\,\mathrm{cm}$ or even $3\,\mathrm{cm}$ will not be pertinent to the task at hand. Choosing a resolution of $24\,\mathrm{cm}$ in this case would lead to premature reactions of the robot and restrict the options for circumnavigating obstacles, since a larger volume in the robot's workspace will be blocked by the perceived interference. Consequently, a voxel resolution of $12\,\mathrm{cm}$ constitutes a sensible choice for detecting objects in the scale of a human hand.

By employing a voxel map evenly partitioning the surveilled space in the vicinity of a robot, sensor fusion can fairly easily be performed based on geometric coincidence. All

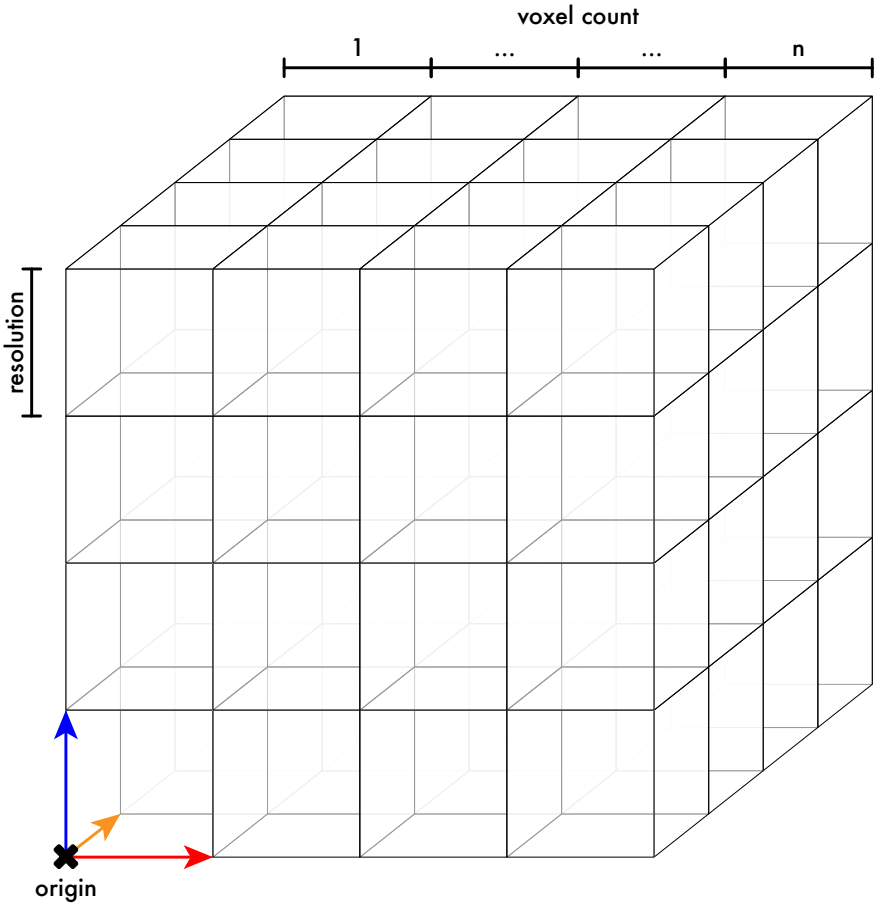



**Figure 4.5.** Geometric specification of the voxel map serving as the global data model. The configuration shown places the origin in the lower left corner of the entire space and is further determined by the desired resolution and number of voxels in each axis.

sensor data located in three-dimensional space can be assigned to the corresponding voxel containing the exact coordinates of the data point, thus satisfying **FR 1**. After all input data has been inserted into the voxel map in this fashion (cf. Figure 4.7), the data contained in each individual voxel can be fused into a new resultant data point for export or further processing, as per **FR 3**. This fusion process can either be performed after all *Inserters* have completed their work by a separate *Processor* or in tandem with the insertion by *Aggregators*.

An example of such a fusion operation can be seen in Figure 4.8. Here the data points from three different sensors (depicted in red, green and blue) coincide within the same voxel, and their exact coordinate values are to be fused into one resultant value. Figure 4.8 shows how the values contained within each voxel can be processed in order to receive a resultant value per voxel (depicted in black), which can later on be used directly or in further calculations.

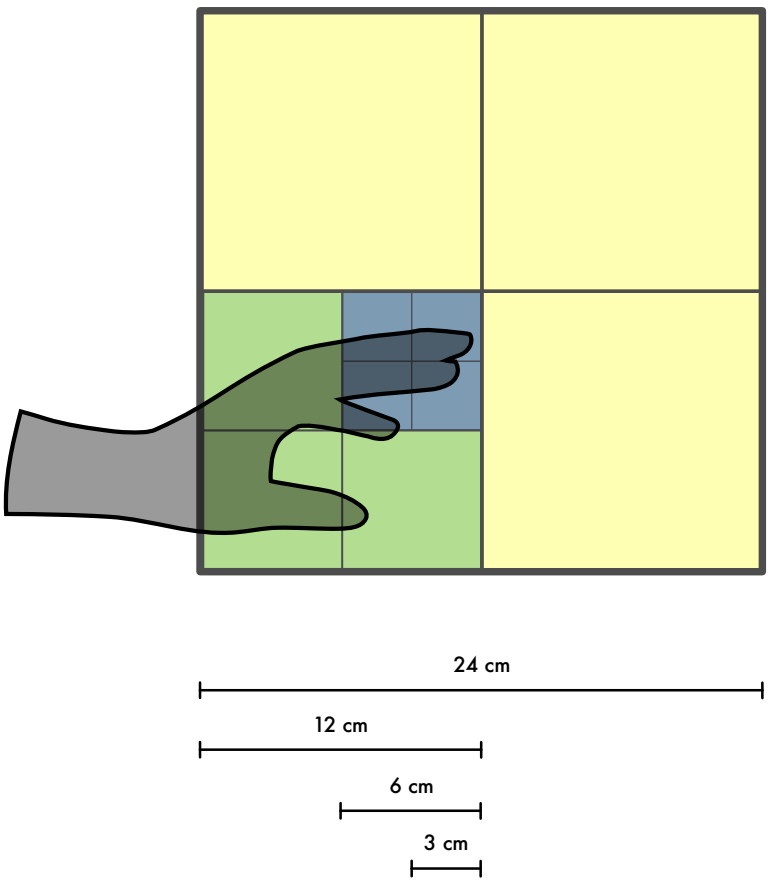



**Figure 4.6.** Impact of different voxel map resolutions on the detection of objects. The image shows a frontal two-dimensional projection of a human hand and the corresponding partitioning of voxel maps of different resolutions.

The storage of a voxel is allocated according to the configured *Modules* of the application (cf. Figure 4.9). Once all *Modules* have been configured, all the output definitions are combined and constitute the data partition of a voxel's storage in the global voxel map model. The hidden partition of a voxel's storage is comprised of all the temporary variables per data field as per the definition of the chosen *Aggregators* for each field. Both partitions together form the per voxel storage definition which is allocated consecutively in memory for every voxel in the voxel map.

### 4.2.6 Static analysis

In order to ensure the correct operation of all modules on their respective input and output data and thus guarantee that **FR 4** and the design-by-contract paradigm can be fulfilled correctly, a static analysis of all modules must be performed once the entirety of metadata is available. The static analysis is one of the methods of enforcing the design-by-contract paradigm in that it checks the modules for compatibility with the existing

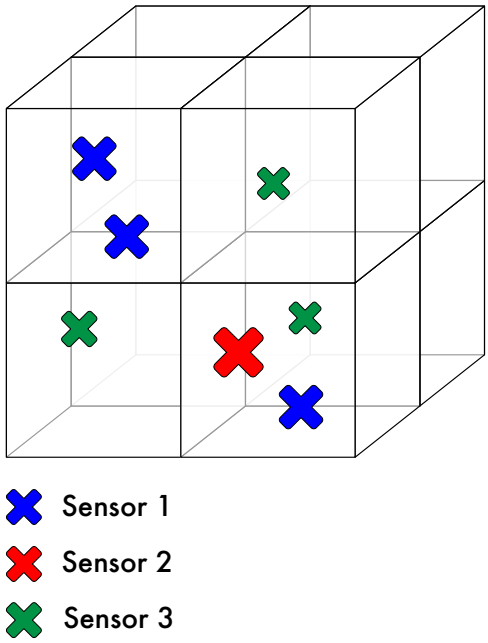



**Figure 4.7.** Example of a 2 x 2 voxel map with data points inserted from three different sensors. Each data point is inserted into the voxel map according to its associated coordinates. All additional data from the original sensor is written to the voxel's storage as well.

input and output data streams and adherence to the contract by the implemented *Method*. The means of reporting the results of the analyses performed by the *StaticAnalyzer* are twofold. One is the direct output of textual messages to the console and the second is the return of the combined maximum error level which occurred during the analysis. The following error levels are defined:

- **OK**
  All definitions and types are correct and the *ModuleInstance* is directly executable without any intervention.
- **WARNING**
  Type conversions must be introduced, which may produce unexpected numerical results, but the *ModuleInstance* can still be executed.
- **ERROR**
  The type definitions are wholly incompatible and the *ModuleInstance* can therefore not be executed. The analysis can, however, continue and still output further errors and warnings should they present themselves. Repeated trial and error runs can ideally be eliminated by this best effort approach to finding all errors and warnings in one analysis process.
- **FATAL**
  An error was encountered which forced the analysis to be cancelled prematurely.

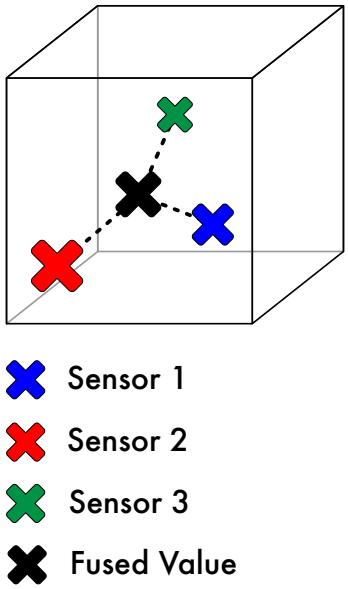



**Figure 4.8.** Fusion of multiple data points from different sensors within the same voxel. The blue, red and green points denote the respective data points from the different sensors, while the black point is the resultant data point of the sensor fusion. In this case the exact coordinates of the individual data points are fused by a mean calculation.

This occurs for example when the number of parameters differs between the definitions, which makes further analysis moot.

An example of the static analysis process for a simple *Inserter* can be seen in Figure 4.10. All the defined input variables of the *Module* are matched against their counterparts in the input data source and the variables read from the global model are matched against the model definition (`checkModuleArgDefinitions()`). Only the specified types are considered when determining their compatibility and not for example the actually used value range of a variable in order to ensure the stable execution of the application. In the same vein, the static analysis is far more strict regarding type compatibility compared to common compilers, which allow the implicit conversions such as from larger integer types to smaller ones, hazarding the consequence of value overflows. Such a determination can be seen with respect to the input variable $y$ in the example (Figure 4.10 bottom left) where the conversion from a 32-bit integer to a 16-bit one could lead to incorrect values later on. The same holds true for floating point types when attempting to convert from 64-bit double precision floating point values to 32-bit single precision ones as for the $x$ variable in the output definition (Figure 4.10 bottom right). Since floating point conversions of integer types are never completely exact, attempting this will lead the *StaticAnalyzer* to produce a warning indicating to the developer that the conversion may lead to a loss of precision, which is the case for the $z$ variable in the input definition (Figure 4.10 bottom left).

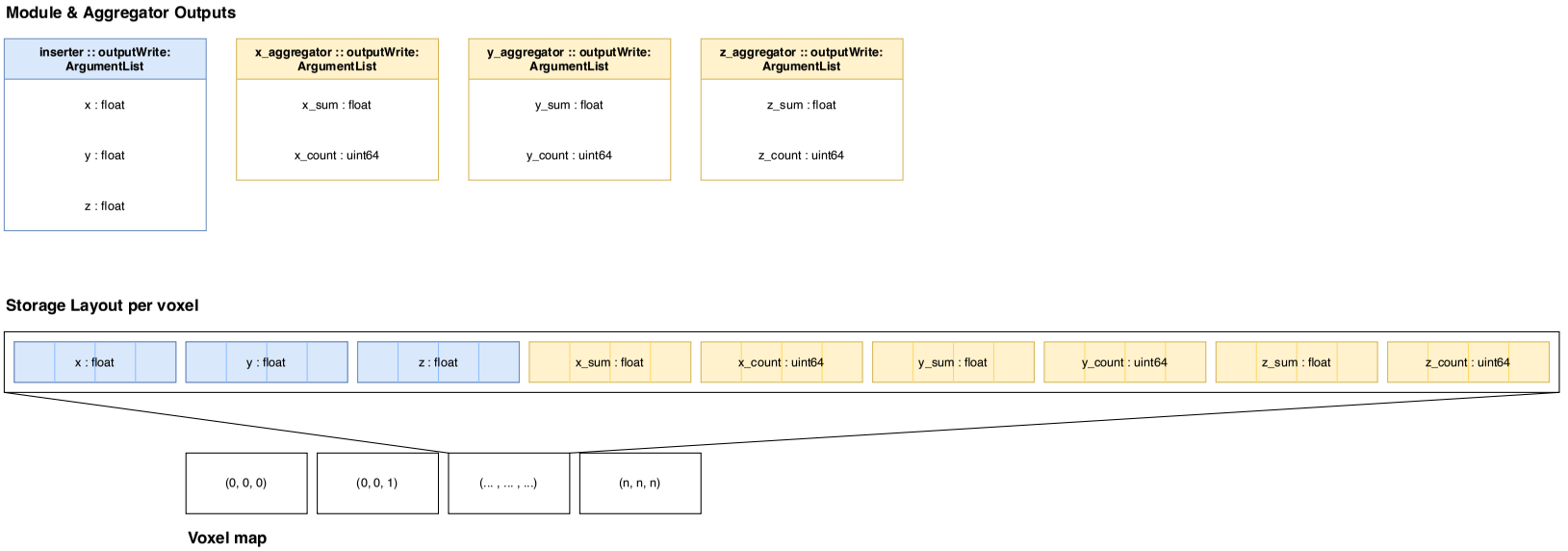



**Figure 4.9.** The storage layout of the global voxel map model in *SensorClouds*. All output definitions of the configured *Modules* are gathered and combined with the temporary variables defined by the *Aggregators*. The resultant storage map is then instantiated once per voxel. This specific example layout shows the storage requirements for a simple inserter for three-dimensional coordinates and their respective *Aggregators* calculating the mean of all inserted coordinates.

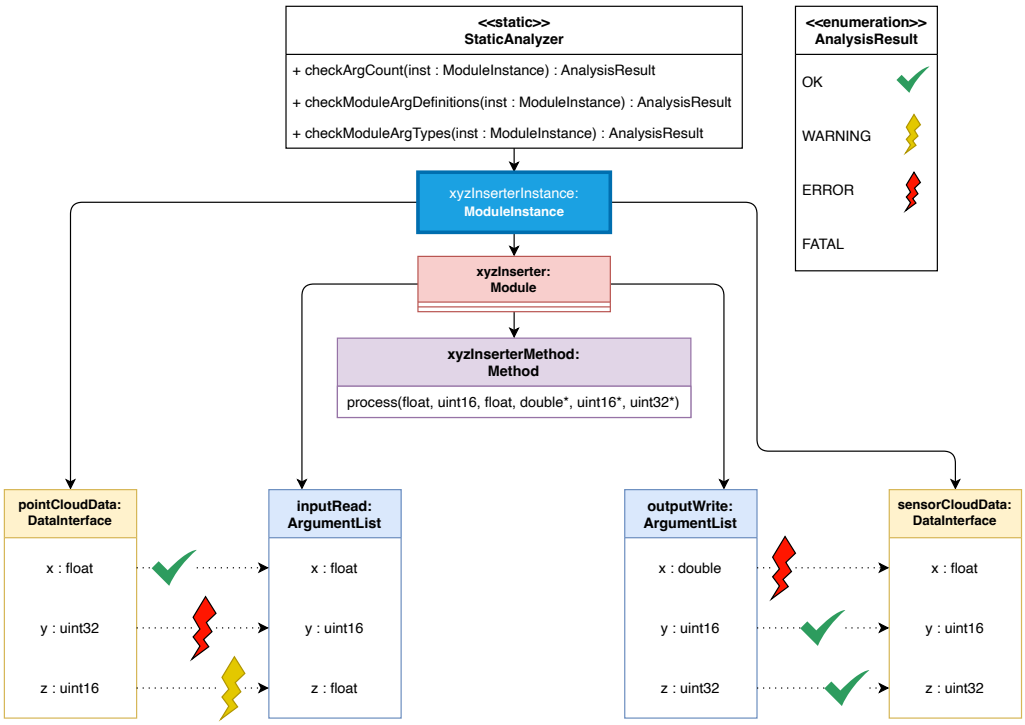



**Figure 4.10.** Results of the static analysis for the example of a simple *Inserter*.

In case all variable definitions match or can be converted to their respective counterparts in the data sources, the executable *Method* of the *Module* must be analyzed for correctness of its defined arguments. First the length of the defined argument list is checked for equality to the number of parameters required to invoke the method (`checkArgCount()`). Any mismatch constitutes a configuration error which cannot be resolved without developer intervention and thus causes the analysis to fail immediately with a **FATAL** result. Finally, the types of the module definition are matched against the types of the method arguments (`checkModuleArgTypes()`). Unlike the previously described check of the definitions against their respective data sources not only the base data types are compared, but also the type modifiers. Variables to be read may only be of a base datatype, whereas those to be written to must be a pointer of the correct base data type. This further enforces the *design-by-contract* principle. Any deviation from the contract which cannot be alleviated with a simple non-altering type conversion is hence penalized and thus constitutes the enforcement of the design-by-contract paradigm.

### 4.2.7 Pipeline

The global execution pipeline is managed by the *SensorCloud* object at runtime. The pipeline stages are a fixed set of ordered execution steps which every *SensorClouds* application cycles through (cf. Figure 4.11). *Aggregators* are automatically executed after every write access by a module and perform their calculations based on the values already present in the global data model and those produced by the *Method* of the module

in question. Both *Inserters* and *Processors* can define optional pre- and post-processing operations which do not require access to the data stored in the global data model and moreover are prohibited from attempting to. Consequently, no aggregation steps are necessary after the respective pre- and post stages of *Inserters* and *Processors* since they cannot write to the global data model.

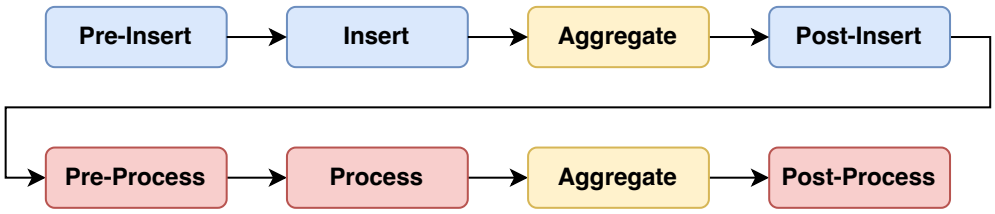



**Figure 4.11.** Global pipeline stages in *SensorClouds* and their execution order.

Every pipeline item (*ModuleInstance*) is automatically placed into its correct position in the execution order. This order is determined first by the pipeline stage it belongs to and secondly by any possible data dependencies of a module. An insertion module projecting a thermal image into three-dimensional space, for example, requires all sources of three-dimensional base data to have been executed before it can perform its task. These dependencies are resolved using a depth-first topological sorting algorithm, such as the one first described by Tarjan [113] in 1976.

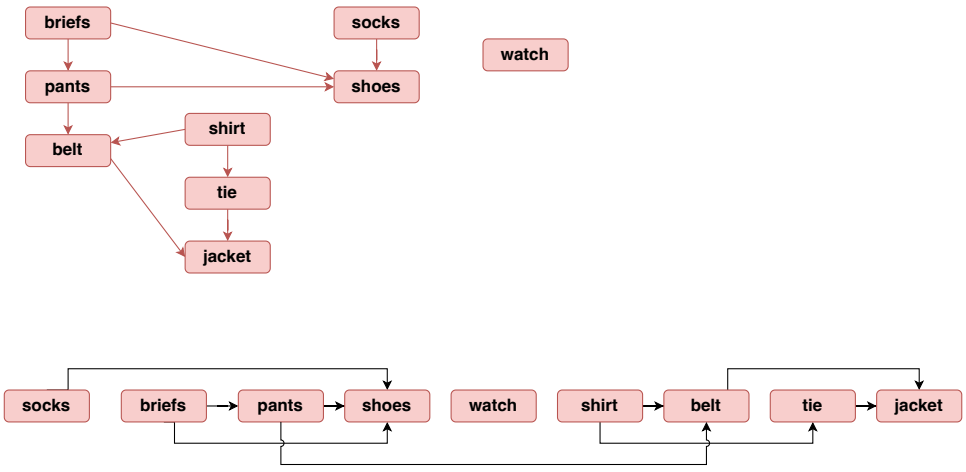



**Figure 4.12.** Example of the depth-first topological sorting algorithm using clothes and their order while getting dressed. (a) shows the directed acyclic graph with the dependencies and (b) shows the result of the sorting algorithm. (Adapted from [33, p. 574])

Cormen et al. [33, p. 573] give an easily approachable example of this topological sorting algorithm by depicting the process of clothing oneself including all the dependencies involved in this process (Figure 4.12a). For example, shoes can not be put on until pants and socks have been previously. Figure 4.12b then shows the linearized output of

the topological sorting algorithm. The algorithm works by traversing the graph from a random node which has not been previously added to the output. From this point onwards, the graph is then recursively traversed following all the dependencies of each node until a leaf is reached. The leaf node is then prepended to the output list and its recursive function call returns to the next higher level. This process is repeated until the originally selected node is reached. Once all nodes in the graph have been added, the algorithm terminates. If at any point during the recursive traversal a node is visited twice, the algorithm terminates prematurely, since this constitutes a circular dependency.

In *SensorClouds* this algorithm is implemented in a generalized class structure which can be reused to topologically sort any list of arbitrary objects which define a single method describing their dependence on other objects of the same type (see Figure 4.13). Any class which is to be given the ability to be sorted topologically must merely extend the *DependencyInterface* and implement the `dependsOn()` method. The interface is templated with the implementing class in order to correctly type the input parameter of the dependency method. The *DependencyResolver* performs the effective topological sorting operation and provides the linearized result list to the invoking instance.

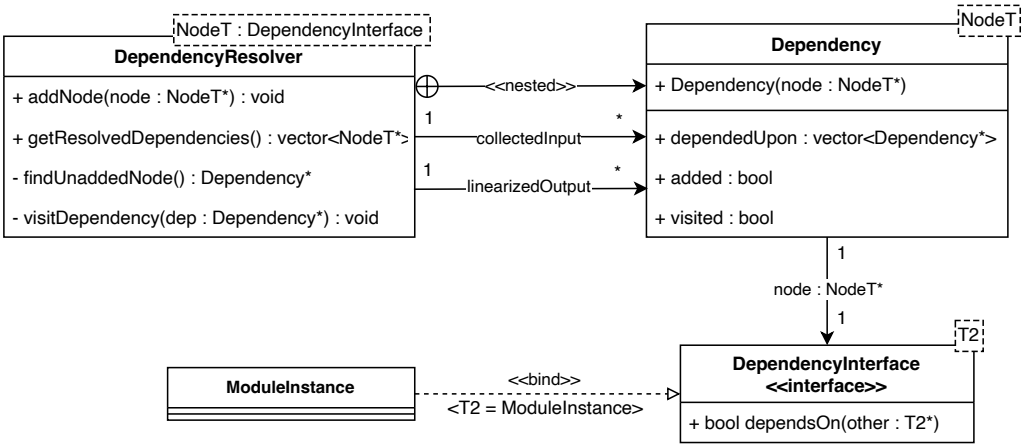



**Figure 4.13.** The dependency resolution architecture of *SensorClouds*.

All objects to be sorted are passed to the *DependencyResolver* which in turn individually wraps them in *Dependency* objects and adds them to its input list (`collectedInput`). This nested inner class of the *DependencyResolver* has the sole purpose of encapsulating all the necessary variables for resolving dependencies without cluttering the objects to be sorted and is thus only visible to the *DependencyResolver* and solely used internally. In order to perform the actual dependency resolution, the dependency graph must first be built. This is accomplished by iterating over all elements of the list and calling the dependency method with each of the remaining elements as parameter. Any nodes producing a positive result are added to the original node's list of elements which depend upon it (`dependedUpon`). Upon this graph structure the actual dependency resolution can be performed thereafter. The specific implementation of the dependency resolution algorithm can be found in Listing 4.1.

```
1  void visitDependency(dependency_ptr node) {
2      if (node->added)            // Finished sorting
3          return;
4      if (node->visited);         // Circular dependency
5          return;
6
7      node->visited = true;
8
9      for (dependency_ptr node_dep : node->dependedUpon) {
10         visitDependency(node_dep);
11     }
12
13     node->visited = false;
14
15     node->added = true;
16
17     linearisedOutput.emplace_front(node);
18 }
```

**Listing 4.1.** Concrete implementation of the dependency resolution algorithm in *SensorClouds*.

In the case of a *ModuleInstance* as depicted in Figure 4.13, the implementation of the dependency method must consider three factors. Firstly, the *ModuleInstance* must check which stage of the pipeline it has been associated with and if the other instance's stage precedes it, since the later stage may only begin execution after the previous stage has completed, regardless of any other factors. If both instances are to be executed within the same pipeline stage, then their output targets must be the same in order to be in dependence on one another, which is the second factor. Finally, the *Model Output* arguments of the other element are compared to the *Model Input* arguments of the current element, since any overlap between these two constitutes a data dependency between the two elements. The dependency resolution is the second method for enforcing the design-by-contract paradigm, since data dependencies clearly define preconditions for a module which then must be executed in the correct order by the *SensorClouds* framework.

The linearized output list of *ModuleInstances* is then executed periodically one after another at runtime. The period of execution can be configured to either be chronological or after one or all sensors have received new data, depending on the application requirements. Selecting the mode of execution when all sensors have received new input data provides basic implicit data synchronization between devices of different polling rates, however no data is altered or interpolated over time by this mode.

## 4.3 Programming Interface for Application Developers

An application developed with the *SensorClouds* framework and realizing the KoARob scenario described in Section 3.2 can be seen in Listing 4.2.

```
1  auto sensorcloud = SensorCloud(256, 0.025, Origin::CENTER,
       ↪ ExecutionPolicy::ON_ALL_NEW_DATA, "/sensorcloud");
2
3  auto rc1 = devices::Roboception160("rc1", "/rc1/depth_points");
4  rc1.registerInserter<insert::SCRGBDInserter>();
5  auto rc2 = devices::Roboception160("rc2", "/rc2/depth_points");
6  rc2.registerInserter<insert::SCRGBDInserter>();
7  auto rc3 = devices::Roboception160("rc3", "/rc3/depth_points");
8  rc3.registerInserter<insert::SCRGBDInserter>();
9  auto rc4 = devices::Roboception160("rc4", "/rc4/depth_points");
10 rc4.registerInserter<insert::SCRGBDInserter>();
11 auto rc5 = devices::Roboception160("rc5", "/rc5/depth_points");
12 rc5.registerInserter<insert::SCRGBDInserter>();
13
14 auto flir = devices::FLIR("flir", "/flir/imagerect");
15 flir.registerInserter<insert::SCThermalInserter>();
16
17 sensorcloud.addDevices(&rc1, &rc2, &rc3, &rc4, &rc5, &flir);
18 sensorcloud.completeDeviceRegistration();
19
20 sensorcloud.registerAggregator<aggregate::MeanAggregator>({"x", "y",
       ↪ "z", "r", "g", "b", "temp"});
21 sensorcloud.registerProcessor<process::ThermalFilter>(&out, 36.1,
       ↪ 37.2);
22
23 sensorcloud.buildPipeline();
24
25 while (ros:ok()) {
26     sensorcloud.executePipeline();
27 }
```

**Listing 4.2.** Application structure in *SensorClouds*.

The first step is to define the global model to be used for all subsequent operations by defining the number of voxels in each dimension, the resolution of each individual voxel and the definition of the coordinate origin of the model. The penultimate parameter defines the execution policy, meaning if the pipeline shall be executed at a fixed time

interval, when one device has produced new data or when all configured devices have. Finally, the topic name to publish the resulting data to is specified.

```
1  auto sensorcloud = SensorCloud(256, 0.025, Origin::CENTER,
       ↪  ExecutionPolicy::ON_ALL_NEW_DATA, "/sensorcloud");
```

Thereafter, the drivers for all present devices must be instantiated and the exacted *Inserters* registered with them. Device drivers receive the user-configurable instance name and the input ROS topic as parameters. The *Inserter* employed in this case is very basic and directly inserts the coordinates and color values of each data point into the corresponding voxel. Since an RGB-D camera internally references the color values to the appropriate distance measurement, no further processing is required.

```
3  auto rc1 = devices::Roboception160("rc1", "/rc1/depth_points");
4  rc1.registerInserter<insert::SCRGBDInserter>();
```

This is repeated for the remaining four cameras in the KoARob cell analogously after which the driver for the thermal imaging camera is instantiated with its suitable *Inserter* capable of projecting the thermal image into three-dimensional space by leveraging a raytracing algorithm to find the relevant occupied voxel.

```
14 auto flir = devices::FLIR("flir", "/flir/imagerect");
15 flir.registerInserter<insert::SCThermalInserter>();
```

All configured devices are then registered with the *SensorCloud* object and the registration process is finalized to indicate to the system that no other devices will be added beyond this point and it has thus all required data about the the connected sensors.

```
17 sensorcloud.addDevices(&rc1, &rc2, &rc3, &rc4, &rc5, &flir);
18 sensorcloud.completeDeviceRegistration();
```

The configuration of the preferred sensor fusion approach is performed by registering the *Aggregator* and the fields it shall be applied to with the *SensorCloud* object. In this case a standard mean *Aggregator* is defined on all the present data fields.

```
20 sensorcloud.registerAggregator<aggregate::MeanAggregator>({"x", "y",
       ↪  "z", "r", "g", "b", "temp"});
```

Finally, the output processing filter is defined which in the case of the KoARob scenario is a filter operating on the temperature value provided by the FLIR camera and is intended to include only those voxels in which a temperature within the range of human body temperature is present, thus filtering all objects and background elements other than humans in the workspace from the acquired sensor data. This *Processor* receives the data output target of the filter operation as well as the upper and lower temperature thresholds as input arguments.

```
21  sensorcloud.registerProcessor<process::ThermalFilter>(&out, 36.1,
        ↪  37.2);
```

Setup concludes with the instruction to build the pipeline for the application, although some parts of this operation may be deferred by the application to a later point in time, depending on the availability of the required data.

```
23  sensorcloud.buildPipeline();
```

Then in the main application loop the execution method of the *SensorCloud* object must be invoked. This internally decides whether to actually execute the pipeline depending on the configured execution policy and the aforementioned deferral of internal configuration steps.

```
25  while (ros:ok()) {
26      sensorcloud.executePipeline();
27  }
```

In summary, the implementation of applications utilizing the *SensorClouds* framework is a fairly straightforward task for application developers, provided the required modules and drivers have been previously implemented.

## 4.4 Related Work

While many highly targeted solutions for sensor fusion applications exist, only a select few have attempted to develop a truly generalized framework for such tasks.

Especially in recent years, a vast amount of research in sensor fusion has focused on the use of neural networks for sensor fusion. While these definitely have their advantages in sensor data processing, as will be shown in Section 5.4, employing them end-to-end in three-dimensional fusion applications encodes geometric and sensor-specific data directly into the neural net, preventing future reuse in other application with differing sensor configurations (cf. [132]).

One example of a more generalized system using neural networks is the work of Chen et al. [25]. They propose the use of a so called *modality agnostic feature sampler* which combines information from multiple sensors based on their three-dimensional location data. While similar to the approach of *SensorClouds* in that all data is combined using three-dimensional coordinates, their solution requires training on individual datasets for each sensor with different neural architectures and fine-tuning on the concrete combination of sensors. The general applicability of this solution is therefore not guaranteed, escpecially since they only tested their approach on fixed datasets.

The point cloud library (PCL) [101] was first developed as integral part of ROS by Willow Garage and has since been spun off into a separate project. Its feature set is divided into multiple distinct modules for various tasks related to sensor data processing and recognition (cf. [93]):

- **Filters**
  The list of supported filters includes the most commonly used filters in image data processing, such as outlier removal, basic cropping to geometric shapes, bilateral filtering for noise reduction or frustum culling for creating virtual camera perspectives.
- **Feature & Keypoint Extraction**
  Various implementations of concepts described in scientific publications (such as e.g., the OUR-CVFH feature histogram [9]) for the extraction of feature vectors and keypoints are integrated into or were originally developed for the PCL. Feature vectors describe the geometric pattern of a point in correlation to the surrounding points. Keypoints, in contrast, are points which are particularly distinct. Together, these two functions can be used to describe large datasets in a compact manner.
- **Segmentation**
  Segmentation algorithms attempt to extract distinct objects from point clouds. Various implementations of segmentation algorithms for this purpose are available in the PCL.
- **Registration**
  When combining multiple point clouds with unknown reference frames these can only be combined by attempting to estimate the coordinate transformation between them first. This is exactly what point cloud registration offers and the PCL contains a number of algorithms fur this purpose as well.

The PCL supports the definition of new point cloud types constructed from constituent structures. This means that a user-specific implementation can define a custom point type using the predefined structure for $x$, $y$ and $z$ coordinates and combine it with another predefined structure or custom data fields. These custom structures can, however, become very complicated, spanning multiple union commands of various structures to be combined. But the main caveat of this approach is the support of the included algorithms for custom point types. While the predefined constituent structures, such as those for coordinates, are typically understood and can be processed by the included algorithms, support for truly custom point types must be manually implemented within each and every algorithm which is supposed to process the custom part of a point definition. The custom point type must also be registered globally with complicated preprocessor macros in order to even be found and recognized by the algorithm implementations. (cf. [91])

In comparison, the data model in *SensorClouds* presents a clear advantage over the tedious process of adding new information types to the PCL. Defining new data fields is as simple as creating a tag. The data type is then individually specified by *Modules*.

While the PCL offers access to various different algorithms, GPU support is still very limited. Development of GPU support is currently ongoing, as logs in the project's github repository show (see [92]), but still not on par with the CPU implementations by far. Due to this fact, the performance of the PCL when processing large datasets is totally insufficient when targeting realtime applications, as will be conclusively demonstrated in the performance evaluation in Section 8.1.

The closest competitor in terms of performance is GPUvoxels [53]. GPUvoxels also utilizes a voxel based data model which all incoming sensor data is inserted into, any additional information from various additional modalities is immediately discarded and can thus no longer be used for further analyses. The data model of GPUvoxels only supports the definition of either binary or probabilistic occupancy for every voxel. Adding support for additional modalities to the library is nigh impossible, since every aspect of it focuses solely on occupancy of voxels. The only differentiation in voxel data is in terms of the position of the robot in the workspace. GPUvoxels uses the tool binvox [78] to convert robot models into voxelized representations of the same, which during runtime can be utilized to determine the distance between the robot structure and obstacles in the environment as well as exclude sensor measurements detecting the robot structure from the collision calculation.

The tool binvox can easily be integrated into the *SensorClouds* architecture in the form of a *Processor* which marks voxels as belonging to the robot structure and calculates the distance between those voxels and the remaining sensor data. The clear advantage of *SensorClouds* lies in the support of multiple modalities in its global data model, as opposed to GPUvoxels, which allows the extraction of valuable data from the combined model after all data has been inserted. The performance of the two frameworks is comparable despite the greater flexibility that *SensorClouds* offers, as will be shown in Section 8.1.

**Chapter Summary**. This chapter explains in depth how the information from simpler, one-dimensional sensors, such as capacitive proximity sensors for the detection of humans in a robot's workspace can be integrated into the three-dimensional data model in *SensorClouds*. Using multiple processing techniques and finite element simulation, the sensor value can be partially reconstructed to reflect the true three-dimensional nature of the measured phenomenon.

# 5 Integration of Capacitive Sensors

Since capacitive sensors can only deliver a single one-dimensional capacitance measurement, various methods must be employed in order to reconstruct the actual three-dimensional nature of the measured value as far as possible. This chapter gives an overview over the techniques utilized for the reconstruction specifically of capacitive sensor values although they can be applied to various other sensor types with only minor modifications as well.

## 5.1 Vicon Motion Capturing

Vicon motion capturing systems [123] developed by Oxford Metrics PLC [86] is a high-precision camera system capable of tracking objects in three-dimensional space through the use of infrared-reflective markers attached to the item to be tracked. Multiple cameras surrounding the desired space within which objects are to be tracked must first be calibrated to each other with the help of a specialized active calibration tool, called "Active Wand". This device is equipped with multiple light emitting diodes, which can be captured by the Vicon cameras on their two-dimensional planar image sensors. The calibration process involves moving the wand through the surveilled space such that all cameras can capture multiple images of the wand in various angles. The control software then triangulates the positions of all installed cameras from these images. With this calibration the Vicon system can achieve sub-millimeter precision when tracking static or dynamic objects in the arena [74]. Vicon also maintains various software packages in its portfolio, all capable of handling specific capturing tasks. The Vicon Tracker software is the tool used to track solely static objects, meaning that the markers attached to an object never change their relationship to each other and thus are mounted rigidly to a fixed object which then moves through space. With Nexus, motion capturing of variable geometries becomes possible, as is the case in tracking humans in the capturing arena.

The Vicon motion capturing system has been used in a wide variety of use cases since its introduction in the late 1970s [124] with applications ranging from medical analyses to digital content creation. Most commonly known is its usage in big budget motion pictures for tracking the movement of human actors on a motion capture stage for later use in the animation of virtual characters by virtual effects (VFX) artists. Notable examples of this include films such as Titanic (1997) [42] in which Vicon motion capturing technology was employed for virtual characters used to make the animated computer model of the ship seem more populated in wide shots. From these rather modest beginnings the technology has evolved so far that it is now commonplace to see completely animated protagonists in feature films, such as is the case in most of the movies produced by Marvel in recent years [29]. Another interesting use in the film industry is that of motion tracking in virtual production systems [125]. These systems can use pre-rendered virtual scenes as the backdrop for the actors' performances. This, however, requires determining the physical location of the camera with respect to the gigantic LED screens forming the backdrop in order to enable the correct projection of the camera's changing perspective while moving through the scene. An example of such a production is the TV show "The Mandalorian", which uses this technology almost exclusively in production (cf. Figure 5.1).

In scientific research, Vicon motion capture technology has been employed extensively in the field of medicine. Motion capturing can be an invaluable medical tool when applied to the analysis of human gait during the treatment of various ailments impacting the motor functions of patients. One such application is the analysis of the angular velocity of the knee joint in patients recovering from strokes by Pomeroy et al. [94]. During physical therapy the progress made by the patient in recovery is typically assessed subjectively,

**Figure 5.1.** Behind the scenes of the production for the TV show "The Mandalorian". Vicon motion tracking cameras are utilized to track the position of the cameras in order to adjust the perspective of the projected backdrop images accordingly. (From [20]. © and ™ Lucasfilm. All Rights Reserved. Used with Permission.)

since not all medical centers have access to expensive and spacious motion capturing arenas. It is for this reason that cheaper alternatives for objective assessment devices have been sought, often by comparing them with the data from a Vicon system. In this work, the researchers investigated the potential use of an electrogoniometer, which is a device to measure the flexibility of a joint in one dimension by using electrically driven sensors, instead of motion capturing. Typically these are based simply on potentiometers or strain gauges to provide the required information on the current angle of the joint. A similar work examined the use of multiple goniometers to assess the range of motion in finger and wrist joints [30]. Both works conclude that the use of motion capture technology is far more accurate than goniometers, yet they are sufficient for use in clinical settings to assess the effect of physiotherapy on patients. In sports medicine, motion tracking can also provide insights into the recovery progress of an injured athlete, but can also even be employed preemptively in order to train at risk athletes in how to prevent potential future injuries [72]. Other research has focused on replacing costly and complex motion capturing technology altogether, instead aiming to replace it with cheaper alternatives such as the Microsoft Kinect [89] or its successors Kinect v2 and Azure Kinect [8]. In order to enable further research by clinicians without access to specific patient groups or a motion capturing arena, various datasets have been recorded

using Vicon cameras containing a multitude of different ailments and scenarios, such as the study of karate students of different skill levels [112], the kinematics of amputees [55] or the success of hip replacements [18] and determining the risk of fall injuries in older adults [22], to name a few.

In robotics research, specifically in human-robot-collaboration, the analysis of human movement is also essential, albeit typically in a far less sophisticated manner. The simplest application of Vicon cameras in robotics is the precise tracking of static (in the sense of the marker configuration) objects in the motion capturing arena. Such is the case for example in the indoor use of unmanned aerial vehicles (UAVs) where the use of GPS is prohibitive [60, 69, 71]. By attaching reflective markers to the frame of quadrocopters their position in space can be determined by the Vicon system at all times. This information can then be fed back to the control system which in turn governs the positional control of the drone. In this manner, even entire swarms of quadrocopters can be located and controlled simultaneously by the Vicon system, provided the marker configurations vary sufficiently in geometry between all the drones. As is the case in medical research, much effort has also been invested in replacing the complex and expensive Vicon system with simpler and more affordable alternatives which can be more easily deployed [51, 136]. The performance of such systems is typically gauged against that of Vicon cameras and is inferior to them. However, they may still be sufficient depending on the concrete field of application.

## 5.2 FEM Simulation

The *finite element method* (FEM), also referred to as *finite element analysis* (FEA), is a computational simulation method which allows the efficient calculation of continuous physical properties in arbitrary geometric structures. The term was first introduced by Clough in 1960 [26] in his paper "The Finite Element Method in Plane Stress Analysis". Here he described the use of a discretization technique for analyzing structural properties of objects more efficiently. Plane stress describes a class of problems in structural engineering for which one dimension can be - at least initially - ignored and thus the calculation of the deformation of a thin plate under load can be examined merely in a two-dimensional cross-section. The effect of the deformative force is then assumed to be the same throughout the remaining structure in the third dimension [97]. Although the mathematical principles had been described by various researchers over the first half of the 20th century, the finite element method only found widespread adoption in the latter half of the century, after aeronautical and civil engineers had achieved promising results with rudimentary computer programs to automate the vast amount of calculations required to properly implement the FEM (cf. [31, p. 10] & [65]). One of the first practical uses of computer aided calculation of physical properties in real-world engineering applications involved the simulation of the structural properties of airplane wings at Boeing [121].

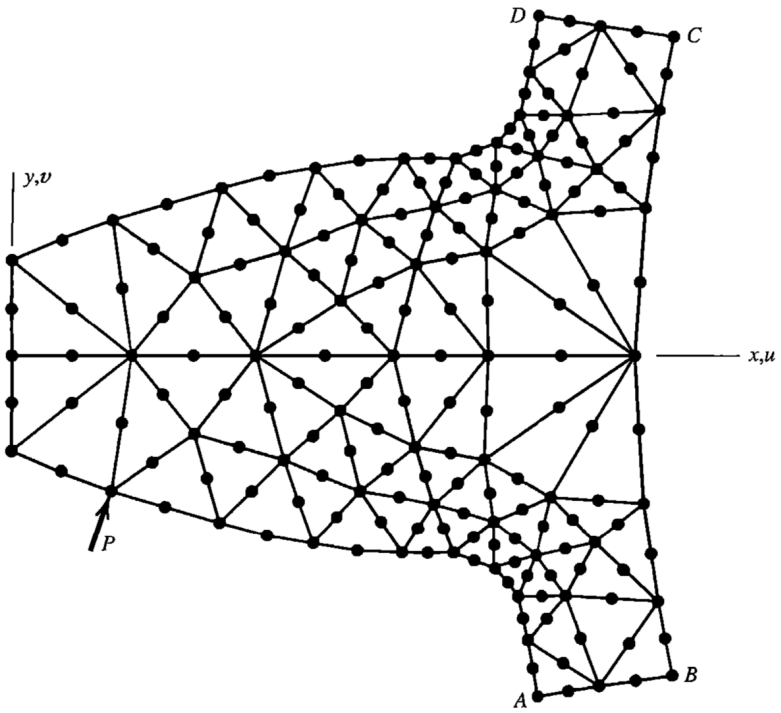



**Figure 5.2.** Approximated mesh of a gear tooth for FEM simulation (From [31, p. 2])

The basic principle of FEM simulation hinges on approximating complex differential equations in infinitesimal space by restricting the calculations to simpler polynomial or even linear equations solved only piecewise for finite elements (cf. [31, p. 1]). A concrete example of how this reduction in complexity can be achieved is depicted in Figure 5.2. In this case a two-dimensional stress analysis under load (denoted by $P$) of a single tooth of a gear is to be performed. The actual geometry is approximated by a mesh and the physics calculations are then performed in each of the nodes, or vertices as they are known in graphics programming, which are the corner points of the elements. The elements of the mesh can be comprised of a variable number of nodes, and even additional nodes along the edges of the elements can be added in order to receive a more accurate approximation of the physical quantities, especially along complex curved geometries. In areas of particular interest to the engineer, the size of the elements can be decreased in order to achieve a finer resolution of the physics simulation. Adding nodes along the sides of a mesh element can additionally aid in the compatibility of meshes when a coarse mesh is no longer sufficient and finer resolution is required, since the six-node triangular element can easily be subdivided into four three-node triangular elements by simply adding connections between the existing nodes. This can be advantageous for seamlessly switching between a coarse mesh for preliminary simulation and evaluation of the results and the final computation at the required level of detail (cf. [31, p. 7f]). Examples of various possible elements for a three-dimensional body can be seen in Figure 5.3.

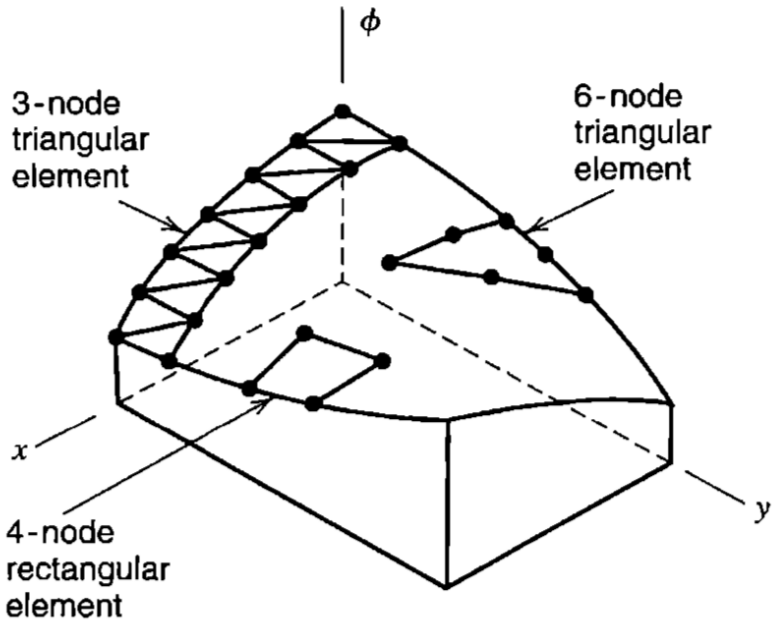



**Figure 5.3.** Depiction of various node types for the approximation of meshes (From [31, p. 8])

According to Cook et al. [31, p. 8]) the procedure of performing a finite element analysis involves the following stages:

- **Problem Classification**
  The analyst must comprehend the problem to be analyzed and which physical properties of the component need to be calculated in order to receive the required results for determining the validity of the examined component.
- **Mathematical Model**
  Correct choice of the employed mathematical models to describe the sought physical characteristics is essential to the successful application of finite element analysis. Depending on geometric composition, required resolution of the results and possible interdependencies of various physical parameters, such as the influence of temperature on other properties, different mathematical models may be more pertinent to the task at hand.
- **Preliminary Analysis**
  Before performing calculations with an FEM program, a preliminary analysis in form of "simple analytical calculations, handbook formulas, trusted previous solutions or experiment" [31, p. 13] should be performed. This can inform the choice of mathematical models and aid in gauging the validity of the simulation results later on.
- **Finite Element Analysis**
  Following these previous steps, the actual computation of physical properties using the FEM can be performed. This stage is again comprised of multiple subordinate stages:
  - *Pre-processing*
    The three-dimensional model of the object within which to simulate the physical properties must first be converted to a mesh with a finite number of discrete elements. This step can typically be left to automated mesh generators included in FEM software suites nowadays, but must be parametrized according to the required type of mesh and its resolution. Manual modifications by the expert analyst always remain possible if the automated generator fails to account for specific requirements of a particular application.
  - *Numerical Analysis*
    FEM software automatically determines the required matrices for the computation of the physical properties to be simulated and performs the calculations on all nodes of the previously generated mesh. The calculations become far more involved should the properties be non-linear or time-dependent.
  - *Post-processing*
    In post-processing, reports of the calculated properties are generated. These can be represented in datasets, plots or three-dimensional models, depending on the requirements of the analyst performing the simulation. FEM software suites contain a vast array of possible output formats for the representation of simulation results.
- **Check the Results**
  The resulting output of the simulation must be rigorously inspected by the analyst

and validated against the expected results determined in the preliminary analysis. Since the results contain additive modeling and approximation errors picked up over the entire process of simulating a real-world object's actual physical properties, the simulation must be closely observed and its results not be taken at face value. An expert analyst must always confirm the validity of the simulation results before proceeding with the engineering task based on them.

- **Expect to Revise**
  Due to the large margin for error, FEM simulations can often produce unexpected and inaccurate results. Because of this, simulating physical properties of real-world objects is more often than not an iterative design process. The mathematical model, mesh discretization and even the problem classification with regard to which physical properties need to be simulated and which may or may not be interdependent can be based on false assumptions of the inner workings of FEM simulation tools and the physical reality being simulated and thus must be adapted accordingly.

Since the early beginnings of FEM simulation in structural engineering, many additional physical domains have been added to commonly available FEM software suites. The most notable fields of application include fluid mechanics, thermodynamics, acoustics and electromagnetics. The field most relevant to this work is that of electromagnetics, which will be described in more detail in Section 5.6.

## 5.3 Physical Background of Capacitive Sensors

Electrical capacitors are indispensable in all forms electronic circuits and their most basic role is the storage of electrical energy. The simplest form of a capacitor is that of the parallel plate capacitor, which consists of two exactly parallel and equally sized conductive plates separated by a dielectric material, such as air, glass or even an electrolyte solution.

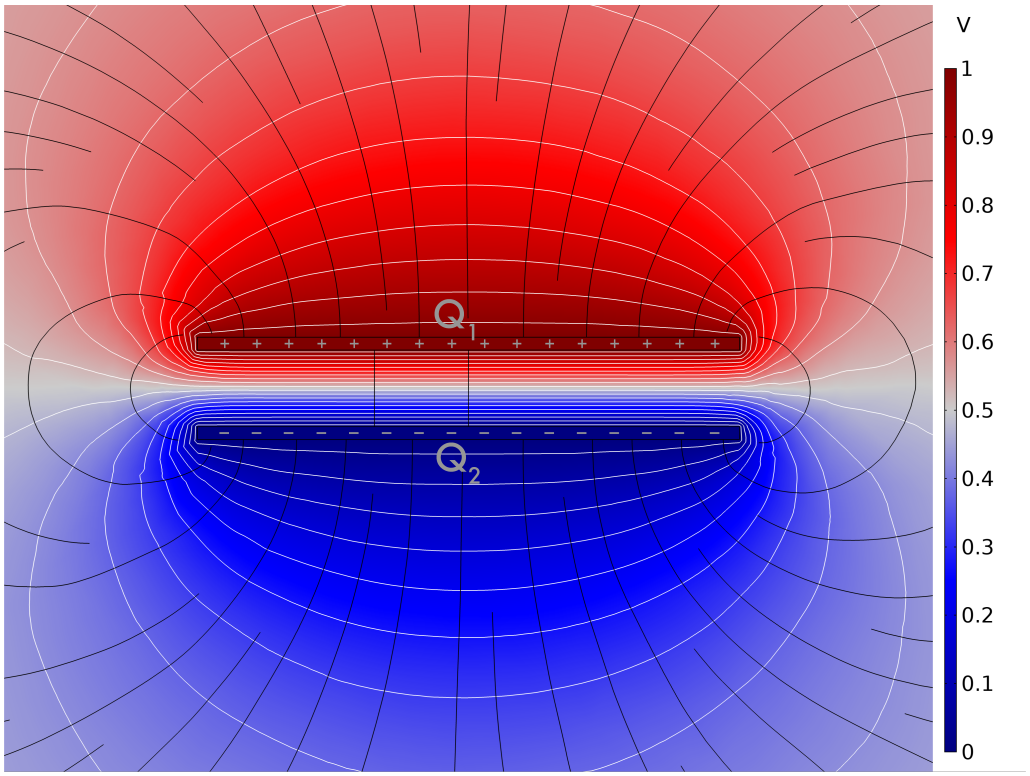



**Figure 5.4.** Electrical field in a parallel plate capacitor. The equal but opposite charges $Q_1$ and $Q_2$ create an electrical field between their respective conductive plates. Also denoted are the equipotential surfaces (white lines) and the electrical field lines (in black) along which the electric force is present. The background gradient symbolizes the electric potential.

According to Coulomb's law, two opposite charges exert a force on each other which can be calculated as

$$F = \frac{Q_1 Q_2}{4\pi\varepsilon_0\varepsilon_r r^2} \tag{5.1}$$

where

$F$ : The resulting force

$Q_1, Q_2$ : The respective charges in Coulomb

$\varepsilon_0$ : The dielectric permittivity in a vacuum, also known as the *electric field constant*

$\varepsilon_r$ : The relative permittivity of the dielectric material between the two charges

$r$ : The distance between the two charges

By applying a voltage across the capacitor, the two plates are charged with equal potential yet opposite polarity which in turn develops an electrical field in the dielectric material inside the capacitor. The special property of a parallel plate capacitor is that it forms a uniform electrical field where the plates face each other directly, which means that the direction of the electrical force is perpendicular to the plates and the electric field strength is equal at every point. This, however, only holds true in the center of a parallel plate capacitor and merely under idealized conditions. (cf. [15, p. 7 ff])

The electrical capacity is defined as

$$C = \varepsilon_0 \varepsilon_r \cdot \frac{A}{d} \tag{5.2}$$

with

$C$ : The capacity in Farads

$\varepsilon_0, \varepsilon_r$ : Permittivities as above

$A$ : The area covered by each of the plates

$d$ : The separation distance between the two plates

The electric energy stored within a capacitor is not only dependent on its capacity, but also on the voltage applied to it and is determined by the formula:

$$E = \frac{1}{2} C V^2 = \frac{1}{2} \varepsilon_0 \varepsilon_r \cdot \frac{A}{d} \cdot V^2 \tag{5.3}$$

with

$E$ : The electric potential energy in Joules

$V$ : The voltage applied to the capacitor in Volt

Every point in space in and around the capacitor plates has a specific potential depending on the voltage applied to the plates. From this scalar voltage $V$, the equipotential surfaces can be determined, namely the surfaces within the electric field where the voltage is the same (cf. Figure 5.4). The electric field strength ($\vec{E}$) is a vector quantity and describes the force ($\vec{F}$) exerted on a specific charge ($q$) at a given point within the field (cf. [15, p. 7ff]):

$$\vec{E} = \frac{\vec{F}}{q} \tag{5.4}$$

Given a specific point in space ($\vec{r}$) the electric field strength can also be calculated from the negative voltage gradient within the field:

$$\vec{E}(\vec{r}) = -\nabla V(\vec{r}) \tag{5.5}$$

In the center of a parallel plate capacitor, this equation can be simplified greatly, since the electric field in that region is uniform:

$$\vec{E} = \frac{V}{d} \tag{5.6}$$

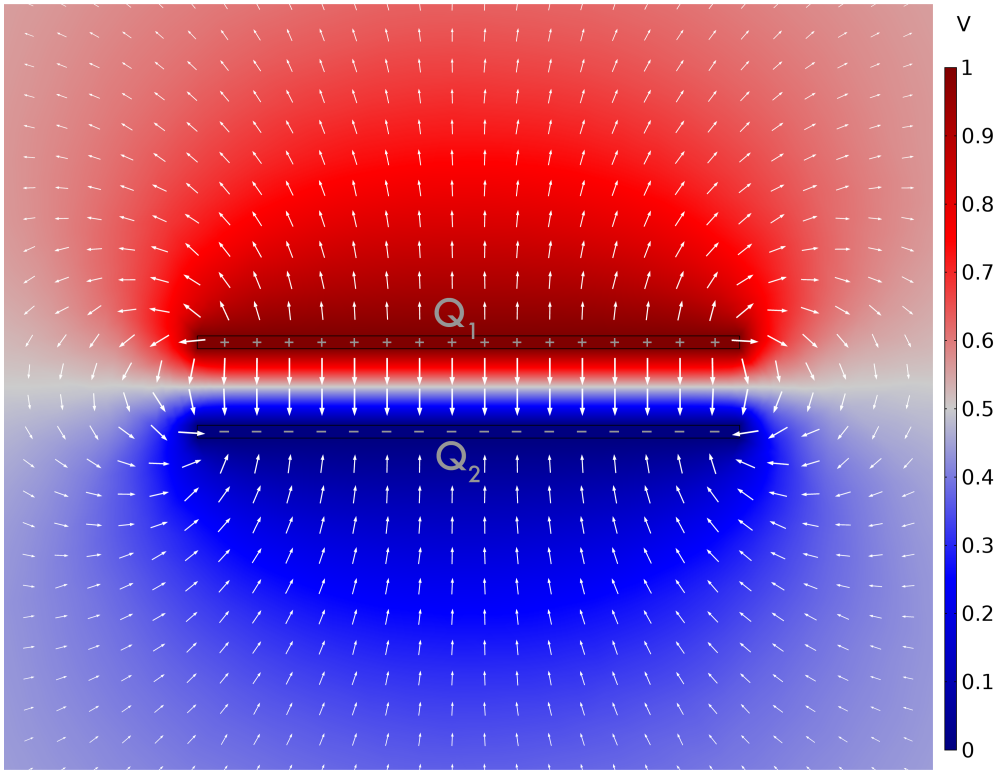



**Figure 5.5.** Visualization of the vector field $\nabla V$ describing the electrical force in every point of the electric field of a parallel plate capacitor. The respective field Strength is denoted by the scale of the individual arrows.

A visualization of the vector field describing the electric field strength in and around a parallel plate capacitor can be found in Figure 5.5, which was created using FEM simulation. The size of each individual arrow is proportional to the electric field strength in that particular point in space. As is evident from the illustration, the field around the parallel plate capacitor is clearly far more perturbed than the homogenous field directly between the two plates. This is the case for most electrode geometries, since all electric field lines terminate at right angles to conductors and a homogenous field can therefore only exist in the overlapping area of two plate conductors [15, p. 24]. While overlapping dual electrode configurations can be very useful in certain applications, such as fluid level [15, p. 140ff] or proximity sensing [15, p. 69ff], they require the two plates of the resulting capacitor to encompass or be encapsulated within the measured object. Remote sensing offers the advantage of the sensor being deployable completely independent

of the measured object, however it requires vastly different electrode configurations, which involve far more complex measurement circuitry and processing.

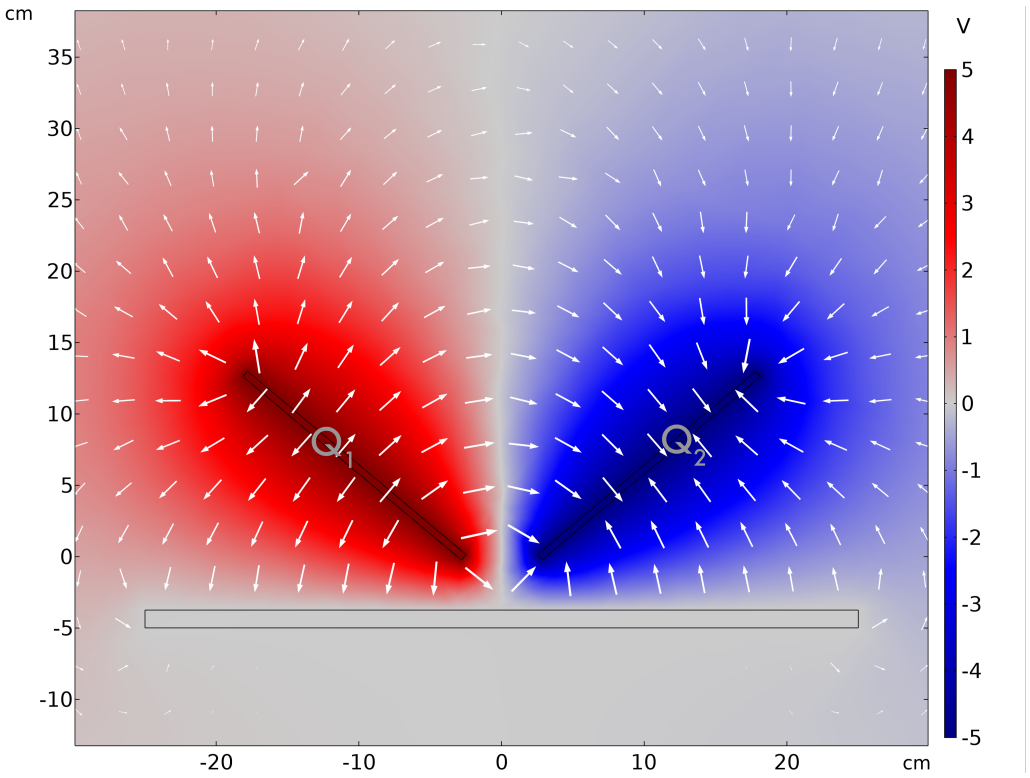



**Figure 5.6.** Visualization of the vector field $\nabla V$ describing the electrical force in every point of the electric field of a capacitor with orthogonal plates. The respective field strength is denoted by the scale of the individual arrows.

Figures 5.6 and 5.7 show the effect of varying the angle of the two capacitor plates to each other. The geometric structure of the electric field in such electrode configurations comparable to that at the outer edges of parallel plates. While there is no longer a homogeneous component to the overall electric field between the two oppositely charged plates, the measurable portion of the field contributing to the capacitance of the sensor gains a far greater reach. This fact, combined with the remote sensing aspect in contrast to parallel plate configurations, makes these types of electrode configurations a sound choice for applications concerning external position detection or distance sensing of conductive or dielectric materials in close proximity to the sensor in the range of typically a few $\mathrm{mm}$. Such configurations are even capable of distinguishing different materials from one another [7], since the dielectric constant differs between materials. Consequently, a system equipped with differential capacitive sensors can differentiate between materials such as metals, plastics, water and even human hands, as long as their dielectric constants differ significantly enough. The inside facing surfaces of a robot gripper are particularly suited for the use of such systems.

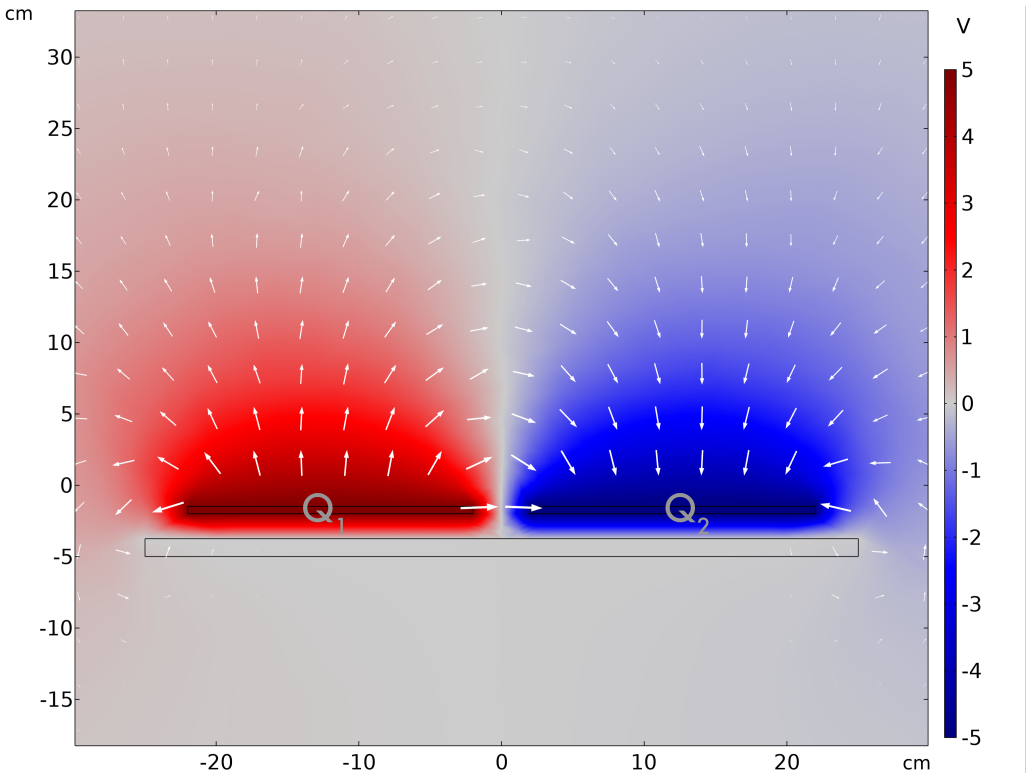



**Figure 5.7.** Visualization of the vector field $\nabla V$ describing the electrical force in every point of the electric field of a capacitor with coplanar plates. The respective field strength is denoted by the scale of the individual arrows.

Single-ended electrode configurations, which are designed to detect the proximity of objects at far greater distances than other arrangements, measure the capacitance of the single electrode against the ground potential. Systems measuring single-ended electrode configurations are also often referred to as being configured in a self-capacitance sensing mode [103, p. 273ff]. A visualization of the electrical field of such a single-ended electrode configuration can be found in Figure 5.8. This increased detection range is possible due to the fact that all conductive and dielectric objects which are to be detected influence the electrical field projected by the capacitive sensor outwards and consequently produce a measurable difference in capacitance of the sensor system. The influence is either caused by the presence of directly (through wired connections) or indirectly (by contact with grounded objects or the ground itself) grounded objects entering the electrical field of the measured capacitor [10, p. 12] or merely by objects with a differing dielectric constant, which is a sufficient influence to be measured with adequately sensitive measurement circuits [15, p. 75].

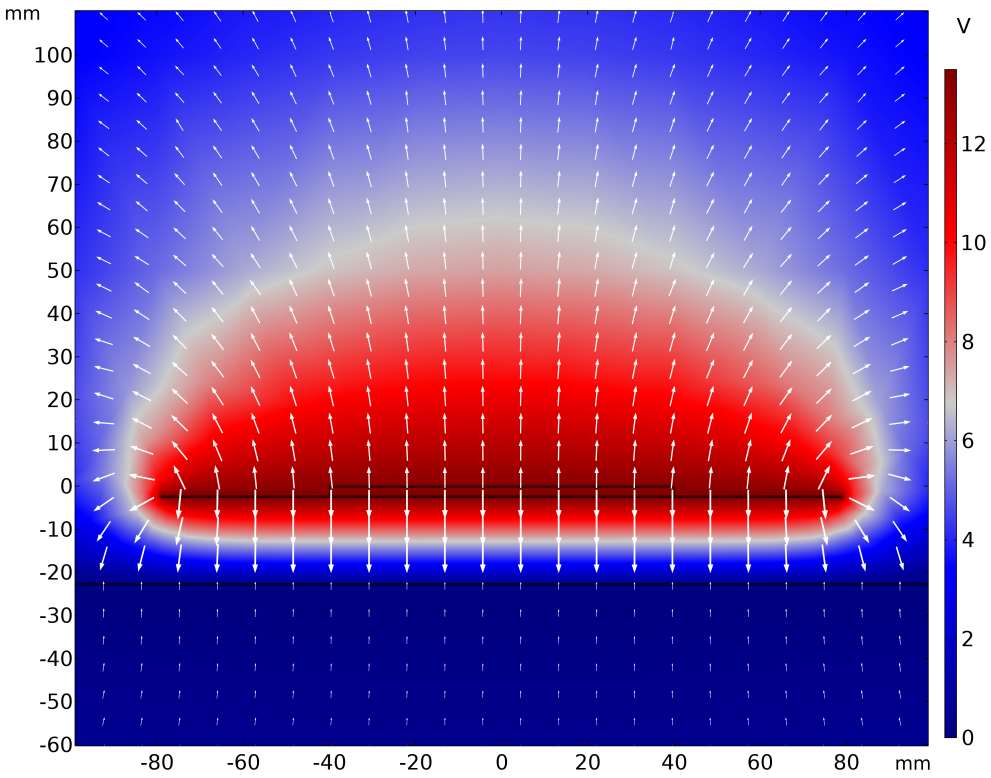



**Figure 5.8.** Visualization of the vector field $\nabla V$ describing the electrical force in every point of the electric field of a single-ended capacitor. The respective field strength is denoted by the scale of the individual arrows.

## 5.4 Intelligent Processing of Capacitive Sensor Data

### 5.4.1 Neural Network for Distance Estimation

The relationship between the measured capacitance of a capacitive sensor and the distance to a particular object, such as a human hand, is directly proportional, albeit not linear. To determine the governing equation of this correlation, various methods can be employed. The simplest and most obvious of these is a manual calibration from live data recorded while entering the electrical field of the sensor and logging the hand's distance to the sensor plane. Performing a non-linear regression analysis on the recorded distances in relation to the respective capacitance values (cf. Figure 5.9) yields the following equation:

$$d = \frac{0.6221299398 * cap}{cap - 2.002569134} + \frac{0.04114530866 * cap}{cap - 2.098833146} \tag{5.7}$$

where $cap$ is the measured capacitance value in picofarads (pF) and $d$ the distance in cm. This function approximates the input data points very well in the required interval $[2.1, 3.1]$ resulting in a maximum error of only $7.4\,\mathrm{mm}$.

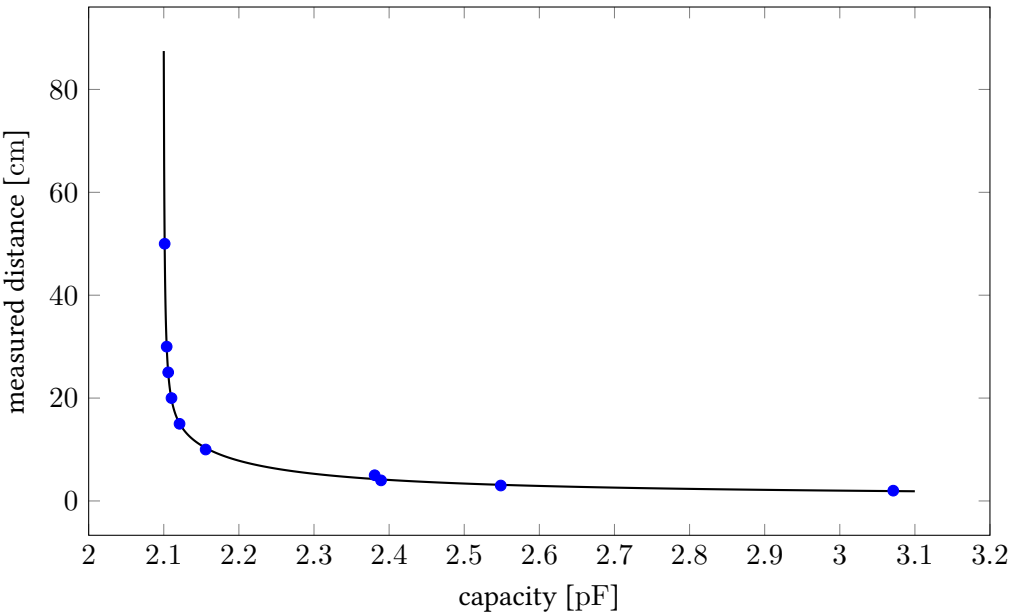



**Figure 5.9.** Approximation of the distance in relation to the capacitance; recorded measurements (blue) and approximated function (black)

The process of manually recording multiple data points and then determining the correct regression function with the least error in approximation, however, is very time consuming and tedious. In order to increase the level of automation in calibrating new electrode geometries, a new system was proposed in [90]. The main premise of this new approach was to eliminate the need for manual data recording and regression analysis by employing neural networks to approximate the distance function based on ground truth data recorded by an external measurement system. The Vicon camera system paired with the Nexus motion capture software was chosen for the task of recording the motion data of a human hand within the electrical field of a capacitive sensor mounted on a robot arm. For the purpose of data recording, initially two capacitive sensors were mounted on one link of a collaborative robot (see Figure 5.10a) from Universal Robots, specifically the UR5 (seen in Figure 5.11). Since the UR5 is comprised of simple joints and tubular aluminum extrusions as links, it is fairly easy to mount additional devices, such as the capacitive sensors utilized here, to the robot's exterior. For the purpose of recording the hand motion in proximity to the capacitive sensors, a sturdy leather welding glove was fitted with multiple markers (see Figure 5.10b) in order to make repeated evaluations easier, since the markers would otherwise need to be attached directly to a human hand each time.

The training data for the neural networks was recorded by setting up the UR5 robot fitted with capacitive sensors in the Vicon motion capture arena and performing multiple approaching and withdrawing motions with respect to the two capacitive sensors (see Figure 5.12). The measured capacitance values were then stored in combination with the distance measurements. These were calculated on the basis of the positional data of the robot link in relation to the marker glove provided by the Vicon motion capture

**(a)** Robot link with capacitive sensors and tracking markers

**(b)** Glove with tracking markers attached

**Figure 5.10.** Capacitive sensors and marker setup for the robot arm and glove to be worn during data recording in order to determine the ground truth distance values for later training of the neural net.

system. Based on this ground truth, the distance function can then be estimated by training the neural net to correlate the distance to the measured capacitance. During execution, the trained neural net is then able to accurately estimate the current distance to a human hand using only the current measured capacitance.

As the relationship between distance and capacitive measurement is close to exponential and the preliminary results were promising, the *Quasi-Newton method* was selected for training. Since the task at hand is a simple function regression analysis, a feed-forward neural network is amply sufficient. This network was configured with two inputs, one for each of the capacitive sensor values, and three outputs for the distances, two of which are the direct distances between the respective sensor electrode plane and the third being the distance to the midpoint between the two sensors. This third value is not the result of actual sensor input, but the results showed that this value could be inferred well from the two actual measurements. The network layout was optimized through experimentation and the best configuration was achieved through three hidden layers with 9, 12 and 9 neurons respectively.

Figure 5.13 shows the results of the distance estimation with the previously described neural network for the midpoint between the two sensors attached to the robot. Three approach motions were performed in this experiment one after another. The plot shows data recorded during runtime with the previously trained neural network for

**Figure 5.11.** The UR5 robot from Universal Robots. A collaborative robot with six axes, capable of manipulating a payload of up to 5kg in the vicinity of humans. Due to its simple geometric construction it is a popular choice for researches seeking to mount additional hardware to a robot quickly and easily. (Image source: [119])

the distance estimation of the midpoint (blue line) and compares it to the Vicon ground truth distance (red dashed line), which was recorded simultaneously for the purpose of verifying the attained results. The deviation between the two datasets and thus the estimation error of the neural network with respect to the actual distance is shown at the bottommost edge of the plot (grey line). Due to the fact that capacitive sensor values become largely unreliable at great distances and far more susceptible to ambient influences, the distance estimation is capped off at a maximum distance of $350\,\mathrm{mm}$. Values beyond that distance are considered out of range. Overall, the neural network for distance estimation achieves a mean error of $20.7\,\mathrm{mm}$ when disregarding all values where the difference between the estimation and the measured ground truth is exactly $0.0\,\mathrm{mm}$, since this only occurs at the explicitly defined cutoff point for the distance data and thus would manipulate the resulting mean error.

### 5.4.2 Compensation of Self-Influence on Capacitive Sensors

The electrical field of capacitive sensors is not only influenced by environmental factors, such as temperature or humidity, but naturally also by any other electrical field emanating from devices in the vicinity of the measurement electrodes or the evaluation electronics. Especially the electromagnetic coils in the rotors of the electric motors powering each axis of the robot the sensors are attached to induce a magnetic field

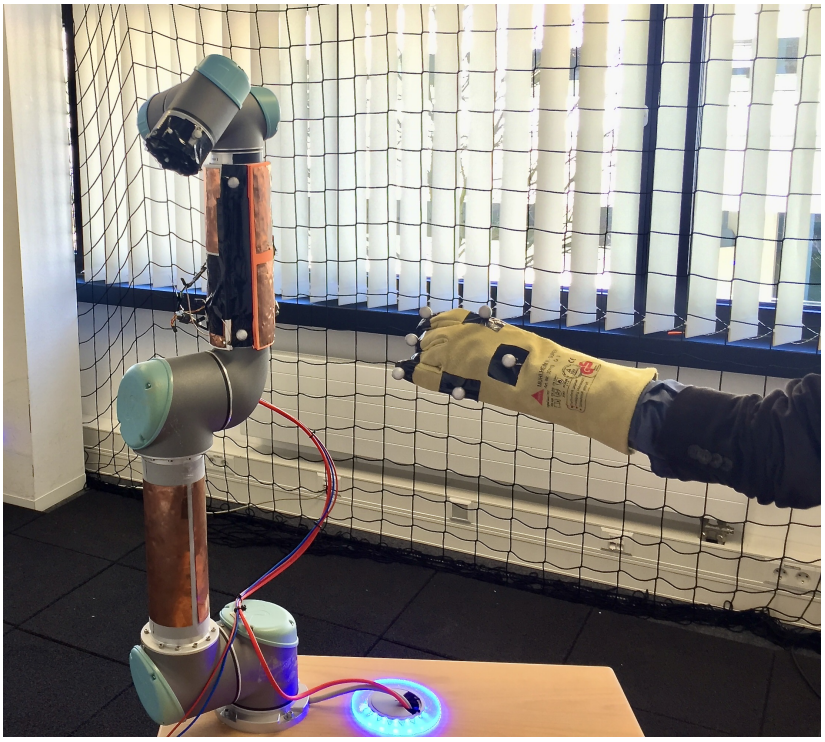


**Figure 5.12.** Experimental setup for the recording of distance data in relation to capacitive sensors measurements. The robot is fitted with reflective Vicon tracking markers around the location of the capacitive sensors, while the human hand position can be determined with the help of the glove also fitted with Vicon markers.

which interferes with the electric field projected by the capacitive sensors. This influence is clearly visible when recording the capacitive measurements while the robot is in motion. Figure 5.14 shows the difference between the base sensor value while the robot is standing still and the measurements taken during the motion of two different axes, most notably one preceding and one subsequent axis with respect to the sensor position on the robot. With the sensors attached to the second link (i.e., between axes 3 and 4 of the UR5 robot) the first axis is the furthest possible from the sensor. However, since it is the most powerful, due to the fact it has to carry the entire robot's weight and any additional payload at the maximal motion speed of the robot, it also induces the largest influence on any electric field in the vicinity (dotted blue line). Nevertheless, its influence is not as apparent as that of the other axis, due to the fact that the influence of electric fields decreases exponentially with distance. The fifth axis is the closest to the sensor electrodes and consequently influences the measurements the most (dotted red line). A fact which can be leveraged in order to ease the compensation of self-influences of the robot axes on capacitive measurements is that these individual influences are additive with respect to the total influence exerted on the sensors. This can also clearly be seen in Figure 5.14, where the sum (dashed black line) of the previously described individual influences (dotted red and blue lines) largely matches the influence exerted by a combined motion of both axes simultaneously (solid gray line). Exploiting this

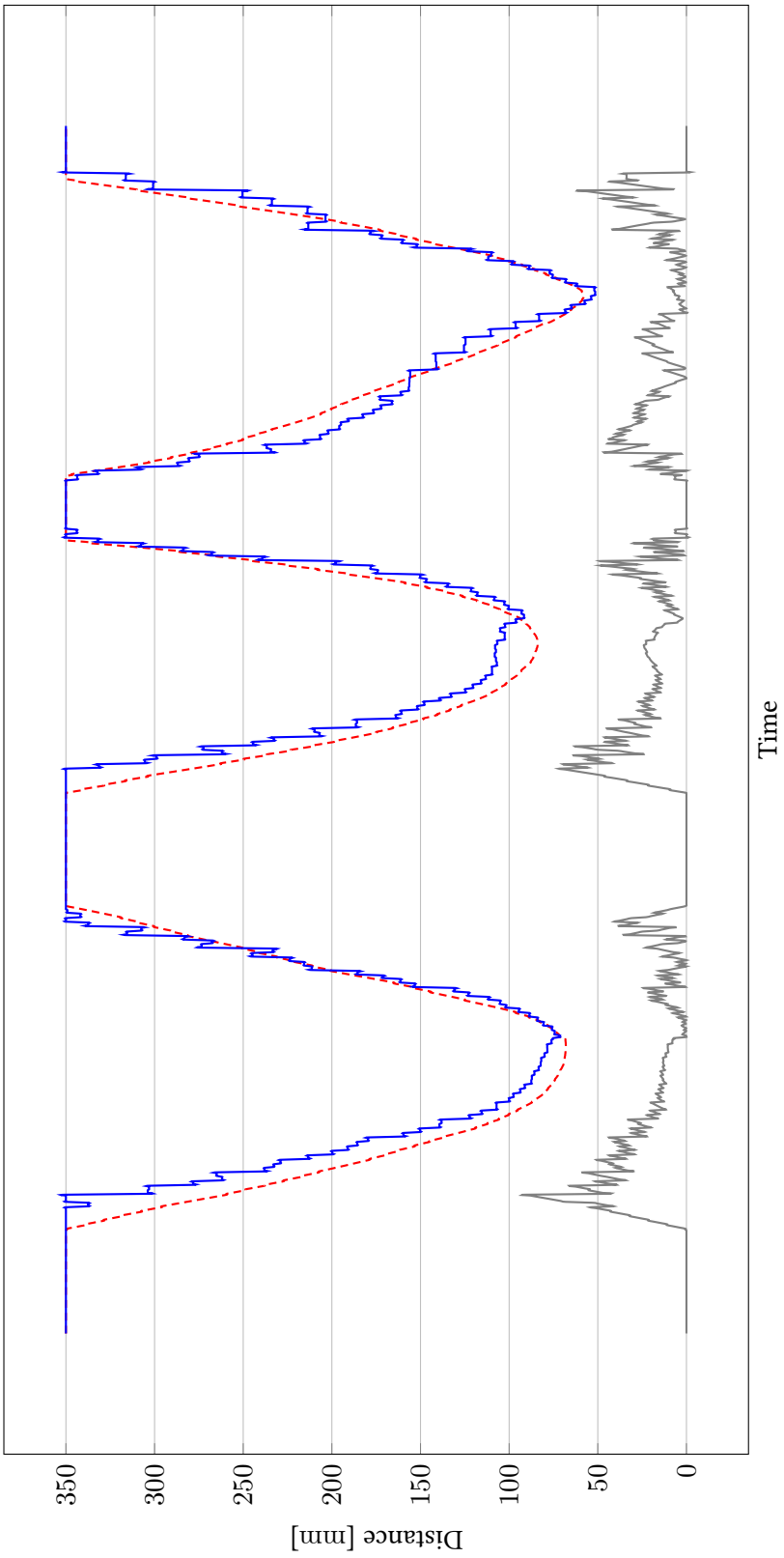



**Figure 5.13.** Distance between the robot (i. e., the capacitive sensors on the link between joint 3 and joint 4) and the palm of a human hand; the graph shows the output of the neural network for the distance estimation (blue, solid) using capacitive sensors and the actual distance measured by Vicon (red, dashed) with respect to the midpoint between the sensor electrodes. The distances from the respective electrode centers behave analogously. The absolute value of the approximation error, i. e., the deviation between the estimated and measured distances, is shown in gray. [90]

principle, a modularized system can be constructed in which the individual influences of each axis can be compensated separately and be subtracted from the incoming raw capacitive sensor value before processing it further, e.g., for distance estimation.

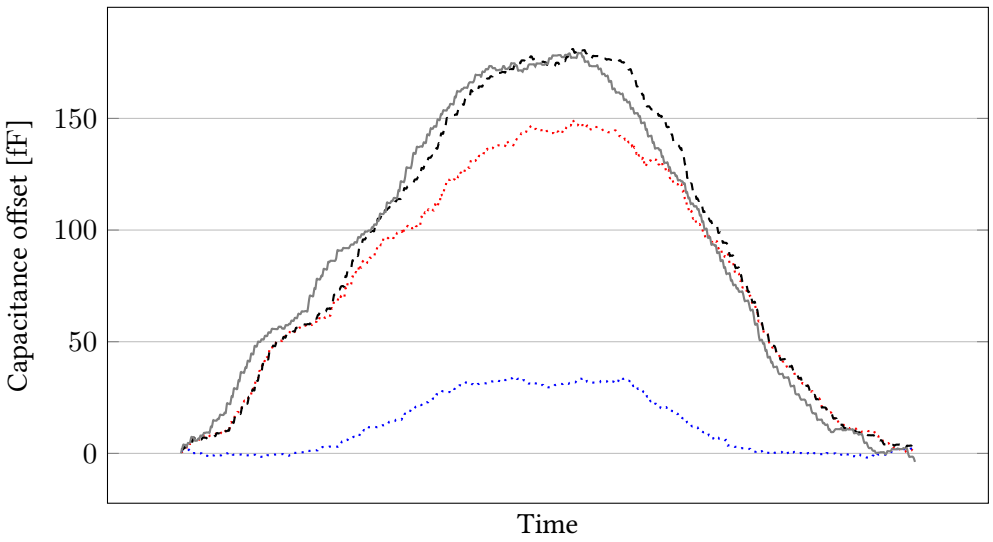



**Figure 5.14.** Comparison of measured sensor values during three different robot motions: The dotted lines show the sensor values during a single axis movement of the first (i. e., lower dotted blue line) and the fifth axis (i. e., upper dotted red line), respectively. The solid gray line shows the sensor measurements during a simultaneous motion of both axes. The dashed black line represents the added measurements of both single movements (i. e., the sum of both dotted lines), which clearly follows the influence of the simultaneous movement (i. e., the solid gray line). [90]

Preliminary experiments, however, showed that the influence exerted on the capacitive sensor measurements by the robot's axes is not only related to the absolute joint position or its current velocity, as one might assume, but rather the combination of both. Figure 5.15 shows this fact very clearly. The data presented here is the result of a testing motion in which the change in capacitance of one sensor was recorded while the robot moved its first axis between $-60°$ and $60°$. Evidently, the capacitance offset is not constant for every discrete angular position of the robot's axis, but rather depends on the angle in combination with the current direction of movement of the axis. This fact is particularly obvious around $0°$ where the motions in positive and negative direction display the starkest difference in measured capacitance. Consequently, the self-influence compensation for a given robot must not only consider the capacitance offset for every possible angle of the axes, but also correlate these values with their angular velocities.

Following the insights gained from the preliminary experiments, two separate neural networks were trained, one for each of the two previously selected axes, for the compensation of self-influences. As was the case with the neural network for the distance estimation, the best network structure was determined experimentally. The final configuration consisted of again a feed-forward neural network but in this case with only two hidden layers with 12 and 24 neurons, respectively. The resulting self-influence

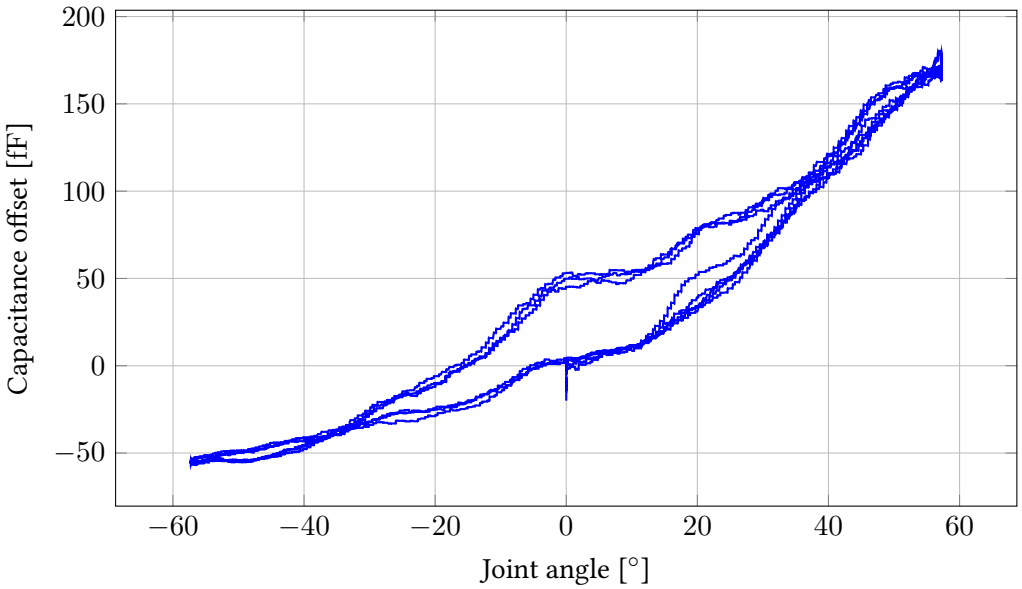



**Figure 5.15.** Influence of the current joint angle and direction of motion on the measured capacitive sensor value. The most significant influence is apparent around $0°$ where the maximum difference between measurements reaches $50\,\mathrm{pF}$ for the same joint angle. The plot shows the result of multiple movements of a single axis. [90]

compensation network receives the current joint position and (signed) velocity as inputs and produces a capacitance offset value (with respect to the base value of the sensor) as output. During training the required capacitance offset value is calculated externally by recording the initial base value before performing any motion and continuously calculating the offset from the base sensor value during the movement of the robot. Capacitive offset, joint position and velocity are recorded together and form the training dataset for the neural network. The performance of the trained neural network was then evaluated in tandem with the network for distance estimation. While the robot performed the same motions as during the data recording phase, a human operator moved their hand in front of the robot in proximity to the sensors. Performance of the self-influence compensation was gauged by determining the resulting distance error by means of external validation through the Vicon tracking system. The results of this evaluation can be seen in Figure 5.16 for the experiment with the self-influence compensation disabled and in Figure 5.17 with it being active.

Evidently, the distance estimation alone (cf. Figure 5.16) is virtually unusable during the motion of the robot, since the influence of the axes' movements is so great, that the robot would most often come to a complete standstill due to a fictitious obstacle created by the effect of the motors on the electrical field of the capacitive sensors. The mean approximation error of the distance estimation while in motion comes to about $169\,\mathrm{mm}$. This is vastly outperformed by the distance estimation with added self-influence compensation as seen in Figure 5.17, where the mean error is merely $38.4\,\mathrm{mm}$.

While this is far worse than the distance estimation when the robot is stationary (mean error: $20.7\,\mathrm{mm}$), it is a vast improvement over the distance estimation when no self-influence compensation is applied.

### 5.4.3 Software Architecture for Training and Runtime Execution

For the implementation of the neural network architectures, the Open Neural Networks Library (OpenNN, [11]) was employed, which facilitated a straightforward implementation of the required network architectures through a comprehensive C++ API. OpenNN itself claims the ability to handle larger datasets and faster training performance compared to competing solutions. The library was integrated into a pre-existing robot controller with real-time capability in order to enable quick response times for the reaction of the robot to the detection of obstacles.

The software architecture for the recording of datasets used in training the neural networks for capacitive sensor data evaluations can be seen in Figure 5.18. All device drivers for the required hardware are real-time software components. This includes the robot driver, whose movements must naturally be commanded in real-time, but also the driver for the Vicon tracking data, since it must be accurately matched against the real-time capacitive sensor measurements in the data acquisition phase (cf. Figure 5.18). Data logging of all relevant input sources for the subsequent training of the neural networks is performed by a dedicated software component which receives the raw capacitive sensor data, the robot's joint positions and velocities as well as the cartesian positions of the capacitive sensors and the tracking glove from the Vicon Tracking driver. All these data points are aggregated by the data logging component, then correlated by their respective time stamps and stored to disk in log files separate per data acquisition session.

Actual training of the neural networks is then performed offline by a separate training application, which reads in one of the log files produced in a data acquisition session and trains the neural network according to the parameters specified in the training application. The trained neural network is then again stored on disk and can later be read in by a third dedicated application, the final productive application.

In the runtime configuration, the architecture is constructed as seen in Figure 5.19. This part of the overall application structure in concrete implementations can be viewed as a separate subsystem which constantly guards the commanded motions of the robot against collisions with humans in the workspace, but otherwise does not need to interact directly with the rest of the specific application. The data from the Vicon tracking component was only used for validation purposes in the runtime configuration. Consequently, in the architecture this data is replaced by the output of the neural network evaluations at runtime, which is twofold. First, the neural network for self-influence compensation receives the raw sensor data and subtracts the share of the robot's self-influence from the raw capacitive sensor value. This corrected value is subsequently passed on to the neural network for distance estimation, which consequently receives a sensor value only containing the external influences and can thus produce a more accurate value. Motion execution is therefore enhanced by a guarding function which

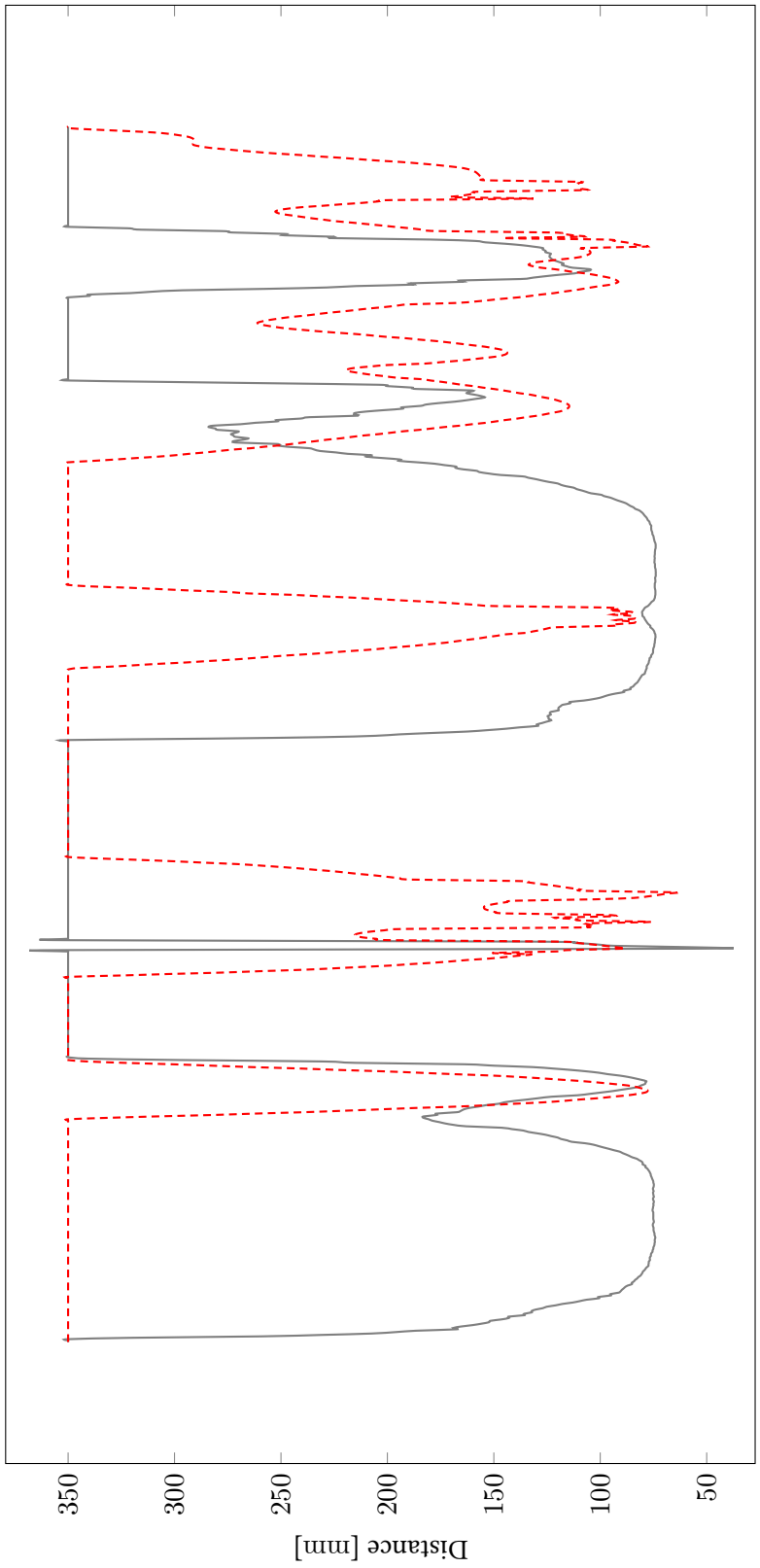



**Figure 5.16.** Comparison between distance estimation (solid gray) and ground truth (dashed red) w/o compensation. [90]

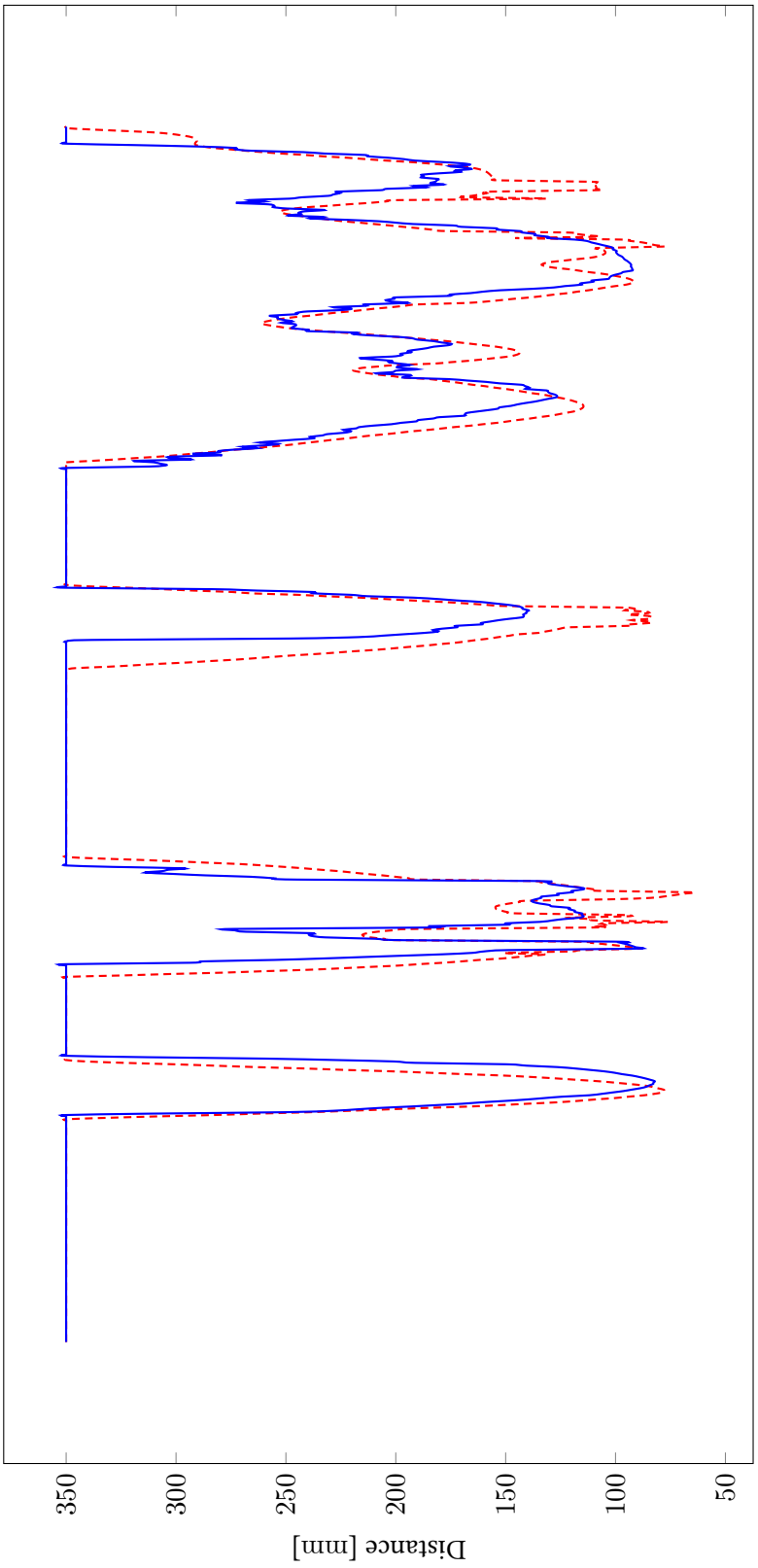



**Figure 5.17.** Comparison between distance estimation (solid blue) and ground truth (dashed red) with compensation. [90]

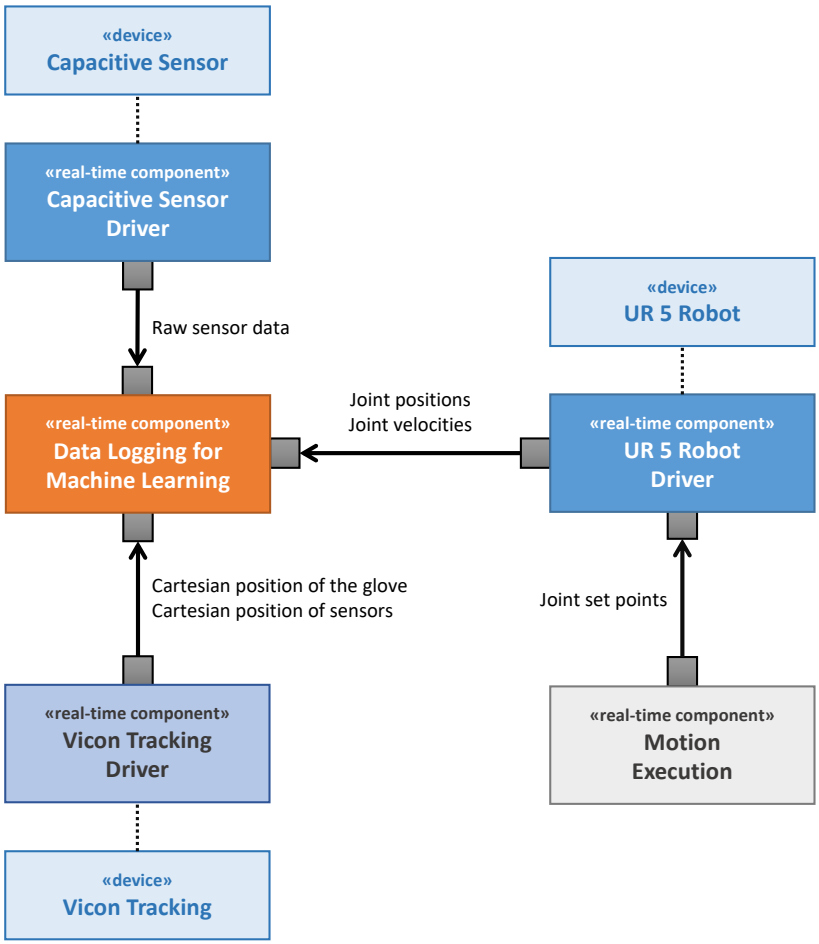



**Figure 5.18.** Software architecture for the recording of training data for capacitive sensors. [90]

continuously processes the incoming distance estimations and checks the currently commanded motion for possible collisions. A possible strategy, which was also employed in the experiments, is to scale the override, which is a percentage scaling of the maximum speed of the robot, depending on the distance to an obstacle.

Linear braking, however, is not very intuitive for people interacting with the robot, since it causes a rather sudden and strong deceleration of the robot at a seemingly arbitrary distance. Users of such systems can of course learn to expect this reaction over time whilst working with the robot. Nevertheless, a more gradual braking curve is preferable, which is flatter at large distances and becomes ever steeper on approach until finally bringing the robot to a complete halt. Figure 5.20 shows an example of such a braking curve as defined by Equation (5.8). This function only produces a viable output in the interval between $10\,\mathrm{and}\,30\,\mathrm{cm}$ as prescribed. Within the two-staged safety concept described in Section 3.2, this braking curve would be applied in the absolute safety stage of the concept, while other, far more advanced techniques need to be employed for the convenience stage. In any case, it is not sufficient to merely alter the robot's speed, but

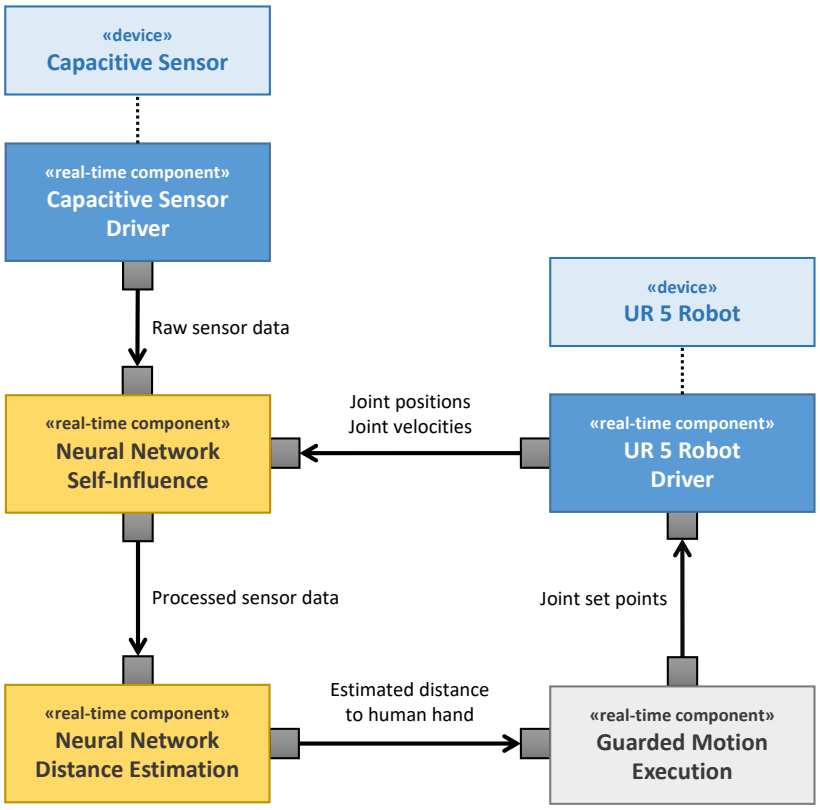



**Figure 5.19.** Software architecture for the runtime execution of the capacitive sensor data calculations. [90]

the convenience stage must actively define alternative robot trajectories in order to successfully curb the need for the absolute safety stage in the majority of cases.

$$t = \frac{x - minDist}{maxDist - minDist}$$

$$Override = (((t * (m - 2)) + (3 - 2 * m)) * t + m) * t \tag{5.8}$$

The implementation and evaluation of the neural networks were performed by Martin Siehler for his bachelor's thesis.

## 5.5 Hardware for Capacitive Sensors

Electrodes for capacitive sensors are utterly simple devices. They typically consist of plain pieces of copper foil or similar shaped metal which are then electrically connected to the circuitry responsible for measuring the actual capacitance. In [54] a customized

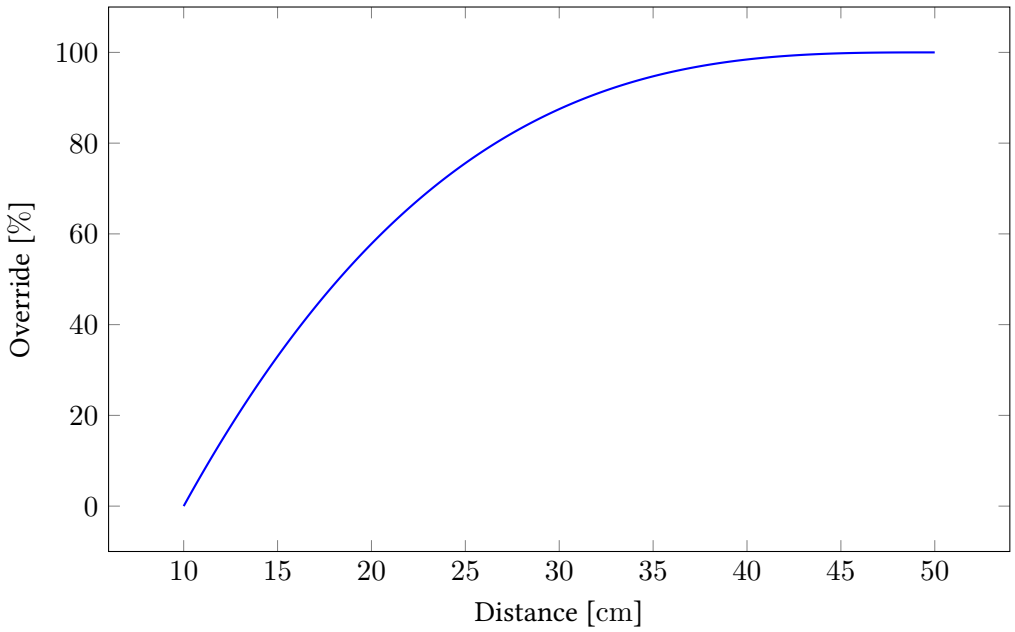



**Figure 5.20.** Improved braking curve for a robot $m = 3$ defined for distances between 10 and 50 cm.

measurement circuit (seen in Figure 5.21) on the basis of the AD7147 integrated circuit (IC) by Analog Devices [10] had been developed and was also employed in the development of the neural networks for intelligent capacitive sensor evaluation [90].

In addition to the actual measurement IC the circuit was fitted with a microcontroller for the configuration and operation of the same as well as smaller pre-processing and communication tasks. While the AD7147 chip was never designed to be used in such an application, but rather for contactless control surfaces like touch-buttons and sliders at very close range, it still performed quite well at larger distances of up to 35 cm as described in the preceding sections.

Inherent to the measurement principle of the AD7147 is the sensitivity to high frequency noise from electronic devices in the vicinity of the attached electrodes. Especially in the frequency range of the excitation signal this was particularly evident.

The developed system consisted of two separate circuit board designs, one of which was the capacitive measurement circuit itself and the other an interchangeable system of base boards (seen in Figure 5.23). The board for the measurement circuit routes all available connections of the microcontroller and the measurement inputs to the attached interconnect ports, which then can be split out to connectors for the attachment of capacitor electrodes or, in the case of the microcontroller ports, routed to communication ports or components mounted on the base board, such as status LEDs.

Although the composition of the employed electrodes was exceedingly simple, consisting only of two sheets of copper foil tailored to fit the sides of the robot structure, manu-

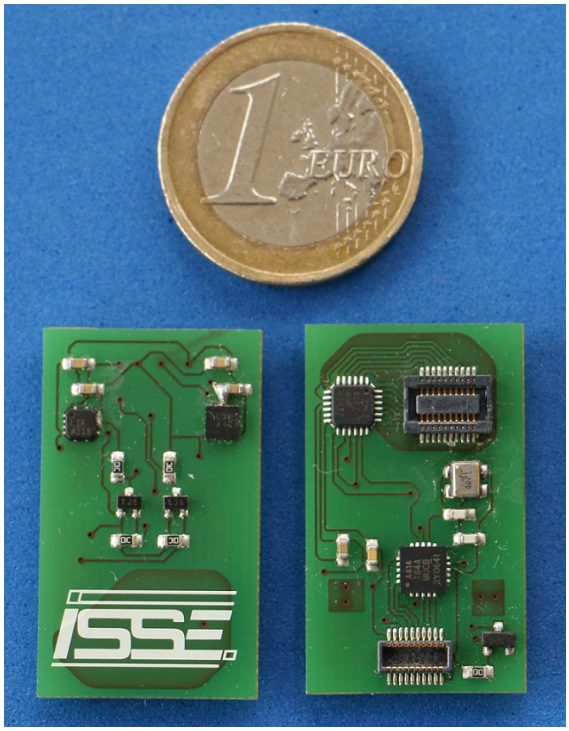


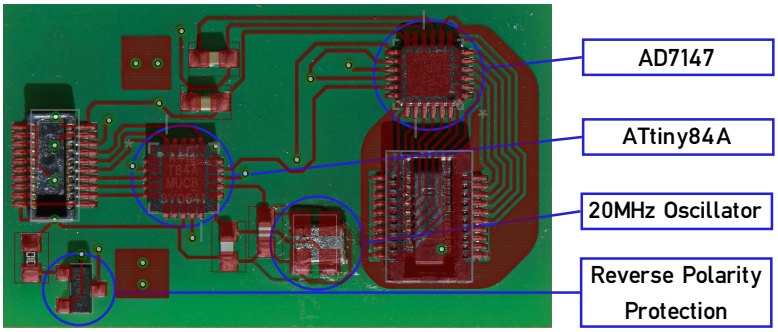

**Figure 5.21.** The capacitive measurement circuit presented in [54].

facturing them and connecting them to the measurement circuit with the appropriate shielded conductor was a fairly involved process as seen in Figure 5.23a.

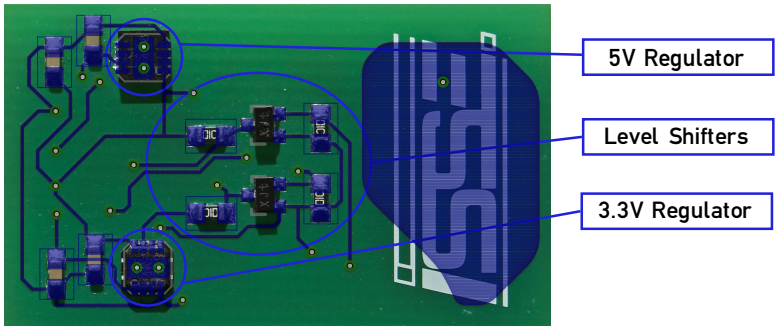
Based on the experience of the previous development of measurement circuits and the experiments performed with various electrode configurations, a concept of an entirely novel type of modular capacitive sensor was developed. The key requirements of this new device were defined as follows:

- **Modular design**
  Multiple sensor modules shall be easily deployable and interconnectable with each other to quickly fashion any type of robot with an external skin of capacitive sensors.
- **Compensation of environmental influences**
  Each module must be able to compensate for external influences as best it can. Optionally, additional electrically shielded modules can be deployed around the robot to act as reference points.
- **Functional Safety**
  The system architecture shall consider the requirements of functional safety in order to create the foundation for a truly redundant sensor system capable of securing robots in production environments.

The foundation of this new circuit design is the FDC2114 capacitive measurement IC from Texas Instruments [117]. Unlike previous ICs, the FDC2114 measures capacitance




**(a)** Front view



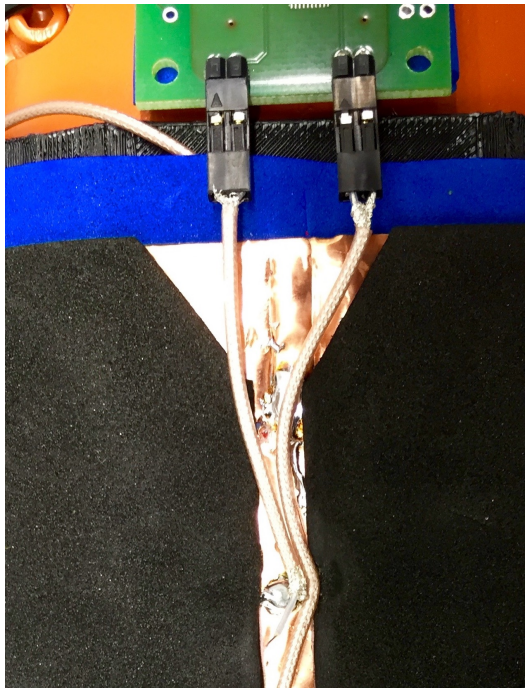

**(b)** Rear view

**Figure 5.22.** Layout of the capacitive measurement circuit presented in [54].

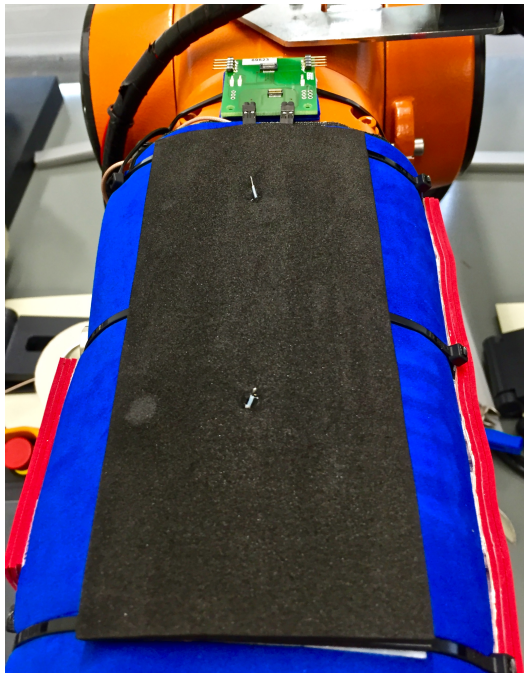
not by storing the induced charge differential and converting the resulting voltage to a digital value, but rather by creating an oscillating circuit with the electrode and measuring the shift in resonant frequency of the entire system induced by objects in the vicinity of the electrode surfaces. This method is far less susceptible to external noise, especially since a narrow band filter can be applied around the frequency of the excitation signal in order to eliminate most influences before the actual signal processing begins. This promises much higher possible detection ranges of up to 60 cm using the same electrode configuration as with the previous measurement circuit as seen in Figure 5.23.

### 5.5.1 Modular Design

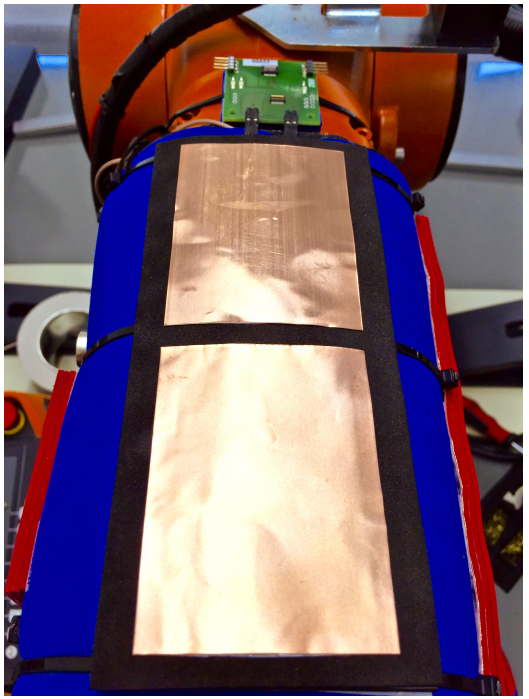
Every capacitive sensing module is devised as a self-contained system which can be deployed on a robot's outer surface. This especially includes the measurement electrodes so that external connections to the module can be limited to power and communication only. The circuit was devised with four square integrated electrodes arranged in a square pattern. Since the functionality of the FDC2114 IC allows for the measurement of different electrode configurations, all four electrodes can be connected directly to it and measured independently of each other. Differential electrode configurations can also be



**(a)** Connection of shielding wire to shielding electrode.



**(b)** Addition of insulating dielectric material as spacing element.



**(c)** Assembled measurement electrodes.

**Figure 5.23.** Construction of two capacitive sensors attached to a robot arm. The sensor consists of two electrodes, one for active shielding and one for the actual measurement, which are separated by a dielectric material in order to increase the projection distances of the electrical field. Connections to the measurement circuit are made by connecting coaxial cables between the circuit and the electrodes. The outer shielding of the coaxial cable transmits the shield signal and hence protects the desired signal carried via the inner core.

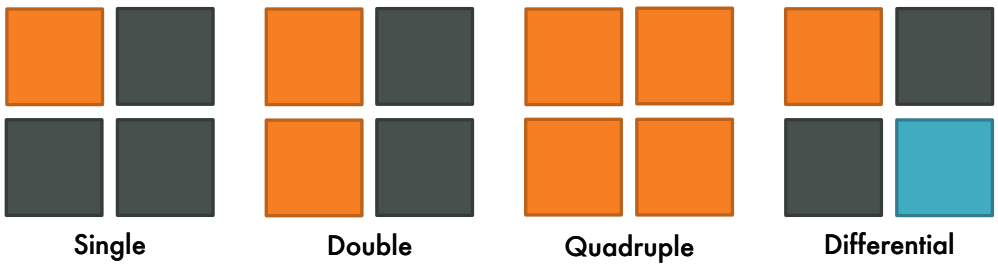



**Figure 5.24.** The capacitive sensor module based on the FDC2114 supports various electrode configurations per module by multiplexing the individual electrodes with the four measurement inputs of the IC.

configured through parametrizing the IC accordingly. The true power of the concept lies in the ability to combine arbitrary arrangements of the integrated electrodes by multiplexing the electrode connections to the measurement inputs of the capacitive measurement IC. Designing capacitive sensor electrodes is invariably a trade-off between the maximum detection range and the maximum resolution the sensors can provide thereafter [95]. Increasing the area of the electrode projects a larger electrical field and thus enables the detection of objects farther away from the sensor. Since space on the surface of a robot arm is severely limited, however, the designer of such a sensor system may choose to prioritize the positional resolution of the capacitive sensors to the detriment of the maximum detection range in order to enable more accurate interaction with humans in close proximity to the robot. The new concept presented here attempts to combine the advantages of both electrode types by enabling the multiplexing of the attached electrodes. As a result of this feature it becomes possible to create a detection cascade over the entire detection range, enabling the detection of objects at greater distances and also with greater positional accuracy as they approach the sensor. A conceptual representation of this cascade method can be seen in Figure 5.25. As long as no object is in proximity to the robot, the sensor combines all available electrodes into a single one in order to deliver the largest detection range achievable by the system. This enables the system to detect the presence of objects at the earliest possible moment. While the object travels closer to the capacitive sensors, the electrode configuration can be gradually adapted with respect to the balance of maximum detection range and the maximum positional resolution of the selected configuration.

A further feature of the integrated module design is the inclusion of electrode interconnect ports which are shielded analog signal connections and allow the connection of additional electrodes to each module. But rather than connect completely external electrodes to the modules, these can be utilized in a far more sophisticated manner by connecting each module's electrode interconnects with each other when multiple modules are deployed together. Each module has eight such interconnect ports such that all electrodes of one module can be read by the neighboring module's measurement IC. The reasoning behind this number of connections will be explained in detail in Section 5.5.2. This functionality allows grouping multiple of the smaller integrated electrodes across connected modules into larger electrodes in order to enable even greater maximum

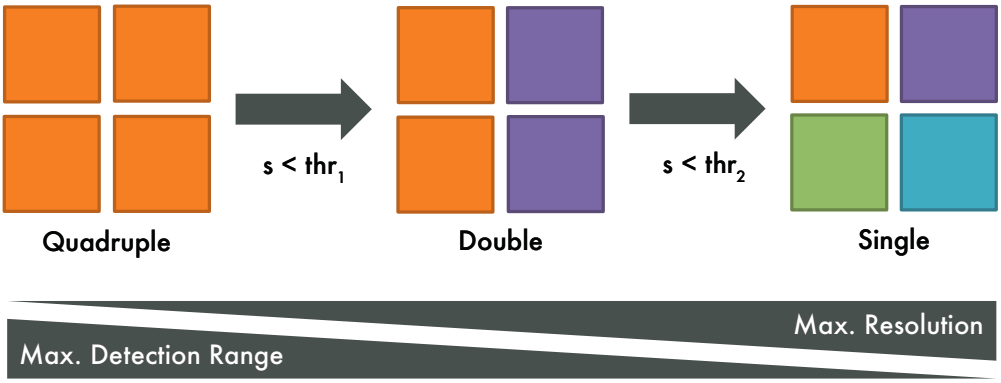



**Figure 5.25.** Concept of the electrode cascade method. For each distance to an object the optimal electrode configuration can be chosen and combined via the multiplexing circuit. All electrodes are combined in the default setting in order to detect objects as soon as possible. With diminishing distance the number of connected electrodes can be gradually decreased according to their respective detection range in order to increase the positional resolution at closer distances. $s$ is the measured distance to the object and $thr_1$ and $thr_2$, respectively, the thresholds below which the next smallest electrode configuration is selected.

detection distances. The previously introduced cascade mechanism can then also be employed across multiple modules, just as well as differential electrode configurations can span across larger areas if the system is configured to use single electrodes from all connected modules in such an arrangement. Examples of such configurations utilizing electrodes across several connected modules can be seen in Figure 5.26.

The system architecture to enable these features employs a number of multiplexing circuits with the goal of allowing the utmost flexibility in the choice of concrete electrode configuration for every application and desired measurement cascade. A conceptual overview over the electronics architecture designed to enable these functionalities can be seen in Figure 5.27. All electrodes are connected to 1x4 multiplexers which supports the connection of a single or any other permutation of the integrated electrodes to be connected to each of the measurement inputs of the capacitive measurement IC. The incoming and outgoing electrode interconnection ports are each equipped with 8x4 multiplexers enabling the use of eight different channels over each connection port and couple each of the eight links with the desired input of the measurement IC.

The best results are, however, achieved if the interconnections of capacitive sensing modules are restricted to modules deployed across a single surface of the robot structure, since only then can the direction of an approaching object be accurately determined. In case multiple modules encompassing an entire link of the robot were to be connected with each other and multiple of their electrodes were to be configured as a single, connected electrode, then the direction of an approaching object with respect to the robot could no longer be determined. While this may be sufficient in certain application scenarios it would egregiously underutilize the capabilities of the presented sensor system.

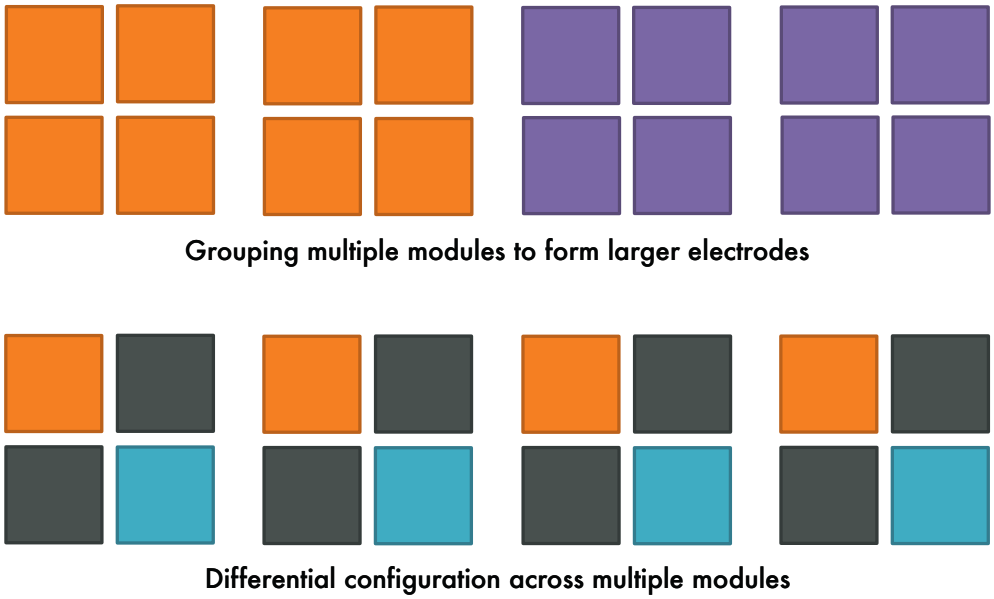



**Figure 5.26.** Possible electrode combinations with the proposed measurement circuitry.

### 5.5.2 Functional Safety

In real-world applications involving human-robot-collaboration, functional safety is of paramount importance. Any robot system operating in the vicinity of human workers must comply with international safety standards and the entire system including the entirety of peripheral components must undergo a risk analysis to gauge the safety compliance of the overall robot application. A common strategy in the development of new robotics applications is to use off-the-shelf components with prior certification according to the relevant safety standards.

While this is by no means the entirety of measures necessary towards creating a certifiably safe capacitive sensor system, it is a required initial step and thus was accounted for in the conceptual design of the modular capacitive sensor system presented here. Via the electrode interconnect ports with a maximum capacity of eight connections, all electrodes can be connected to the capacitive measurement IC of the neighboring module and vice-versa. In this fashion it becomes possible to redundantly measure all electrodes, thus eliminating any faulty measurements which could be provided by one of the measurement ICs if it was utilized alone. The resulting capacitance measurements can then be verified by one or both of the integrated microcontrollers or even by a higher-level control system independent of the capacitive sensor modules. Figure 5.28 shows the signal path between the electrodes and the measurement IC under regular circumstances, while Figure 5.29 shows how the signal path is modified by the multiplexers in order to connect the electrodes of both sensor modules to the measurement IC of the respective neighboring module. Since the measurement cycle is fairly short, it can be assumed that the resulting values are within the same range, barring any noise

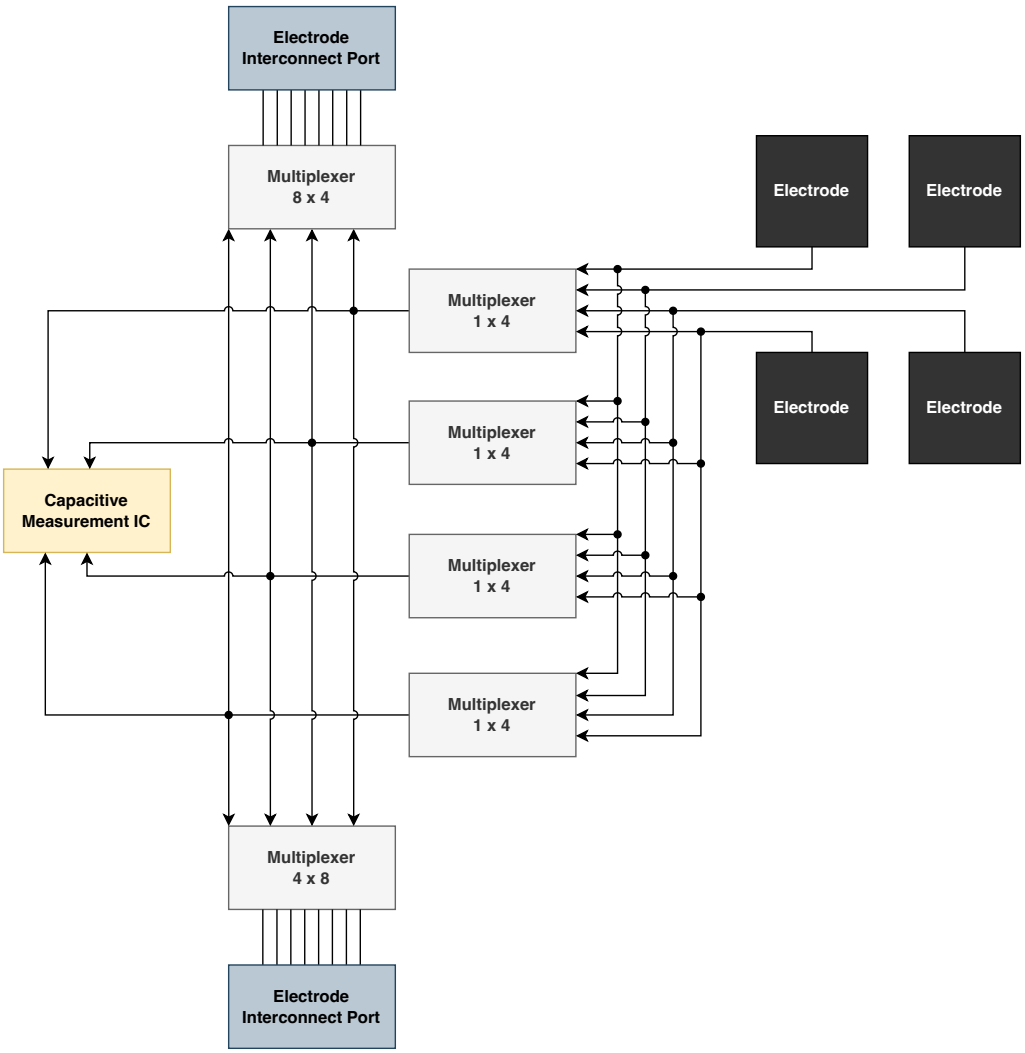



**Figure 5.27.** Functional block diagram of the novel capacitive measurement module.

picked up by the electrodes. Consequently the verification of values against each other can allow for a certain tolerance before claiming a fault in the system.

In terms of redundancy pertaining to safety standards the presented concept can detect faults in any of the subordinate components and is thus a valid first step towards a functionally safe capacitive sensor system. Should the interconnect ports or any of their multiplexers fail, the result of the capacitive measurement would differ vastly, since the overall area of the connected electrodes would differ greatly from the direct connection to the local IC. The correct function of the capacitive measurement ICs themselves is verified by swapping the connected electrodes between its own and the neighboring IC. A failure of the multiplexers directly connected to the electrodes can be detected by periodically cycling the connected electrodes into one measurement input and verifying the proportional increase in overall capacitance. Any failure in the microcontrollers can be detected through cross-checking their respective measurements and calculations

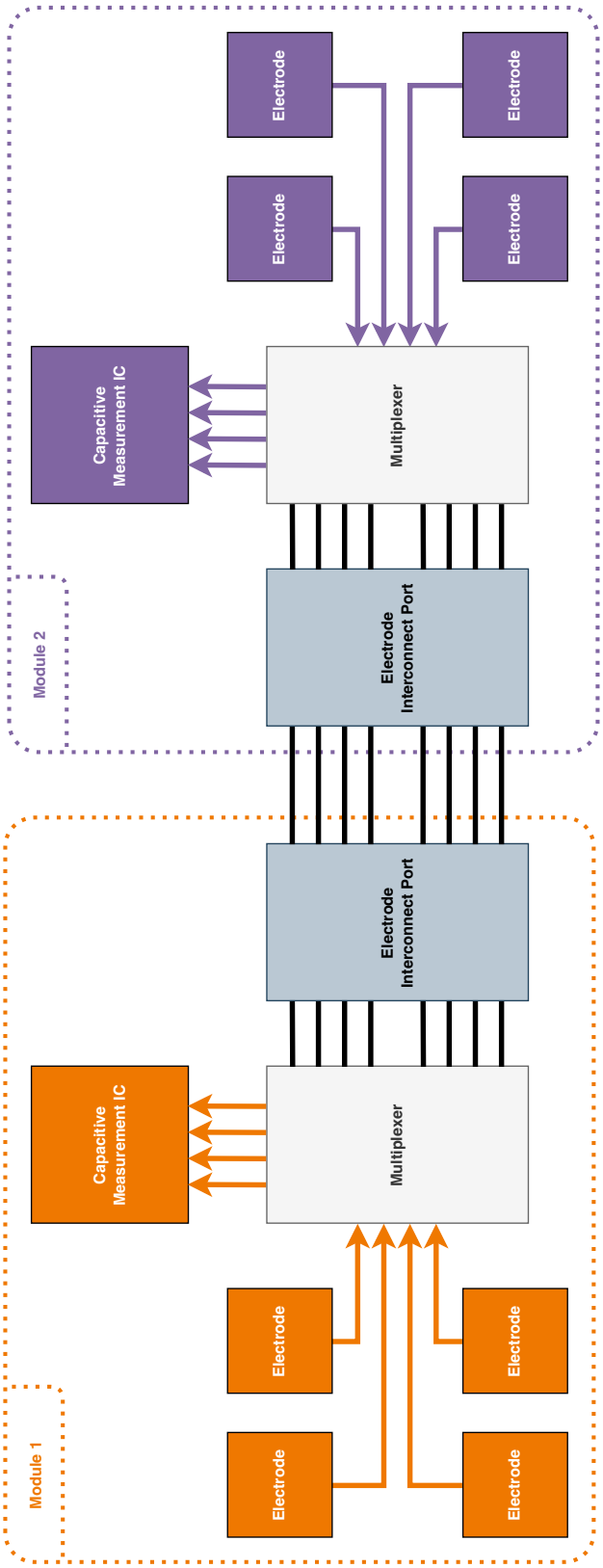



**Figure 5.28.** Depiction of the signal flow path through the main components of the capacitive sensor module in the default case in which the four integrated electrodes are directly connected to and measured by the module's own capacitive measurement IC.

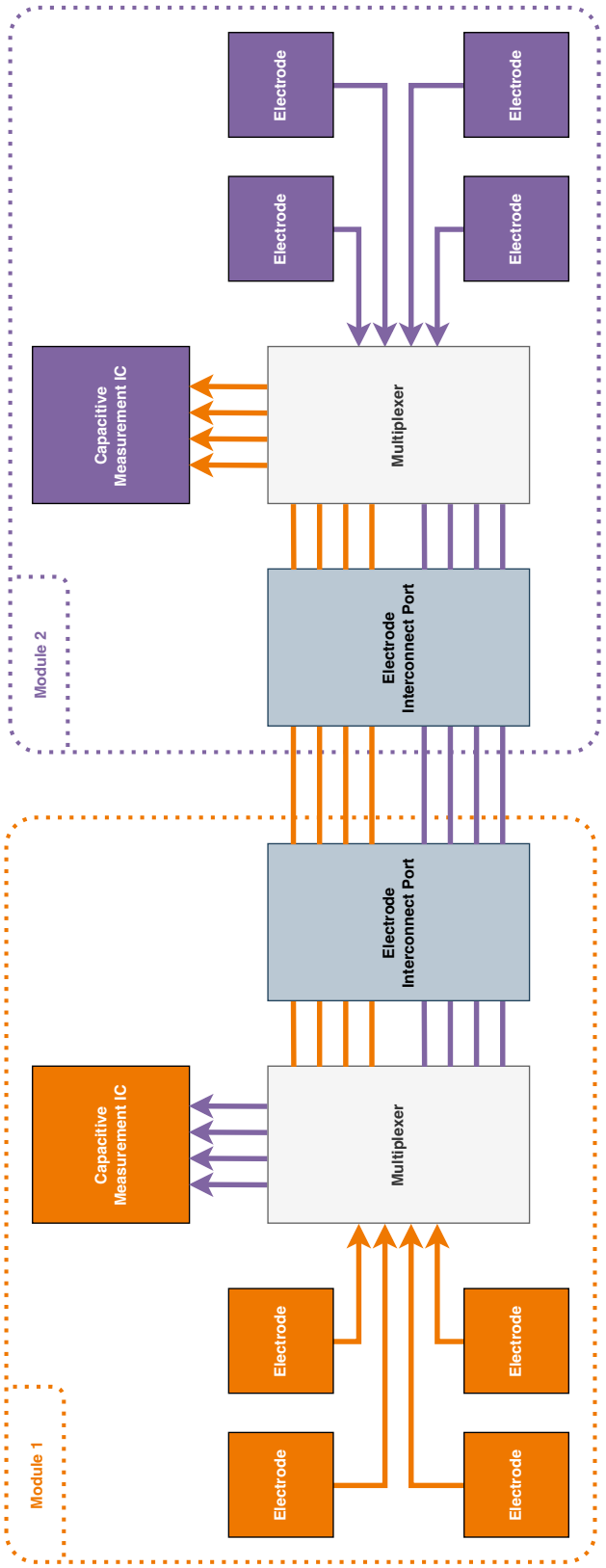



**Figure 5.29.** Depiction of the signal flow path through the main components of the capacitive sensor module during the safety relevant redundancy check. Signals from each module's integrated electrodes are routed to the neighboring module's capacitive measurement IC via the multiplexers and the electrode interconnect ports. Value congruency can subsequently be verified by the integrated microcontrollers or optionally a higher control instance.

against each other in the overarching control system. Alternatively, inherently fail-safe microcontrollers can be employed as an added safety measure.

### 5.5.3 Compensation of Environmental Influences

Temperature, humidity and atmospheric pressure variations in the vicinity of capacitive sensors heavily influence the measured capacitance in such sensors, since these influences alter the dielectric constant of the capacitor [116, p. 25f]. To compensate for these the previous capacitive sensor system [54] already utilized an additional electrode stored near the control cabinet of the robot and encased in a shielded enclosure acting as a Faraday cage in order to guard it from any grounded or electrically active influences in the vicinity. In this manner only the influences exerted by the environment on the capacitive sensors, specifically of temperature and humidity are registered by the shielded sensor. Recording the initial capacitance value measured by the shielded sensor consequently allows the continuous compensation of environmental influences at runtime by subtracting the difference between the initial and current value of the shielded capacitive sensor from any other values retrieved from any sensors deployed on the robot structure. Before the introduction of the central reference electrode for compensation, environmental influences had been seriously detrimental to the operation of the robot, since the base capacitance level of all attached sensors would slowly increase over time, leading the sensor system to perceive non-existent objects and as a consequence slow down the robot, eventually causing a complete standstill.

Figure 5.30 shows how the central reference sensor can be used to compensate large changes in capacitance due to temperature and humidity changes in the environment. The measured capacitance of a single sensor is depicted in blue and clearly includes large changes in the base capacitance in this specific case induced by opening the door to the lab where the experiment was being conducted, thus leading to a rapid change in environmental conditions due to a sudden air exchange with the hallway outside the lab (seen around index $1500$). This significant change is also recorded by the shielded reference electrode (green line) and can thus easily be filtered from measured capacitance value by subtracting the difference between the current and base values of the reference electrode from the sensor input, the result of which is depicted in black. The actual approach of an object towards the sensor electrode (between about index $2500$ and $2800$) is not recorded by the reference electrode and thus still present in the calculated output value after the environmental compensation is applied. To further smooth the measurement value before relaying it to further processing stages a sliding average calculation can be applied in order to receive a more stable value and filter out any sudden changes caused by inevitable noise in the sensor system (depicted in gray).

While this technique proved an effective measure to eliminate gradual changes in room temperature, humidity and atmospheric pressure, the influence of heat sources in direct vicinity of the measurement circuit or electrodes, such as the internal components of the robot emitting heat beneath the mounted sensors cannot be accounted for in this fashion. The new capacitive sensor concept can account for environmental influences in the immediate surroundings of the sensors in one of two ways. The first is to

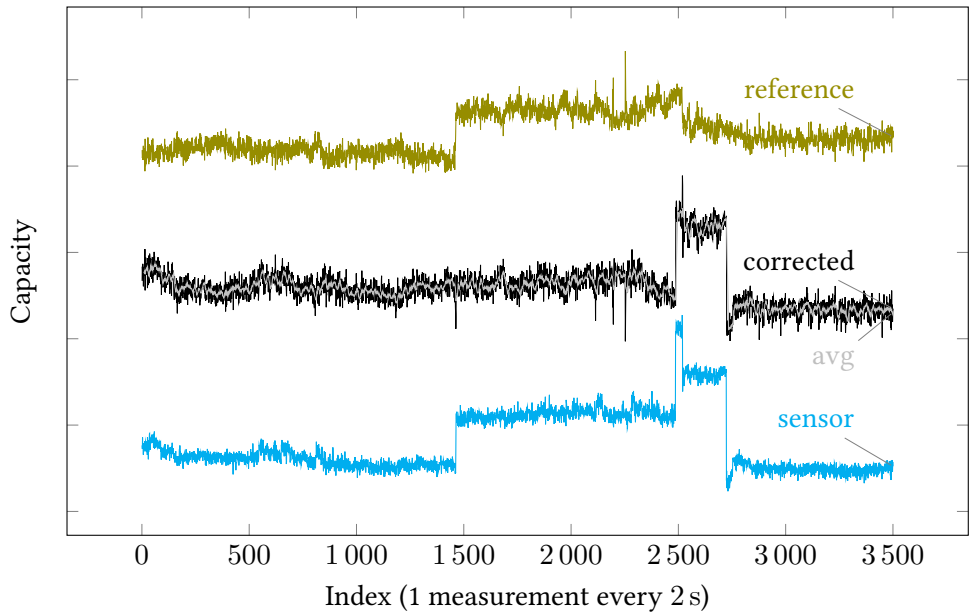



**Figure 5.30.** Results of the compensation procedure for environmental influences.

utilize the FCD2114 IC's input specifically designed for the purpose of connecting offset capacitances to it. The value of any capacitance connected to this input will automatically be subtracted from the digitally converted output value of all connected sensors, if so configured. While this may seem like the obvious solution to the problem of environmental influences in direct proximity to the sensor, it must be noted that fixed capacitors for use as components in electronic circuits will invariably react differently to external influences than single ended capacitors made from metal foil even if they have the identical nominal capacitance. This may, however, be sufficient compensation, depending on the application, and fairly easy to implement, since the reference capacitor can be directly mounted on the circuit board as a standard electronic component. The second and far more accurate compensation technique is similar to the previously introduced shielded reference electrode. By deploying entirely separate but shielded modules on the robot, the local environment influences can be compensated in the same manner as the global ones previously. If a sufficient number of free layers are available within the printed circuit board of the capacitive sensor module, such a reference electrode with its accompanying shield could even be directly integrated into the module. Such a construction would deliver the most accurate external influence data possible from within each and every module, which would then additionally account for local temperature gradients along the robot structure.

## 5.6 FEM Simulation of Capacitive Sensors

The concept of *SensorClouds* is to congregate and, if desired by the application developer, fuse all incoming sensor data streams into a unified global model of a robot's

surroundings. Integrating the one-dimensional sensor measurement provided by capacitive sensors into this three-dimensional world model directly hardly offers any productive benefit to the performance of the application. Since the physical properties measured by capacitive sensors are actually located in three-dimensional space, this information can arguably be at least partially reconstructed by other means. In pursuit of such a methodology, a simulation of the physical properties of the capacitive sensors employed was constructed using the *finite element method* (FEM). The software product used to accomplish this task was COMSOL Multiphysics [28], which is a powerful FEM simulation tool covering a wide array of physics simulation fields, from electromagnetics and structural mechanics to chemical engineering and more.

Figure 5.31 shows a visualization of the electrical field lines as well as the equipotential lines for a single-electrode configuration measuring the capacitance of the electrode against the virtual ground potential.

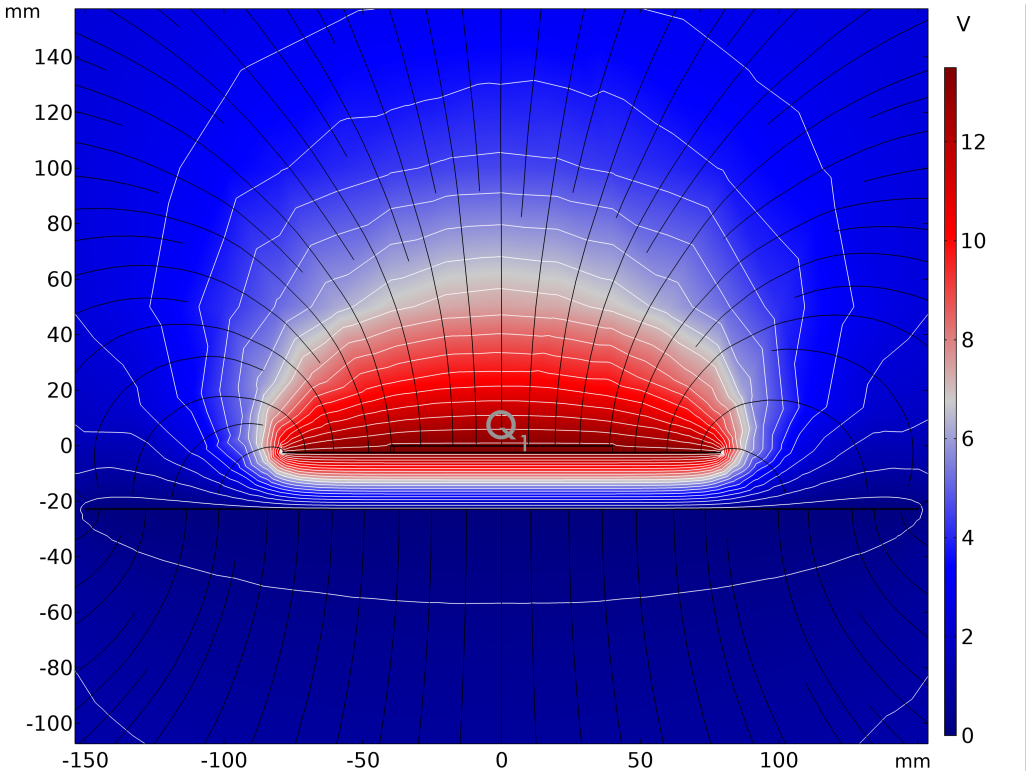



**Figure 5.31.** Electrical field for a single-electrode configuration. The charge $Q$ creates an electrical field with respect to the virtual ground potential. Also denoted are the equipotential surfaces (white lines) and the electrical field lines (in black) along which the electric force is present. The background gradient symbolizes the electric potential.

Based on the understanding of the physical characteristics of capacitive sensors and the electrical field they project (as described in Section 5.3) it was hypothesized that a partial three-dimensional reconstruction of the capacitive sensor data could be achieved by

determining the intersections of equipotential surfaces with the electrical field lines for a given electrode geometry. These intersections could then be inserted into the global world model as discrete points belonging to a point cloud for the capacitive sensor. An example of this principle can be seen in Figure 5.32. This visualization only shows a two-dimensional cross-section of the entire three-dimensional field for better clarity.

This representation is not entirely accurate with respect to a foreign object entering the electrical field, since all objects which can be measured exert their own individual influence on the electrical field And thus alter the field and equipotential lines. It is in fact, however, a valid approximation of the locations in which the same influence is exerted by the electrical field of the capacitive sensor, at the very least within the accuracy threshold capacitive sensors are able to achieve. Hence it serves as a sound model of the possible three-dimensional locations a foreign object may be located at, given a concrete capacitance value.

In the three-dimensional case, the intersections must not be only calculated between two lines, but between equipotential surfaces and the electrical field lines pervading the entire volume surrounding the measurement electrode. Due to the very nature of FEM simulations, the data gained from its computations is not represented by continuous surfaces, but rather by discretized points which are the very same produced by the automatic mesh construction employed to reduce the computational complexity of physical calculations. Consequently, the simulation results for both the equipotential surfaces and the electrical field lines do not align perfectly numerically, since the sample spaces for the simulation of various physical properties need not match exactly with each other. To overcome this limitation, a post-processing tool was developed, which simply reads in the output data of the sampled equipotential surfaces and the sampled electrical field lines and calculates the intersecting points between the two datasets. Specifically, the points of the equipotential surfaces are chosen as the determining factor, since these are always exactly on the, albeit approximated, correct surface. For each of these points, a corresponding point from the field line dataset is chosen, which must lie within the configurable tolerance range pertaining to the euclidean distance between the two sample points. These intersections are then stored in a comma-separated-value (CSV) file, within each row contains values for the x, y and z coordinates as well as the electrical potential in V as determined by the equipotential surface the intersection is located upon. The resulting file can then be read in at runtime and stored within a data structure such as a map, which associates the voltage levels with all relevant coordinates and makes it fairly simple to retrieve the entire relevant point cloud for further processing at runtime.

This method produces a total of 254.582 intersection points with a tolerance parameter of $15\,\mathrm{mm}$ across a range of 26 discrete voltage levels (every half V from $0\,\mathrm{V}$ to the maximum of $13\,\mathrm{V}$ for this particular sensor) for the simulation results of the simple circular single-electrode capacitive sensor as seen in Figure 5.31. The simulation of the electrical field lines was configured to achieve uniform density at a separating distance of $5\,\mathrm{cm}$, which means that a new field line is inserted when the previous ones diverge by more than double the configured distance. A visual representation of the resulting intersections point cloud for all voltage levels combined can be found in Figure 5.33.

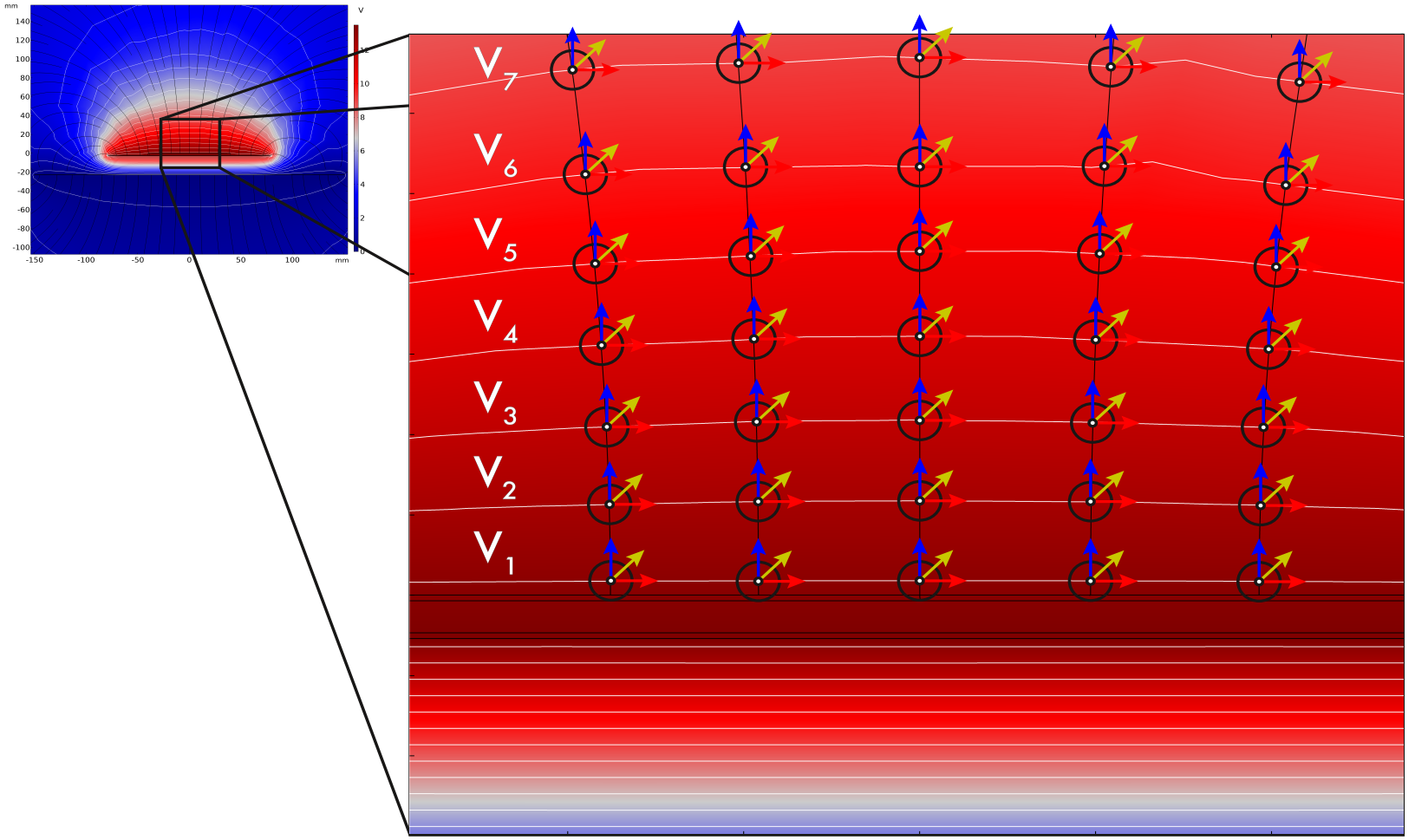



**Figure 5.32.** Enlarged section of the FEM simulation for a single-electrode configuration. The section is extracted from Figure 5.31, seen in the upper left corner. For each equipotential line (denoted by the corresponding voltage level $V_n$) a point cloud can be extracted by determining the intersections between the equipotential line and the electrical field lines.

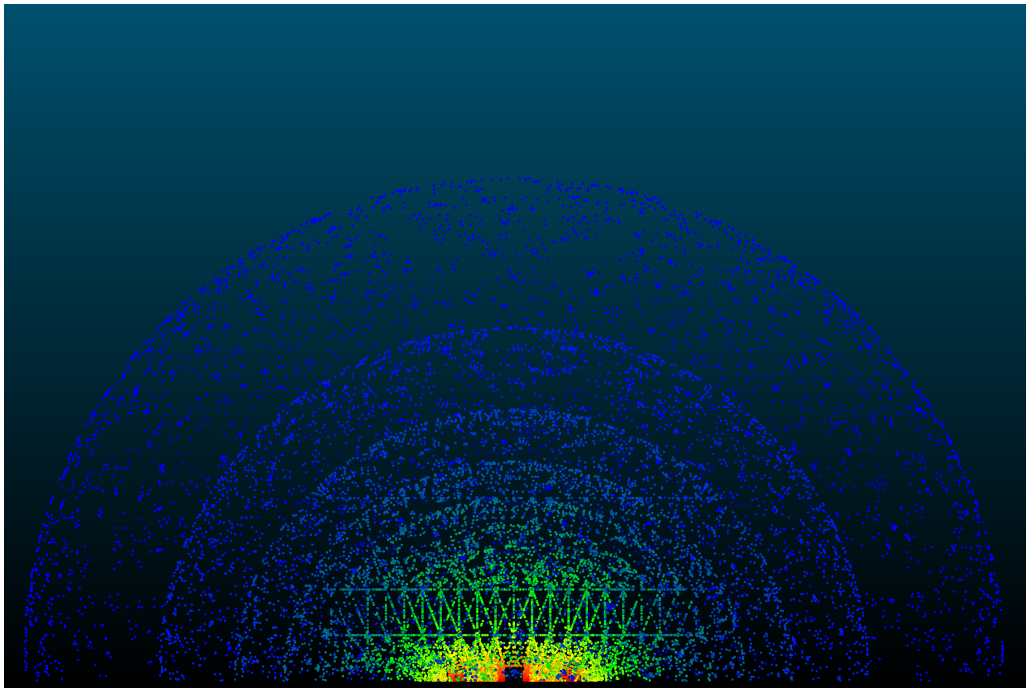


**Figure 5.33.** Output of the intersection tool for the simulated data of a simple disc-shaped single-electrode sensor.

The equipotential surfaces remain clearly visible, even in the sparsely sampled form of a point cloud. Moreover, a regular, triangular pattern can be discerned towards the bottom center of the image which is the product of the mesh generation algorithm used in the FEM simulation tool.

Whereas this specific depiction may suggest the methodology presented hitherto is superfluous, since the equipotential surfaces are evidently indistinguishable from concentric half-spheres, yet it truly coruscates once applied to electrode geometries of greater complexity. An example of this can be seen in Figure 5.34 in which the point clouds of all sensors attached to the SCHUNK LWA robot arm employed in the research project SINA (cf. Section 3.1) are depicted in their entirety.

Depending on the type of capacitive sensor and accompanying measurement circuit, the voltage levels output by the simulation as the pure physical representation of the characteristics of the electrostatic phenomena may need to be mapped to actual values output by the sensor. Should the sensor system itself output a concrete voltage value, then a plain offset may be sufficient to match the two values. If, however, the measurement circuit provides an exact capacitance measurement, a more involved matching procedure between the two scales may be required. Another approach would be to simply perform the matching on the basis of distance to the electrode surface, a step which may be incorporated into the distance estimation as described in Section 5.4.1.

Approximating the possible three-dimensional locations for a capacitance sensor in this manner and storing the results as a map of discrete sensor values with their cor-

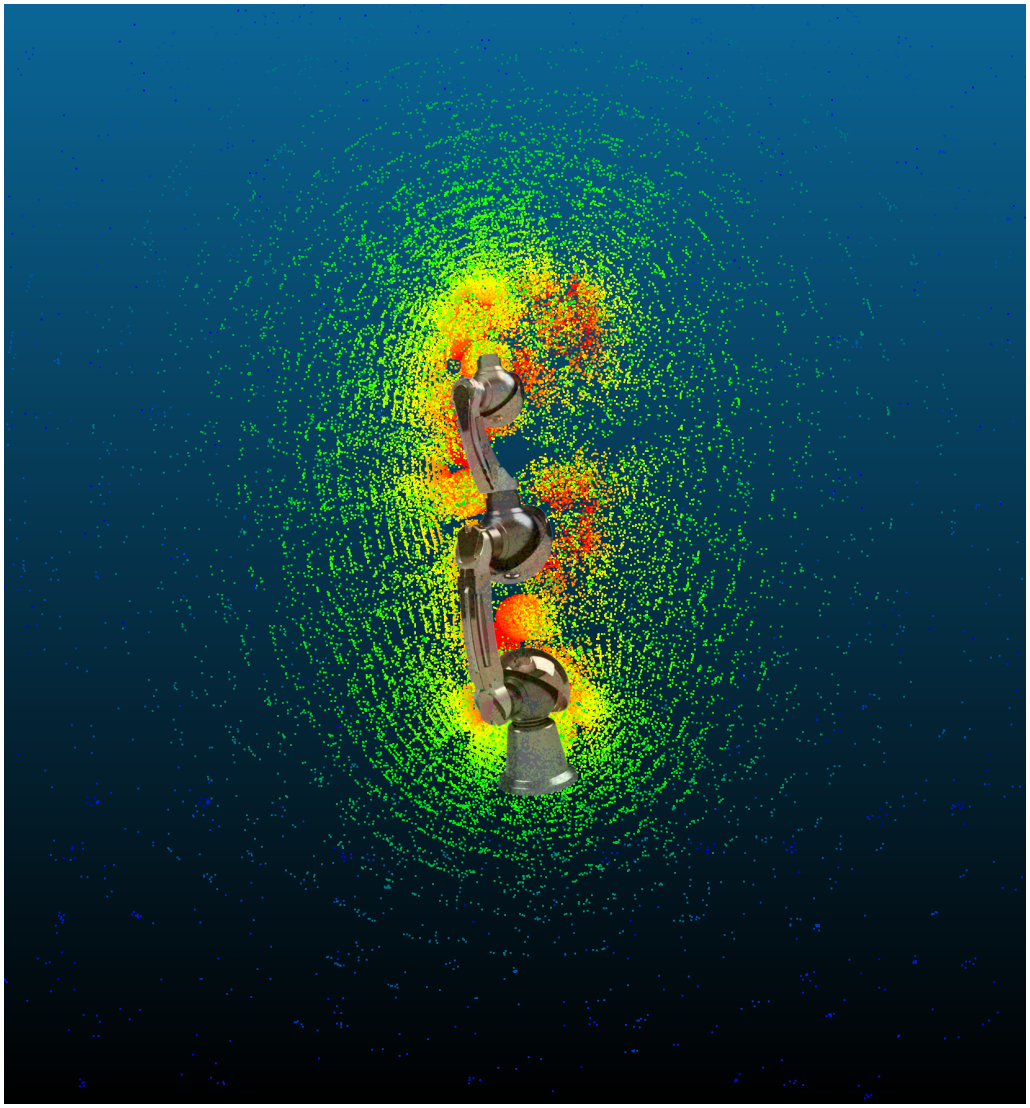


**Figure 5.34.** Point cloud generated for the capacitive sensors attached to the SCHUNK LWA robot arm (overlayed semi-transparently) used in SINA.

responding list of possible three-dimensional locations is also a very computationally inexpensive method for achieving the desired results. While far more extensive and sophisticated physics simulations of capacitive sensors could be performed and would produce results with higher accuracy, these would also require much more computing performance at runtime. One such possible enhancement would be to store the discretized values for the various equipotential surfaces, as mentioned previously, but interpolate a more accurate point cloud at runtime for every capacitance value measured. This would require matching the points contained in the point clouds of differing size for every level to each other and interpolating the required points based on their distance to the points of the higher and lower level, respectively.

## 5.7 Integration of Capacitive Sensor Data into the *SensorClouds* Model

Contrary to all three-dimensional sensor data discussed in previous chapters, which is the result of true three-dimensional sensor measurements and can thus be accurately associated with a specific location in space, the resulting point cloud derived from the FEM simulation of a capacitive sensor is merely the representation of all *possible* locations for a given measurement value. To integrate this unique type of point cloud data into the overall environment model in the sense of the *SensorClouds* architecture, a number of strategies can be employed.

The simplest of these is treating the data as if the occupancy probability of all possible points is 100%. This is the most conservative of all approaches, since most of the points will not in fact represent an obstacle in the vicinity of the robot, but this may be a desirable safety trait depending on the concrete application. A far more sophisticated approach involves probabilistic calculations and can be employed to pinpoint objects more accurately either by cross-referencing the capacitive sensor data with that of other sensors (ideally of differing modalities) or with a number of other capacitive sensors.

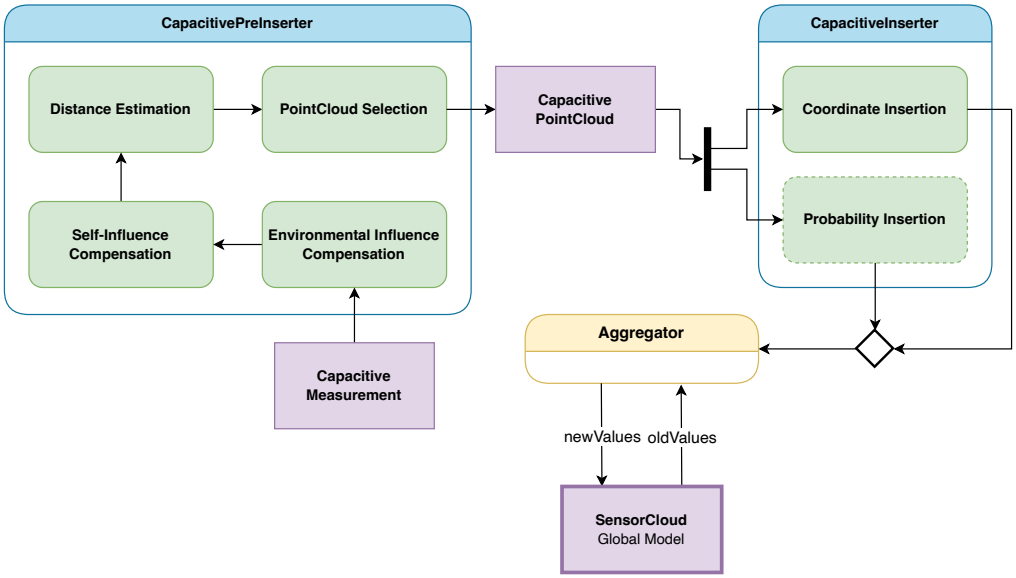



**Figure 5.35.** Activities involved in the processing of capacitive sensor data and their places within the *SensorClouds* architecture.

In order to integrate capacitive sensors into the global *SensorClouds* data model in the manner thus far described, the implementation of two *Modules* is necessary as seen in Figure 5.35. The first is an instance of *PreInserter*, which is responsible for handling all tasks related to the processing of the raw capacitance value received from the measurement circuit and subsequently selecting the pertinent point cloud from the stored dataset. Secondly, the actual *Inserter* which inserts both the coordinate data of the capacitive sensor point cloud as well as any desired probabilistic parameters into the

global three-dimensional data model must be implemented. The *PreInserter* for capacitive sensor data encompasses all the processing steps presented over the course of this chapter, beginning with the environmental influence compensation (Section 5.5.3), since this is the most basic compensation step and definitively applies globally to all capacitive sensors connected to the system. It adjusts the base capacitance level according to the drift of the measured value of a shielded reference sensor and constitutes a simple offset calculation. The self-influence compensation (Section 5.4.2) is applied thereafter and eliminates any offset in measured capacitance caused by the robot structure or the electric motors contained in its axes. Due to its non-linearity, this compensation is performed by a neural network according to the robot's current position and velocity. Subsequently, the distance to a calibrated reference object (typically the human hand) is determined by an additional neural network with the aim of equating a measured capacitance value to a real-world distance from the human entering the workspace of the robot. Finally, the capacitive *PreInserter* selects the appropriate point cloud from the dataset generated by the intersection tool on basis of the results of the FEM simulation performed for the specific electrode geometry employed (Section 5.6). This point cloud is then later inserted into the global *SensorClouds* data model by the capacitive *Inserter* which adds the coordinates from the capacitive point cloud to the appropriate voxels of the model alongside any additional probabilistic parameters desired for further processing.

The methods described in this chapter can naturally also be harnessed in order to integrate other sensor types, provided they are applicable to the modality in question. Especially the FEM simulation can, by its very nature, be utilized to simulate any number of different physical properties and, consequently, concrete sensor implementations exploiting their effects. In summary, this chapter has presented exemplarily how a one-dimensional sensor type can be integrated into a three-dimensional environmental model through several sophisticated processing techniques.

## 5.8 Related Work

In recent years, much research has focused on the use of capacitive sensor skins as a method of securing robots during interaction with humans in the workspace. Due to their relatively low cost and ability to be deployed on the robot itself, eliminating any occurrence of occlusion through other objects in the vicinity or even the robot structure, they present a viable option as safety systems in many robotic applications.

There are even some commercial applications of capacitive sensors in the form of robot skins available on the market today. One such system is the APAS assistant from BOSCH Rexroth [98], which is safety certified according to the current standards for human-robot-collaboration and can detect approaching humans safely at a distance of up to $5\,\mathrm{cm}$ and hereby prevent collisions before they occur. It must be noted, however, that the robot arm of the APAS assistant had to be limited to a motion speed of $0.5\,\mathrm{m/s}$ in order to pass certification as a safe collaborative robot system. Another commercially available system is the KR 5 SI from MRK-Systeme [59, 81] which additionally incorporates safe tactile sensors as a second modality and uses capacitive sensors to stop the robot

before direct contact occurs, if possible. It should be noted that a safety certified capacitive sensor system is currently under development by MRK-Systeme, which was also partly advanced over the course of the project SINA described in Section 3.1. Both these systems use a soft outer skin above the sensors to reduce impact forces in case a collision is not preventable. An additional commercial product is available from FOGALE robotics [45], which is capable of detecting humans in the workspace reliably at distances of up to $20\,\mathrm{cm}$, much less than the first generation of capacitive measurement circuits described in this work (cf. Section 5.5). A more recent scientific publication by FOGALE in cooperation with the university of Montpeiller, however, claims a maximum detection range of $30\,\mathrm{cm}$ for their capacitive sensor skin [73]. Moreover, their system allows the implementation of three-dimensional position sensing when multiple smaller electrodes are employed and even enables rudimentary gesture control through such configurations, which is also possible to realize with the second generation of sensor electronics presented in this work by introducing gesture recognition algorithms into the signal processing of the modular electrode configuration (cf. Section 5.5.1).

Scientific research into the use of capacitive sensors in HRI applications has been ongoing for the last few decades. One of the first instances of such a system being considered was during the course of an evaluation of various proximity sensors as collision avoidance system for manipulators during space missions. The survey conducted by Volpe and Ivlev [126] in 1994 at the Jet Propulsion Laboratory at the California Institute of Technology concluded that a capacitive sensor skin offered many advantages pertaining to coverage area and reliability. While they tried to evaluate existing commercial sensors in their survey, the capacitive sensor from Capacitec [23] was produced on special order due to the fact that the desired detection range of up to $15\,\mathrm{cm}$ was otherwise unavailable at the time. As the review on capacitive proximity sensors by Moheimani et al. [80] reveals, research in this field has seen a significant resurgence in recent years with the number of publications per annum increasing from about 20 in 2015 to around 175 in 2021, although a large portion of these concerns the development of sensor systems for applications other than robotics, such as human interface devices as control inputs for various purposes. Capacitive sensors have even been used in non-technical research areas, such as behavioral analysis, in order to non-invasively and harmlessly detect the positions, pokes or pecks of various animals in behavioral experiments [13].

A large amount of research in the field of capacitive sensors for HRI has focused on systems for robotic grippers. Here the main goal is to create near field proximity sensors capable of distinguishing between humans and the object which is to be grasped. These sensor systems are often coupled with additional sensors, detecting modalities such as force or vision in order to augment the detection of humans and objects in the environment of the robot. One such system is the one developed by Mamaev et al. [70], which builds upon their previous work of creating a custom modular capacitive and tactile sensor board capable of material detection [6] and couples it with a camera system in order to find and hand certain objects over to a human on command. In this specific publication, however, the overall system is merely capable of handling empty ceramic mugs, which are more easily differentiable from a human hand with their capacitive sensor system, since no fluid is involved. Pagoli et al. [87] developed a flexible capacitive

sensor capable of being deployed on soft robotic grippers, which is advantageous in many gripping applications, especially with previously unknown objects, since they, by their very nature, adapt their geometry to the object they are currently grasping. This enables a very flexible robotic gripper without the need for specialized fingers for each object to be gripped and even grasping objects which could not typically be grasped at all with classic grippers. Constructed from a number of different layers containing silicone, conductive strip electrodes and sheets of paper coated in conductive ink, the material cost of these proposed sensors is very low, however the manufacturing process is quite involved, since they need to be built up manually layer by layer. Such electrodes could easily be attached to the electrode interconnect ports of the measurement circuit presented in this work, thus enabling its use in grippers for soft robotics as well.

Further research into robotic applications of capacitive sensors focuses almost solely on human-robot-interaction and their use as safety or interaction systems. Lee et al. [67], for example, presented a dual-mode capacitive sensors system aimed at deployment on a robot arm or gripper using two layers of orthogonal sensing lines similar to the configuration found in capacitive touch screens. In contrast to touch screens the top layer of line electrodes is used in groups for proximity detection, while the intersections between the top and bottom sensing lines are used for tactile measurements. This is possible due to the fact that the sensing lines are all embedded in a flexible rubber casing, allowing the lines to be slightly deflected within the carrier material and thus actually contact each other when pressure is applied to the outer layer of the carrier. Schlegl et al. [104] presented a highly reactive capacitive proximity sensor system for use as collision avoidance system for human-robot-collaboration. This sensor system is moreover also capable of distinguishing various known material types in a differential electrode configuration and can switch between single-ended and differential mode during runtime in order to change from proximity sensing to near field and material detection.

Scholl et al. [105] proposed using support vector machines (SVM) in order to learn quality measures of capacitive sensor values and were thus able to reduce the positional error from previously $11.6\,\mathrm{mm}$ with their measurement circuit down to $7.4\,\mathrm{mm}$. They compared their results to the author's previous work [90] and are indeed better overall in the distance estimation error. It should, however, be noted that the average error of both approaches was misrepresented in their comparison, since they only calculated the mean error in the limited detection range of up to $200\,\mathrm{mm}$, whereas the author's previous work specified the mean error in the range up to $350\,\mathrm{mm}$. Due to the exponential nature of the correlation between distance and capacitance it is self-evident that the error increases at larger distances, as can also clearly be seen in Figure 5.13. The normalized mean error value for comparison in the range up to $200\,\mathrm{mm}$ of the author's approach comes to about $15\,\mathrm{mm}$ Moreover, Scholl et al. stated that their approach generally outperforms the author's approach, all while completely disregarding the influence of the measurement circuit employed. The performance of the second generation of capacitive measurement circuitry is expected to vastly outperform that of the first one due to its greatly improved resistance to electromagnetic interference. The resulting increase in signal-to-noise ratio in the capacitive sensor values consequently leads to a

smaller error in distance estimation when compared to the ground truth measurements obtained from an external tracking system.

Regarding the FEM simulation of sensors in general or capacitive sensors specifically, its uses in current research focus primarily on evaluating a sensor's fitness for the intended application a priori, rather than evaluating sensor input data in realtime with data from an FEM simulation. Especially in the research field developing novel capacitive sensors concepts, electrode designs or electrode materials it is quite common to find images of FEM simulations validating the desired performance characteristics of the proposed designs. Schlegl et al. [104] showed the different performance characteristics of the single ended and differential modes of their sensor system with the help of FEM simulation. Lee et al. [67] determined the optimal electrode configuration in terms of which lines to group into measurement electrodes by simulating all of them and selecting the configuration with the farthest reaching electrical field. Chuanyang et al. [47] used FEM simulation to gauge the performance of small capacitive sensors embedded in a prosthetic hand during approach and subsequent contact of an object to be manipulated. In concrete terms, the influence of an approaching object on the potential distribution in the electrical field was simulated in order to validate the suitability of the developed sensor for grasping objects with a prosthetic hand. Schöffmann et al. developed a custom FEM simulation to aid in the design of proximity sensors and application setups containing these. However, they simulate capacitive sensors with the model of an orthographic camera, which leads to greatly simplified approximations of the actual physical properties of capacitive sensors. This may be an acceptable level of precision for their demonstrated use-case of close proximity interactions with humans in assistive healthcare applications, but it does not generalize well to the physical characteristics of capacitive sensors at larger distances as has been shown in this chapter.

**Chapter Summary**. This chapter presents additional software not required in order to employ the *SensorClouds* architecture, but which offer a significant increase in convenience for certain tasks. An infrastructure for automating machine learning tasks was developed, as well as a query language for easier retrieval of specific data points from the global data model.

# 6 Additional Software

## 6.1 Infrastructure for Automated Machine Learning

Over the course of the development and repeated training of the neural networks for intelligent capacitive sensor data evaluation (see Section 5.4) it became evident that the typical workflow in such developments still comprises many laborious manual procedures. As shown in Figures 5.18 and 5.19 and described in detail in the accompanying section (cf. Section 5.4.3), two distinct applications were developed: one for recording the data required for the training of the neural network and another for the actual execution of the robot task. This distinction is largely superfluous if the supporting software architecture employed to complete such tasks enables seamless switching between data recording and execution - or at least for verification of the learning results - within a single application.

### 6.1.1 Typical Machine Learning Workflow

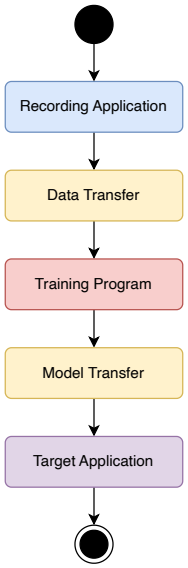



**Figure 6.1.** Typical machine learning workflow: The overview over the high-level activities which need to be performed in sequence. The subactivities are each further detailed in Figures 6.2 to 6.6.

Especially in light of steadily growing network architectures in machine learning and ever larger data sets utilized to train them [108], more powerful and decentralized computers with large storage capacities are typically employed in order to speed up the turnaround time of training results. The use of such decentralized systems, however, still involves significant manual effort in the form of data transfer, setup on the remote machine and retrieval of the trained model for local execution. A typical workflow for the process of collecting data, training the model on it and then using that to predict values during execution is depicted in Figures 6.2 to 6.6, while the high-level overview of the sequence of subactivities is shown in Figure 6.1. The person developing such a robot application with supervised machine learning must first implement and execute a

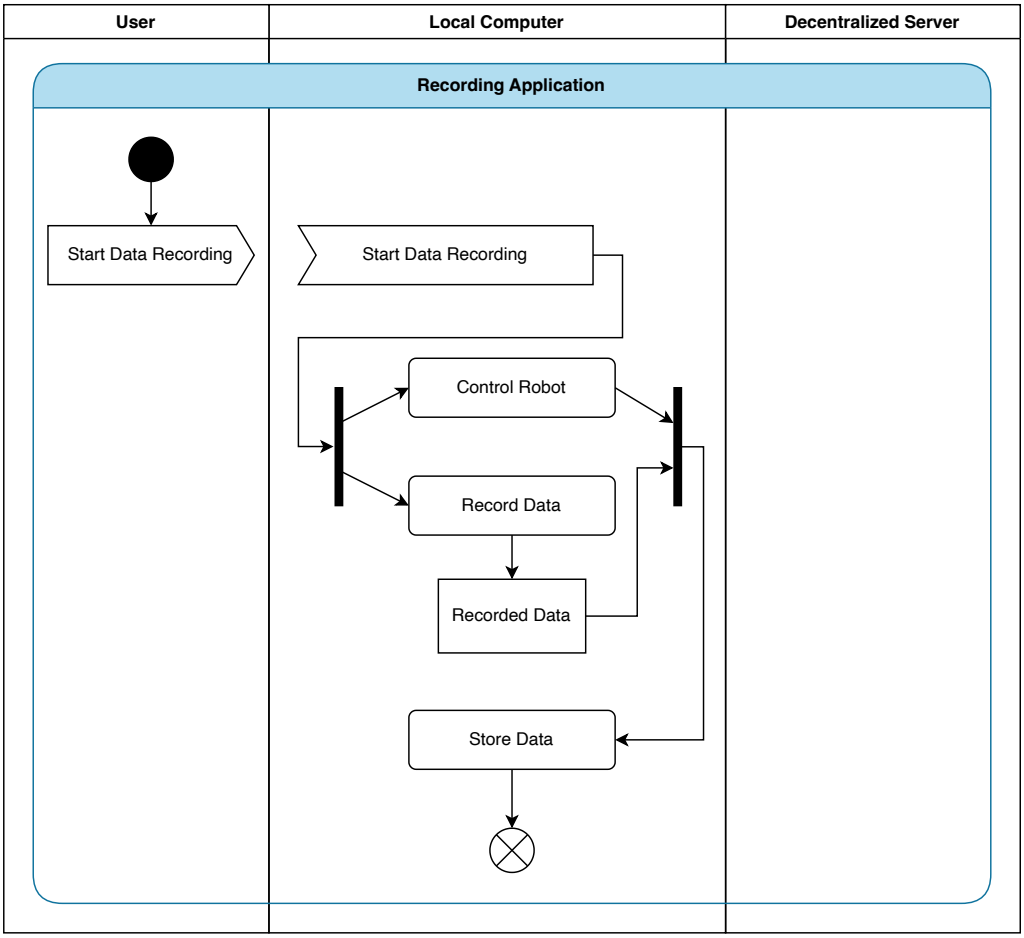



**Figure 6.2.** Typical machine learning workflow: The training data recording application.

program which records the training data, meaning all input sensor data and the ground truth values to which a correlation is to be learned later on (cf. Figure 6.2). Oftentimes the robot must be controlled in parallel with the recording process in order to obtain sensor data in multiple robot configurations and during motions. The recorded dataset must subsequently be manually transferred to the decentralized server's storage before training can begin (cf. Figure 6.3). Training is performed on the server by another custom application which is pointed at the previously stored training data and must be explicitly configured on the device by the user (cf. Figure 6.4). After training is complete, the user must again explicitly retrieve the trained model from the server and transfer it to the local machine for use in the target robot application (cf. Figure 6.5). In effect, the final target application (cf. Figure 6.6) only differs from the training application in the sense that it replaces the ground truth data with the predictions from the machine learning model, although the code overhead created by the data acquisition routines typically leads developers to maintain two separate applications for training and execution.

Particularly in supervised learning, it is completely obvious that the variables predicted by the machine learning model are, of course, the same that the ground truth delivers, as

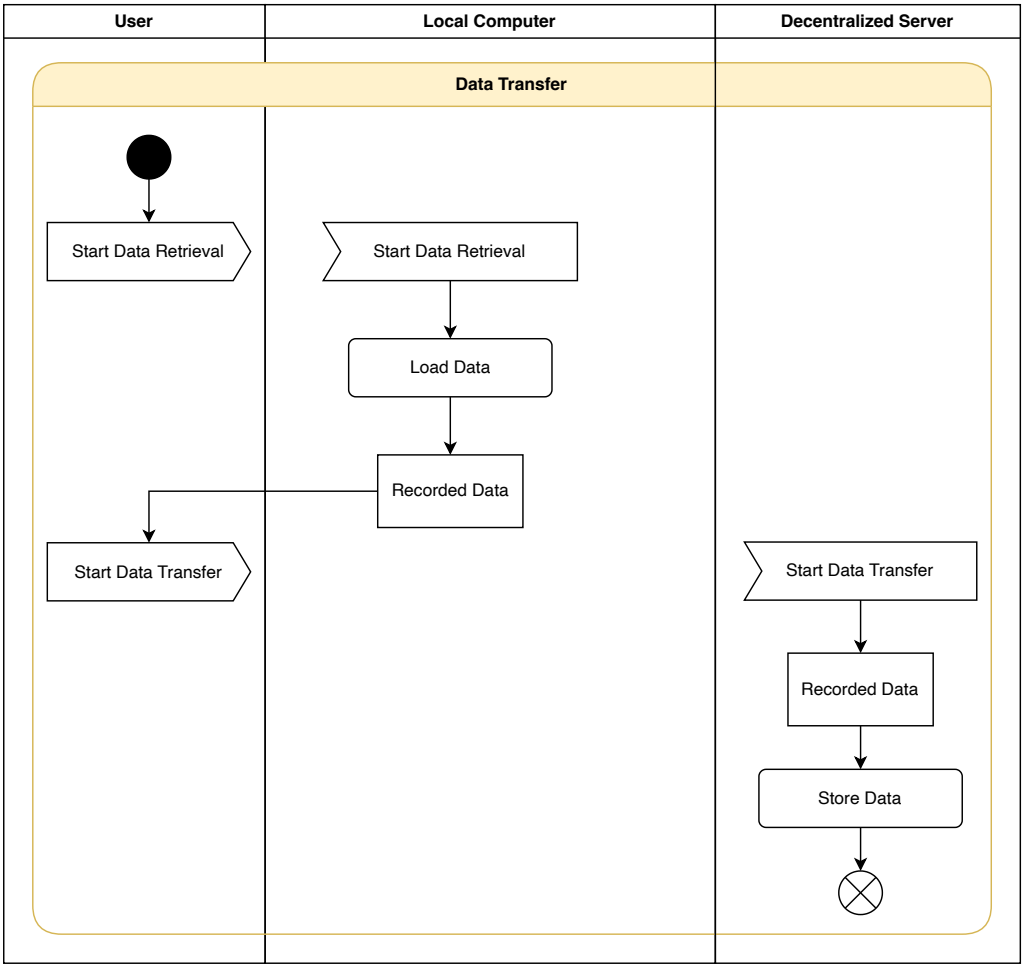



**Figure 6.3.** Typical machine learning workflow: Transfer of training data to the server.

this is the output variable which is to be learned by the system. Due to this fact, it should be possible to implement applications performing actual tasks whose runtime sensor data is recorded alongside data from a verified ground truth system and subsequently replace the data stream from the ground truth with the model output for productive deployment. All this should require only changing a parameter of the application to switch from training to execution. In order to achieve this improvement over typical machine learning workflows within *SensorClouds*, a supporting software architecture was developed in order to automate machine learning tasks as far as possible and seamlessly integrate the data acquisition and model execution into applications based on the *SensorClouds* framework. This is hereafter referred to as *SCML* (*SensorClouds* Machine Learning). It should be noted at this point that the architecture presented here is not in any way to be confused with AutoML techniques which attempt to automatically create models, preprocess input data and train the aforementioned model without intervention by a data scientist. Instead, this architecture aims at automating the logistic aspects of machine learning. AutoML techniques can, however, be utilized

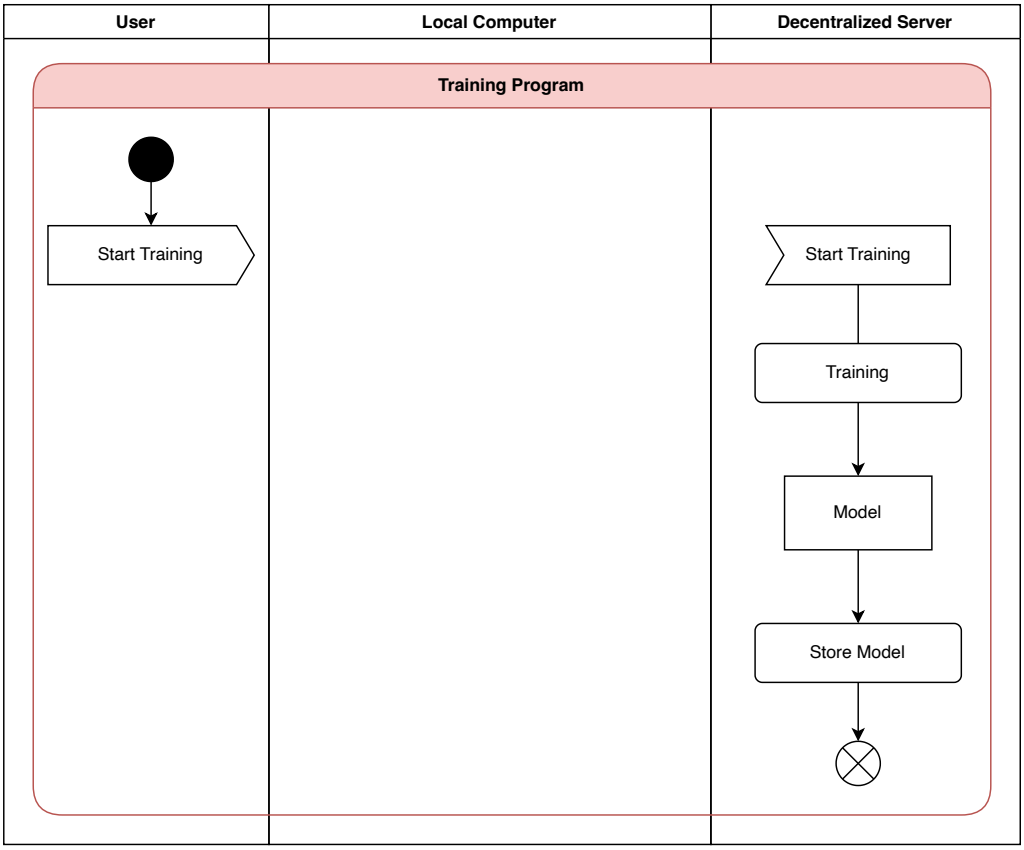



**Figure 6.4.** Typical machine learning workflow: The training process on the server resulting in a trained model in storage.

within this system to perform the actual data processing and model creation tasks in the appropriate instances.

### 6.1.2 SCML architecture

The SCML architecture is devised as a set of services running inside Docker containers [36]. Docker containers are, put most simply, isolated processes running atop the base operating system but completely sandboxed from the remaining system in terms of execution and filesystem management, which includes all dependent packages an application running within a docker container might require. An image built from such a container specification can then easily be deployed to any number of systems, as it does not rely on the concrete configuration of the target system, since all packages and even the base system (e.g., running Ubuntu 18.04 inside a Docker container on a host running Ubuntu 20.04) are completely contained within the pre-built image. Generally, the architecture is split into three main components (cf. Figure 6.7) which can all be deployed on a single device or, more expediently, onto three separate devices. The first is the main environment, which contains the actual executable used for training and execution of the robot application and attached to that, through the inclusion of

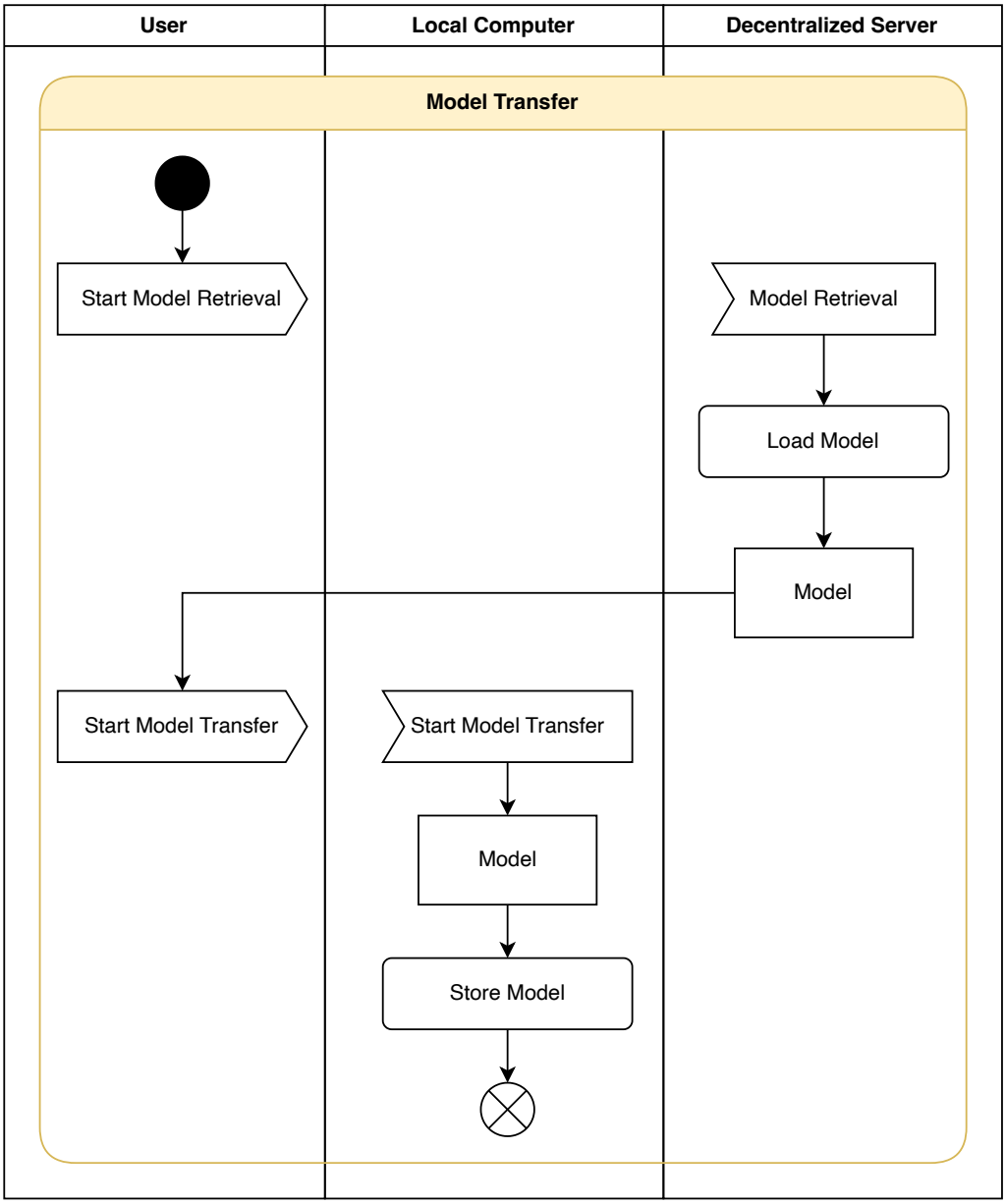



**Figure 6.5.** Typical machine learning workflow: Transfer of the trained model from the server to the local machine.

the SCML library, an instance of the Docker manager responsible for deploying the necessary containers on other target devices. Secondly, the storage device is in charge of capturing all training data as well as storing it and the resulting model after training. It is also responsible for generating various identifiers, which will be explained in further detail later. The final device is the concrete training device which can be a local machine for preliminary testing, but will most often be a central high-performance computing server best suited for training complex machine learning models. Any desired machine

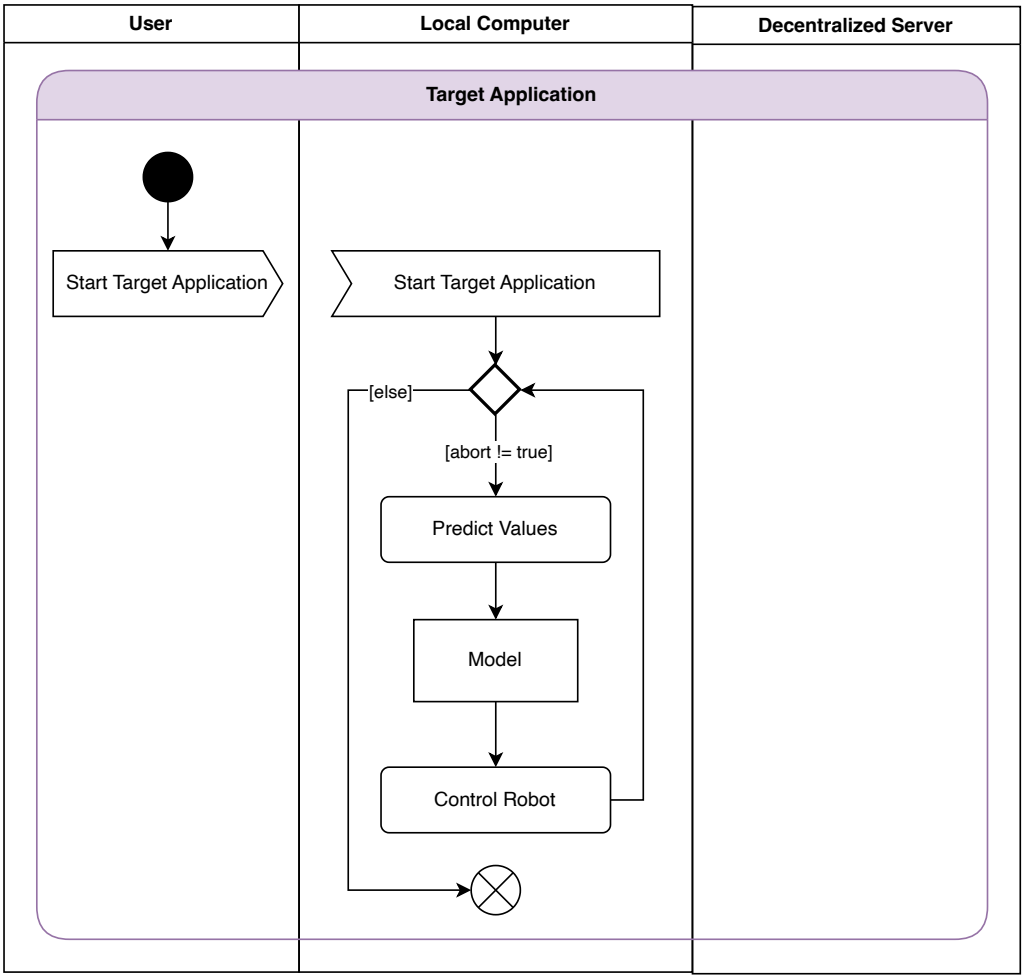



**Figure 6.6.** Typical machine learning workflow: Execution of the final target application with predictions from the trained model.

learning framework can easily be wrapped with a simple command interface which enables compatibility with SCML and subsequently be deployed as a Docker image representing a self-contained execution unit for training with that particular framework. Every software packaged in this manner can then automatically be deployed to the training device on request by an application.

The storage service also handles a second vital task: identification. All components of the SCML architecture must be assigned a unique identifier in order to be able to retrieve any data later on. Consequently, every application building upon this architecture, every dataset recorded and every machine learning model trained receive such an identifier. This enables referencing the correct data using these identifiers. Regarding only a single application and only a few datasets for training, their necessity might not be quite as obvious, since the entries would be easily distinguishable in and of themselves. Considering a busy lab environment, however, with multiple people each running multiple experiments, iteratively improving machine learning models with repeated

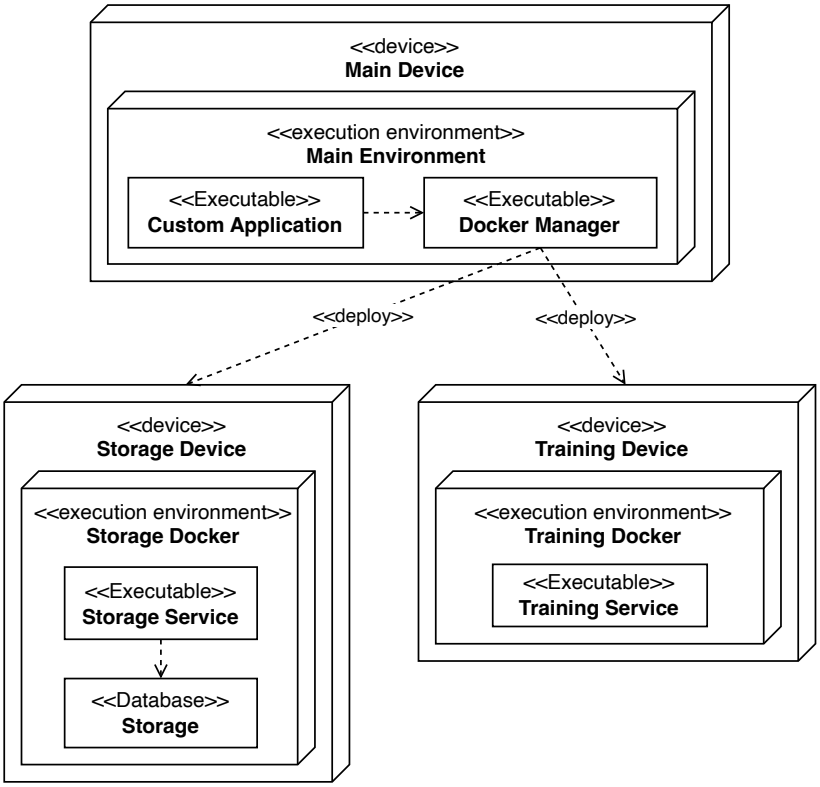



**Figure 6.7.** Deployment of software components in SCML. The architecture differentiates between three different devices, although the execution of all components involved can also be performed by a single device, if so desired.

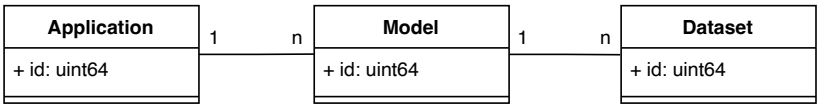



**Figure 6.8.** Identifier types in SCML. A single application can have multiple different models associated with it which in turn can each have multiple datasets associated with them.

dataset recordings, the need for a more sophisticated identification system becomes entirely apparent. A machine learning model can have multiple datasets associated with its training such as when the model is supposed to be trained with multiple different scenarios or when it is trained offline and additional data is used at runtime to perform fine-tuning through online learning. An application in turn can have multiple models associated with it, either since their architectures differ completely or only multiple parameter configurations are to be evaluated. The unique identifier of an application enables quick identification of the relevant models and datasets within the system relevant to the current task. With the identifier of a model, the training or execution of an application can seamlessly switch between different ones simply by configuring the

correct target identifier. The relationship between the various identifiers is depicted in Figure 6.8.

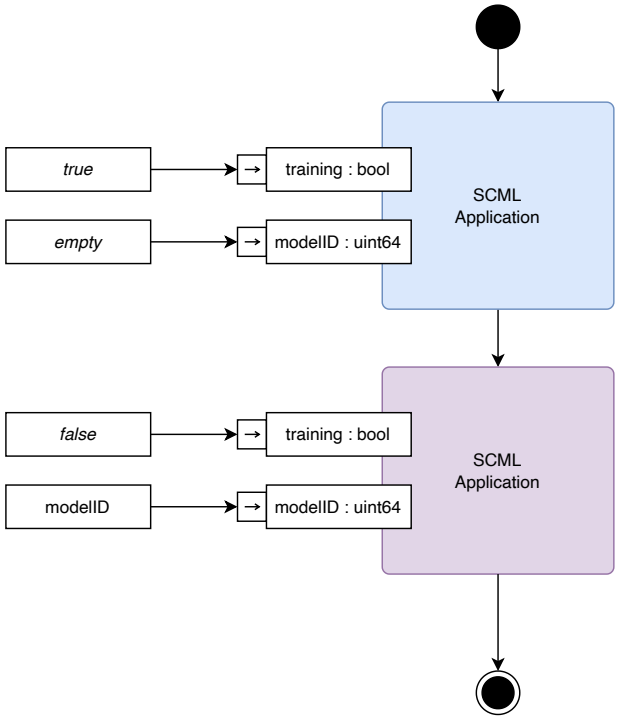



**Figure 6.9.** Workflow in SCML: The two high-level activities are performed in sequence. The subactivities are each further detailed in Figures 6.10 and 6.11.

Employing the SCML architecture results in the simplified workflow depicted in Figure 6.9. The number of manual interactions required of the user is reduced from five to two instances compared to the typical workflow (cf. Figure 6.1). Also worthy of note is that it becomes possible to reduce the number of different applications (modeled as high-level activities) from three to a single one. The remaining two steps in the typical workflow are entirely manual and must be performed correctly by the user in order for the next application to function properly, i.e., the user is responsible for copying data to the correct location on the target system and specifying the path to the required data in the configuration of the following application. The new workflow for the recording of training data is shown in Figure 6.10. With SCML all previous manual intervention steps are automated. This means that the user must only start the desired application in training data recording mode and all subsequent steps are handled by SCML. Although the main, training and storage devices are all depicted as separate instances, they can still be one and the same device, regardless. In training data recording mode, the SCML-based application handles the transfer of data to the storage device automatically and subsequently starts the user-configurable training with the machine learning framework of choice deployed in a Docker container to the training device. After training is complete, the model is then saved back to the storage device for future use by the target application (cf. Figure 6.11). Since the trained models can

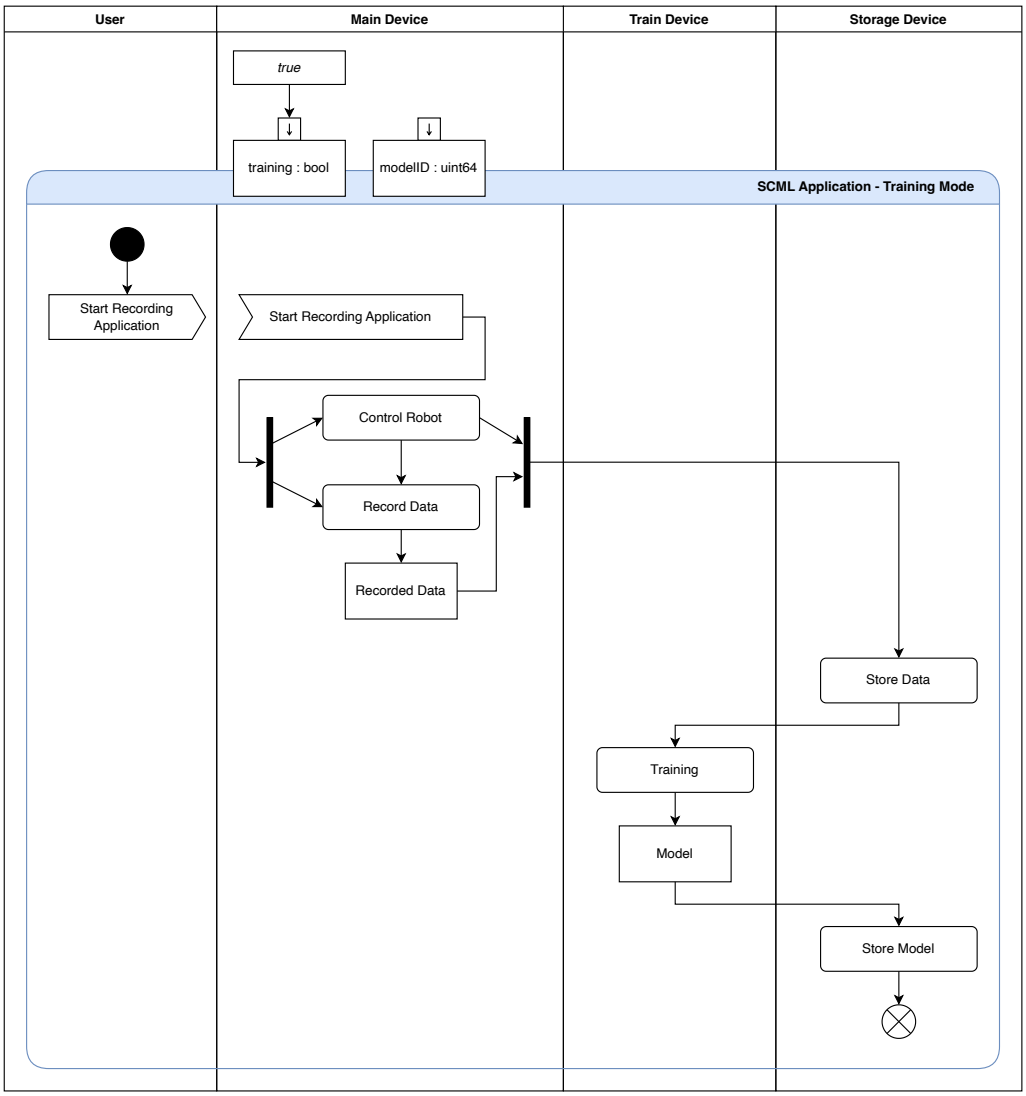



**Figure 6.10.** Workflow in SCML: The training application, which handles data recording, transfer and learning automatically and internally. The input parameters are `true` for training and empty for the `modelID`, since no model exists yet.

easily be referenced via their unique identifiers, the desired model can automatically be retrieved from the storage device during execution of the target application and predictions based on the incoming sensor values can be made for use in the control of the robot.

### 6.1.3 Related Work

While the deployment of Docker containers for machine learning tasks is a common practice, particularly in cloud solutions, they typically still have to be handled manually, meaning the data must be transferred explicitly by the user, and specific training applica-

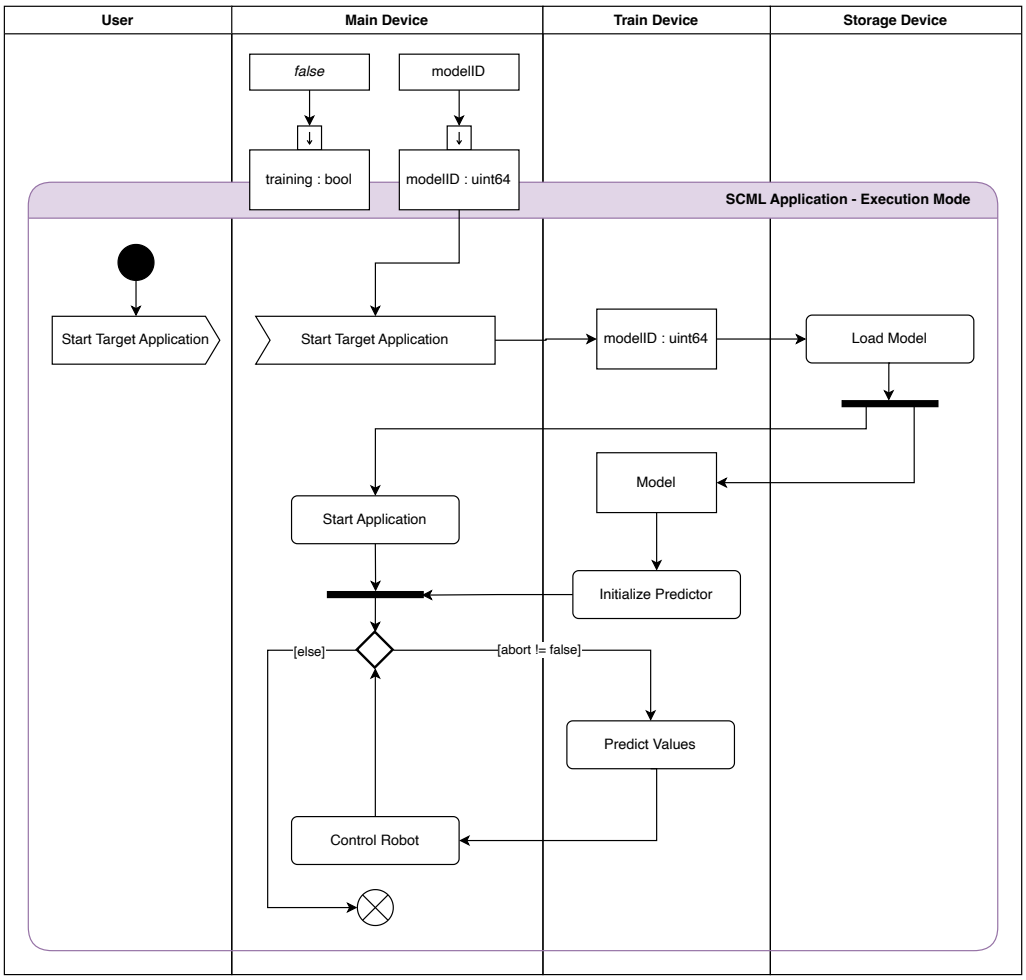



**Figure 6.11.** Workflow in SCML: The target robot application, which handles model retrieval and prediction automatically and internally. The input parameters are `false` for training and the desired `modelID` to be used during execution for the prediction.

tions must be developed and deployed to the cloud server, executed there and the trained model subsequently retrieved from it and loaded into another application for inference at runtime. Some similar frameworks for automating machine learning infrastructure tasks have been developed at the same time, none of them, however, offer both a degree of versatility and level of automation comparable to SCML. AzureML from Microsoft [76], for example, allows training of machine learning models exclusively on Microsoft's own cloud servers. Local datasets can be used, yet typically data storage must be handled by another service and is not handled by the machine learning framework itself. Another option is MLflow [79], an open source platform for automating and collaborating on machine learning workflows. Like AzureML, MLflow can automate machine learning tasks and is compatible with various typical machine learning frameworks, such as SciKit, PyTorch or TensorFlow, yet it also does not handle data storage directly and instead relies on external data providers and manual connections with these. TFX is

an "end-to-end platform for deploying production ML pipelines" [48] and aimed at production deployment of trained machine learning models. TensorFlow was developed by Google and TFX is only compatible with TensorFlow and no other machine learning frameworks. TFX also relies on external storage providers for storing all training and evaluation data. In contrast to these solutions, SCML offers maximum extensibility through a simple machine learning framework interface, can be deployed on any device and handles all data storage internally. In this manner, multiple experiment applications can be associated with multiple datasets and models, and the framework keeps track of all these instances for future reference.

The exemplary implementation of the previously described architecture for automated machine learning infrastructure tasks was performed by Martin Siehler during the course of his work as a student assistant.

## 6.2 Query Language for Data Selection and Retrieval (ScQL)

Since not all processing algorithms operate on single data points and require data from surrounding points or voxels, *Modules* can define *DataSets* as the input and output parameters. These may contain not only one data point but multiple. The selection and retrieval of these data points may not be performed by each module itself since this would require direct access to the global model's storage by every module and consequently would break the design-by-contract paradigm. To overcome this problem, a data retrieval mechanism was developed, which can either be employed through manual definition of selection rules and filters, or via a structured query language for voxel-based datasets, named the *SensorClouds Query Language (ScQL)*.

The Structured Query Language (SQL) as it is known nowadays was originally developed by Chamberlin and Boyce at the IBM Research Library and published in 1974 [24]. At that time it was still known under the name SEQUEL, which stands for "Structured English Query Language". The goal of this query language was to simplify the retrieval of data from large databases and make them accessible for programmers and non-programmers alike by making the queries more easily understandable with common English terms for typical operations on the tables of databases. Previously, queries typically had to be formulated in terms of relational algebra which stemmed from the underlying principle of emerging database systems of the time, which were starting to be developed as relational databases. This move away from the typical hierarchical or network based databanks of the time was instigated by an article published by Edgar Codd in 1970 [27], who also worked at the IBM Research Lab in San Jose at the time of publishing. Codd proposed a relational database system and demonstrated how the entirety of operations on such a system could be greatly simplified compared to the previously existing database systems, since these required special access programs for all operations, which had to be rewritten every time the structure of the database changed. SQL is standardized today in ISO/IEC 9075-1:2023, which is split into eleven parts covering various areas of the query language from the very basics up to the management of external data sources [4].

Since then, many languages akin to SQL have been developed for various applications, all serving as a simple interface for retrieving data from complex systems. RDF [128], for example, is a graph-based data model for representing semantic information on websites in a subject-predicate-object syntax. These graphs can become very large and thus too unwieldy to search manually for the relevant connections and semantic relationships sought by the requester. To this end, a number of query languages have been developed with the goal of providing a far simpler method of retrieving the desired information from large knowledge graphs stored in the RDF format, which have Since all been consolidated into the SPARQL Protocol and RDF Query Language (SPARQL) [127]. SPARQL has been the official recommendation for RDF query languages of the W3C consortium since 2008. SPARQL allows querying RDF graphs much in the same manner as relational databases, but it also allows e.g., extracting valid RDF subgraphs from larger ones with the `CONSTRUCT` keyword, which is not found in standard SQL syntax and thus an application specific extension of an otherwise similar syntax. Egenhofer

[38] proposed a spatial querying language (Spatial SQL) in 1994 for data retrieval from Geographic Information Systems (GIS), which are spatially enabled relational databases, meaning they can process and even index data by locations and timestamps expressed in earth's spacetime, specifically geographic coordinates on earth (latitude, longitude and elevation) and a date and time expressed in a standard timezone. Spatial SQL is a domain-specific extension to standard SQL in that it allows the formulation of queries and the definition of procedures based on geometric information. Geographic Information Systems are nowadays commonly available from large database vendors as extensions to their regular database products and as such also include extensions to the standard SQL syntax in order to enable spatial queries and procedures to be performed. Among others, Oracle [85] and Microsoft [77] offer such products.

### 6.2.1 Architecture

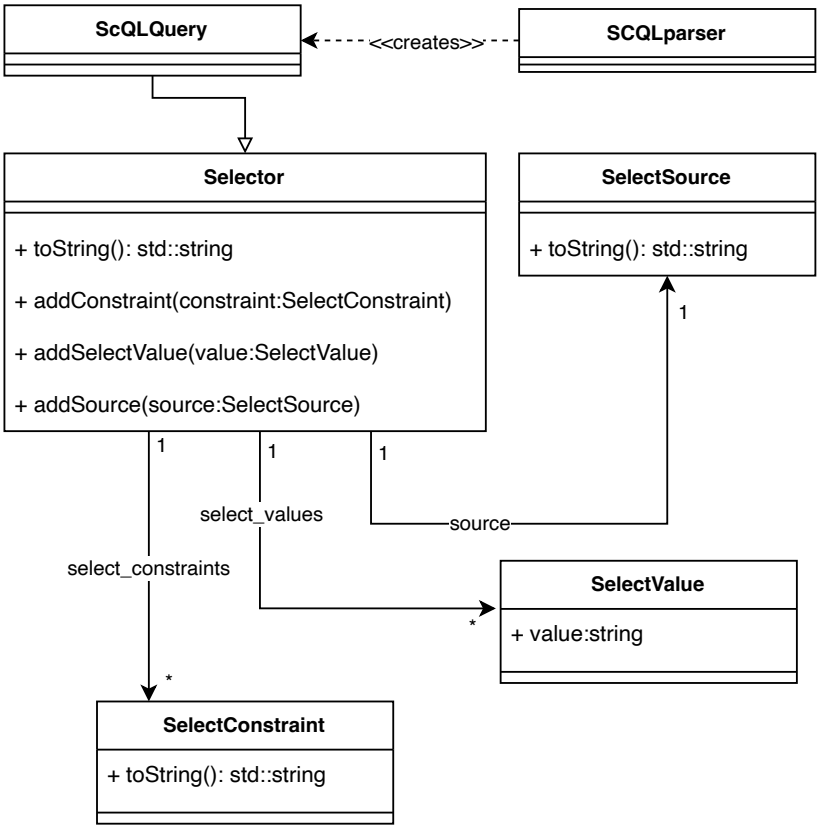



**Figure 6.12.** General overview over the ScQL architecture.

The general architecture overview of the ScQL querying system can be seen in Figure 6.12. The central concept is the query object (*ScQLQuery*), which in turn is derived from the *Selector* class. This could theoretically be extended to include various other statement types for future implementations of ScQL queries in different areas of the *SensorClouds* architecture, such as `INSERT` queries similar to classic SQL for simple *Inserters* requiring

no more than a simple write operation mapping between input and output data. A selection statement is comprised of a *SelectSource* that specifies from which area the voxels should be selected, a number of *SelectConstraints* which determine the filter rules by which voxels are to be selected and finally the list of *SelectValues* which determine the data fields of interest to be included in the output of the selection process.

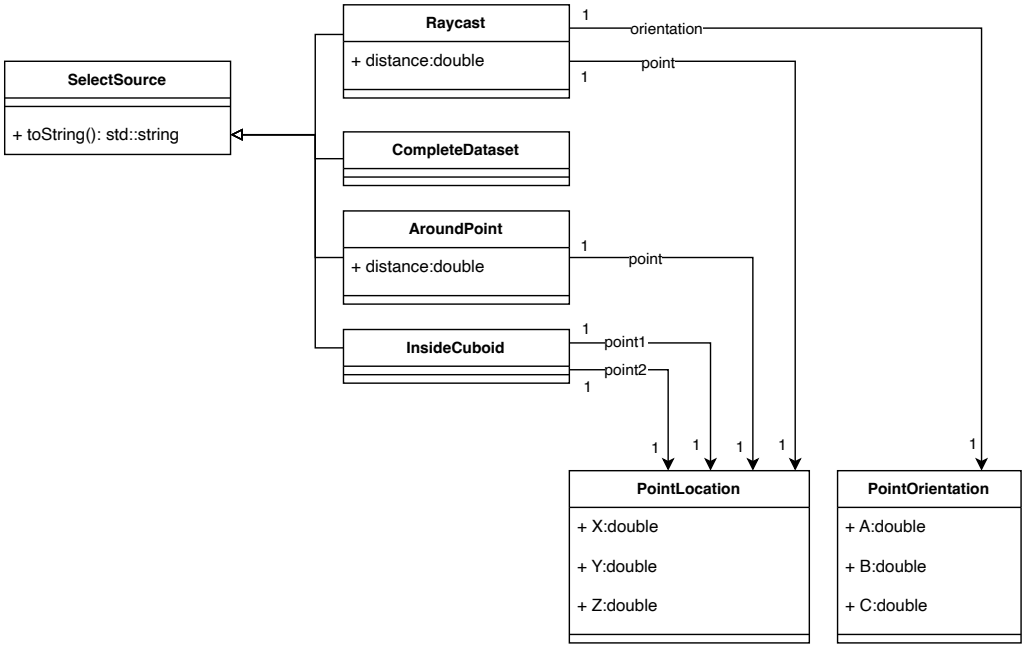



**Figure 6.13.** Definition of the geometric constraint *SelectSource* in ScQL.

*SelectSource* specifies from which source the data shall be selected in terms of geometric constraints (cf. Figure 6.13). Either the complete dataset (*CompleteDataset*) shall be searched for voxels matching the filter criteria of the query or the source can be further constrained to include only voxels that fall within a specific geometric definition. This can be around a certain point in space (*AroundPoint*) within a specific radius or inside a defined cuboid (*InsideCuboid*), meaning a number of voxels contained between two given points. Finally, the selection can even be constrained to the path along a raycast through the voxel map (*RayCast*) matching all encountered voxels or only those matching the search criteria. The specific geometric parameters of all selection operators are defined by the classes *PointLocation* and *PointOrientation*, the former being the most commonly required concept. This concept of various geometric definitions for the region of interest is furthermore easily extensible with custom definitions of geometric constraints imposed on the voxel selection process.

*SelectConstraints* define the actual filter criteria which must be matched against the data point contained in each individual voxel. These can, for example, be simple boolean operators, which logically combine other constraints with each other, such as logical `AND` and `OR` operators, as seen in Figure 6.14. Since these are modeled as derivative of the class *SelectConstraint* themselves and have two associated constraints for the

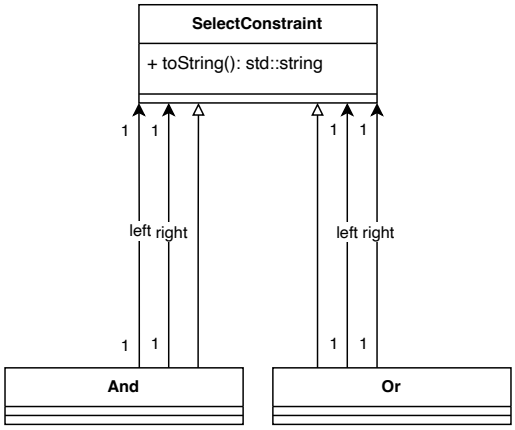



**Figure 6.14.** Definition of boolean operations in ScQL.

left-hand and right-hand sides of the logical operation, respectively, an arbitrary depth of recursive logical operations is possible to define.

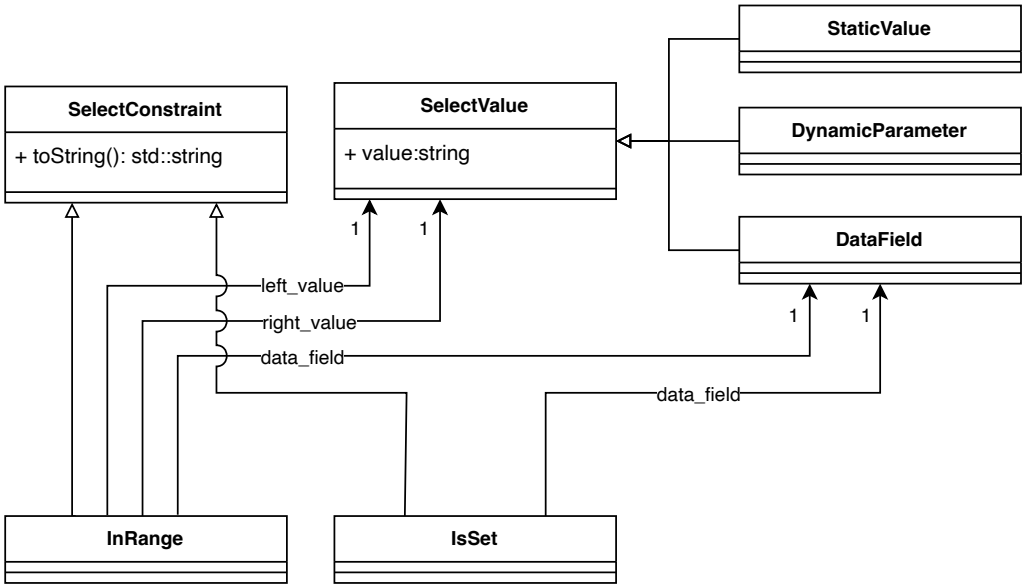



**Figure 6.15.** Definition of range and existence operations in ScQL.

In order to filter data points by the existence of a certain data field within it, the operator *IsSet* (cf. Figure 6.15) can be employed in order to select, e.g., all voxels containing an entry for temperature stemming from a thermal camera image. The *DataField* class is used to specify the correct data field for selection, so in the previous example this would hold a reference to the field `temp`. Since the *IsSet* operator only checks for existence of a data field, but completely ignores its value, it is only suitable for the selection of data points used in other processing steps thereafter. The *InRange* operator, however, works by comparing the value of a data field with two other values, which can either be

specified as *DynamicParameter* or as a *StaticValue*, also depicted in Figure 6.15. While a static value is merely a numeric value to be compared against, a dynamic parameter is in fact a value that is passed to the selection algorithm by the *Module* in which it is defined and thus is only assigned a specific value at runtime in accordance with the processing algorithm of the concrete *Module*.

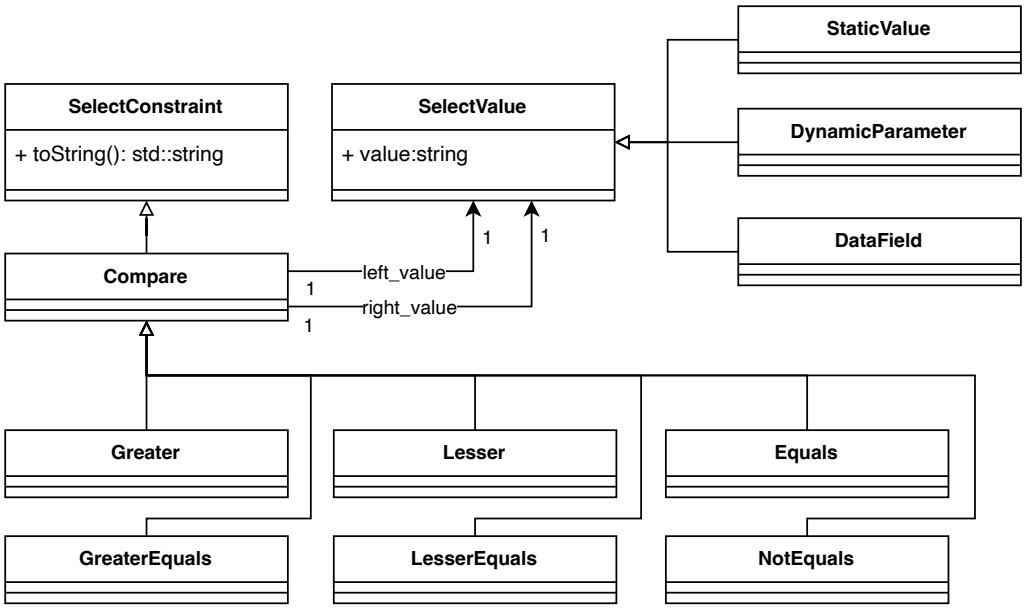



**Figure 6.16.** Definition of comparator operations in ScQL.

Finally, the *Compare* operator allows comparisons between two different values and encompasses the typical options for value comparison, as depicted in Figure 6.16. The values again can be any one of the previously described *SelectValue* derivatives, so e.g., a *PointValue* can be compared to a static value or a dynamic parameter in the selection process.

Since the geometric constraints are far more likely to filter out the largest portion of the entire data for the final selection process, these are applied first. This saves checking every voxel for their compliance with the specified constraints only to then throw away most of the matching voxels again because they are not even in the region of interest. This, in consequence, also saves computational cost, since logical and value constraints typically require more calculations.

### 6.2.2 Query Language and Parsing

The actual query language ScQL is based upon the class architecture described in the previous section. Each of the described concepts in the architecture equates to a part of the query language. As an example, a query to select all voxels containing temperature information from the entire data model would be formulated as follows:

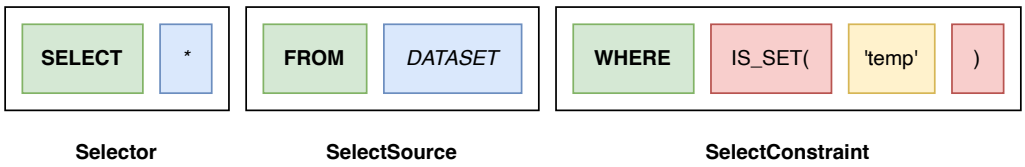



**Figure 6.17.** Simple ScQL `SELECT` statement for the retrieval of all voxels containing temperature information.

The built-in star ($*$) operator (operators denoted in blue) stands for the inclusion of all data fields from the selected voxel in the output dataset, as is the case in classic SQL statements. Together with the `SELECT` keyword (language keywords denoted in green) this constitutes the equivalent to the *Selector* class of the architecture and the basis for the remaining statement. The *SelectSource* portion determines the geometric location at which the selection process shall be performed and is executed before any further filters, since this leads to a greatly improved performance, as mentioned in the previous section. Similar to classic SQL, but not representing a specific table in a relational database, the `FROM` keyword is used to specify the desired geometric location. `DATASET` is also a built-in protected keyword which denotes that the entire dataset is to be searched for results matching the constraints. Finally, any further required constraints can be specified in the `WHERE` clause, as is also the case for filter rules in classic SQL. As the example's goal is to retrieve all voxels containing temperature information, yet not inspect the temperature value for any further conditions, the `IS_SET` *SelectConstraint* (operations denoted in red) is sufficient to achieve the intended result, namely selecting only those voxels in which a value for `'temp'` has been defined (parameters denoted in yellow).

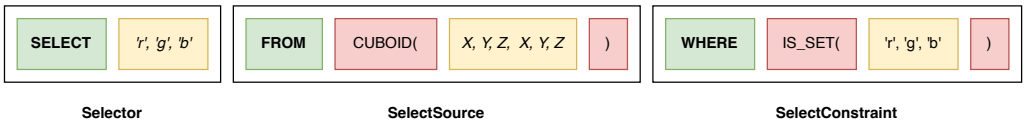



**Figure 6.18.** Simple ScQL `SELECT` statement for the retrieval of all voxels containing color information within a set cuboid.

Adding a more restrictive geometric constraint and filtering by the existence of more than one data field leads to a query such as the one depicted in Figure 6.18. Here, the `CUBOID` operator is employed to limit the search for matching voxels to a specific area circumscribed by the two points described by cartesian coordinates ($X$, $Y$ and $Z$), which are specified subsequently in the operator call. The `IS_SET` operator can also be called with multiple data fields as parameter list and the operator then checks each voxel for previous write operations to *all* of the fields specified.

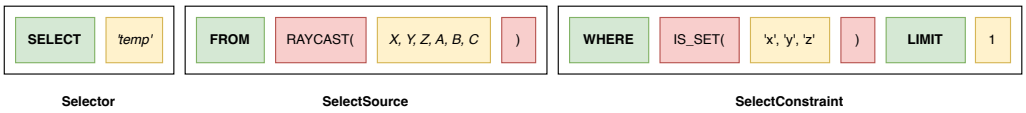



**Figure 6.19.** Advanced ScQL `SELECT` statement for the retrieval of the first voxel containing coordinates along a raycast.

A more complex `SELECT` operation can be found in Figure 6.19. Here the goal is to select the temperature data field from the first voxels along a raycast. The source of the cast ray is defined by $X$, $Y$ and $Z$ coordinates, while the orientation the ray should be cast in is specified by the orientation defined by the three angles $A$, $B$ and $C$, which are the respective rotations around the cartesian axes determined by the origin point of the raycast. Both these definitions are passed to the `RAYCAST` operation which constitutes the geometric constraint in this particular query. The remaining constraints limit the selection process to only include a voxel which has had coordinate information inserted into it (`IS_SET` operator) and limits the result set to a single voxel, meaning the first voxel matching the search criteria along the path of the raycast is to be returned. This is in fact the ScQL statement required for the insertion of two-dimensional camera data into the three-dimensional global data model. To accomplish this, a virtual ray from the spatial location of the camera sensor is projected along the axis determined by the projection matrix retrieved by the intrinsic calibration of the camera. The first point in three-dimensional space, which contains coordinate information from a depth sensor also included in the application setup, can usually be assumed to also be the one seen by the color camera, and thus the color information can be added to the matching voxel. This does not account for all circumstances, such as that the views of the two sensors could be at oblique angles to each other and therefore the camera cannot necessarily directly see the object which the depth camera is detecting, but this could be addressed by including the normals in all sensor data insertion. The normal of a sensor measurement is the addition of a vector describing which point in space the measurement was taken from. With this additional information, the *SelectConstraint* could be amended to only accept voxels whose normal vector has a small enough angular distance to the incoming ray that the camera sensor being projected can technically even see the point seen by the depth sensor.

The implementation of the parser for this language was performed with the help of ANTLRv4 [88], which is a parser generator for languages defined in simple grammar files. ANTLR generates small programs in a programming language of choice, which are in turn capable of parsing the language specified in the grammar file. Each parsed segment is passed to the appropriate responsible class by means of a visiting operation, a process which is modeled after the commonly known Visitor pattern. Parameters are passed along to the visiting operation. In this fashion, the parser can instigate the construction of the required hierarchy of object instances of the classes defined in the ScQL architecture (cf. Section 6.2.1) and these in turn can subsequently execute the necessary operations at runtime. Data selection operations in *SensorClouds* can consequently either be defined manually via *Selector* objects with all required constraints

added to it, or a simple ScQL statement. Since these selection operations are included in the definitions of *Modules* which are JIT compiled for increased efficiency, they too benefit from compiler optimizations and the known values of many variables at runtime of the base program.

The implementation of the ScQL query language and the accompanying parser were performed by Moritz Hofer within the scope of his internship project.

**Chapter Summary**. This chapter gives an in-depth look at a selection of key details from the *SensorClouds* reference implementation. Performance, flexibility and ease of use as well as the balance between them were the driving factors in all implementation decisions.

# 7 Reference Implementation

## 7.1 Compute Unified Device Architecture (CUDA)

By the year 2009, GPUs had started overtaking CPUs in terms of raw computational power. However, prior to the introduction of arithmetic logic units (ALUs) capable of executing general purpose computing instructions, GPUs were highly application specific devices designed specifically for the hardware acceleration of calculations for the display of three-dimensional objects on the computer screen. The first step towards general purpose applications was the introduction of shader programs, which allowed the manipulation of visual effects calculated by the GPU for the first time. These gave far more creative freedom to developers, since they were no longer entirely bound by the processing formulae implemented in hardware on the GPU, yet were still very limited in terms of what could be calculated in which step of the rendering pipeline. The main purpose of GPUs was at this point still the rendering of three-dimensional objects represented as large collections of polygons. Shaders are small programs which enable the modification of how these polygons are finally rendered to the display and allow developers to implement various shading (hence the name), lighting, and particle effects. (cf. [32, p. 11ff])

CUDA is a proprietary programming interface from NVIDIA and enables the implementation of programs for general purpose computing on GPU hardware. It was first introduced in 2007 and in first iteration only a C programming interface was available. Since then, C++ has been added to the officially supported languages and many more third-party wrapper libraries have been developed, such as for Python, Java or MATLAB. Specifically for C++ it must be noted, however, that merely a selection of the typically heavily utilized standard libraries are supported. Code using extended features of the C++ standard library cannot be used in the same files as CUDA code, since the compiler does not recognize them. Yet the main drawback of choosing CUDA is that programs developed with it are only executable on GPUs produced by NVIDIA. An alternative which is conceptually similar and follows the same basic principles with respect to the programming and memory models is OpenCL (Open Computing Language). It was originally developed by Apple and first released two years after CUDA in 2009. Its development has since been handed over to the Khronos Group, a non-profit open source organization also responsible for the development of the OpenGL (Open Graphics Language), which is a graphics API that e.g., handles the compilation and execution of shaders in the graphics pipeline of GPUs. While OpenCL has similar capabilities to CUDA, it cannot offer the same level of platform specific optimization, since it has to maintain compatibility with hardware from multiple vendors. A clear advantage of OpenCL is that it is not limited to GPUs, but can also be executed on multi-threaded CPUs and mobile ARM processors alike. Many vendors offer standard-compliant OpenCL implementations for their hardware architecture, including AMD, ARM, Qualcomm, Samsung and even NVIDIA themselves [118]. This multi-platform compatibility, however, results in some operations and setup procedures which involve more manual work from developers and do not quite perform as well as their CUDA counterparts without significant architecture dependent optimizations to the OpenCL version of the program [40], which consequently contradicts true portability of the developed applications. (cf. [32, p. 13ff])

### 7.1.1 Programming Model

Computing architectures are typically classified by Flynn's Taxonomy first proposed in 1966 [43] and later extended in 1972 [44] to describe them in terms of the relationship between execution units or processors and the data streams they execute instructions on. Since then the classification has been extended with subclasses, e.g., by Duncan in 1990 [37], in order to differentiate various emerging architectures further. Duncan's taxonomy does not add to the four basic classes originally proposed by Flynn, but rather subdivides them further. The four basic classes introduced in Flynn's basic taxonomy from 1966 are as follows: (cf. [43][44][37])

- **Single instruction stream, single data stream (SISD)** Single-Core CPUs, which were standard in consumer computers up until the early 2000s, are an example of an SISD architecture in this classification system. A single-core processor processes instructions serially, meaning one after the other and one at a time. It executes this instruction on a single element from the input data stream. This means that at any given point in time a single processing unit executes a single instruction on a single data point.
- **Single instruction stream, multiple data streams (SIMD)** The same instruction is executed by an arbitrary number of processors on individual data streams. This is useful for the parallelization of workloads which have to process a large amount of data in exactly the same way, such as when iterating over arrays or performing matrix calculations. GPUs are typically classified as having an SIMD architecture. NVIDIA however makes a further distinction and classifies the architecture employed in their GPUs as "single instruction, multiple threads" (SIMT). The difference between SIMD and SIMT is that the processors are not independent of one another, but execute all instructions in lockstep, meaning that all processors can always only execute the same instruction on their respective data points and not be at different points in the program. In consequence, this means that all processors complete their work at the same time, but whenever a branching condition is encountered, all processors not entering the branch due to the evaluated condition on their individual data must halt execution until those that do enter have exited the branch again (branch divergence).
- **Multiple instruction streams, single data stream (MISD)** Multiple processors execute instructions on the same input data stream. While very few implementations of this architecture have ever existed, it is the predominant construction method for safety-critical computer systems, such as flight computers or industrial functional safety equipment. NASA, for example, constructed the main computer cores of the space shuttle after this principle [111]. While the calculations performed by the processors independently are largely identical, they differ in the safety critical portion where the results are checked by the respective other processors for consistency with their own results. Hence, they operate on different instruction streams while actually performing the identical calculations on a single input data stream.

- **Multiple instruction streams, multiple data streams (MIMD)** Modern multicore CPUs are constructed after the MIMD architecture in the sense that multiple independent threads are executed on multiple (potentially) disparate data streams. Any multiprocessor system from integrated application-specific ICs (ASICs) for use in modern smartphones up to the highest levels of high-performance computing clusters, also referred to colloquially as "supercomputers", employ the MIMD architecture in various specific implementations of the basic principle (cf. [96, p. 14]).

The two architectures with multiple input data streams (SIMD, MIMD) can operate completely unimpeded as long as the input data to the individual processors is actually entirely independent from each other. Once data dependencies exist between data written by one processor or thread and read back by another one or simultaneous write access is possible, more complex programming paradigms have to be employed in order to synchronize memory accesses across the entire system. While the ultimate goal is always to develop truly parallel and lock-free programs, multiple execution units operating on the same dataset will always have to be synchronized in one form or another. The simplest form is to lock data completely, e.g., with a mutex, as long as one thread is working on it. All other threads trying to access the same data point simultaneously typically have to enter a spin-lock and wait for the other thread to complete its work until they can continue with their execution. While this synchronization method is data focused, it is also possible to synchronize different threads by controlling their execution. In condition synchronization, a thread sleeps until another thread wakes it back up again once a predefined condition has been met. Unless a monitoring thread is periodically checking the condition and can control the wakeup sequence of the other thread, the sleeping thread never wakes up, since it cannot monitor the condition by itself. Barrier synchronization in contrast defines a point in the program execution at which all threads must wait until all other threads have also arrived at it. This is also the method used extensively in CUDA's execution control and is, for example, the mechanism by which branch divergent code is synchronized. The end of a condition is hereby implicitly defined as a barrier synchronization point which must be reached by all threads before execution can continue. (cf. [96, p. 152ff])

An overview of program execution in SIMT architecture of CUDA enabled GPUs is given in Figure 7.1. Kernels, which are the programs to be executed by the GPU (Device) in parallel, are always started by a program running on the CPU (Host), which is thereafter free to execute any other serial code which may be needed while the GPU computes the results. The host program is also responsible for any data transfers to and from the device. The execution hierarchy differentiates four levels of thread groups, the lowest of which is a warp. Warps are units of 32 threads which are executed in lockstep. The previously mentioned synchronization upon branch divergence is also only applicable on the level of warps, since multiple warps are executed independently of each other. Above that, threads are grouped into blocks which can, in the current versions of CUDA hardware, comprise a maximum of 1024 threads. This is due to the fact that a block must always fit onto a single streaming multiprocessor (SM), which is NVIDIA's denomination for a processing unit in its GPUs, and cannot be distributed across multiple SMs. A single

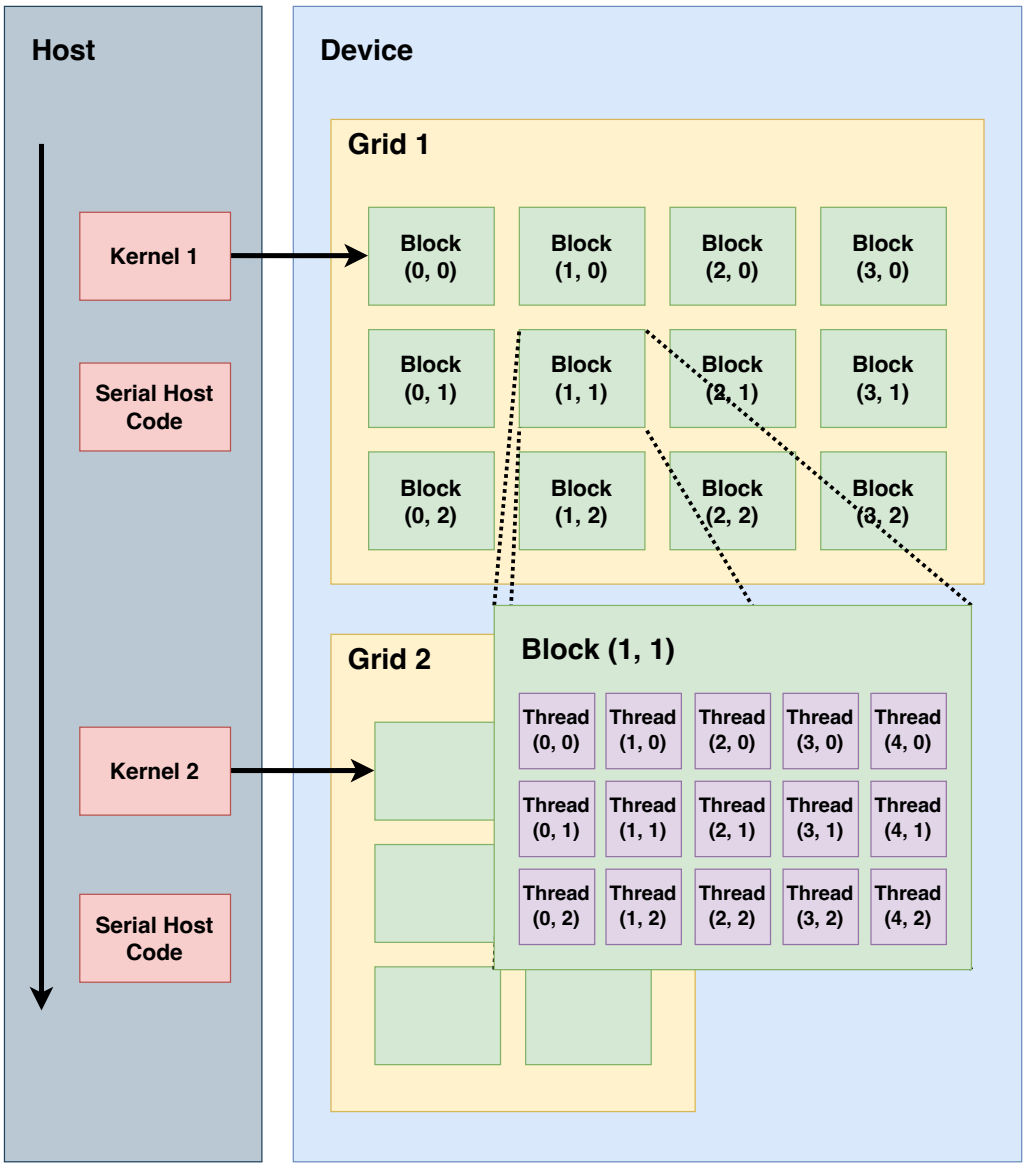



**Figure 7.1.** The execution model of CUDA with the program timeline between CPU and GPU programs.

streaming multiprocessor executes a single instruction on up to 1024 data elements per instruction cycle. Modern NVIDIA consumer GPUs such as the RTX 4090 have up to 126 streaming multiprocessors on a single die, meaning they can theoretically execute 129 024 threads simultaneously. However, other constraints such as memory speed and transfer latency when copying data from the main working memory to the GPU can impact the maximum achievable throughput. (cf. [32, p. 69ff])

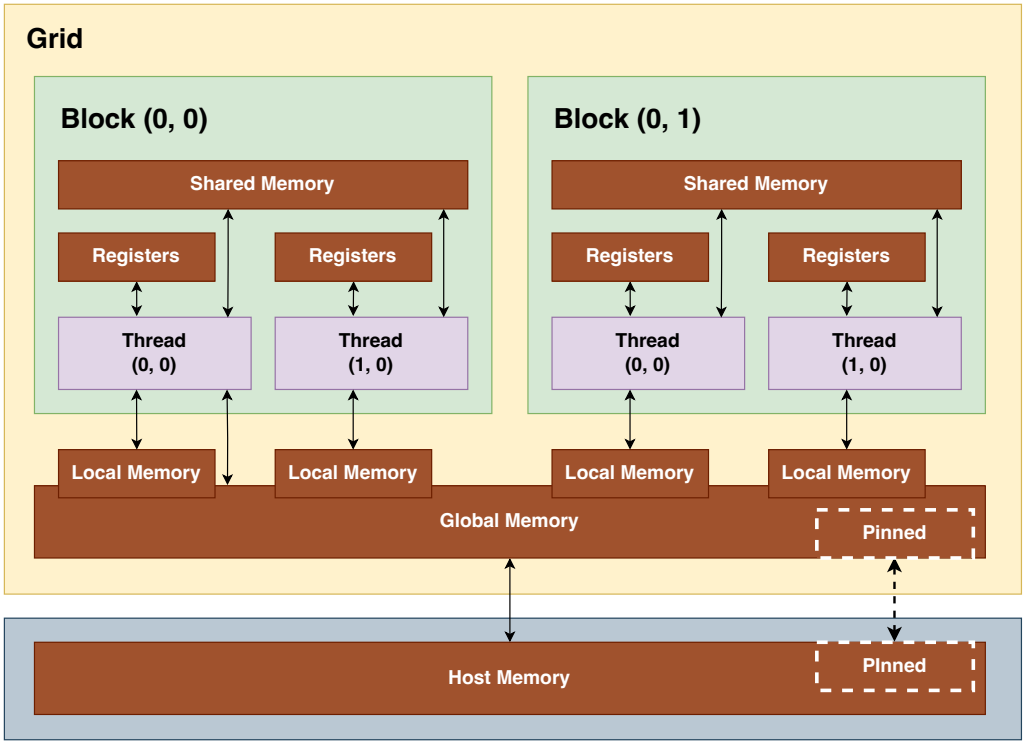



**Figure 7.2.** The memory model in CUDA with the various memory types accessible by each thread.

### 7.1.2 Memory Model

The memory model in CUDA, depicted in Figure 7.2, differentiates three main types of memory on the device, which can be accessed by all threads, however some memory types have certain access restrictions imposed on them, and they all differ in access speeds. The fastest memory type is, as in any processor architecture, the bank of registers available to every processor. Each streaming multiprocessor of the GPU has a fixed number of registers available to all kernels running on the SM at the same time. At compile time, the number of registers required by a CUDA kernel is calculated by the compiler and fixed thereafter. The number of registers each thread can occupy for its calculations is thus variable, but the SM can only schedule as many blocks as their register requirements allow to execute simultaneously. Shared memory is effectively an on-chip cache which is explicitly controlled by the program code, rather than the GPU hardware itself. Its effective bandwidth is roughly five times slower than direct register access [41, p. 111]. All threads within a block (maximum of 1024 threads) can access the same portion of shared memory which makes it ideal for passing data between threads with low latency. Should multiple blocks be executed at the same time on one SM, then access to shared memory is restricted to the portion reserved for the block the accessing thread is a part of. Shared memory access across blocks is prohibited. (cf. [32, p. 107ff])

Global memory is the largest but slowest region of memory available to CUDA kernels and is mainly used for loading datasets from the CPU onto the graphics card for later processing and for storing large amounts of result data before passing it back to the CPU for further processing. Global memory is about ten times slower than shared memory [41, p. 111]. Individual threads are assigned local memory, which is actually stored in global memory and consequently suffers from the same high latency problem. It does, however, have some special properties that differ from regular global memory. Local memory is only used in certain cases, also referred to as "register-spilling", in which a thread, for example, allocates an array at runtime. Since the size of the array was not known at compile-time and thus no registers could be reserved to hold the array, it must be stored elsewhere. The term local is used, since this portion of memory is only accessible by the thread it belongs to and cannot be accesssed by other threads or even other kernels. (cf. [32, p. 107ff])

A significant performance boost in memory transfers between host and device in CUDA can be achieved by *pinning* certain regions of host memory. Memory pages which have not been pinned can be swapped out to disk, meaning that it does not actually reside in the working memory of the CPU anymore, but has been written to the main storage device. This typically occurs when the working memory is full and the page in question has not been accessed for a longer period of time. Pinning a memory page prevents this mechanism from being applied to the locked region ensuring that it is always directly available in the working memory. Additionally, this means the GPU is capable of directly accessing this memory without intervention from the CPU using a functionality called *direct memory access* (DMA), which is part of the PCI-Express standard GPUs are connected to the CPU with. Consequently, DMA provides a significant speedup in memory transfer latency, since no data has to pass through the CPU and bind computing resources there, but rather is directly requested by and transferred to the GPU via its main bus system. (cf. [32, p. 334ff])

A significant portion of every early CUDA application was concerned with explicit memory management. To ease this burden on developers, the concept of *unified memory* was introduced. Mapped memory is a function of the CUDE framework which eliminates the need for explicit transfers of data by the developer in the correct direction before the data is intended to be processed either by the device or the host. With unified memory, one need only reserve the required amount of memory once, while the CUDA framework takes over managing the allocation of memory on both sides as well as the transfer between the two when required. The concept is based on a unified virtual address space, meaning the GPU can attempt to access any memory address across the entire system, including those of the CPU. However, since the memory is not actually physically available to the GPU, every access first results in a page fault, which prompts the CUDA framework to migrate the requested page from CPU to GPU memory. This, of course, introduces significant performance overhead, which can however be mitigated somewhat by manually prefetching the soon to be accessed pages but cannot completely match the performance of explicit copy operations. (cf. [102])

## 7.2 Programming Interface for Module Developers

### 7.2.1 Module Implementation

When implementing a new type of module a developer must create two separate classes. The first is the internal framework representation of the module which defines all metadata for the module, and the second is the actual processing kernel which contains the implementation of the atomic operation to be performed when the module is executed. An example of the metadata class can be seen in Listing 7.1, in this case a simple *Inserter*, which is derived from the class *Inserter*, in turn a derived class of *Module*. This type hierarchy serves to constrain the possible types of input and output definitions since not all are applicable to every type of module.

```
1  class SCXYZInserter : public SCInserter {
2
3      SCXYZInserter() : SCInserter(
4          "SCXYZInserter",
5          pipeline::SCGPUMethod(&_inserter, "process",
               ↪ &SCXYZInserterImpl::process),
6          SCInserterInfo(
7              ArgumentList(
8                  Field{sensorclouds::tags::x, DataType::FLOAT},
9                  Field{sensorclouds::tags::y, DataType::FLOAT},
10                 Field{sensorclouds::tags::z, DataType::FLOAT}
11             ),
12             {},
13             ArgumentList(
14                 Field{sensorclouds::tags::x, DataType::FLOAT},
15                 Field{sensorclouds::tags::y, DataType::FLOAT},
16                 Field{sensorclouds::tags::z, DataType::FLOAT}
17             )
18         ),
19         { "SCXYZInserterImpl.h" }
20     ) {}
21
22     SCXYZInserterImpl _inserter;
23 };
```

**Listing 7.1.** Definition of an *Inserter*.

The constructor of the newly created *Inserter* must call the base constructor with the required configuration parameters. The first parameter is the name which the *Module* should be referenced by. It merely serves identification purposes in logs and debugging

messages and can therefore be chosen at will by the module developer. Secondly, the module must specify the concrete *Method* which contains its processing function. Unless otherwise created previously, this object can be created directly inline in the constructor call of the *Module* class. The third parameter required to create a new *Module* is the meta-information on the read and written fields as detailed in Section 4.2.3. In this example only *Data Input* (lines 7-11) and *Model Output* (lines 13-17) are defined, since no data must be read from the global model by this *Inserter*. Hence, the *Model Input* parameter is left empty (line 12). Finally, any header files which need to be included in the JIT compilation process in order for all relevant classes and implementations to be found, can be specified as a list of strings. A *Method* in turn, requires three parameters in order to be constructed: a reference to the object previously instantiated upon which the *Method* shall be executed at kernel runtime, its name as a string and a pointer to the class member function to be executed, since the extraction of parameter information within C++ itself requires this in order to be able to resolve the method at application compile-time.

```
1  class SCXYZInserterImpl : public
       ↪ memory::SCMemoryItemizable<SCXYZInserterImpl> {
2
3      void process(float xr, float yr, float zr, float* x, float* y,
           ↪ float* z) {
4          if (!isnan(xr) && !isnan(yr) && !isnan(zr)) {
5              *x = xr;
6              *y = yr;
7              *z = zr;
8          }
9      }
10
11 };
```

**Listing 7.2.** Implementation of an *Inserter* processing kernel.

Listing 7.2 shows the implementation of the actual processing kernel for the previously defined simple coordinate *Inserter*. The *SCXYZInserterImpl* is derived from the class *SCMemoryItemizable*, which is merely a convenience wrapper around *MemoryItem* which allows the automatic construction of an accompanying memory item during construction of another object simply by deriving from it (line 1). First of all, this eliminates the need to explicitly call the constructor of the managing memory item, but it also is technically necessary, since the object being passed to the memory item is still being constructed and therefore incomplete, and hence cannot be passed as a parameter yet. The member function of this class must now only define its parameters according to the specification provided in the *Inserter* as seen above (line 3), lest errors occur later during kernel compile-time and the static analysis as described in Section 4.2.6. The

very simple processing function in this example merely checks if the received input data fields are actually valid numbers (line 4) and if so passes them on directly to the output variables (lines 5-7).

### 7.2.2 Argument Definition

In the preceding examples, the simplest form of argument definition has consistently been used, which is the specification of individual fields of the input and output data. This is, however, only the case when no special *Selector* is required and standard iteration over all input data points and direct matching to the coordinate space of the global model for output data are desired. Before presenting the details on the usage of the *DataSet* concept, a brief explanation of the simpler list of single fields, seen in Listing 7.3, is given.

```
1  // Argument list definition (for x only)
2  ArgumentList(
3      // A single data field from the data point to be referenced
4      Field{
5          // Tags identify data fields
6          sensorclouds::tags::x,
7          // Internal type representation
8          DataType::FLOAT
9      }
10 )
```

**Listing 7.3.** Definition of a simple *ArgumentList* with one data field.

Data fields are always identified by tags in *SensorClouds* (line 6). While data fields in the incoming point cloud messages from ROS are identified entirely by name with strings, frequent string comparisons are a significant detriment to performance of applications, since the two strings to be compared both have to be iterated over completely (at least until the end of the shorter one) for every single comparison operation. The expected data type of this field is specified with the internal type enumeration introduced in Section 4.2.2. Each field required for the *Module* is defined in this manner and added to the *ArgumentList* (line 1) for the appropriate entry in the meta data definition (*Data Input*, *Model Input* or *Model Output*).

A more complex example of an *ArgumentList* can be found in Listing 7.4 in which an ScQL statement is utilized to select the desired insertion point in the global model through a raycast operation (lines 5-6). Of note here is that the *ScQLquery* object (line 4) is merely an extension of the *DataSet* object, with the only addition being that it accepts a string parameter for the query upon construction and can be queried by the processing function at kernel runtime. All remaining behavior of the class is identical

```
1  ArgumentList(
2
3      // ScQLquery is a SCDataSet with a query string
4      ScQLquery{
5          "SELECT 'temp' FROM RAYCAST($1, $2, $3, $4, $5, $6) " +
6              "WHERE IS_SET('r', 'g', 'b')",
7          // Data fields specified separately for type information
8          Field{
9              sensorclouds::tags::temp,
10             DataType::UINT32
11         }
12     }
13
14 )
```

**Listing 7.4.** Definition of an *ArgumentList* with an *ScQLquery*.

to that of the *DataSet*. The placeholders for dynamic variables are denoted by a dollar symbol ('$') followed by the ordinal number of the argument. The arguments must then be specified in that particular order when the query is later executed. The individual fields expected as a return to the query operation must still be explicitly specified (lines 8-12) in order for *SensorClouds* to receive the required type information and be able to determine compatibility between the requesting *Module* and the data contained in the global model (cf. Section 4.2.6).

The accompanying *Inserter* processing kernel is depicted in Listing 7.5. The argument definition must now specify an *ScQLquery* object with the required data fields as template parameters (line 1). Since this is an *Inserter* for projecting a thermal image into three-dimensional space and annotating pre-existing spatial data with temperature information, it is built upon a raytracing operation. The resulting projection matrix of the required calibration is stored in the device object. With this information, the device can calculate the outgoing vector for the raycast from the current pixel coordinated on the camera sensor (line 4). The elements of the resulting vector are then passed to the query operation as the dynamic parameters (line 6). Finally, the temperature value of the current pixel can then be written to the appropriate place in the global data model (line 8).

```
1  void process(uint32_t temp_in, uint16_t x_in, uint16_t y_in,
       ↪ ScQLquery<Field<sensorclouds::tags::types::temp, uint32_t>>
       ↪ query) {
2      if (!isnan(temp)) {
3          // Projection matrix is stored in device object
4          auto vec = device.project(x_in, y_in);
5          // Prepared query is executed with dynamic parameters from
               ↪ kernel
6          auto result = query.select(vec.x, vec.y, vec.z, vec.a, vec.b,
               ↪ vec.c);
7          // Write operation
8          result[sensorclouds::tags::temp] = temp_in;
9      }
10 }
```

**Listing 7.5.** *Inserter* querying the model and inserting thermal information at the site of the result.

## 7.3 Details of the *SensorClouds* Reference Implementation

The following section details some select aspects of the *SensorClouds* reference implementation, which aim at fulfilling the non-functional requirements detailed in Section 4.1.2.

### 7.3.1 Memory Handling

For the purposes of memory integrity and ease of development, the *SensorClouds* architecture provides automatic mechanisms for memory management that are also used internally within the framework. The main component of this is a custom smart pointer implementation named *MemoryItem*. The base functionality is identical to that of the smart pointer implementation of the C++ standard library (`shared_ptr`) in the sense that it maintains a reference to a data object and a global reference counter for that object. Access to the variable is only possible via the smart pointer in order to prohibit external manipulation of the data object and the correct functionality of the smart pointer's memory management functions. Every time the smart pointer is explicitly copied or assigned to a new variable the reference counter is incremented. The smart pointer object is actually copied, except for the reference counter, which must be the identical instance across all copies of the smart pointer. To this end, the reference counter must be allocated on the heap (`new` allocator) so it is independent from individual copies of the smart pointer. Each instance of the smart pointer object only holds a raw pointer reference to the heap allocated counter which is trivially copyable. Every time a copy of the smart pointer is deleted the reference counter is decremented until the count is finally zero, which means the object is no longer referenced from any other and can

be destroyed. A *MemoryItem* can either be initialized with an already existing object or array of which it subsequently takes ownership or constructs a new instance of the specified type upon its own construction.

What sets the *MemoryItem* concept (cf. Figure 7.3) apart from the standard library version of a smart pointer, though, is the compatibility with and extended management functions in the context of CUDA applications. In order for data to be transferred between CPU and GPU within CUDA, the required memory for the data to be transferred must be previously allocated before the transfer can take place. The *MemoryItem* class takes care of the necessary allocations automatically upon construction and does so for both the managed data object as well as itself, so that the management functions are also available on both CPU and GPU. After all allocations have been performed, the *MemoryItem* has four different pointer addresses stored: one pointer to itself and the data object on the CPU as well as the pointer to the *MemoryItem* and the managed data object on the GPU side. The memory item also offers a convenient `transfer()` method which handles the transfer of data from the host to the device and vice-versa depending on the direction parameter passed to it. In this manner, developers need not keep track of individual pointer addresses for individual objects, since this is all handled by the smart pointer internally. This automatism also extends to any subordinate dependent objects registered with a superordinate object, so that the transfer of data need only be initiated for the superordinate object, and the rest is handled automatically by the *MemoryItem*.

Memory pinning, which pins pages of the CPU's working memory for faster transfers to and from the GPU than regular dynamic memory would allow, is also automatically managed by a central instance of the *MemoryManager*, which all individual memory items utilize for all operations concerning pinned memory and global memory functions. The *MemoryManager* is responsible for keeping track of all pinned memory regions and handling the addition and removal of such regions on request by a *MemoryItem*. Since memory can only be pinned once, lest an error occur, the memory manager must also calculate possible intersections between requested regions, e.g., when a pinned object is contained within another object which is to be pinned, and pin the remaining memory not included in the overlapping section.

And finally, in order to ease the development of applications and *Modules* with the *SensorClouds* framework, the *MemoryItem* provides a custom dereferencing operator which automatically detects whether the code is currently being executed on the CPU or the GPU and returns the appropriate pointer to the underlying data object. This leads to cleaner and easier to read code, since data accesses do not each have to manually check which computing device they are being executed on and also increases the portability of *Module* code, since the data access through the smart pointer becomes agnostic of the executing device.

### 7.3.2 Input Data Flow

In the reference implementation, data input is restricted to ROS topics, although all interfaces are explicitly exchangeable with arbitrary input drivers, pursuant to **FR 6**.

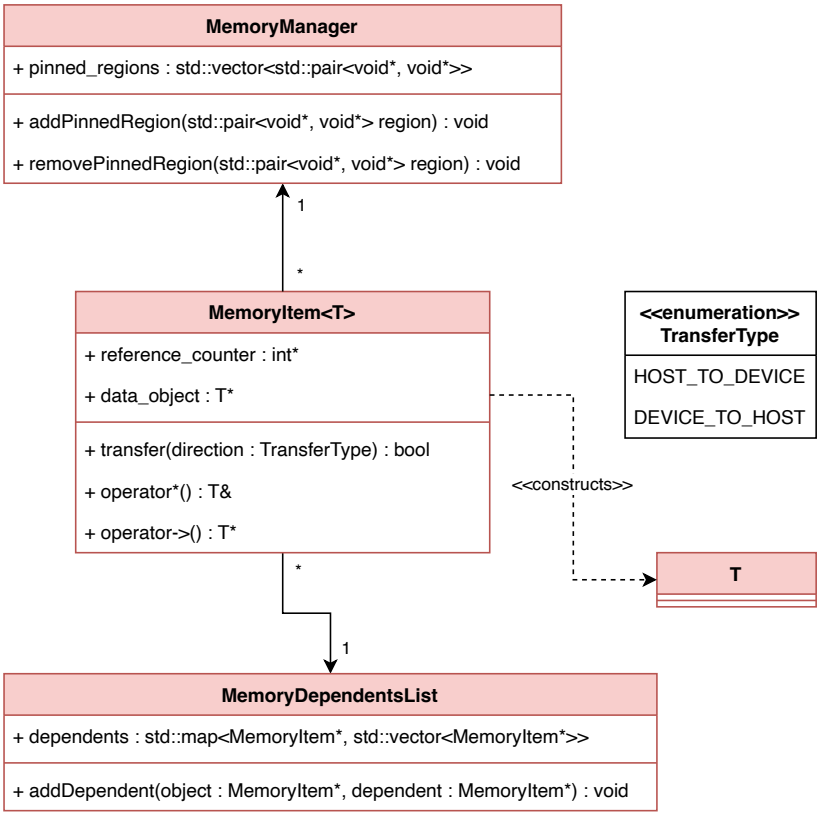



**Figure 7.3.** Overview of the MemoryItem architecture.

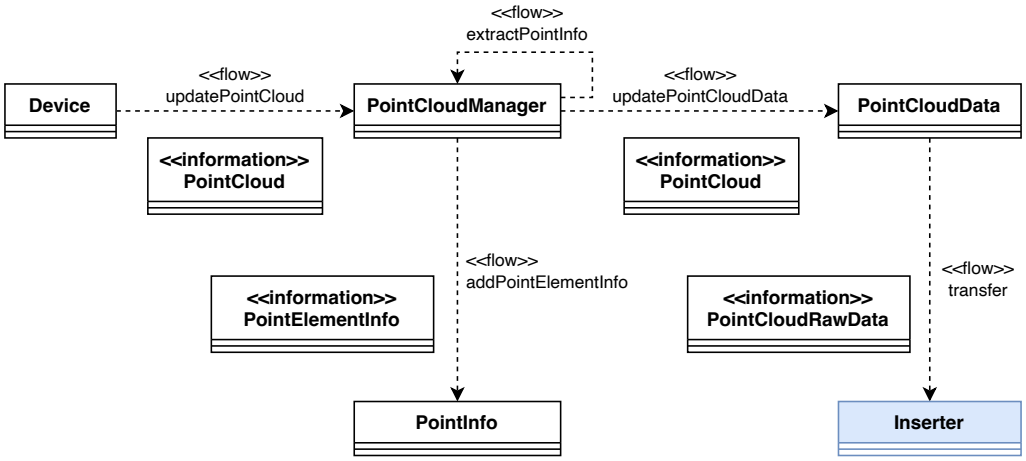



**Figure 7.4.** Information flow of a point cloud through the various stages of the processing pipeline.

The information flow of a point cloud message from its reception via a ROS topic to its transfer to the GPU for processing by the appropriate *ModuleInstance* is depicted in Figure 7.4. Every *Device* driver is responsible for handling concrete data input handling

as well as calibration and other device specific data and providing access to this data to other classes in the application. As such, the *Device* receives the point cloud message from ROS directly and then hands it off to the *PointCloudManager*. This class is in charge of extracting the meta-information stored within the point cloud message upon reception of the first message, since this information is immutable at runtime per the implementation of messages in ROS itself. The meta-information is represented in the *PointInfo* object which stores the information on an entire data point. This information is in turn a list of individual *PointElementInfo* objects, which hold the information on each individual data field, such as the data type, its size and the name of the field defined in the ROS message. The *PointCloudManager* also acts as a container for both the meta-information, which it alone holds, as well as a reference to the associated *PointCloudData* object which is a CUDA-compatible object and hence can be used in GPU code, such as the processing function of an *Inserter*. Since the meta information is only required for the code generation process described in the following section, the *PointCloudData* can transfer only the raw point cloud data in form of a byte array contained in the ROS message to the GPU. This data is then subsequently read in by the *SensorClouds* control loop executing the processing method of the *Inserter* on each data point.

### 7.3.3 Code Generation

For every *ModuleInstance* in the developed application, code for JIT compilation is automatically generated by the framework upon completion of the configuration process. In this context the following section will differentiate between different compile-time and runtime definitions. A graphical overview of these and how they interact with each other is depicted in Figure 7.5. *Application runtime* denotes the execution of the overall application based upon the *SensorClouds* framework, with the *application compile-time* specifying the build process of said application. During application runtime a second compile-time instance occurs, namely for the compilation of CUDA kernel code generated for the execution of *ModuleInstances* at application runtime with the available configuration data and meta-information on the incoming data streams. This is called *kernel compile-time*. After all kernels have been compiled, the main execution loop of the application begins in which data is transferred to the GPU, the kernels are executed and finally the results are transferred back to the CPU. This step is then called *kernel runtime*. The entire generated kernel code for the execution of a simple *Inserter* is shown in Listing 7.12. The following section will, however, provide concrete excerpts from the complete example to illustrate specific aspects as they are discussed.

The basic structure of the CUDA kernel generated for each *Module* consists of the following steps:

1. **Creation of local arrays**
   Listing 7.6 shows exemplary output of the JIT code generator for a simple *Inserter* adding point cloud coordinates to the global model. The necessary data points from input and output datasets are copied from the GPU's global memory to local memory for further processing (lines 15 & 25). Since the length of the required

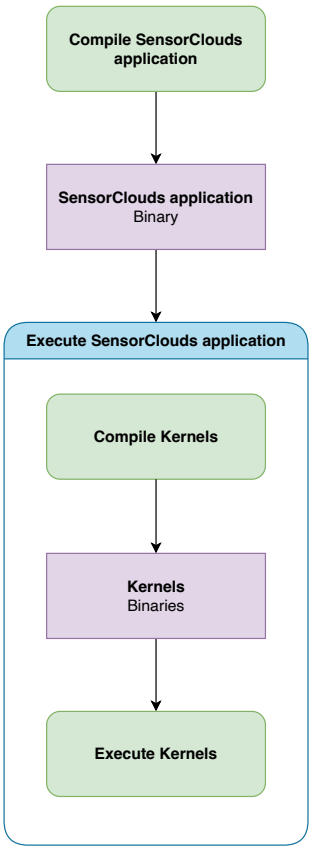



**Figure 7.5.** Overview of the application and kernel compilation and execution timelines.

arrays is known at compile time (lines 6-7), these fit into the register bank of the streaming multiprocessor. Should larger arrays be required, an explicit transfer of data to shared memory might be advisable if the implicit caching mechanism should fail. In each iteration a thread copies the appropriate data points into these local arrays.

2. **Creation of correctly typed variables**
   Since the memory layout is completely dynamic and determined during runtime of the base application, the actual data storage can only be implemented as a raw byte array. In order to be able to call the strongly typed processing methods defined by *Modules*, strongly typed variables must be created which store copies of the data fields contained in the raw array. The meta information on the layout of the raw data arrays is defined as described in Section 4.2.5 and is used to cast a pointer to the raw array first to the correct pointer type and then dereference it in order to be able to copy the value to the variable. The helper variables required by the *Aggregators* are created analogously. Listing 7.6 already showed the creation of the three input variables (lines 10-12) and the assignment operation to them from the local array. Variables to be written to, which are needed by both the *Method* output and *Aggregators*, are slightly more complex in their setup as shown

```
1  // Pointers to data objects are passed to the kernel function
2  auto& INPUT = **reinterpret_cast<
3                      SCMemoryItem<SCPointCloudData>*>(INPUT_POINTER);
4  auto& OUTPUT = **reinterpret_cast<
5                      SCMemoryItem<SCSensorCloudData>*>(OUTPUT_POINTER);
6
7  // Local arrays
8  uint8_t inputElement[20];
9  uint8_t outputElement[48];
10
11 // Input variables
12 float x_inR;
13 float y_inR;
14 float z_inR;
15
16 // Read the next element and copy to local array
17 memcpy(inputElement, INPUT.readFromIndex(i), 20);
18
19 // Assign input variables from data copied to local array
20 x_inR = ( *( (float*)  &inputElement[0] ) );
21 y_inR = ( *( (float*)  &inputElement[4] ) );
22 z_inR = ( *( (float*)  &inputElement[8] ) );
23
24 // Simple case of mapping directly to input coordinates
25 auto currentIndex = OUTPUT.readFromCoords({x_inR, y_inR, z_inR});
26 // Copy data point at input coordinates to local array
27 memcpy(outputElement, currentIndex, 48);
```

**Listing 7.6.** Creation of and copy operation to local arrays.

exemplarily for the output variable $x$ in Listing 7.7. The variable `x_outW` (line 4) is the one which the write operation of the *Method* is cached into before later being processed by the *Aggregator*. Since the variable of the primitive type `float` would simply be copied by value if it were to be passed directly to a function, a pointer to it is required (`x_outW_ptr` in line 4 with its assignment in line 5). The previous value of the data field before the operation was performed can be required by certain *Aggregators* in order to perform their calculations and hence a variable to store the old value is required also. These declarations all occur outside of the main kernel processing loop, since they are static for the duration of a kernel execution. Inside the loop for every iteration, the written variable must first be reset from the value it may have been assigned in the previous iteration (line 12). The old value variable as well as the helper variables of the *Aggregator* must be

updated in every iteration of the loop with the values copied from the respective data points from the output data (lines 14-18).

```
1  // Only for variable x. y and z defined analogously
2  // Before loop:
3  // Declaration of strongly typed variables...
4  // ...for the variables written to the global model
5  float x_outW, *x_outW_ptr = &x_outW, x_outW_old;
6  x_outW_ptr = &x_outW;
7  // ...for the helper variables of the Aggregators
8  float x_sum_agg, *x_sum_agg_ptr = &x_sum_agg;
9  uint32_t x_count_agg, *x_count_agg_ptr = &x_count_agg;
10
11 // Inside loop:
12 // Reset written variable on every loop iteration
13 x_outW = 0.0f;
14 // Assign appropriate value from local array to variable
15 x_outW_old = ( *( (float*)  &outputElement[1] ) );
16
17 // Same procedure for the halper variables of the Aggregators
18 x_sum_agg = ( *( (float*)  &outputElement[20] ) );
19 x_count_agg = ( *( (uint32_t*)  &outputElement[16] ) );
```

**Listing 7.7.** Creation of correctly typed variables for further use by *ModuleInstances* and *Aggregators*.

3. **Retrieval of *Method* and *Aggregators***
   Since *ModuleInstances* can hold state information, which might be initialized on the host side and then needed on the device side, all *Method* objects are each automatically wrapped in a *MemoryItem*. In consequence, the JIT code cannot simply create new objects for the employed *Methods* but must rather retrieve the pre-existing ones transferred from the host. To this end the raw pointers of the objects on the device side are included in the generated code and cast to the appropriate class type for further use. The same process must be carried out for every required *Aggregator*.

4. **Execution of *Module* function**
   Once all the required variables for the execution of the concrete processing function have been defined, the *Method* of the *Module* of the can be called with the correctly typed local variables as parameters. The call is performed on the concrete object reference created in the previous step. Since the CUDA compiler (nvcc) aggressively inlines methods in order to minimize the number of context

```
// Retrieval of the previously created
auto& moduleInst = *reinterpret_cast
    ↪ <SCMemoryItem<SCXYZInserterImpl>*>(140214001477632);

// Aggregator for the data field x
auto& aggregator_1 = *reinterpret_cast
    ↪ <SCMemoryItem<SCMeanAggregatorImpl>*>(140214001470464);
// Aggregator for the data field y
auto& aggregator_2 = *reinterpret_cast
    ↪ <SCMemoryItem<SCMeanAggregatorImpl>*>(140214001471488);
// Aggregator for the data field z
auto& aggregator_3 = *reinterpret_cast
    ↪ <SCMemoryItem<SCMeanAggregatorImpl>*>(140214001472512);
```

**Listing 7.8.** Retrieval of *Method* objects and *Aggregators*.

switches during the execution of a kernel, the variables are not actually copied at runtime of the JIT compiled program, but rather the method implementation is inlined at the call point and operates directly on the local variables. The generated code for calling the *Method* of the current *Module* is shown in Listing 7.9. It receives the three coordinate variables from the input data as read-only pass-by-value parameters and the output coordinate variables as pointers to the local variables as discussed previously.

```
moduleInst->process(x_inR, y_inR, z_inR, x_outW_ptr, y_outW_ptr,
    ↪ z_outW_ptr);
```

**Listing 7.9.** Execution of the *Method* of the current *Module*.

5. **Execution of *Aggregator* functions**
   Analogous to the execution of the *Module* function, all necessary *Aggregator* methods are called thereafter. They receive the value previously stored in the global model, the new value produced by the method of the *Module*, the location to write their result to (which is a pointer to the local variable) as well as pointers to any helper variables the *Aggregator* has defined and are needed to perform the intended calculation. Listing 7.10 shows the calls to all three *Aggregators* needed for the coordinate insertion.

6. **Commitment of results to global storage**
   After all calculations have been performed by the *Module* and the defined *Aggregators*, the results are then copied back into the global storage where they

```
aggregator_1->aggregate(x_outW_old, x_outW, x_outW_ptr,
    ↪ x_sum_agg_ptr, x_count_agg_ptr);
aggregator_2->aggregate(y_outW_old, y_outW, y_outW_ptr,
    ↪ y_sum_agg_ptr, y_count_agg_ptr);
aggregator_3->aggregate(z_outW_old, z_outW, z_outW_ptr,
    ↪ z_sum_agg_ptr, z_count_agg_ptr);
```

**Listing 7.10.** Execution of the *Aggregator* methods defined for the global model and relevant to the concrete *Module*.

can subsequently be loaded again by other *Modules* or finally copied back to the CPU as result set either entirely or a selection of data points from it. The local variables are sequentially written back to the local array copy of the global data. To this end, the appropriate index of the local array copy must again be cast to the appropriate pointer type of the data field and subsequently dereferenced in order for the value to be written back correctly into the byte array. Finally, the local array copy is written back into the global model storage, and the next loop iteration begins or, if all data points assigned to this thread have been processed already, the kernel terminates at this point.

```
// Only for variable x. y and z defined analogously
// Write back the output of x
( *( (float*)  &outputElement[1] ) ) = x_outW;

// Write back new values of helper variables
( *( (float*)  &outputElement[20] ) ) = x_sum_agg;
( *( (uint32_t*)  &outputElement[16] ) ) = x_count_agg;

// Copy local element back to global model storage
memcpy(currentIndex, outputElement, 48);
```

**Listing 7.11.** Write back operations from the variables to the local array and finally to the global model.

In keeping with the design-by-contract paradigm and fulfilling the requirement of modularity (**FR 4**), no *Module* or *Aggregator* can ever access any variable or part of the global data model not previously defined in their specification. This fact is self-evident when regarding the code generated by *SensorClouds* for JIT compilation. All read and write accesses to the input point clouds as well as the global model are cached locally to the executing thread and only then passed on to concrete processing functions. There

is no way that a module developer can gain access to the raw data arrays, since their location is hidden from any executed method in this fashion.

The generated code also makes it very clear that not only the application developer, but also the module developer never have to deal with the specifics of parallel programming, since the parallelization is handled completely by the *SensorClouds* framework, hereby fulfilling **NFR 1**. Module developers can thus focus entirely on implementing the processing function for a single data point and application developers can focus on building their concrete sensor fusion and processing application.

```
// Kernel function definition with pointers to data objects as parameters
__global__
void run(uintptr_t INPUT_POINTER, uintptr_t OUTPUT_POINTER) {

    auto& INPUT = **reinterpret_cast<SCMemoryItem<SCPointCloudData>*>(INPUT_POINTER);
    auto& OUTPUT = **reinterpret_cast<SCMemoryItem<SCSensorCloudData>*>(OUTPUT_POINTER);

    uint8_t inputElement[20], outputElement[48];

    float x_inR;
    float y_inR;
    float z_inR;

    float x_outW, *x_outW_ptr = &x_outW, x_outW_old;
    float y_outW, *y_outW_ptr = &y_outW, y_outW_old;
    float z_outW, *z_outW_ptr = &z_outW, z_outW_old;

    float x_sum_agg, *x_sum_agg_ptr = &x_sum_agg;
    uint32_t x_count_agg, *x_count_agg_ptr = &x_count_agg;

    float y_sum_agg, *y_sum_agg_ptr = &y_sum_agg;
    uint32_t y_count_agg, *y_count_agg_ptr = &y_count_agg;

    float z_sum_agg, *z_sum_agg_ptr = &z_sum_agg;
    uint32_t z_count_agg, *z_count_agg_ptr = &z_count_agg;

    auto& moduleInst = *reinterpret_cast <SCMemoryItem<SCXYZInserterImpl>*>(140214001477632);

    auto& aggregator_1 = *reinterpret_cast
        ↪  <SCMemoryItem<SCMeanAggregatorImpl>*>(140214001470464);
    auto& aggregator_2 = *reinterpret_cast
        ↪  <SCMemoryItem<SCMeanAggregatorImpl>*>(140214001471488);
    auto& aggregator_3 = *reinterpret_cast
        ↪  <SCMemoryItem<SCMeanAggregatorImpl>*>(140214001472512);

    for (size_t i = blockIdx.x * blockDim.x + threadIdx.x; i < INPUT.size(); i += blockDim.x *
        ↪  gridDim.x) {
        memcpy(inputElement, INPUT.readFromIndex(i), 20);
        x_inR = ( *( (float*)  &inputElement[0] ) );
        y_inR = ( *( (float*)  &inputElement[4] ) );
        z_inR = ( *( (float*)  &inputElement[8] ) );

        memcpy(outputElement, currentIndex, 48);

        x_outW = 0.0f;
        x_outW_old = ( *( (float*)  &outputElement[1] ) );

        y_outW = 0.0f;
        y_outW_old = ( *( (float*)  &outputElement[5] ) );

        z_outW = 0.0f;
        z_outW_old = ( *( (float*)  &outputElement[9] ) );

        moduleInst->process(x_inR, y_inR, z_inR, x_outW_ptr, y_outW_ptr, z_outW_ptr);

        x_sum_agg = ( *( (float*)  &outputElement[20] ) );
        x_count_agg = ( *( (uint32_t*)  &outputElement[16] ) );

        y_sum_agg = ( *( (float*)  &outputElement[28] ) );
        y_count_agg = ( *( (uint32_t*)  &outputElement[24] ) );

        z_sum_agg = ( *( (float*)  &outputElement[36] ) );
```

```
59          z_count_agg = ( *( (uint32_t*)  &outputElement[32] ) );
60
61          aggregator_1->aggregate(x_outW_old, x_outW, x_outW_ptr, x_sum_agg_ptr,
                ↪  x_count_agg_ptr);
62          aggregator_2->aggregate(y_outW_old, y_outW, y_outW_ptr, y_sum_agg_ptr,
                ↪  y_count_agg_ptr);
63          aggregator_3->aggregate(z_outW_old, z_outW, z_outW_ptr, z_sum_agg_ptr,
                ↪  z_count_agg_ptr);
64
65          ( *( (float*)  &outputElement[1] ) ) = x_outW;
66          ( *( (float*)  &outputElement[5] ) ) = y_outW;
67          ( *( (float*)  &outputElement[9] ) ) = z_outW;
68
69          ( *( (float*)  &outputElement[20] ) ) = x_sum_agg;
70          ( *( (uint32_t*)  &outputElement[16] ) ) = x_count_agg;
71
72          ( *( (float*)  &outputElement[28] ) ) = y_sum_agg;
73          ( *( (uint32_t*)  &outputElement[24] ) ) = y_count_agg;
74
75          ( *( (float*)  &outputElement[36] ) ) = z_sum_agg;
76          ( *( (uint32_t*)  &outputElement[32] ) ) = z_count_agg;
77
78
79          memcpy(currentIndex, outputElement, 48);
80      }
81  }
```

**Listing 7.12.** Entire JIT code generated for a simple *Inserter*.

### 7.3.4 Dataset Handling

*DataSets* and, by extension *ScQLquery* objects, must be generated specifically for the data fields and their types at kernel compile-time, since all the necessary information is only available at that point in the application lifecycle. For each *DataSet* required, code for a derived type of the base class *SCDataSet* is generated, which holds the specific information on the data layout of the global model and access operations for the data fields specified by the *Module* and *only* for these.

The goal of the code generation for *DataSets* is to enable access to a selection of data points from a larger pool of data while maintaining the access restrictions imposed on *Modules* in accordance with the design-by-contract paradigm. Concretely, this access shall be in the form of a two-dimensional array access, where the parameter of the first access denotes the index of the data point to be accessed from the result set and the second access specifies the data field:

```
dataset[42][sensorclouds::tags::x];
```

Listing 7.13 shows the exemplary code generator output for a *DataSet* with access to the cartesian coordinates of a data point in the global model. The struct *SCDataSet_1*

(line 24) is templated, just as the equivalent *ScQLquery* in Listing 7.5 was. The template parameters are the field definitions of the configured data fields the *Module* has requested access to. Since it is derived from the base class *SCDataSet*, it can be used in its place, but can polymorphically overwrite the array access operator (`operator[]`, lines 26-28) parametrized with the index of the element to be accessed. This allows the concrete implementation of the *DataSet* to return a specialized version of the proxy object *SCPointContainer_1* (lines 1-20), which in turn provides an access operator for the individual fields of the selected data point. The template parameter for the `operator[]` method does not have to be explicitly specified since C++ added so called *class template argument deduction* (CTAD) [34] in C++20, which enables the compiler to infer the template argument from the constructor's parameters if the template argument is used as a parameter type. Wherever possible, the `constexpr` specifier is used, which tells the compiler to evaluate expressions at compile-time rather than at runtime. The consequence of this, however, is that all variables within the expressions marked with this specifier must be known at compile-time and must be constant. Since the dynamic aspect of the *SensorClouds* architecture is fully resolved at kernel compile-time, this is not a hindrance, since all necessary information is available at this stage of the application lifecycle. This is also the other reason for the use of tags, which are simple structs denoting a specific data field, and are employed in the *SensorClouds* architecture to access data fields. Were the generated code to produce string comparisons for the name of each field in the `if...else if...` statement (lines 9-15), differentiating between the data fields, then the code could not be evaluated at compile-time and would suffer a significant loss in performance. However, since the types of the tags as well as the access points and indices are all known at compile-time, the compiler can directly replace any data access proxied by the *DataSet* with direct memory access, all while still keeping the access restrictions in place. Hence, this construct enables both high performance and flexibility while restricting *Module* access to the pre-arranged data fields.

While the use of a proxy object might seem counterintuitive from a performance standpoint, preliminary evaluations showed that compilers can seemingly optimize a two-dimensional array access with two different array subscript operators over two different objects far more aggressively than other alternatives, such as the function call operator with two arguments. The performance of various access methods was evaluated by performing 10 000 000 data access operations with each of the alternatives and recording the total execution time over all iterations, since every single iteration is way too quick for accurate time measurement with standard methods. As point of comparison, a very complex dynamic runtime-evaluated implementation was pitted against the JIT compiled implementations of the same operations. The results of this evaluation are depicted in Table 7.1.

This JIT-compiled *DataSet*, however, creates a different problem for developers using the *SensorClouds* framework, since the objects they want to use in their programs don't exist before the kernel compile-time. This means that developers could only verify that their code uses the correct fields and data types by running the entire application and checking if any errors occur during the kernel compilation phase. Moreover, some IDEs, such as Clion from JetBrains [58] can display type information on variables inline

```
1  struct SCPointContainer {
2
3      constexpr SCPointContainer_1(const uint8_t *data) :
           ↪ _rawDataPointer(data) {}
4
5      template<typename Tag>
6      constexpr
7      inline
8      auto const operator[](Tag &tag) const {
9          if constexpr (std::is_same_v<Tag,
               ↪ sensorclouds::tags::types::x>) {
10             return ( *( (float*) &_rawDataPointer[2] ) );
11         } else if constexpr (std::is_same_v<Tag,
               ↪ sensorclouds::tags::types::y>) {
12             return ( *( (float*) &_rawDataPointer[6] ) );
13         } else if constexpr (std::is_same_v<Tag,
               ↪ sensorclouds::tags::types::z>) {
14             return ( *( (float*) &_rawDataPointer[10] ) );
15         }
16     }
17
18     const uint8_t *_rawDataPointer;
19
20 };
21
22
23 template<typename... Ts>
24 struct SCDataSet_1 : SCDataSet<Ts...> {
25
26     constexpr auto operator[](size_t index) {
27         return SCPointContainer_1(_contents_pointers_array[index]);
28     }
29
30     uintptr_t _contents_pointers_array[42];
31
32 };
```

**Listing 7.13.** Generated code for one specific instance of a *DataSet* for access to three-dimensional coordinates.

within the code structure while it is being developed. CLion even invokes the static portion of the C++ compiler in order to resolve even complex type hierarchies for inline

| Implementation | Time [ms] | Time [µs] | Time [ns] |
|---|---|---|---|
| `operator[][]` | 53 | 53 269 | 53 269 638 |
| `operator()` | 11 | 11 548 | 11 548 231 |
| `operator[][] (JIT)` | 0 | 1 | 1114 |
| `operator() (JIT)` | 0 | 1 | 1832 |

**Table 7.1.** Results of the evaluation of operator performance in dynamic and JIT code. Times are total execution time for 10 000 000 iterations each.

display. This feature also requires type information which is finally available at kernel compile-time to be available during development of the *SensorClouds* application.

In order to support this feature in accordance with **NFR 4**, a static mock implementation of the *DataSet* class was included in the *SensorClouds* framework, whose only purpose is to mirror back the correct types from dynamically created data structures for programmer convenience. Developers program their applications against this definition of *DataSet* which is then swapped for the concrete implementation at kernel compile-time, guaranteeing a seamless transition between the two. What makes such a feature possible is the template meta-programming library [35] introduced in C++11. It enables the creation of small meta-programs which argue only on type information, but which can solve complex tasks relating to dynamic type hierarchies at compile-time, which increases compilation times by a fair amount, but makes the executable code that much more efficient.

```
1  template<typename... Ts>
2  struct SCPointContainer {
3
4      template<typename Tag>
5      constexpr typename VariadicContainsType<Tag, Ts...>::type
           ↪  operator[](Tag& tag) {
6          return ( ( typename VariadicContainsType<Tag, Ts...>::type )
               ↪  0 );
7      }
8
9  }
```

**Listing 7.14.** The mock implementation for *SCPointContainer* for compatibility with IDEs.

Analog to the code section shown for the concrete implementation of *DataSet* generated for JIT compilation shown in Listing 7.13, the static mock version found in Listing 7.14 also uses a *SCPointContainer* object, which overloads the array access operator in order to enable access to a specific data field of the current data point, the implementation of which can be found in Listing 7.14. The fields to be accessed through a *DataSet* are

defined as template arguments of the type *Field*, which in turn each has two template arguments for the data field's tag and the desired type:

```
1  SCDataSet<
2      Field<sensorclouds::tags::types::x, float>,
3      Field<sensorclouds::tags::types::y, float>,
4      Field<sensorclouds::tags::types::z, float>
5  > dataset;
```

**Listing 7.15.** Declaration of a *DataSet*.

Hence, in order to retrieve the correct data type to a tag passed to the access operator in the mock implementation for IDE compatibility, the template arguments have to be searched first for the *Field* type with the correct tag and its type then returned to the caller, which is exactly what the template meta-function *VariadicContainsType* does. To be syntactically correct, however, the access operator method must also actually return a value, which is why the type retrieval meta-function is used multiple times in the implementation of the operator method. The first instance serves the definition of the operator's return type (line 5), while the second instance is used to cast the mock value $0$, which is convertible to any other type, to the correct type for the requested data field.

The implementation of the *VariadicContainsType* meta-function is shown in Listing 7.16. At the beginning of this listing, an excerpt from the *Field* class is shown (lines 2-6) to illustrate how the type information used by *VariadicContainsType* is stored within it. By copying the template parameter to a local type with the `using` directive, it can later be accessed directly on every instance of the class during compilation. The correct function of the following implementation of *VariadicContainsType* requires the definition of structurally overlapping meta-classes each containing a type by the same name but with different definitions, one void and one the actual type (lines 9 & 12). The algorithm employed for *VariadicContainsType* is based on recursion through partial template specialization, which is the practice of defining a class with a broad template first, and then progressively specializing the types of the template arguments for various cases. With variadic template arguments, this can be exploited to process one element of the type list after another. First the base type needs to be declared (lines 16-17). The meta-function is called with the searched tag and the variadic argument the *DataSet* was defined with as template parameters (line 20). Within the function, the currently split off type's tag is checked for a match with the searched tag (line 22), and if it does indeed match, the data type is returned (line 24). If not, the recursion continues (line 26). Finally, if the tag is not contained within the list, the type `void` is returned (lines 31-32).

```
// Template types are saved in the Field class for later reference
template<typename Tag, typename Type>
struct Field {
    using TagType = Tag;
    using TypeType = Type;
}

// Wrapper for void type (not found)
template<typename... Vs> struct MakeVoidType { typedef void type; };

// Wrapper for concrete type
template<typename T> struct MakeType { typedef T type; };


// Base type declaration
template < typename Tp, typename... List >
struct VariadicContainsType : MakeVoidType<List...> {};

// Split off the first type declaration of the variadic template parameter
template < typename Tp, typename Head, typename... Rest >
struct VariadicContainsType<Tp, Head, Rest...>
        : std::conditional< std::is_same<Tp, typename Head::TagType>::value,
                // If searched type is found, return the type wrapper
                MakeType<typename Head::TypeType>,
                // if note, continue recursively without 'Head'
                VariadicContainsType<Tp, Rest...>
        // Finally return the actually determined type
        >::type {};

// Searched type was not found
template < typename Tp >
struct VariadicContainsType<Tp> : MakeVoidType<Tp> {};
```

**Listing 7.16.** Implementation of the *VariadicContainsType* template meta-function for type resolution within *DataSets* during development.

**Chapter Summary**. This chapter evaluates the *SensorClouds* architecture as well as its accompanying reference implementation by a number of key factors. First, the raw performance is compared against competing frameworks in a quantitative evaluation, and secondly the applicability to typical sensor fusion applications is qualitatively evaluated. Finally, the fulfillment of requirements is discussed.

# 8 Evaluation

## 8.1 Quantitative Evaluation

Due to the fundamental technological differences between the compared frameworks discussed in Section 4.4, the qualitative evaluation of the frameworks can only be based on the lowest common denominator between them, which is the execution time of an insertion operation for a single point cloud into the respective underlying global data model.

The hardware setup for the performance evaluation of the three frameworks consisted of the following components:

- **CPU: Intel Core i7-8700K**
  with 6 physical cores and 12 threads running at a base frequency of 3.70 GHz and able to reach a sustained maximum boost frequency of 4.70 GHz. It is an 8th generation Intel processor (Coffee lake) with a 14 nm lithographic process.
- **RAM: Corsair Vengeance**
  64 GB (2 * 32 GB) of RAM running at a clock frequency of 3000 MHz in dual channel configuration.
- **GPU: NVIDIA RTX 2080 Super**
  Being the first generation of NVIDIA GPUs with dedicated raytracing cores for advanced lighting effects, it also possesses 3072 CUDA general purpose computing cores as well as 384 dedicated Tensor cores for machine learning applications. The GPU is built upon the Turing architecture from NVIDIA and was manufactured by TSMC using their 12 nm FinFET fabrication process. The 2080 Super is also equipped with 8 GB of GDDR6 on-board video memory capable of transfer speeds of up to 496 GB/s.

The software setup consisted of a base Ubuntu 20.04 Desktop installation with g++ 9 as a standard C++ compiler and the NVIDIA toolkit 11.2 with nvcc as the CUDA compiler. In order to ease in the building process, the project was compiled using the newer CMake version 3.10.2, since it has native support for CUDA and can thus aid immensely in the correct configuration of the build process.

| Framework | Avg [µs] | Median [µs] | StdDev [µs] | Min [µs] | Max [µs] |
|---|---|---|---|---|---|
| *SensorClouds* | 25.460 | 17.632 | 18.821 | 11.392 | 158.080 |
| GPUvoxels | 15.270 | 14.496 | 4.124 | 12.672 | 210.496 |
| PCL | 1218.544 | 442.935 | 9052.105 | 248.251 | 171 473 |

**Table 8.1.** Results of the evaluation (point cloud with 43 941 points; 10 000 executions each).

Table 8.1 shows the results of the quantitative evaluation in which the insertion procedure was executed 10 000 times consecutively for each framework individually. The input data was a static point cloud containing 43 941 points stored in the appropriate working memory, meaning no read access to the main storage of the computer was necessary at runtime.

## 8.2 Applicability to Sensor Fusion Tasks

Since *SensorClouds* in and of itself is solely an architecture for sensor fusion and not a concrete set of algorithms, this chapter will give an overview of how various types of such algorithms can be implemented within the *SensorClouds* architecture. In literature three distinct abstraction levels of sensor fusion processes are typically described [50, p. 22f][39, p. 7][68]:

- **Low-level or raw data fusion**
  In this case, the raw sensor values are directly combined with one another. This is of course only possible if a direct mapping between sensor values is immediately apparent or can be created by processing the values in such a manner that they are transformed into a common unit of measurement. The expectation of this fusion is that the resulting output contains more information than the inputs of the operation individually hold. Any raw sensor inputs measuring the same physical property can, for example, be directly fused with each other, such as distances in three-dimensional space acquired from depth sensors.
- **Intermediate-level or feature level fusion**
  Intermediate-level fusion is based on pre-processed sensor values which are then subsequently fused into a common data point. On this level the data to be fused already contains semantic information in the form of e.g., detected edges, corners or objects, depending on the feature extraction method employed. The resulting features are then fused into a common representation of the surveilled space in form of a feature map which can subsequently be used for further processing such as obstacle avoidance. Fusing multiple sources of features promises to produce a more reliable and complete dataset which subsequently leads to more robust decision making e.g., with respect to objects in the path of a robot which are to be avoided.
- **High-level or decision fusion**
  High-level fusion is employed in order to combine the suggested decisions from multiple sources. These in turn base their suggestions on either raw data or extracted features and propose a plan of action for the controller of the system to execute. Fusing decisions, e.g., through voting or statistical methods promises more robust decision making of the overall system, since the failure of one source to produce a decision proposal or a wrong decision can be compensated by the other sources. An example of this technique are the airbag systems found in automobiles. In order to prevent the accidental or excessive deployment of airbags due to sensor faults or minor impacts, respectively, the controller only fires the airbag inflation charges if more than one crash sensor detects an impact. An additional acceleration sensor within the vehicle must also detect the impact in order for the airbags to be deployed, thus preventing accidental firing in case of defective crash sensors or strong electromagnetic interference.

Through the high level of customizability offered by the *SensorClouds* architecture in terms of data storage and processing, all three abstraction levels of sensor fusion are easily realizable within it. The implementation of raw data fusion is the most obvious,

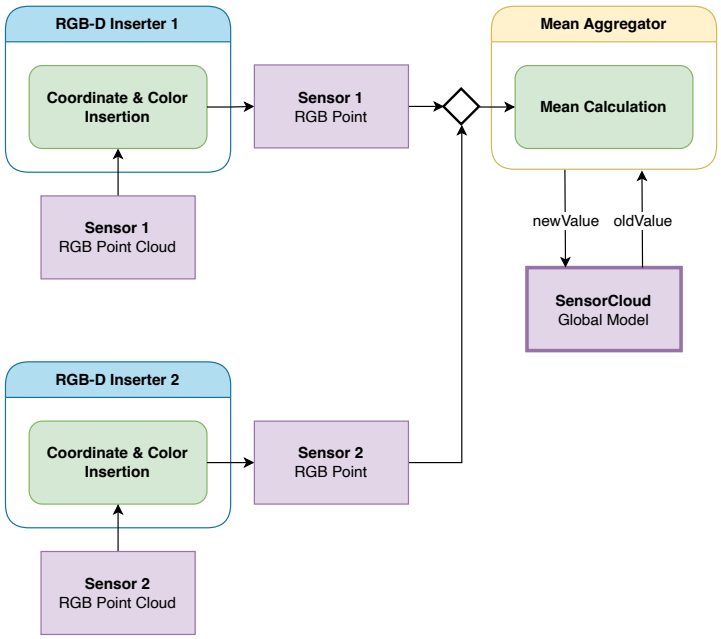



**Figure 8.1.** Activities involved in sensor fusion by spatial coincidence. This is the default in *SensorClouds*.

in case data from multiple sensors measuring the same phenomenon is to be fused. All data input streams are fed to the appropriate *Inserters* and subsequently fused with a simple *Aggregator* calculating the mean value of the measured property per voxel. The addition of capacitive sensor values in the conservative form of merely inserting the point cloud containing all possible values into the global data model can also be interpreted as raw data fusion, since the actual property of capacitance is not considered, but rather a safety zone of occupied voxels is created which makes the data equivalent to any other sensor data providing information on three-dimensional occupancy in the environment.

Should the probabilistic nature of capacitive sensors and all other sensors involved additionally be considered this would constitute an elevation to feature level fusion, since all data points are endowed with additional information beyond the mere measurement of the respective physical property. Completely abstract features, such as detected corners, edges or even objects can also easily be added to the *SensorClouds* data model by simply adding custom bit-fields to represent the applicable detections per voxel as well as a custom *Module* to generate the appropriate entries at each location. This can be either an *Inserter* if the required features can directly be extracted from each individual sensor data input stream or a *Processor* computing the features for each voxel after all input data has been entered into the global data model.

In order for high-level fusion to be implemented in form of a voting system within *SensorClouds* the *Aggregator* must simply maintain a distinct counter per voxel for each inserted modality. Then the model output can be filtered according to an arbitrary amount of required votes regarded as the threshold of certainty to base subsequent

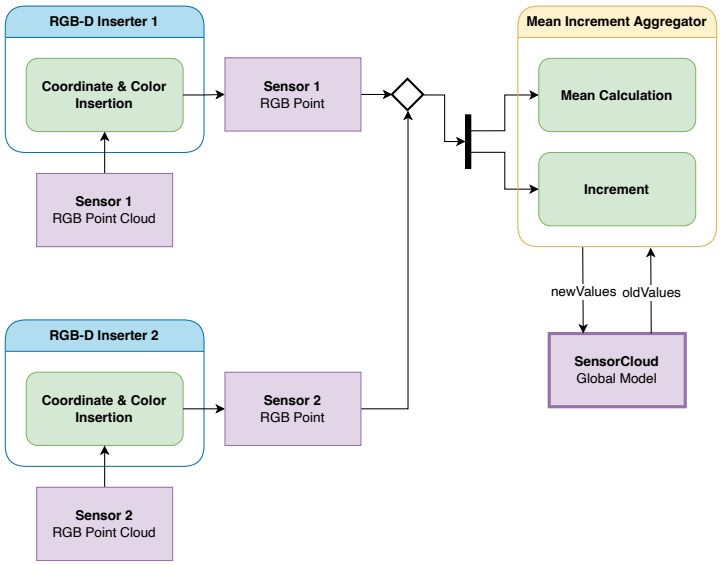



**Figure 8.2.** Activities involved in sensor fusion with an insertion counter.

decisions on (cf. Figure 8.2). Another possible implementation would be to maintain separate data fields per voxel for each sensor by defining a custom *Inserter* which differentiates properly between different sensors. Such a configuration is especially useful when it is necessary to maintain the information which sensor did or did not measure a certain property at a certain point in space. The actual decision level fusion process would consequently have to be performed by a custom *Processor* capable of understanding this semantic definition at a later stage (cf. Figure 8.3).

The former differentiation of sensor fusion levels hinges on how the data is processed throughout the entire pipeline and at which point in the pipeline or at which abstraction level the data is fused. Another classification of fusion processes distinguishes three different types based on the physical phenomena and the relationship of sensors to each other [39, p. 8f]:

- **Complementary**
  The relationship between multiple sensors is called complementary if their respective output data completes the overall image of the surroundings. This typically implies multiple sensors of the same type or at least measuring the same phenomenon, such as a number of cameras covering different areas of the space to be observed. Complementary fusion is ordinarily straightforward, since the individual inputs must only be added to each other in order to receive the complete picture of the environment.
- **Competitive**
  Competitive or redundant configurations are characterized, in contrast to complementary configurations, by multiple sensors observing the same phenomena in the same space. A competitive configuration can also be present when measurements from one and the same sensor taken at different instants are combined

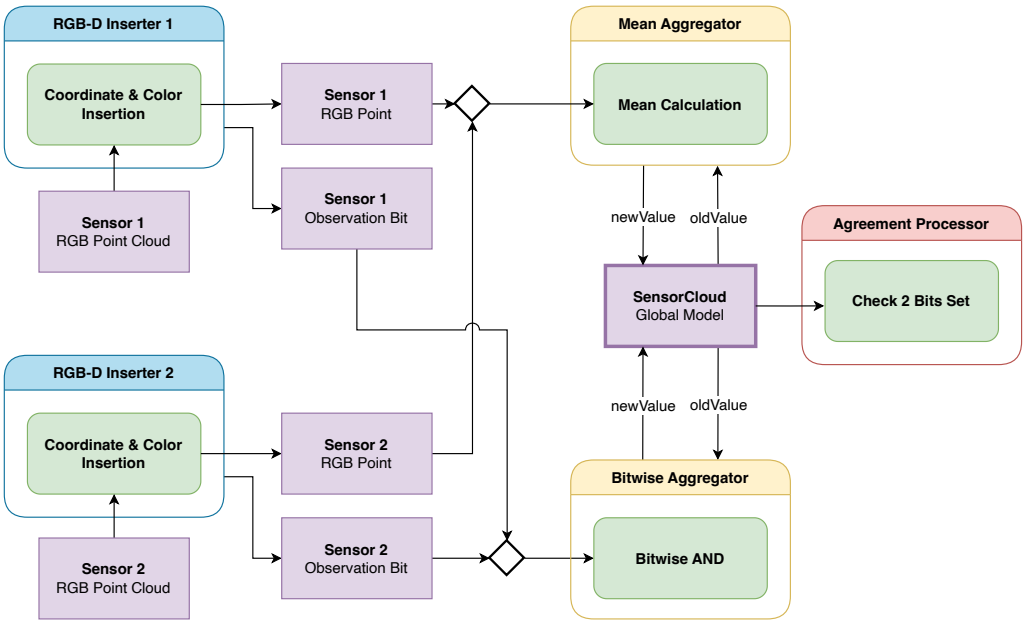



**Figure 8.3.** Activities involved in sensor fusion with a bitmask operation.

with each other. A special case of competitive fusion is fault tolerance which can either be implemented as a compensation or as a safety system. The compensation configuration simply allows one sensor to take over for another one in case it encounters a system fault and provides the required data in its stead. The safety configuration in accordance with the standard ISO 13849-1 [2] is more involved and requires redundant controllers cyclically cross-checking each other's results as well as the sensor inputs from two different sources. Should an error be detected in any of these steps, the emergency stop signal must be sent immediately to all components of the overall system.

- **Cooperative**
  When different sensor modalities are combined with each other, it is called cooperative fusion. While it is far more difficult to properly implement a cooperative sensor configuration due to the varying accuracies of the participating sensors and thus the resulting lower overall accuracy, these configurations also hold the promise of producing a combined view of the environment that would not have been possible to observe with each of the sensors involved on their own. Such a configuration is present, for example, within the Microsoft Kinect and similar RGB-D sensors, which combine the data from the built-in depth sensor with the images from an RGB camera in order to receive an RGB point cloud, or more specifically, three-dimensional data with added color values.

The *SensorClouds* architecture is also capable of supporting the processing of all different types of sensor configurations as described above through the development of appropriate *Modules* suited to the respective tasks. Complementary sensor fusion is the simplest possible type and can be realized in the same manner as the example

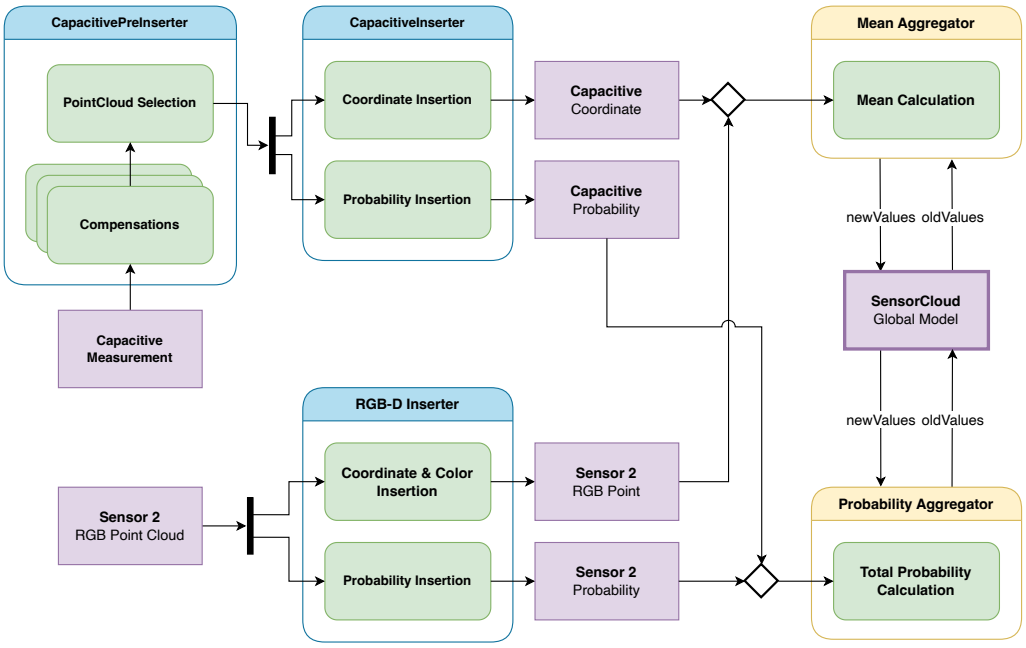



**Figure 8.4.** Activities involved in sensor fusion for capacitive sensor data with accompanying probability data.

given for low-level-fusion seen in Figure 8.1. Since the main premise of *SensorClouds* is the combination of all sensor data into one single global data model on the basis of three-dimensional locations of individual measurements, complementary sensor fusion is one of the inherent underlying paradigms of the entire architecture and thus trivial to implement.

Competitive fusion or redundancy of sensors can for example be used as a validation technique, in order to ensure correct positioning of sensors at runtime. If multiple cameras are deployed, all overlooking the same workspace of a robot with some degree of overlap in their respective fields of view, then such a validation routine is easy to implement. The required architecture is the same as depicted in Figure 8.3, with the only addition being that the *Agreement Processor* need only check one single voxel for agreement, yet must confirm that all bits are set for this voxel, meaning every camera in the system has detected an object at that point in space. The choice of this reference object should fall on something which is within the overlap area of the fields of view and improbable to be obscured by passing people or objects at runtime. In this manner the positioning of the cameras can be continuously validated at runtime in order to ensure the data fused in the global model continues to be accurate with respect to the relative transformation between the various sensors employed.

Depicted in Figure 8.4 is an example which covers both competitive and cooperative fusion in the same application. Since a system with both cameras (outside-in-sensors) as well as capacitive sensors mounted to the robot structure (inside-out-sensors) need not rely on a single one of these sensors to ensure operator safety in the workspace of the robot, their combination in the global model can be categorized as competitive

fusion. Both cameras and capacitive sensors observe humans in the environment and are intentionally used in tandem in order to compensate each other's inherent deficiencies, thus creating a competitive or redundant sensor configuration in which the goal is to have at least one sensor value at all times in the relevant areas. Should the goal however be to improve the accuracy of the capacitive sensors by correlating the probabilistic point clouds generated from the FEM simulation with measurements from depth cameras or similar three-dimensional sensors, then the configuration shifts from a competitive to a cooperative configuration. An example in which data from two sensors is combined in order to obtain previously unattainable information is the integration of thermal imaging into the global model. Three-dimensional sensors cannot perceive thermal information, since specialized imaging sensors are required to accomplish this task. On the other hand, thermal cameras have no way of perceiving depth, and stereoscopic configurations would be very difficult to realize due to the lack of clear edges and focus in thermal images, since heat inevitably bleeds into the surrounding areas. Hence, the combination of these two sensor types, which is easily realizable with the *SensorClouds* architecture, is a prime example of cooperative fusion, since these two types of sensors in combination produce previously unobtainable data in form of a thermal depth image.

## 8.3 Fulfillment of Requirements

In this segment of the evaluation, the fulfillment of the requirements imposed on the *SensorClouds* framework in Section 4.1 will be discussed.

### 8.3.1 Functional requirements

**FR 1 Integration of all sensor data into a global model**
All data input coming into a *SensorClouds* application is inserted into the global model. This includes any sensor modality or other type of information defined by the *Modules* configured for the developed application. The framework is also agnostic to the information contained within the data fields, meaning that raw data, as well as extracted features or decision markers can be processed equally. The only limitation to the amount of data fields that can be handled by the framework lies in the available memory on the processing hardware.

**FR 2 Real-time execution**
The results of the quantitative evaluation clearly show that the *SensorClouds* framework's performance is comparable to competing solutions while providing far more flexibility. The typical computational overhead caused by increased flexibility is eliminated by generating highly application specific code which is then compiled at application runtime.

**FR 3 Dynamic fusion of various sensor modalities**
*SensorClouds* enables the dynamic fusion of all sensor modalities, provided the data provided by the sensor contains three-dimensional coordinate information in order to locate the measurements taken in space. Should this information not be directly available, this work has also shown, using the example of capacitive sensors, how to reconstruct the three-dimensional information by means of physics simulations.

**FR 4 Modular and reusable architecture**
Since all the concrete algorithmic functionality is encapsulated within *Modules* in *SensorClouds*, the requirement of modularity is fully satisfied. Various extensions, such as to the ScQL query syntax, are possible with only minimal alterations to the framework's code base. Due to the fact that all data field definitions are kept abstract and the data type definitions for *Modules* are clearly defined following the design-by-contract paradigm the architecture and the *Modules* implemented for it are highly reusable. *Modules* can be applied to any data containing the required data fields, and differing types between *Module* and the provided data are automatically converted if possible.

**FR 5 Scalable Parallel Execution**
Through the use of the CUDA framework and the modularization of processing functions, the architecture and any applications developed with it are scalable on parallel architectures. Executing a *SensorClouds* application on a newer device with more CUDA cores can only lead to an increase in performance, since more data points can be processed at the same time. Architectural improvements of the CUDA hardware can also be taken advantage of, since even programs developed

for older versions of the hardware can still be executed on current hardware without issues and benefit from performance enhancements of the hardware without alteration. Since *Modules* are JIT-compiled at application runtime, improvements to the compiler or the specific architectural components of the hardware are taken advantage of directly.

**FR 6 Compatibility with established software ecosystems**
The input and output formats of *SensorClouds* are compatible with the point cloud definition of ROS and consequently are directly convertible to that of the PCL. Other drivers for input and output data can be added by extension of the appropriate interface classes. Aside from that, any application developed in C++ or with appropriate binding infrastructure (such as Python calling C++ code) can interface with a *SensorClouds* application directly. Hence, *SensorClouds* is fully compatible with any established ecosystems, such as ROS or the PCL.

### 8.3.2 Non-functional requirements

**NFR 1 Abstraction from parallel computing**
No module developer ever has to develop procedures for the parallel processing of data points, since the *SensorClouds* framework completely abstracts from these concepts and only requires module developers to specify the desired algorithm for processing a single data point. Everything required for the parallel execution of algorithms, data handling and transfer as well as dependency resolution is handled by the framework automatically.

**NFR 2 Programming paradigm**
No application developer must ever interface with raw sensor data directly when developing with *SensorClouds*. Applications are comprised of definitions of connected devices as well as insertion and processing modules.

**NFR 3 Keep with industry standards**
Great care was taken in the development of the *SensorClouds* architecture and its accompanying reference implementation to abstain from utilizing any external tooling other than what the standard build toolchain and the IDE provide out of the box. No external tooling is required in order to build a *SensorClouds* application. A CUDA compiler and the essential build tools are entirely sufficient.

**NFR 4 IDE support**
The reference implementation also went to great lengths to ensure that developers can always rely fully on the support and correct function of their IDE when developing applications with *SensorClouds*. Even when the definition of a concrete type cannot even exist at the time of development, the framework provides a static mock implementation a priori, e.g., for the use of *DataSets*, in order to enable the continued support by enhanced IDE analysis tools.

**Chapter Summary**. This chapter concludes this work by recapitulating the achievements presented throughout and provides an outlook on future developments.

# 9 Conclusion and Outlook

This work has presented *SensorClouds*, a modular and realtime-capable framework for the processing of multi-modal sensor data in applications for human-robot-collaboration. The architecture enables application developers to quickly and easily create programs employing complex sensor fusion methods with very little programming effort. Processing kernels provided by module developers can be arbitrarily combined in applications to create the desired output data. The interoperability of these modules is guaranteed through the enforcement of data access contracts following the design-by-contract paradigm. Data dependencies between modules are automatically resolved into the correct execution order and mismatched data types between different contract definitions are automatically converted if this is safely possible, thus increasing the compatibility of modules with various sensors and other modules.

The goal of creating a unified environment model as the basis for future robot actions was achieved by enabling the inclusion of sensors without inherent three-dimensional information in the three-dimensional global data model of the framework. Two-dimensional camera images can be projected into the coordinate space of the global data model and their data added at those points in the model which already contain three-dimensional information. One-dimensional sensors, such as e.g., capacitive or ultrasonic sensors, can also be added to the global model by applying the appropriate preprocessing and the results of a physics simulation determining the specific characteristics of the sensor model.

Retrieval of results and complex data selection within modules can be performed with the help of the ScQL query language for spatial and conditional filtering of sensor data contained in the global data model. Training of machine learning models can be almost fully automated by employing the ScML architecture, which supports the automatic data recording, training and prediction in supervised learning applications.

The results of the evaluation showed that, while far more complex in terms of the dynamic data model, the performance of the *SensorClouds* framework is competitive

with with the best performing alternative approach, which can, however, only process occupancy and no additional sensor information.

In summary, the *SensorClouds* framework is built upon a modular and highly flexible architecture which can easily by extended to provide additional functionality through processing modules. This flexibility does not impact the performance of the reference implementation, however, since all modules are JIT-compiled at runtime, reducing the overhead during execution since many parameters are known by the time the modules are compiled and thus regarded as constant.

Robot implementations will likely rely more and more heavily on data from multiple sensors in the future, in order to navigate and understand their surroundings and plan their future actions accordingly. *SensorClouds* can play an important role in the rapid and stable deployment of applications making heavy use of sensor fusion and data processing. Provided enough community interest can be generated, the framework could become an integral part in the development of such applications and offer a wide array of processing modules to application developers, easing the integration of sensor fusion even further.

Future possible advancements to the research presented in this work include the further development of the presented concept for capacitive sensor hardware in order to increase the detection range and resolution of capacitive sensors mounted on the robot. This also opens up new possibilities for sensor data processing, since the new generation of measurement circuits promises to be far less susceptible to noise. The *SensorClouds* architecture could be extended to allow even more flexibility in terms of modifying certain internal behaviors of the framework with special modules. This would also enable the definition of new ScQL query commands through the use of the appropriate module. The reference implementation of *SensorClouds* can be further optimized, which could lead to even better and more consistent performance.

# Bibliography


[1] Robots and robotic devices - safety requirements for industrial robots - part 1: Robots (iso 10218-1:2011), 2012. URL https://www.beuth.de/de/norm/din-en-iso-10218-1/136373717.

[2] Safety of machinery - safety-related parts of control systems - part 1: General principles for design (iso 13849-1:2015), 2015. URL https://www.beuth.de/de/norm/din-en-iso-13849-1/230387878.

[3] Robots and robotic devices - collaborative robots (iso/ts 15066:2016), 2016. URL https://www.beuth.de/de/vornorm/iso-ts-15066/250504228.

[4] Information technology — database languages sql — part 1: Framework (sql/framework) (iso/iec 9075-1:2023), 2023. URL https://www.iso.org/standard/76583.html.

[5] T. Adams and R. Layton. *Introductory MEMS: Fabrication and Applications*. Springer US, 2014. ISBN 9781489984210. URL https://books.google.de/books?id=B6PnrQEACAAJ.

[6] H. Alagi, S. E. Navarro, M. Mende, and B. Hein. A versatile and modular capacitive tactile proximity sensor. In *Proc. 2016 IEEE Haptics Symposium (HAPTICS)*, pages 290–296. IEEE, April 2016. doi: 10.1109/HAPTICS.2016.7463192.

[7] H. Alagi, A. Heiligl, S. E. Navarro, T. Kroegerl, and B. Hein. Material recognition using a capacitive proximity sensor with flexible spatial resolution. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6284–6290. IEEE, 2018.

[8] J. A. Albert, V. Owolabi, A. Gebel, C. M. Brahms, U. Granacher, and B. Arnrich. Evaluation of the pose tracking performance of the azure kinect and kinect v2 for gait analysis in comparison with a gold standard: A pilot study. *Sensors*, 20(18):5104, 2020.

[9] A. Aldoma, F. Tombari, R. B. Rusu, and M. Vincze. Our-cvfh–oriented, unique and repeatable clustered viewpoint feature histogram for object recognition and 6dof pose estimation. In *Pattern Recognition: Joint 34th DAGM and 36th OAGM Symposium, Graz, Austria, August 28-31, 2012. Proceedings 34*, pages 113–122. Springer, 2012.

[10] Analog Devices, Inc. Ad7147 data sheet. URL http://www.analog.com/media/en/technical-documentation/data-sheets/AD7147.pdf. (accessed on 09.08.2023).

[11] Artificial Intelligence Techniques. Open neural networks library, 2023. URL http://www.opennn.net/. (accessed on 09.08.2023).

[12] K. D. Backer, T. DeStefano, C. Menon, and J. R. Suh. Industrial robotics and the global organisation of production. 2018. doi: https://doi.org/https://doi.org/10.1787/dd98ff58-en. URL https://www.oecd-ilibrary.org/content/paper/dd98ff58-en.

[13] S. W. Badelt and A. P. Blaisdell. Capacitive sensors for detecting proximity and response. *Behavior Research Methods*, 40(2):613–621, 2008.

[14] Basler AG. Basler tof camera - user's manual. URL https://www.baslerweb.com/fp-1614239921/media/downloads/documents/discontinued_cameras/discontinued_cameras___manuals/AW00133810000_Tof_GigE_UM.pdf. (accessed on 09.08.2023).

[15] L. K. Baxter. *Capacitive sensors - Design and Applications*. IEEE Press Series on Electronics Engineering. IEEE Press New York, 1997.

[16] B. E. Bayer. Color imaging array, July 20 1976. US Patent 3,971,065.

[17] H. Becker. Robotik in der gesundheitsversorgung: Hoffnungen, befürchtungen und akzeptanz aus sicht der nutzerinnen und nutzer. *Pflegeroboter*, pages 229–248, 2018.

[18] A. Bertaux, M. Gueugnon, F. Moissenet, B. Orliac, P. Martz, J.-F. Maillefert, P. Ornetti, and D. Laroche. Gait analysis dataset of healthy volunteers and patients before and 6 months after total hip arthroplasty. *Scientific Data*, 9(1):399, 2022.

[19] Boston Dynamics. About boston dynamics. URL `https://bostondynamics.com/about/`. (accessed on 09.08.2023).

[20] British Cinematographer Magazine. Vicon and ilm partnership utilised to full effect on latest project: Lucasfilm's the mandalorian. URL `https://britishcinematographer.co.uk/vicon-and-ilm-partnership-utilised-to-full-effect-on-latest-project-lucasfilms-the-mandalorian/`. (accessed on 09.08.2023).

[21] Bundesministerium für Bildung und Forschung. Ergebnissteckbrief sina, 2020. URL `https://www.interaktive-technologien.de/service/ergebnissteckbriefe/ergebnissteckbriefe-ara/ergebnissteckbrief-sina`. (accessed on 09.08.2023).

[22] P. E. Caicedo, C. F. Rengifo, L. E. Rodriguez, W. A. Sierra, and M. C. Gómez. Dataset for gait analysis and assessment of fall risk for older adults. *Data in brief*, 33:106550, 2020.

[23] Capacitec, Inc. Capacitive sensors for non-contact displacement measurement. URL `https://www.capacitec.com/`. (accessed on 09.08.2023).

[24] D. D. Chamberlin and R. F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74, pages 249–264, New York, NY, USA, 1974. Association for Computing Machinery. ISBN 9781450374156. doi: 10.1145/800296.811515. URL `https://doi.org/10.1145/800296.811515`.

[25] X. Chen, T. Zhang, Y. Wang, Y. Wang, and H. Zhao. Futr3d: A unified sensor fusion framework for 3d detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 172–181, 2023.

[26] R. W. Clough. The finite element in plane stress analysis. *Proc. 2ˆ< nd> ASCE Confer. On Electric Computation, 1960*, 1960.

[27] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[28] COMSOL. Comsol multiphysics® simulation software. URL `https://www.comsol.com/comsol-multiphysics`. (accessed on 09.08.2023).

[29] J. Condit. 100 years of motion-capture technology. URL `https://www.engadget.com/2018-05-25-motion-capture-history-video-vicon-siren.html`. (accessed on 09.08.2023).

[30] J. R. Cook, N. A. Baker, R. Cham, E. Hale, and M. S. Redfern. Measurements of wrist and finger postures: a comparison of goniometric and motion capture techniques. *Journal of applied biomechanics*, 23(1):70–78, 2007.

[31] R. Cook, D. Malkus, M. Plesha, and R. Witt. *Concepts and Applications of Finite Element Analysis*. Wiley, fourth edition, 2001. URL `https://books.google.de/books?id=S1DczwEACAAJ`.

[32] S. Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[33] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms, fourth edition*. MIT Press, 2022. ISBN 9780262046305. URL https://books.google.de/books?id=drZNEAAAQBAJ.

[34] cppreference.com. Class template argument deduction (ctad) (since c++17), . URL https://en.cppreference.com/mwiki/index.php?title=cpp/language/class_template_argument_deduction&oldid=151592. (accessed on 09.08.2023).

[35] cppreference.com. Metaprogramming library (since c++11), . URL https://en.cppreference.com/mwiki/index.php?title=cpp/meta&oldid=156416. (accessed on 09.08.2023).

[36] Docker Inc. Docker - overview. URL https://docs.docker.com/get-started/. (accessed on 09.08.2023).

[37] R. Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, 1990. doi: 10.1109/2.44900.

[38] M. J. Egenhofer. Spatial sql: A query and presentation language. *IEEE Transactions on knowledge and data engineering*, 6(1):86–95, 1994.

[39] W. Elmenreich. An introduction to sensor fusion. *Vienna University of Technology, Austria*, 502:1–28, 2002.

[40] J. Fang, A. L. Varbanescu, and H. Sips. A comprehensive performance comparison of cuda and opencl. In *2011 International Conference on Parallel Processing*, pages 216–225. IEEE, 2011.

[41] R. Farber. *CUDA application design and development*. Elsevier, 2011.

[42] /Film. How titanic used motion capture technology before robert zemeckis obsessed over it. URL https://www.slashfilm.com/1190213/how-titanic-used-motion-capture-technology-before-robert-zemeckis-obsessed-over-it/. (accessed on 09.08.2023).

[43] M. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966. doi: 10.1109/PROC.1966.5273.

[44] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[45] FOGALE robotics. Sensitive surfaces for human / robot interactions & cooperation. URL https://www.fogale-robotics.com/pdf/Sensitive_surfaces.pdf. (accessed on 09.08.2023).

[46] Fraunhofer-Institut für Produktionstechnik und Automatisierung IPA. Fraunhofer ipa - wir produzieren zukunft - fraunhofer ipa. URL https://www.ipa.fraunhofer.de/. (accessed on 09.08.2023).

[47] C. Ge, B. Yang, L. Wu, Z. Duan, Y. Li, X. Ren, L. Jiang, and J. Zhang. Capacitive sensor combining proximity and pressure sensing for accurate grasping of a prosthetic hand. *ACS Applied Electronic Materials*, 4(2):869–877, 2022. doi: 10.1021/acsaelm.1c01274. URL https://doi.org/10.1021/acsaelm.1c01274.

[48] Google LLC. Tfx | ml production pipelines | tensorflow. URL https://www.tensorflow.org/tfx. (accessed on 09.08.2023).

[49] B. K. Gunturk, J. Glotzbach, Y. Altunbasak, R. W. Schafer, and R. M. Mersereau. Demosaicking: color filter array interpolation. *IEEE Signal processing magazine*, 22(1):44–54, 2005.

[50] D. Hall and J. Llinas. *Multisensor data fusion.* CRC press, 2001.

[51] M. Hamer. *Scalable localization and coordination of robot swarms.* PhD thesis, ETH Zurich, 2019.

[52] E. Hering, G. Schönfelder, S. Basler, K.-E. Biehl, T. Burkhardt, T. Engel, A. Feinäugle, S. Fericean, A. Forkl, C. Giebeler, et al. Geometric quantities. In *Sensors in Science and Technology: Functionality and Application Areas*, pages 147–372. Springer, 2022.

[53] A. Hermann, F. Drews, J. Bauer, S. Klemm, A. Roennau, and R. Dillmann. Unified gpu voxel collision detection for mobile manipulation planning. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4154–4160, 2014. doi: 10.1109/IROS.2014.6943148.

[54] A. Hoffmann, A. Poeppel, A. Schierl, and W. Reif. Environment-aware proximity detection with capacitive sensors for human-robot-interaction. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 145–150. IEEE, 2016.

[55] S. Hood, M. K. Ishmael, A. Gunnell, K. Foreman, and T. Lenzi. A kinematic and kinetic dataset of 18 above-knee amputees walking at various speeds. *Scientific data*, 7(1):150, 2020.

[56] K. Ikeuchi. *Computer Vision: A Reference Guide.* Computer Vision. Springer US, 2014. ISBN 9780387307718. URL `https://books.google.de/books?id=AILVXwAACAAJ`.

[57] Intel Corporation. Intel® realsense™ depth camera d435i. URL `https://www.intelrealsense.com/depth-camera-d435i/`. (accessed on 09.08.2023).

[58] JetBrains s.r.o. Clion - a cross-platform ide for c and c++. URL `https://www.jetbrains.com/clion/`. (accessed on 09.08.2023).

[59] R. Koeppe, D. Engelhardt, A. Hagenauer, P. Heiligensetzer, B. Kneifel, A. Knipfer, and K. Stoddard. Robot-robot and human-robot cooperation in commercial robotics applications. In *Robotics Research. The Eleventh International Symposium: With 303 Figures*, pages 202–216. Springer, 2005.

[60] O. Kosak, C. Wanninger, A. Hoffmann, H. Ponsar, and W. Reif. Multipotent systems: Combining planning, self-organization, and reconfiguration in modular robot ensembles. *Sensors*, 19(1), 2019. ISSN 1424-8220. doi: 10.3390/s19010017. URL `https://www.mdpi.com/1424-8220/19/1/17`.

[61] KUKA AG. Hrc in the production of the bmw group, . URL `https://www.kuka.com/en-de/industries/solutions-database/2017/06/solution-systems-bmw-dingolfing`. (accessed on 09.08.2023).

[62] KUKA AG. Technical data technical data - kr 16, . URL `https://www.kuka.com/-/media/kuka-downloads/imported/8350ff3ca11642998dbdc81dcc2ed44c/db_kr_16_en.pdf`. (accessed on 09.08.2023).

[63] KUKA AG. Many wrenches make light work: Kuka flexfellow will provide assistance during drive train pre-assembly, . URL `https://www.kuka.com/de-de/unternehmen/presse/news/2016/10/vw-setzt-auf-mensch-roboter-kollaboration`.

[64] T. Kuroda. *Essential principles of image sensors.* CRC press, 2017.

[65] M. G. Larson and F. Bengzon. *The finite element method: theory, implementation, and applications*, volume 10. Springer Science & Business Media, 2013.

[66] D. Lee. Samsung and lg go head to head with ai-powered fridges that recognize food. URL https://www.theverge.com/2020/1/2/21046822/samsung-lg-smart-fridge-family-hub-instaview-thinq-ai-ces-2020. (accessed on 09.08.2023).

[67] H.-K. Lee, S.-I. Chang, and E. Yoon. Dual-mode capacitive proximity sensor for robot application: Implementation of tactile and proximity sensing capability on a single polymer platform using shared electrodes. *IEEE sensors journal*, 9(12):1748–1755, 2009.

[68] Z. Liu, G. Xiao, H. Liu, and H. Wei. Multi-sensor measurement and data fusion. *IEEE Instrumentation & Measurement Magazine*, 25(1):28–36, 2022.

[69] S. Lupashin, M. Hehn, M. W. Mueller, A. P. Schoellig, M. Sherback, and R. D'Andrea. A platform for aerial robotics research and demonstration: The flying machine arena. *Mechatronics*, 24(1):41–54, 2014.

[70] I. Mamaev, D. Kretsch, H. Alagi, and B. Hein. Grasp detection for robot to human handovers using capacitive sensors. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 12552–12558, 2021. doi: 10.1109/ICRA48506.2021.9560970.

[71] A. Mashood, M. Mohammed, M. Abdulwahab, S. Abdulwahab, and H. Noura. A hardware setup for formation flight of uavs using motion tracking system. In *2015 10th International Symposium on Mechatronics and its Applications (ISMA)*, pages 1–6. IEEE, 2015.

[72] J. McConnell, C. Donnelly, S. Hamner, J. Dunne, and T. Besier. Effect of shoulder taping on maximum shoulder external and internal rotation range in uninjured and previously injured overhead athletes during a seated throw. *Journal of Orthopaedic Research*, 29(9): 1406–1411, 2011.

[73] K.-E. M'Colo, B. Luong, A. Crosnier, C. Néel, and P. Fraisse. Obstacle avoidance using a capacitive skin for safe human-robot interaction. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6742–6747. IEEE, 2019.

[74] P. Merriaux, Y. Dupuis, R. Boutteau, P. Vasseur, and X. Savatier. A study of vicon system positioning performance. *Sensors*, 17(7):1591, 2017.

[75] B. Meyer. *Design by contract*. Prentice Hall Upper Saddle River, 2002.

[76] Microsoft. Azure machine learning, . URL https://azure.microsoft.com/de-de/products/machine-learning/. (accessed on 09.08.2023).

[77] Microsoft. Spatial data, . URL https://learn.microsoft.com/en-us/sql/relational-databases/spatial/spatial-data-sql-server?view=sql-server-ver16. (accessed on 09.08.2023).

[78] P. Min. [binvox] 3d mesh voxelizer. URL https://www.patrickmin.com/binvox/.

[79] MLflow Project. Mlflow - an open source platform for the machine learning lifecycle. URL https://mlflow.org/. (accessed on 09.08.2023).

[80] R. Moheimani, P. Hosseini, S. Mohammadi, and H. Dalir. Recent advances on capacitive proximity sensors: From design and materials to creative applications. *Journal of Carbon Research*, 8(2), 2022. ISSN 2311-5629. doi: 10.3390/c8020026. URL https://www.mdpi.com/2311-5629/8/2/26.

[81] MRK-Systeme GmbH. Features safeinteraction. URL https://mrk-systeme.de/produkt/safeinteraction. (accessed on 09.08.2023).

[82] Neobotix GmbH. Neobotix: Homepage, . URL https://www.neobotix-robots.com/. (accessed on 09.08.2023).

[83] Neobotix GmbH. Mobile robot mpo-700, . URL https://www.neobotix-robots.com/products/mobile-robots/mobiler-roboter-mpo-700. (accessed on 09.08.2023).

[84] Open Robotics. Basic concepts - ros2 documentation. URL http://docs.ros.org/en/humble/Concepts/Basic.html. (accessed on 09.08.2023).

[85] Oracle. Spatial database. URL https://www.oracle.com/database/spatial/. (accessed on 09.08.2023).

[86] Oxford Metrics PLC. Software for smart sensing | oxford metrics. URL https://oxfordmetrics.com/about. (accessed on 09.08.2023).

[87] A. Pagoli, F. Chapelle, J.-A. Corrales-Ramon, Y. Mezouar, and Y. Lapusta. Large-area and low-cost force/tactile capacitive sensor for soft robotic applications. *Sensors*, 22 (11), 2022. ISSN 1424-8220. doi: 10.3390/s22114083. URL https://www.mdpi.com/1424-8220/22/11/4083.

[88] T. Parr. *The Definitive ANTLR 4 Reference [book version 2.0]*. Pragmatic, 2014. ISBN 978-1-93435-699-9. URL http://gen.lib.rus.ec/book/index.php?md5=f8750c0a556b8bd8767c80c36b93d9c3.

[89] A. Pfister, A. M. West, S. Bronner, and J. A. Noah. Comparative abilities of microsoft kinect and vicon 3d motion capture for gait analysis. *Journal of medical engineering & technology*, 38(5):274–280, 2014.

[90] A. Poeppel, A. Hoffmann, M. Siehler, and W. Reif. Robust distance estimation of capacitive proximity sensors in hri using neural networks. In *2020 Fourth IEEE International Conference on Robotic Computing (IRC)*, pages 344–351. IEEE, 2020.

[91] Point Cloud Library. Adding your own custom pointt type - point cloud library documentation, . URL https://pcl.readthedocs.io/projects/tutorials/en/master/adding_custom_ptype.html. (accessed on 09.08.2023).

[92] Point Cloud Library. Point cloud library (pcl), . URL https://github.com/PointCloudLibrary/pcl. (accessed on 09.08.2023).

[93] Point Cloud Library. Point cloud library, . URL https://pointclouds.org/. (accessed on 09.08.2023).

[94] V. Pomeroy, E. Evans, and J. Richards. Agreement between an electrogoniometer and motion analysis system measuring angular velocity of the knee during walking after stroke. *Physiotherapy*, 92(3):159–165, 2006.

[95] R. Porins and P. Apse-Apsitis. Capacitive proximity sensor maximum range dependence on geometry and size. In *2020 IEEE 8th Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, pages 1–4, 2021. doi: 10.1109/AIEEE51419.2021.9435768.

[96] T. Rauber and G. Rünger. *Parallel Programming: for Multicore and Cluster Systems*. Springer Berlin Heidelberg, 2013. ISBN 9783642378010. URL https://books.google.de/books?id=UbpAAAAAQBAJ.

[97] E. L. Reiss and S. Locke. On the theory of plane stress. *Quarterly of Applied Mathematics*, 19(3):195–203, 1961.

[98] Robert Bosch GmbH. Apas assistant inline: Durchgängiger apas assistant inline: Durchgängiger materialfluss ohne schutzzäune. URL https://s-dc-de.resource.bosch.com/media/de/press_release_1/2019_1/maerz_4/PI_010_19_APAS_Hannover_Messe_de.pdf. (accessed on 09.08.2023).

[99] Roboception GmbH. Der rc_visard stereosensor, 2023. URL https://roboception.com/de/rc_visard/. (accessed on 09.08.2023).

[100] ROS.org. Pcl overview. URL http://wiki.ros.org/pcl/Overview. (accessed on 09.08.2023).

[101] R. B. Rusu and S. Cousins. 3d is here: Point cloud library (pcl). In *2011 IEEE International Conference on Robotics and Automation*, pages 1–4, 2011. doi: 10.1109/ICRA.2011.5980567.

[102] N. Sakharnykh. Maximizing unified memory performance in cuda. URL https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/. (accessed on 09.08.2023).

[103] T. Schlegl and H. Zangl. Capacitive sensing for safety applications. In *Technologies for Smart Sensors and Sensor Fusion*, pages 259–282. CRC Press, 2014.

[104] T. Schlegl, T. Kröger, A. Gaschler, O. Khatib, and H. Zangl. Virtual whiskers—highly responsive robot collision avoidance. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5373–5379. IEEE, 2013.

[105] C. Scholl, A. Tobola, K. Ludwig, D. Zanca, and B. M. Eskofier. A smart capacitive sensor skin with embedded data quality indication for enhanced safety in human–robot interaction. *Sensors*, 21(21), 2021. ISSN 1424-8220. doi: 10.3390/s21217210. URL https://www.mdpi.com/1424-8220/21/21/7210.

[106] SCHUNK GmbH & Co. KG. Powerball leight weight arm lwa 4.6, mm-award 2012, deutsch, . URL https://www.youtube.com/watch?v=aqBR_J6P-pI. (accessed on 09.08.2023).

[107] SCHUNK GmbH & Co. KG. Schunk – kompetenzführer für spanntechnik, greiftechnik und automatisierungstechnik, . URL https://schunk.com/de/de. (accessed on 09.08.2023).

[108] J. Sevilla, L. Heim, A. Ho, T. Besiroglu, M. Hobbhahn, and P. Villalobos. Compute trends across three eras of machine learning. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2022. doi: 10.1109/IJCNN55064.2022.9891914.

[109] SICK AG. Capacitive proximity sensors cm, . URL https://www.sick.com/de/en/capacitive-proximity-sensors/capacitive-proximity-sensors/cm/cm30-16bap-kw1/p/p244255. (accessed on 09.08.2023).

[110] SICK AG. 2d-lidar-sensorenlms1xx / indoor, . URL https://www.sick.com/de/de/lidar-sensoren/2d-lidar-sensoren/lms1xx/lms100-10000/p/p109841. (accessed on 09.08.2023).

[111] A. Spector and D. Gifford. The space shuttle primary computer system. *Commun. ACM*, 27(9):872–900, sep 1984. ISSN 0001-0782. doi: 10.1145/358234.358246. URL https://doi.org/10.1145/358234.358246.

[112] A. Szczęsna, M. Błaszczyszyn, and M. Pawlyta. Optical motion capture dataset of selected techniques in beginner and advanced kyokushin karate athletes. *Scientific Data*, 8(1):13, 2021.

[113] R. E. Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2): 171–185, 1976.

[114] Teledyne FLIR LLC. Flir a-series. URL https://www.flir.com/products/a400-a700-science-kits/?model=85903-0102&vertical=rd+science&segment=solutions. (accessed on 09.08.2023).

[115] R. Tellez. A history of ros (robot operating system). URL https://www.theconstructsim.com/history-ros/. (accessed on 09.08.2023).

[116] E. Terzic, J. Terzic, R. Nagarajah, M. Alamgir, E. Terzic, J. Terzic, R. Nagarajah, and M. Alamgir. Capacitive sensing technology. *A Neural Network Approach to Fluid Quantity Measurement in Dynamic Environments*, pages 11–37, 2012.

[117] Texas Instruments Incorporated. Fdc2214 4-ch, 28-bit, capacitance to digital converter. URL https://www.ti.com/product/FDC2214. (accessed on 09.08.2023).

[118] The Khronos® Group Inc. Api adopters - opencl. URL https://www.khronos.org/conformance/adopters/conformant-companies#opencl. (accessed on 09.08.2023).

[119] Thinkbot Solutions LLC. Universal robots ur5 - medium-sized basic cobot. URL https://cdn.shopify.com/s/files/1/0253/6200/6096/products/Universal_Robots_UR5_1200px_800x.jpg?v=1572361438. (accessed on 09.08.2023).

[120] B. Thormundsson. Mobile cobots market volume worldwide from 2021 to 2030. URL https://www.statista.com/statistics/1378948/mobile-cobots-market-volume/. (accessed on 09.08.2023).

[121] M. J. Turner, R. W. Clough, H. C. Martin, and L. Topp. Stiffness and deflection analysis of complex structures. *journal of the Aeronautical Sciences*, 23(9):805–823, 1956.

[122] Velodyne Lidar, Inc. Ultra puck - proven, versatile, robust. URL https://velodynelidar.com/products/ultra-puck/.

[123] Vicon Motion Systems Ltd UK. Motion capture cameras | the full range from vicon, . URL https://www.vicon.com/hardware/cameras/. (accessed on 09.08.2023).

[124] Vicon Motion Systems Ltd UK. About us, . URL https://www.vicon.com/about-us/. (accessed on 09.08.2023).

[125] Vicon Motion Systems Ltd UK. Vicon x ilm vicon x ilm - breaking new ground in a galaxy far, far away., . URL https://www.vicon.com/resources/case-studies/vicon-x-ilm/. (accessed on 09.08.2023).

[126] R. Volpe and R. Ivlev. A survey and experimental evaluation of proximity sensors for space robotics. In *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, pages 3466–3473 vol.4, 1994. doi: 10.1109/ROBOT.1994.351037.

[127] W3C - RDF Data Access Working Group. Sparql 1.1 query language. URL https://www.w3.org/TR/sparql11-query/. (accessed on 09.08.2023).

[128] W3C - RDF Working Group. Rdf 1.1 concepts and abstract syntax. URL https://www.w3.org/TR/rdf11-concepts/. (accessed on 09.08.2023).

[129] J. Wallén. *The history of the industrial robot.* Linköping University Electronic Press, 2008.

[130] Y.-H. Weng and Y. Hirata. Ethically aligned design for assistive robotics. In *2018 IEEE International Conference on Intelligence and Safety for Robotics (ISR)*, pages 286–290, 2018. doi: 10.1109/IISR.2018.8535889.

[131] J. Wiles. What is new in artificial intelligence from the 2022 gartner hype cycle. URL https://www.gartner.co.uk/en/articles/what-is-new-in-artificial-intelligence-from-the-2022-gartner-hype-cycle. (accessed on 09.08.2023).

[132] D. J. Yeong, G. Velasco-Hernandez, J. Barry, and J. Walsh. Sensor and sensor fusion technology in autonomous vehicles: A review. *Sensors*, 21(6):2140, 2021.

[133] Z. Zhang. Active calibration. In K. Ikeuchi, editor, *Computer Vision - A Reference Guide*, Computer Vision, pages 1–5. Springer US, computer vision: a reference guide edition, 2014. ISBN 9780387307718. URL https://books.google.de/books?id=AILVXwAACAAJ.

[134] Z. Zhang. Camera model. In K. Ikeuchi, editor, *Computer Vision - A Reference Guide*, Computer Vision, pages 77–80. Springer US, computer vision: a reference guide edition, 2014. ISBN 9780387307718. URL https://books.google.de/books?id=AILVXwAACAAJ.

[135] Z. Zhang. Perspective camera. In K. Ikeuchi, editor, *Computer Vision - A Reference Guide*, Computer Vision, pages 588–589. Springer US, computer vision: a reference guide edition, 2014. ISBN 9780387307718. URL https://books.google.de/books?id=AILVXwAACAAJ.

[136] P. Zhao, C. X. Lu, B. Wang, N. Trigoni, and A. Markham. 3d motion capture of an unmodified drone with single-chip millimeter wave radar. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5186–5192. IEEE, 2021.

# List of Figures

# Listings

# Own Publications


[1] D. Bermuth, A. Poeppel, and W. Reif. Scribosermo: Fast speech-to-text models for german and other languages. *arXiv preprint arXiv:2110.07982*, 2021.

[2] D. Bermuth, A. Poeppel, and W. Reif. Finstreder: Simple and fast spoken language understanding with finite state transducers using modern speech-to-text models. *arXiv preprint arXiv:2206.14589*, 2022.

[3] D. Bermuth, A. Poeppel, and W. Reif. Jaco: An offline running privacy-aware voice assistant. In *2022 17th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pages 618–622. IEEE, 2022.

[4] B. Eberhardinger, A. Habermaier, A. Hoffmann, A. Poeppel, and W. Reif. Toward integrated analysis & testing of component-based, adaptive robot systems. In *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 301–302. IEEE, 2016.

[5] C. Eymüller, J. Hanke, A. Hoffmann, A. Poeppel, C. Wanninger, and W. Reif. Towards a real-time capable plug & produce environment for adaptable factories. In *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–4. IEEE, 2021.

[6] C. Eymüller, J. Hanke, A. Poeppel, and W. Reif. Towards self-configuring plug & produce robot systems based on ontologies. In *2023 9th International Conference on Automation, Robotics and Applications (ICARA)*, pages 23–27. IEEE, 2023.

[7] M. Filipenko, A. Poeppel, A. Hoffmann, W. Reif, A. Monden, and M. Sause. Virtual commissioning with mixed reality for next-generation robot-based mechanical component testing. In *ISR 2020; 52th International Symposium on Robotics*, pages 1–6. VDE, 2020.

[8] J. Hanke, C. Eymüller, A. Poeppel, J. Reichmann, A. Trauth, M. Sause, and W. Reif. Sensor-guided motions for robot-based component testing. In *2022 Sixth IEEE International Conference on Robotic Computing (IRC)*, pages 81–84. IEEE, 2022.

[9] A. Hoffmann, A. Poeppel, A. Schierl, and W. Reif. Environment-aware proximity detection with capacitive sensors for human-robot-interaction. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 145–150. IEEE, 2016.

[10] A. Poeppel, C. Eymüller, and W. Reif. Sensorclouds: A framework for real-time processing of multi-modal sensor data for human-robot-collaboration. In *2023 9th International Conference on Automation, Robotics and Applications (ICARA)*, pages 294–298. IEEE, 2023.

[11] A. Poeppel, A. Hoffmann, M. Siehler, and W. Reif. Robust distance estimation of capacitive proximity sensors in hri using neural networks. In *2020 Fourth IEEE International Conference on Robotic Computing (IRC)*, pages 344–351. IEEE, 2020.

[12] M. Schörner, C. Wanninger, R. Katschinsky, S. Hornung, C. Eymüller, A. Poeppel, and W. Reif. Uav inspection of large components: Determination of alternative inspection points and online route optimization. In *2023 IEEE/ACM 5th International Workshop on Robotics Software Engineering (RoSE)*, pages 45–52. IEEE, 2023.

[13] M. Stüben, A. Poeppel, and W. Reif. External torque estimation for mobile manipulators: A comparison of model-based and lstm methods. In *2022 Sixth IEEE International Conference on Robotic Computing (IRC)*, pages 95–102. IEEE, 2022.

[14] C. Wanninger, S. Rossi, M. Schörner, A. Hoffmann, A. Poeppel, C. Eymüller, and W. Reif. Rossi a graphical programming interface for ros 2. In 2021 *21st International Conference on Control, Automation and Systems (ICCAS)*, pages 255–262. IEEE, 2021.