TECHNISCHE UNIVERSITÄT CHEMNITZ

Faculty of Computer Science

Media Computing

Bachelor's Thesis

Object Localization from 3D Point Clouds

Richard Siegel

Chemnitz, March 19, 2018

Examiner: Dr. Danny Kowerko
Supervisor: Dr. Danny Kowerko

Siegel, Richard
Object Localization from 3D Point Clouds
Bachelor's Thesis, Faculty of Computer Science
Chemnitz University of Technology Chemnitz, March 2018

Acknowledgement

I would like to thank Dr. Danny Kowerko for giving me the opportunity to write my thesis at the junior professorship Media Computing and letting me thereby contribute to the localizeIT project.

For his advice and support, I thank Robert Manthey.

I would also like to thank the Intenta GmbH for supplying enhanced software interfaces for the professorship, without which the work on this thesis would not have been possible.

Last but not least, I want to thank my fiancée Elvira for giving me strength and my parents for all their years of efforts and support for my education.

Contents

List of Figures

List of Tables

1. Introduction

The audiovisual laboratory of the junior professorship Media Computing at Chemnitz University of Technology, as described in [HMH+], was designed to develop and improve acoustic source localization with microphone arrays (described in [ZHK17]). At the core of the laboratory is a frame cage ($L \times W \times H = 4.26 \times 3.756 \times 3.497m$) inside of which sensors are installed and sound sources are placed. To support development and evaluation of acoustic localization methods, a prototype software for the localization of loudspeakers within the frame cage in the laboratory environment must be designed, utilizing the available optical 3D-sensors. This software is designed, implemented and evaluated in this thesis. In case the optical system is reliable and precise enough, it may serve as a ground truth for tests and evaluations of the acoustic localization system in development.

Whether the optical system is capable of serving as the ground truth for the evaluation of the acoustic localization is dependent on the accuracy of positions measured by the optical system and the level of accuracy being tested acoustically. To effectively support series of tests, a program for the optical localization of loudspeaker must be capable of exporting several object positions of a test setting at once. The localization software prototype, created in this thesis, provides documentation of the laboratory test settings by generating a basic, exportable, 3D-model of the laboratory and object positions. As part of the user interface a model is constantly providing visualization of the available data and manipulations applied to it.

An essential requirement for the creation of the software prototype is utilizing existing laboratory equipment. Besides audio technology the laboratory is equipped with 10 optical Intenta S2000 smart sensors. These sensors are highly specialized in tasks such as access control and generation of statistical data[1]. However, due to the support of Intenta GmbH for the junior professorship Media Computing, images and point clouds[2] created for sensor internal operations are available for development of the prototype software.

[1] Information about the S2000 is found in a brochure provided by the sensor developer Intenta: `https://www.intenta.de/files/inhalt/de/sensor-systeme/brochure-INTENTA-S2000-EN-20160524.pdf`

[2] Point clouds represent 3D-positions. Details about the S2000 point cloud are found in 2.1. For further information about point clouds form optical sensors see 3.1 and 3.2.

2. Laboratory, Stereo Sensor and API

This chapter introduces the hardware available for software development. The optical sensors and software requirements for working with their sensor API are explained in section 2.1. An example program for streaming connections to the sensor (supplied by Intenta) is presented in section 2.2. Lastly, in section 2.3, the audiovisual laboratory itself and the computer system for development and its configuration, along side the installed connection to the sensors, is described.

2.1. The Intenta S2000

The S2000 sensor is an intelligent monitoring sensor developed for tasks such as room surveillance, access control and recognition of potentially dangerous situations, e.g. analyzing human behavior in save areas for working with robots [BPS17]. The sensor is capable of counting people and differentiating them by size. Dimensions of the S2000 (presented in fig. 2.1) measure $200 \times 70 \times 33mm$, while it weights ca. $500g$ and has a maximal power consumption of $5W$. To connect to the sensors their Ethernet port can be used, which is also capable of supplying the power needed for the S2000 to operate via Power over Ethernet. The recommended mounting height is $2,50 \rightarrow 7m$ above the ground.[1]

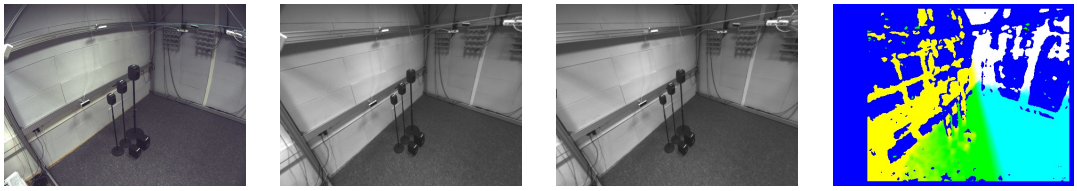
Internally the S2000 creates a 3D point cloud of its environment based on its two cameras with a 97^o field of view [REK18]. 2D-Images recorded with the sensor consist of 512×384 or 1280×960 distinguishable picture elements, pixels [Gra99, p. 569][2]. As illustrated in fig. 2.2 the sensor provides an unprocessed color image, two calibrated gray-scale images and point cloud values stored within an image structure. For each sensor camera a calibrated images is generated (see fig. 2.2b and 2.2c). The image structure containing the point cloud information includes the coordinates of every calibrated left camera image pixel in 3D-space. To represent the 3D-coordinates of the point cloud in the image (visible in fig. 2.2d) the three coordinate values are stored in place of the three color values, red, green and blue of an RGB-color image. Consequently the image structure of the point cloud maps the 2D-positions of pixels inside the image from the left sensor camera (fig. 2.2b) into a 3D-coordinate system

[1] This information is based on the S2000 brochure by Intenta: `https://www.intenta.de/files/inhalt/de/sensor-systeme/brochure-INTENTA-S2000-EN-20160524.pdf`

[2] Detailed information about the fundamentals of 2D digital image representation is found in chapter two of [Jä13].

Power Supply	Power over Ethernet (IEEE 802.3af)	**Mounting Height**	2,50 – 7m
Power Consumption	max. 5W	**Enclosure**	Aluminium
EMC	EN55022	**Degree of Protection**	IP 20, optional IP 65
Ethernet	1000MBit, RJ45	**Dimensions (L x W x H)**	200 x 70 x 33mm
Switched In-/Outputs	4 digital IOs	**Weight**	ca. 500g
USB Port	USB 2.0, Type Host	**WiFi (optional)**	802.11b/g
Serial Port	2-Wire RS485	**Logical Protocols**	Message API, CSV, RTP, RTSP, FTP, HTTP, HTTPS, SMTP...

Figure 2.1.: The S2000 smart sensor by the Intenta GmbH (at the top) and its specifications (at the bottom). The circular gabs in the black layer, behind the glass covering the sensors face, mark the positions of its cameras. Image and table in this figure have been adopted from the Intenta S2000 brochure, available at `https://www.intenta.de/files/inhalt/de/sensor-systeme/brochure-INTENTA-S2000-EN-20160524.pdf`.



(a) 1280 × 960 pixels. The raw RGB-color image form the left sensor camera.

(b) 512 × 384 pixels. The calibrated gray-scale image from the left sensor camera.

(c) 512 × 384 pixels. The calibrated gray-scale image from the right sensor camera.

(d) 512 × 384 pixels. The RGB-color representation of the sensor point cloud.

Figure 2.2.: Four images extracted from the S2000 sensor via the SVP2-Protocol, visualized with OpenCV. Fig. 2.2d is the color representation of coordinate values transmitted inside an image structure, where the red, green and blue values are x, y and z coordinates of the point cloud. The other three images display regular color and brightness (gray-scale) images.

when the color values of the point cloud images equal 2D-position (fig. 2.2d) are interpreted as 3D-coordinates. Likewise, every point cloud pixel position in 3D-space is stored with the same index as the corresponding brightness value inside the left sensor camera image. The common index is determined by the common image size of 512 × 384 for both, left sensor image and point cloud. Due to the ability of the calibrated left camera image (fig. 2.2b), it is also referred to as point cloud texture in this thesis.

To configure the S2000 sensors an HTTP interface, protected by logging credentials, is used. Any conventional modern web browser, on a computer connected to the sensor via Ethernet, may be used to connect to this sensor interface by navigating to the preset static IP address incorporated into the standard configuration of the S2000. Security relevant settings, like the logging credentials and the IP configuration are adjustable once logged in. Automatic sensor calibration and complex functionaries of the sensor, including the detection and counting of people within areas of the sensors wide field of view, are also configurable through the browser. The images and point cloud, currently streaming from the sensor, are visible among the configuration menus and visually support the effect of changes applied to settings. Extracting the raw point cloud data from the sensor is currently not possible for the general Intenta costumer, since the S2000 was designed primly to supply its users with statistical, non visual data.

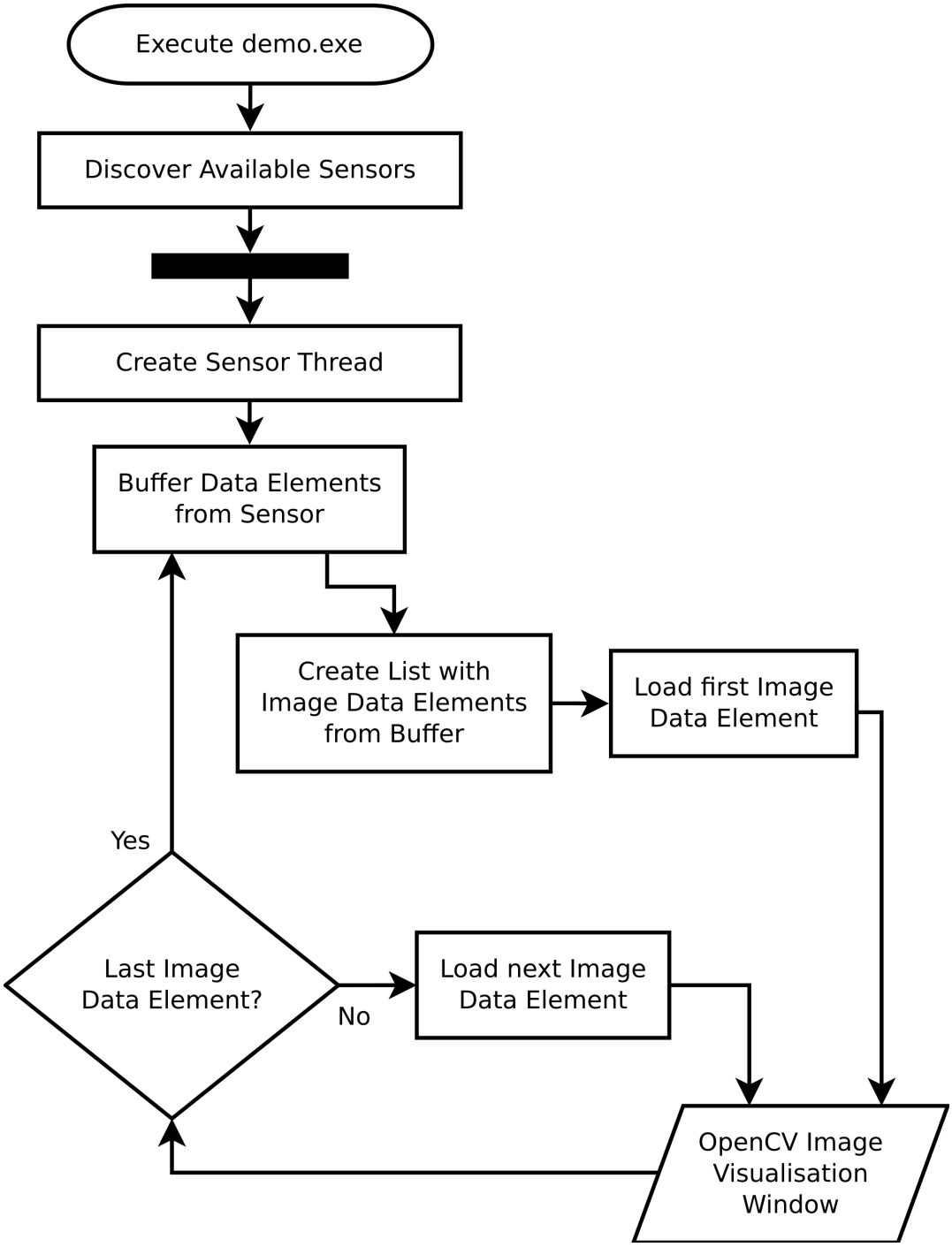


Figure 2.3.: This flowchart illustrates the essential algorithm of the SVP2_API example "Multisensorconnection". Execution begins on the top left. Once running, the program does not terminate by itself, it constantly refreshes the visualized sensor data (symbolized on the bottom right). The black bar above "Create Sensor Thread" indicates parallelism. Thus, one thread for each sensor.

2.2. S2000 API and Multisensorconnection Demo

To gain access to sensor data and the point cloud image representation, Intenta utilizes the SVP2-Protocol. Due to the partnership of the Intenta GmbH with the junior professorship Media Computing through the project localizeIT[3], the professorship was permitted to use selected tools, which provide access to the S2000 for development. The Intenta program SVPManager, similar to the HTTP interface of the sensor, allows to configure sensor settings. However, unlike the regular sensor interface the SVPManager provides an option to change the control mode of the S2000, such that the point cloud is transmitted via the SVP2-Protocol. Once the sensors control mode was changed appropriately, the Intenta SVP2_API enables access to images and point cloud data through c++ code. The most essential part of the SVP2_API is the *Sensor* class, it acts as a mediator, data from the sensor is stored and extracted from it. In order to use the *Sensor* class for data extraction, the API supports several phases needed to initiate actual sensor data access, such as, automatic sensor detection, connecting to sensors, starting a streaming client and loading available sensor data frame-wise. To illustrate and test the capabilities of the API, Intenta created a demo-project which makes use of the OpenCV library (see 3.3) to display the images, and point cloud image representation, as they are provided by the sensor. The simplified structure of this API example, called "Multisensorconnection", is illustrated in fig. 2.3 as a flowchart. After discovering the available sensors, a thread for each sensor is created. Within these threads a streaming connection to the sensor is running inside loops. New available sensor data is repeatedly buffered and the image elements contained inside every such buffered data set is displayed. If the appropriate control mode is selected for each sensor within the SVPManager, images similar to the examples in fig. 2.2 are visualized for each sensor by the API example. To connect to a specific sensor, an argument, consisting of protocol and sensor IP is added, when the program is executed from Microsoft's command line (e.g. "demo.exe svp2://192.168.10.52"). The default development environment for the SVP2_API is Visual Studio 15 2017 for 64-Bit Microsoft Windows installations. To compile the program, OpenCV must be installed. Version 3.4.0 of OpenCV supports the example.

Due to the complexity of the API and limited availability of documentation outside of the Intenta GmbH, the *Multisensorconnection API Demo* is utilized as the foundation of the prototype software developed in this thesis. It does not include functionality for 3D-visualization of point clouds or provide assistance for general localization tasks. However, it provides a constant stream of buffered sensor data, including the current point cloud image. The integration of OpenCV into the *Multisensorconnection API Demo* for the visualization simplifies the implementation of additional OpenCV functionality (see 3.3).

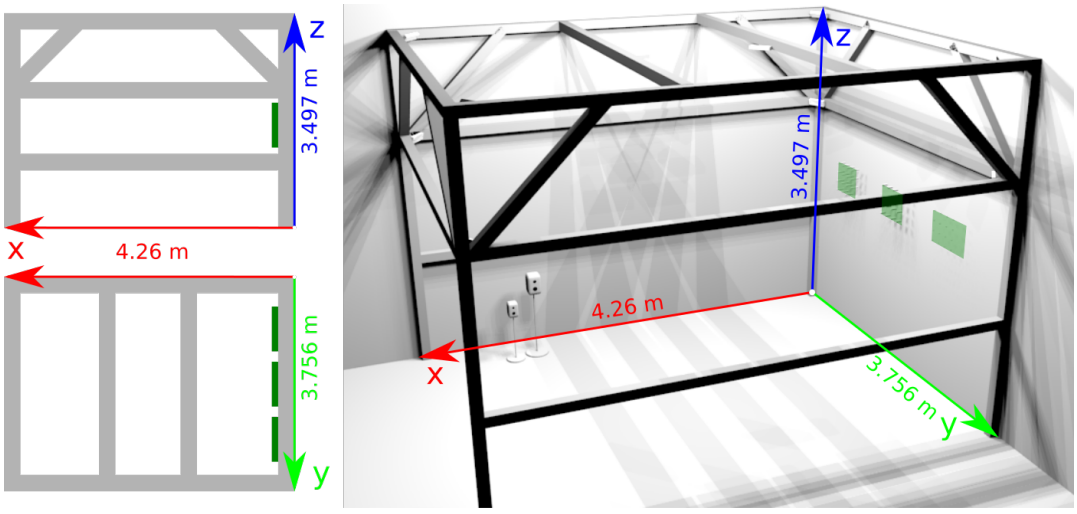
[3]Find more information about the localizeIT initiative at `https://localize-it.de/initiative/`

2.3. Laboratory Hardware Setup

For evaluations of acoustic localization methods the laboratory includes 16 loudspeakers, as specified in tab. 2.1, to act as sound sources. The five different types of loudspeakers are also visible in fig. 2.4a. The Genelec models are installed on stands with adjustable height; TANNOY loudspeakers remain on the ground. An optical localization system must be able to determine position areas of these five types of sound sources within several positions in the laboratory. Fig. 2.4b presents a schematic plan and a 3D-model of the laboratory frame cage, which illustrates the coordinate system used to describe object positions (e.g. loudspeaker positions). Object positions consist of three floating point number values in meters, one for each axes of the coordinate system. The frame cage installed in the laboratory with the dimensions $L \times W \times H = 4.26 \times 3.756 \times 3.497m$ is where the sensors are installed and the experiments conducted. Fig. 2.4 illustrates illustrates this frame cage.

The sole optical data source in the audiovisual laboratory, capable of producing a streams including 3D-data, is the S2000. Thus, the development environment for the creation of the prototype software is largely determined by the exclusive SVP2_API for the Intenta sensor. Hence, a 64-Bit installation of Microsoft Windows 8.1 Pro, running Visual Studio 15 2017 and OpenCV 3.4.0 for C++ development with the SVP2_API, is utilized. Additionally the opencv_contrib (see 3.3) module and OpenGL 4.6.0 (see 3.4) were installed during the curse of development. The Computer running the software is equipped with a 3.40GHz Intel(R) Core(TM) i7-4770 CPU, 32GB of RAM and an NVIDIA GeForce GT 740 graphics card. The 10 Intenta sensors installed inside the frame cage are connected to the development computer via Ethernet. two netowrk switches are utilized to combine the signals from all 10 sensor connection cables into the single connection cable plugged into the computer. The netowrk switches are also connected to an Ethernet power adapter and supply the power from the adapter to the sensors. Static IPs have been configured for the development computer and every connected S2000. The sensors are installed at about 2.45 m above the ground with different view angles.

(a) This photograph presents the testing area inside the frame cage. All types of laboratory loudspeakers are visible on the center left. The Genelec 8010 AP, 8020 CPM and 8030 BPM (installed on stands) and below them, the TANNOY Reveal 402 and 502 are visible in that order, form left to right. Furthermore several S2000 sensors are visible, one installed directly above the loudspeakers.



(b) The 3D-model on the right gives an overview of the frame cage visible in fig. 2.4a. On the left a schematic side view (top) and top view (bottom) of the cage illustrate, how object positions should be measured. The models in this representation are not to scale.

Figure 2.4.: The frame cage inside the audiovisual laboratory. The red, green and blue arrows represent the dimensions of the cage along the x, y and z axes, $4.26 \times 3.756 \times 3.497m$. Fig. 2.4a highlights the position of the axes inside a photograph of the frame cage, while fig. 2.4b illustrates the axes positions using models. The coordinate origin is the intersection of the illustrated arrows in the corner of the laboratory. Microphone arrays positions are highlighted with green quadrangles.

Type	Genelec 8010 AP	Genelec 8020 CPM	Genelec 8030 BPM	TANNOY Reveal 402	TANNOY Reveal 502
Quantity	2	8	2	2	2
Watt	25	20	40	50	75
Hz	74 - 20000	66 - 20000	58 - 20000	56 - 48000	49 - 43000
Length (mm)	116	142	178	212	238
Width (mm)	121	151	189	147	184
Height (mm)	180	227	284	240	300

Table 2.1.: Technical specifications of the laboratory loudspeakers. Addopted and modified from [REK18]

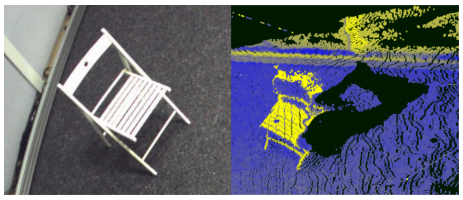
3. Fundamental 2D and 3D Processing Concepts and Libraries

In this chapter the concepts and libraries necessary for the development of the object localization and representation, based on point clouds, are described. Methods for processing 2D-images and 3D point clouds are considered and combined in 3.1, 3.2 and 3.3. The visualization tools for 3D-models with point clouds and other 3D-geometry is explained in 3.4. 3D-model interchange file formats are introduced in 3.5. Lastly, 3.6 explains a transformation method which synchronizes 3D-models such as 3D point clouds with a reference coordinate system.

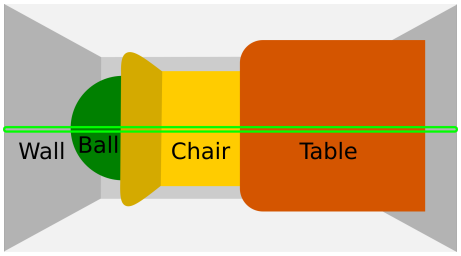
3.1. Principles of Optical Localization from 2D to 3D

Object localization through image processing is a demanded in many research areas, such as media retrieval, robotics, virtual reality and the development of surveillance systems. To find object positions inside 2D-images several algorithms exist. Such algorithms rely on feature extraction operations applied to the image, inside of which objects are located. Binarization techniques similar to the described one in [SP00], and edge detection algorithms like canny edge (described in [Can83]) are used in feature detection. Based features inside an image, which is being searched for an object, characteristics of the object are compared to the image. The Surf-Algorithm [GDP10] is a popular detection algorithm, which uses these principles. It is capable of locating a reference image inside an images, distorted by perspective. In controlled environments optical markers simplify the detection of objects. These markers are designed to be easily and robustly recognizable by simple feature detection algorithms. One type of such optical markers is discussed in greater detail in 3.3.2.

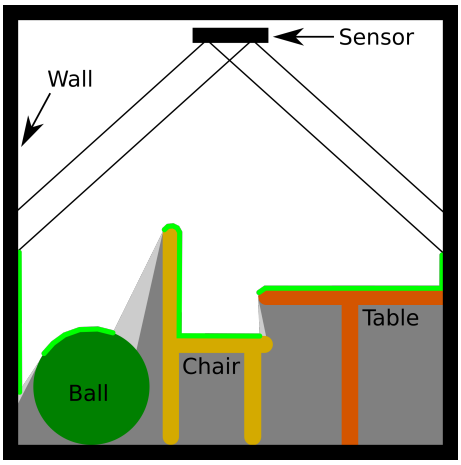
Apart form localization within 2D-images, stereo camera systems enable 3D-position estimation through triangulation. Similar to human two eyed vision, two cameras, installed at a specified distance to each other, record parallel images in stereo vision systems. The cameras of the Intenta S2000 (see fig. 2.1) are an example for a conventional stereo vision camera setup. The object positions on 2D-images is recorded side by side, similar to human, two eyed, vision, and the distance value between the cameras enable the calculation of distance to the stereo camera as described in [MV08]. Since the precise calculation of 3D-positions depends on accurate 2D-location detection, the S2000 sensor calibrates its wide angle camera images (see



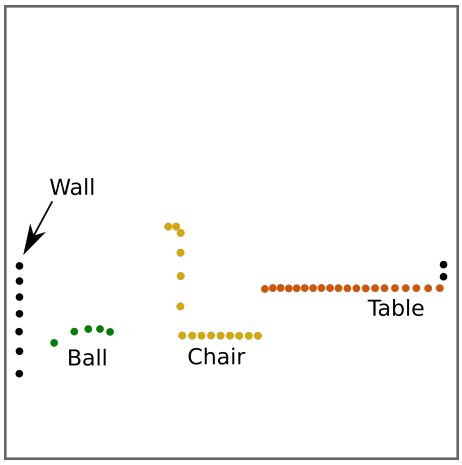
(a) The image on the left is the sensors view. On the right side a shadow like section, on the floor, and missing parts on the back side of the chair, illustrate stereo sensor artifacts. The brightness value of vertices in the point cloud is represented in yellow and blue. The black areas do not contain vertices.



(b) A schematic sensor view towards an example setting. The light green line highlights the section of the sensors view, which is further explained in fig. 3.1c and results in the point cloud vertices depicted in fig. 3.1d.



(c) A side view towards the setting illustrated in fig. 3.1b. The light green line highlights the visibility, to the sensor, and is visible for the sensor perspective in fig. 3.1b. The gray areas are invisible to the sensor cameras.



(d) The point cloud section corresponding to fig. 3.1c. Only the sections of objects, visible to the sensor cameras (as in fig. 3.1b) are represented in the point cloud. Significant parts of objects are lost and lead to misrepresentation.

Figure 3.1.: The technical limitations of point clouds from optical stereo camera sensors. Fig. 3.1a presents an example of characteristic artifacts in stereo sensor point clouds. In fig. 3.1b, 3.1c and 3.1d a schematic explanation for stereo sensor point cloud computation, explaining the artifact, is illustrated.

fig. 2.2b). The distortion caused by the wide angle lens, as visible in fig. 2.2a, would otherwise not allow the triangulation of accurate 3D-positions.

By applying the distance triangulation to every detected point, which occurs in both stereo images, a point cloud is calculated by the S2000. A more popular 3D point cloud sensor based on stereo cameras is the Microsoft Kinect, which uses an infrared laser (unlike the S2000) emitter to project point patterns into the room for more effective stereo point matching. Research about the Kinect shows, that the accuracy of distance estimations decreases noticeable with the distance from the sensor. At 5 m distance, 4 cm of random error occur. [Kho11]

Points in 3D-space, inside a point cloud, are also called vertices, like 3D-positions connected to complex geometry (see 3.4.2). Apart from technical limitations concerning the accuracy, stereo sensor point clouds localize only vertices originating from the pixels within the stereo camera images (as previously described). This results in characteristic artifacts, blank spots within the 3D-model, created based on point cloud data. Fig. 3.1 illustrates this problem. The two cameras of a stereo sensor have very similar views as visible in fig. 2.2. One of the camera images is used to assign brightness values to the vertices as described above and is considered the sensors main view towards the recorded scene. Fig. 3.1a presents such a main view of the sensor and blank areas within the point cloud from the same sensor as they are visible from perspectives other than the main views. A hypothetical main view is depicted in fig. 3.1b and only a slice of its resulting point cloud (marked in light green) is illustrated in fig. 3.1d. The three example objects, a ball, a chair and a table, appear misrepresented, when compared to the schematic side view of objects and sensor in fig. 3.1c. Visible in this side view, are the objects as, ideally, they should be described in any recorded 3D-model of the room including point clouds. However, only the object parts visible to the sensor (highlighted in light green in fig. 3.1c) are presented in the point cloud. Notice, that potentially undesired objects, like the walls of a room will be represented in point clouds as illustrated in fig. 3.1d. For the localization of objects, based on point cloud data, this means, that not the actual object position, but rather the position of object parts, visible within the sensors view, is directly extractable form point clouds. Point clouds, like images, do not contain information about which parts of them represent a specific object. The mapping from vertex to represented object must be generated with additional algorithms, similar to object detection and localization in 2D-images. However, even if 2D-image algorithms for object localization are utilized, with precalculated point clouds associated to the 2D-images, mapping of 2D-positions to 3D-space is greatly simplified.

3.2. Localizing Objects within Point Clouds

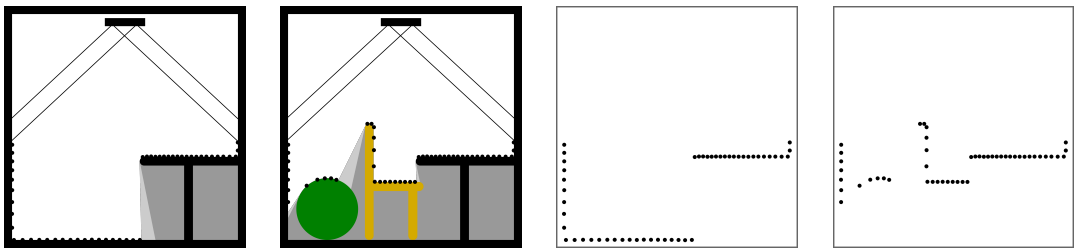
As described in 3.1 vertices within a point cloud are not grouped into objects and similar to 2D-images point clouds usually contain background information like symbolized by the room and primarily its walls in fig. 3.1. Additionally stereo sensors usually dislocate a few vertices, which than appear to be in random positions within the point cloud data. A common tool for to overcome the mentioned problems and further analyze point cloud data is the Point Cloud Library, PCL.[1] As the name suggests the open source project, PCL was created for processing point clouds. Filtering points, reconstructing surfaces and object shapes or the creation of Range images based on point clouds, the library features algorithms for a wide range of operations based on point cloud data. PCL is implemented as a C++ library with full support for the Robot Operating System, ROS. It is commonly used in robots, which utilize point cloud sensors to analyze their environment. To keep the implementation of point cloud passing algorithms compact, PCL uses a processing pipeline. [RC11]

The pipeline interface of PCL (as described in [RC11]) is structured in the following matter:

1. Creation of a processing Object such as filters, feature estimation, surface reconstruction or segmentation.
2. Definition of the input cloud.
3. Configuration of various parameters.
4. Calling *compute, filter, segment* or similar functions to create the output.

However, the complexity of point cloud library and its abilities far exceed the requirements for the creation of a prototype software for object localization from 3D point clouds. The data structure native to PCL is not supported by the current Intenta SVP2_API available for development. Thus, the conversion of point cloud vertices into the PCL structure must be implemented to utilize the library on live streams from the sensor. Such a self implemented conversion would require testing against non modified point cloud data, to avoid failures within conversion. Hence, only after the implementation of a PCL free prototype software for the S2000 (to test against), potentially arising problems within a PCL based software (for the S2000) could be found. Alternatively a point cloud could be exported into the PCD (Point Cloud Data) file format native to PCL, but this would hinder an effective prototype software development process, as no direct, quick, live testing of scenario changes would be possible.

[1] Point Cloud Library is well documented at `http://pointclouds.org/documentation/`

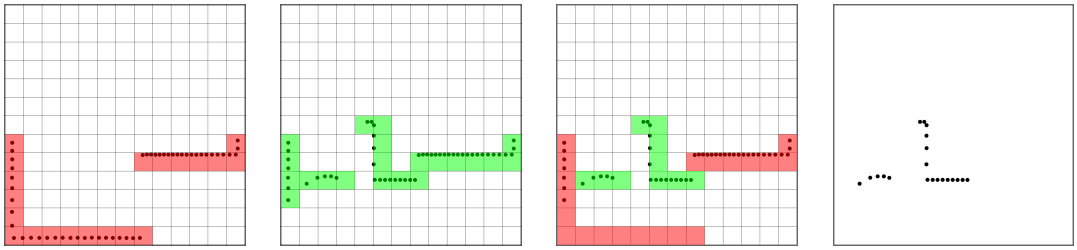


(a) An Empty Room Setting

(b) A Room with Objects

(c) Point Cloud of 3.2a

(d) Point Cloud of 3.2b

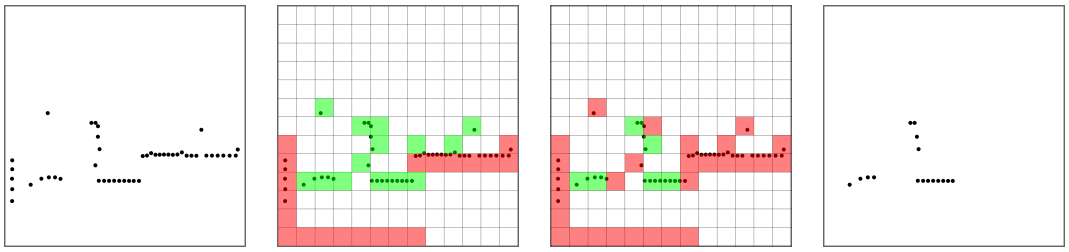


(e) Mask from 3.2a Vertex Areas

(f) The Vertex Areas from 3.2b

(g) Mask from 3.2e Applied to 3.2f

(h) Resulting Point Cloud after 3.2g



(i) Point Cloud of 3.2b with Noise

(j) Mask from 3.2e Applied to 3.2i

(k) Density Filter Applied to 3.2j

(l) Resulting Point Cloud after 3.2k

Figure 3.2.: The concept of masks and density filters for the localization in point clouds is illustrated in this figure. Similar to fig. 3.1 a 2D-section of example point clouds is schematically illustrated. Highlighted in light green are areas containing unmasked and unfiltered point cloud vertices. Areas highlighted in red are masks and filters, which mark where vertices are removed. Fig. 3.2a, 3.2c and 3.2e illustrate mask creation. In fig. 3.2b, 3.2d and 3.2f the primary point cloud is created. Most remaining subfigures illustrated how masks and filters are applied.

A simplified approach to the localization of objects within the point cloud is to separate the cloud into subareas which are than further analyzed. This has similarities to the first step of the You Only Look Once approach for detecting objects in 2D-images (presented in [RDGF16]), where the classification of subareas, created by a 2D-grid, is used to determine more advanced bounding boxes. While 2D-images are divided into squared subareas by 2D-grids, the vertices of a 3D point cloud are divided into cubed subareas by 3D-grids. Nevertheless, in order to simplify the illustration in the schematic example fig. 3.2, the 3D-cubes are symbolized by squares. Small 3D-subareas are desirable, as they increase the resolution of positions retrieved from the point cloud. To remove background objects from point clouds a mask is utilized. As illustrated in fig. 3.2a, the optical sensor has to record a point cloud of the empty environment. Provided the environment does not change, the sensor position and orientation remains the same, the point cloud of the environment (see fig 3.2c) is used to create a list of point cloud subareas which will serve as a valid mask. Fig. 3.2e highlights all the subareas of the point cloud, which are part of the mask in red. Once one such mask has been created it is sufficient for multiple point clouds until the environment or the sensor orientation changes. Objects like the ball and chair inside fig. 3.2b are then recorded by the sensor. As explained in 3.1 and illustrated in fig. 3.1 the resulting point cloud (fig. 3.2d) contains only partial representations of the objects and the environment. By removing vertices in previously masked subareas from all the subareas containing vertices (fig. 3.2f), as illustrated in fig. 3.2g, the resulting point cloud is theoretically reduced to the partial object representations it included before (see fig. 3.2h). However, not all vertex positions within point clouds are not necessarily located accurately. It is important to determine sufficiently big subarea size to handle vertex displacements. Apart from minor locating errors, which most probably still fall within a masked subarea, a few vertices appear randomly displaced in real point cloud recordings. Such displaced points are referred to as noise. A more realistic point cloud corresponding to the setting in fig. 3.2b is illustrated in fig. 3.2i. As illustrated in fig. 3.2j, even a small amount of noise leads to multiple undesired additional subareas of the point cloud, which are not covered by the mask. Consequently a density filter is introduced to remove the noise containing areas. Such a filter removes all the subareas containing less then an adjustable number of vertices. The example in fig. 3.2k illustrates, how a density filter removes all subareas containing less then 2 vertices additionally to the subareas removed by mask. Fig. 3.2j illustrates the remaining vertices of the point cloud visualized in fig. 3.2i, after both, mask and density filter, have been applied. Due to an uneven distribution of density within the point cloud, depending on suffice distance and orientation towards the sensor, low density areas are stronger effected by density filters. Notice that information from the low density parts of the chair in the center of fig. 3.2j is lost, compared to the unfiltered idealized example in fig. 3.2h. In the creation of

the prototype software for object localization from 3D point clouds, it is assumed, all vertex containing subareas are individual objects. Thus, fig. 3.2k contains six objects, highlighted in green. Further development into a method, which does not split large objects, is possible. Nevertheless complex PCL solutions are to be considered as well, if the prototype performance is unable to satisfy required accuracy. Apart from 3D vertex position analysis, the image assigning brightness values to the vertices holds enough information to perform 2D-image detection and localizing algorithms. The 2D-positions are then mapped into the point cloud along with brightness values. Since the Intenta provides a sensor connection example containing OpenCV, the computer vision library is used for image analysis in prototype development.

3.3. Localizing Markers within the 2D Image of the Point Cloud Texture in OpenCV

The *Multisensorconnection Demo* for the S2000 (see 2.2) provides sensor image visualization with OpenCV. Among the images visualized by the *Multisensorconnection Demo* is the calibrated gray-scale image from left stereo sensor camera, the point cloud texture. As described in 2.1, 2D-coordinates from this texture image are applied to the point cloud image in order to retrieve the 3D-coordinates. In this section OpenCV and the ArUco module (see 3.3.2) for the library are introduced as means to extracting 3D texture image coordinates.

3.3.1. Fundamentals of the OpenCV Library

OpenCV is a popular computer vision library with a high range of functionality, including basic image processing functions and algorithms for tasks such as feature matching or face detection[PBKE12]. The OpenCV core module provides the base of the library. It includes arithmetic functions and data structures, and is only one of many modules included in OpenCV. Other common modulus include user interface functions, image reading and writing tools, feature detectors, calibration tools, object detection and more. Furthermore optional modules may be added to the set of standard modules, which are provided within OpenCV installations. One of the repositories for such additional modules is called *opencv_contrib* and contains the *ArUco* module which is further described in section 3.3.2. [Lag14]

Within OpenCV several data types are defined. To store position the *CvPoint* types are used. These structures contain two to three members (x,y and optionally z) of integer or floating point types. Same OpenCV types give hints to the number of members and member types in their names (e.g. *cv::point2f* stores its position into two float members). *CvSize* types are similar to *CvPoint*, however there members are called *width* and *height* and store size as the name suggests. *CvRect* combines

both *CvPoint* and *CvSize* to save the position and size of a rectangle. The *CvScalar* type holds a maximum of four values with double precision. [BK08, p. 31]

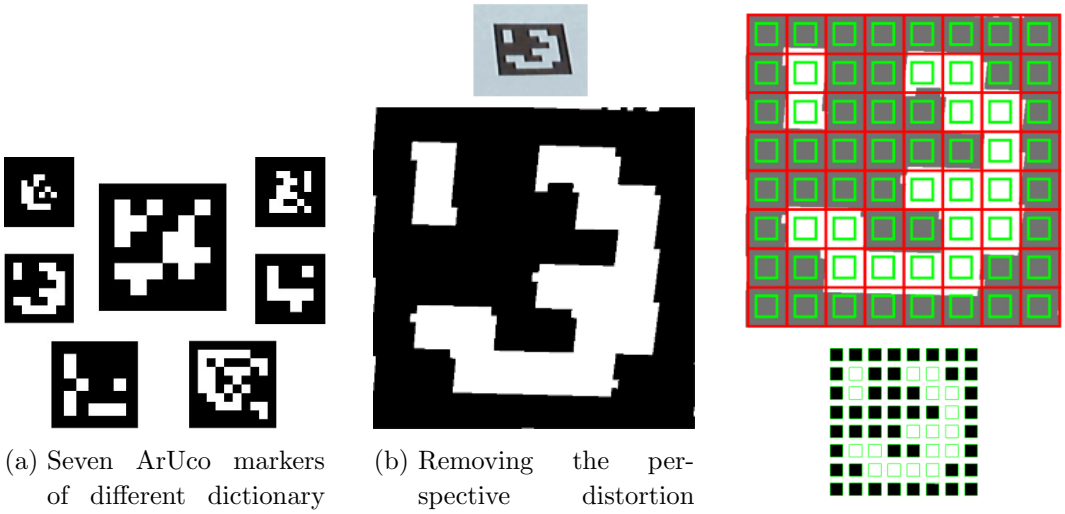
The OpenCV Matrix

Matrices are one of the fundamental data types in OpenCV. They vary in size (rows and columns) and are utilized to store image data. An important property of *cv::Mat* is the ability to define the data type contained within the matrix on construction. This also allows types capable of storing multiple channels to be used within the matrix and thus, the representation of RGB color values. Matrices with one row or column are used to represent vectors, as there is no additional types for such vectors in OpenCV. Internally *cv::Mat* stores its content using a *data* array, which is interpreted according to the values of *width*, *height* and other header settings. Matrices can be created with *cvCreateMat()*, or copied using *cvCloneMat()*. [BK08, p. 33-34] Documentation and further details about the *cv::Mat* are found in the book *Learning OpenCV: Computer Vision with the OpenCV Library*[BK08].

3.3.2. Marker Detection and Localization with ArUco

Camera based augmented reality applications require algorithms, that estimate the camera position. This is needed to properly place a model of the world on top of the image recorded by the camera. Through this model the illusion of placing virtual objects within the real world (as recorded by the camera) becomes possible. The ArUco library (a module developed for OpenCV) was created and optimized to for such tasks. To calculate the camera position and mark the areas, which can be utilized to position virtual objects, ArUco uses fiducial markers. These markers consist of black and white squares, which are interpreted as binary values. Each combination of binary values is in turn mapped to the identification value of the marker. To simplify and accelerate marker tracking, a border of black squares surrounds the variable section of every code. The marker structure is illustrated in fig. 3.3. An advantage of ArUco markers, in comparison to other fiducial markers is, that the existing binary values used to determine the marker index have a maximized hamming distance. For multiple sizes of ArUco markers an algorithm for generating marker dictionaries calculates marker patterns, which are distinctive (due to maximized hamming distance). This reduces the probability of falsely assigned marker indices. Rotated markers are likewise taken into account. This results in the stable tracking of markers and their rotation. [GJMSMCMJ14]

ArUco markers are independent of color channels and convey information about the position and orientation of themselves or the objects which they identify. Therefore utilizing the ArUco markers may has benefits for tasks other than the creation of virtual reality.



(a) Seven ArUco markers of different dictionary sizes. The marker in the center has a size of 6×6 squares encoding its id.

(b) Removing the perspective distortion of a marker and binarizing it to prepare bit-extraction.

(c) Extracting the bits of information encoded into the ArUco marker.

Figure 3.3.: Fiducial markers of the ArUco library and the extraction of marker values are visualized. Images have been adopted/modified from `https://sourceforge.net/projects/aruco/`.

Once the ArUco module has been set up with OpenCV, multiple markers within on image can be detected with the function *detectMarkers()*. It takes the arguments image (of type cv::Mat), dictionary (of type aruco::Dictionary), corners (a vector of Point2f), ids (a vector of integer), detectorParams (of type aruco::DetectorParameters) and rejected (also a vector of Point2f). The argument dictionary defines the size (see fig. 3.3a) and number of markers, which are part of the dictionary, which the function used to find markers. If the dictionary does not match the one of the markers which are visible within the image, markers will not be located, because the step of bit extraction (see fig. 3.3c) in marker analysation fails. The argument corners contains the positions of all four corners of a marker. While the rejected argument is similar, it contains the corners of markers which have been rejected by the algorithm. The ids argument stores the dictionary index of the found marker in corresponding order to the corners argument.

3.4. Visualization of 3D-Models

For the visualization of 3D-models, graphic libraries are utilized. Within the Microsoft environment Direct3D is the native graphics library, it does not support other operating systems. Next to Direct3D, OpenGL is the most widely accepted API for

graphic visualization. It is often applied in the implementation of research projects. The OpenGL shading language is platform independent. [Che08, p. 253,284,336]

Since the audiovisual laboratory is not limited to Microsoft Windows in the choice of its operating systems, OpenGL will be of benefit in case Intentas SVP_API becomes platform independent in the future. OpenGL does not include windowing systems or methods to receive user inputs directly in order to keep it operating system independent. The ability to visualize complex geometric objects is not part of OpenGL, but basic geometric primitives (see 3.4.2) can be combined to create complicated 3D-models. [SG09]The data types included into OpenGL ensure that defined type sizes do not vary between platforms, however many can be substitute with non OpenGL types if platform portability is not required for an application.[2] FreeGLUT is a common platform independent windowing system for OpenGL [SG09]. This thesis documents theory and implementation for OpenGL version 3.6.0.

3.4.1. Positioning, Scaling and Rotating of 3D-models with OpenGL

OpenGL utilizes matrix operations to position, scale and rotate objects. Matrices have the ability to store such operations, which are referred to as transformations. Essentially the multiplication of a matrix and a vector, applies the transformation to the resulting vector, which replaces the initial vector. This is covert in more detail later in this subsection. The model view matrix is a 4×4 matrix used to describe transformations. [Bus03]

Theorem 3.4.1 (Buss, 2003)**.** *"A transformation on $\mathbb{R}^2$ is any mapping $A : \mathbb{R}^2 \to \mathbb{R}^2$. That is, each point $x \in \mathbb{R}^2$ is mapped to a unique point, $A(x)$, also in $\mathbb{R}^2$."*

The transformations represented within the model view matrix are designed to map points $x \in \mathbb{R}^3$ into $\mathbb{R}^3$. However, in order to create not only linear but also affine transformations when projected into $\mathbb{R}^3$, the transformation operations in OpenGL take place inside a 4×4 matrix and thus in $\mathbb{R}^4$.

[2] Information found at `https://www.khronos.org/opengl/wiki/OpenGL_Type`

Theorem 3.4.2 (Buss, 2003)**.** *"Let A be a transformation. A is a linear transformation provided the following two conditions hold:*

1. *For all* $\alpha \in \mathbb{R}$ *and all* $x \in \mathbb{R}^2$,$A(\alpha x) = \alpha A(x)$.
2. *For all* $x, y \in \mathbb{R}^2$, $A(x + y) = A(x) + A(y)$.

Note that $A(0) = 0$ *for any linear transformation A. This follows from condition 1 with* $\alpha = 0$*."*

To understand the concept of affine transformations, the group of linear transformations should be understood first. The ability of a transformation to perform a translation (as described later in this section), disqualifies a transformation from being identifiable as linear. A simple translation is performed by adding a vector to the points that are transformed.

Theorem 3.4.3 (Buss, 2003)**.** *"A transformation A is affine provided it can be written as the composition of a translation and a linear transformation. That is, provided it can be written in the form* $A = T_u B$ *for some* $u \in \mathbb{R}^2$ *and some linear transformation B.*
In other words, a transformation A is affine if it equals

$$A(x) = B(x) + u$$

with B a linear transformation and u a point. Because it is permitted that $u = 0$*, every linear transformation is affine. However, not every affine transformation is linear. In particular, if* $u \neq 0$*, then transformation A is not linear since it does not map 0 to 0."*

Affine transformations are among the most fundamental mathematical concepts in computer graphics. In order to display a 3D-scene on the 2D-canvas OpenGL utilizes a rendering pipeline which involves multiple affine transformations. Creating a 3D-model out of vertices is only the first step of OpenGLs rendering pipeline. Displaying the resulting 2D-image is the last step. [Bus03]
Further details about the rendering pipeline and affine transformations can be found in *3D Computer Graphics: A Mathematical Introduction with OpenGL* [Bus03].

Matrix Vector Product

As mentioned before the transformations stored inside the model view matrix can be applied to a vertex through multiplication. Usually a transformation is applied to

every vertex of a group which is interpreted as one geometric primitive (see 3.4.2). To calculate the matrix product with a 4×4 matrix, a four dimensional vector has to be defined. The x, y, and z values of the vector are adopted from the represented vertex, while the forth value is set to 1. For multiplicative operations 1 is the neutral element and therefore acts as a placeholder. Since only 3D-transformations are desired the resulting vector (the matrix vector product) has to be reduced from a four dimensional vector to a 3D-vertex again. For linear transformations the forth dimension can be ignored since it remains unchanged.
The computation of the matrix vector product is defined as follows:

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} a \cdot x + b \cdot y + c \cdot z + d \cdot 1 \\ e \cdot x + f \cdot y + g \cdot z + h \cdot 1 \\ i \cdot x + j \cdot y + k \cdot z + l \cdot 1 \\ m \cdot x + n \cdot y + o \cdot z + p \cdot 1 \end{pmatrix}$$

Scaling

Changing the size of objects consisting of multiple vertices is achieved by multiplying every dimension of the vertices with the same value s. Setting $s > 1$ results in the object being scaled up, $s < 1$ shrinks the object and $s = 1$ has no effect. Such a scaling process is illustrated in fig. 3.7c in section 3.6. If different values are multiplied to each dimension the shape of the scaled object may be flattened or stretched as a result, however the transformation is linear in any case. OpenGL creates a transformation matrix which results in the matrix vector product mirroring the described operation of multiplying a scaling value to all three relevant dimensions.

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} s_x \cdot x \\ s_y \cdot y \\ s_z \cdot z \\ 1 \end{pmatrix}$$

Rotation

The Matrix for the rotation around an axis $\boldsymbol{u}$ is defined as follows in the book *3D Computer Graphics: A Mathematical Introduction with OpenGL* [Bus03]. The axis $\boldsymbol{u}$ is a vector that is automatically normalized by OpenGL when the rotation function is called with a non unit vector.

$$R_{\theta,\boldsymbol{u}} = \begin{bmatrix} (1-c)\boldsymbol{u_1^2} + c & (1-c)\boldsymbol{u_1 u_2} - s\boldsymbol{u_3} & (1-c)\boldsymbol{u_1 u_3} + s\boldsymbol{u_2} & 0 \\ (1-c)\boldsymbol{u_1 u_2} + s\boldsymbol{u_3} & (1-c)\boldsymbol{u_2^2} + c & (1-c)\boldsymbol{u_2 u_3} - s\boldsymbol{u_1} & 0 \\ (1-c)\boldsymbol{u_1 u_3} - s\boldsymbol{u_2} & (1-c)\boldsymbol{u_2 u_3} + s\boldsymbol{u_1} & (1-c)\boldsymbol{u_3^2} + c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where: $c = \cos\theta$; $s = \sin\theta$ and $\boldsymbol{u} = \frac{\boldsymbol{u}}{||\boldsymbol{u}||}$

This enables OpenGL to define matrices for rotations in every possible direction without the need for combining rotation matrices by computing a matrix matrix product. An example rotation process is illustrated in fig. 3.7b in section 3.6. Similar to scaling rotations could be performed using only 3D-matrices, both transformations are linear.

Translation

A vertex v may be translated by adding a vector $\boldsymbol{u}$. As illustrated with multiple vertices in 3.7a in section 3.6, where e.g. the green arrows is equal to $\boldsymbol{u}$ and $B_0 = v$. Translations are by definition no linear transformations but nevertheless affine. The fourth dimension of the model view matrix is used avoid the additional vector addition, which would be required to perform translations with 3D-matrices and vectors.

$$\begin{bmatrix} 1 & 0 & 0 & \boldsymbol{u_1} \\ 0 & 1 & 0 & \boldsymbol{u_2} \\ 0 & 0 & 1 & \boldsymbol{u_3} \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ 1 \end{pmatrix} = \begin{pmatrix} v_1 + \boldsymbol{u_1} \cdot 1 \\ v_2 + \boldsymbol{u_2} \cdot 1 \\ v_3 + \boldsymbol{u_3} \cdot 1 \\ 1 \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix}$$

Note that the matrix matrix product as described in the next paragraph may cause changes in the values of the last row of a transformation matrix when non linear transformations are involved. Consequently the fourth value of the vector $\boldsymbol{r}$ resulting from the matrix vector product may not always equal 1. If $\boldsymbol{r_4} \neq 1$ the conversion of four dimensional vector to 3D-vertex must be executed as follows:

$$v_{trans} = \begin{bmatrix} \frac{r_1}{r_4} & \frac{r_2}{r_4} & \frac{r_3}{r_4} \end{bmatrix}$$

Matrix Matrix product

So far the benefit of realizing OpenGLs transformations with matrix operations is not obvious. Scaling and translating can as described in the according paragraphs be calculated without matrices, however for each vertex every transformation world need to be applied to one by one. For n vertices and k transformations the complexity of the algorithm would be $O(n \times k)$ and therefore highly dependent on k. This situation is improved by utilizing matrices to store the transformation information. Multiplying various transformation matrices results in a single transformation matrix, which still moves all vertices as if they would have been transformed step by step using the original transformations. Thus, only k matrix multiplications reduce the complexity (of performing k transformations on n vertices) to $O(n)$ since effectively only one transformation has to be applied per vertex now.

The Matrix Stack

OpenGL always uses the current model view matrix to perform transformations on the vertices of geometry objects. To alter the transformation for individual objects, matrix stats can be saved using a stack as illustrated in line 10. Changes applied to the model view matrix can be undone by resetting the matrix to the saved states present in the stack. This is done in line 17.

```
1  void display(){
2      drawBackground();
3
4      glLoadIdentity();                    // load identity matrix M:=I
5      glScalef(1.2f,1.2f,1.2f);            // matrix multiply M:=M*S
6
7      glColor3f(0.0f, 0.0f, 0.0f);         // use black color
8      drawExample();                       // draw object
9
10     glPushMatrix();                      // save matrix X:=M
11
12      glColor3f(1.0f, 0.0f, 0.0f);        // use red color
13      glScalef(0.5f,0.5f,0.5f);           // matrix multiply M:=M*S
14      glTranslatef(0.3f,0.3f,0.0f);       // matrix multiply M:=M*T
15      drawExample();                      // draw object
16
17     glPopMatrix();                       // reset matrix to M:=X
18
19     glColor3f(0.0f, 0.0f, 1.0f);         // use blue color
20     glRotatef(33.0f,0.0f,0.0f,1.0f);     // matrix multiply M:=M*R
21     drawExample();                       // draw object
22
23     glFlush();
24 }
```

This OpenGL code example draws the same object three times in three different colors, as presented in fig. 3.4. All transformations applied to the object are provided by the model view matrix M. The OpenGL functions for scaling, translating and rotating

Extracting Transformed 3D-Models from OpenGL

Understanding the model view matrix is critical for the export of transformations (as visualized through OpenGL). The library does not allow the direct extraction of individual transformed vertex positions unless they are projected onto the 2D-canvas used to visualize them. Thus, to extract and store the transformed vertex positions in 3D-space, transformations must be calculated separately for the purpose of storing them. Luckily OpenGL provides a method for extracting transformation matrices. Based on the extracted matrix and the non transformed vertices, the entire transformed 3D-model is calculated and then extracted. With the OpenGL

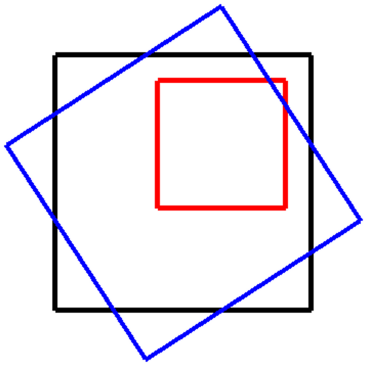


Figure 3.4.: The output of an OpenGL program, that illustrates the effect of a set of matrix operations: The black square shows the reference position, while the red square is scaled and translated. The blue square is not scaled and not translated, but rotated.

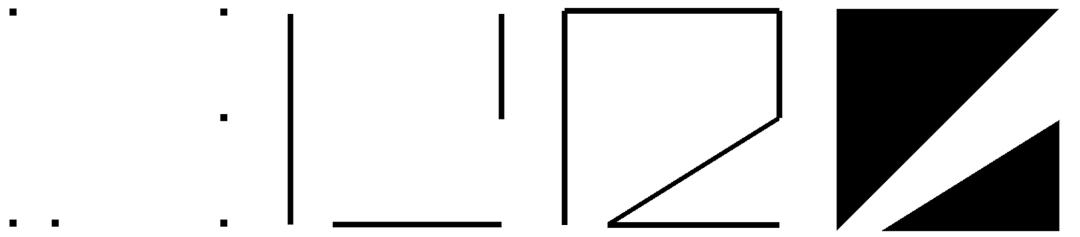
function call *glGetFloatv(GL_MODELVIEW_MATRIX, ptr)* a pointer to the matrix array (ptr) containing the model view matrix is retrieved. The array representing the 4×4 matrix contains 16 GLfloat values. The values of the matrix are listed column-wise inside the matrix, the left most matrix column reaching from array index 0 (at the top) to 3 (at the bottom). The right most matrix column consists of the last array values, with the indices from 12 (at the top) to 15 (at the bottom).

Since OpenGL utilizes graphics hardware to speed up matrix computations, performing an unnecessary amount of matrix operations externally with CPU computations slows down an application needlessly.

3.4.2. Visualization of 3D-Model Vertices

Once the windowing system FreeGLUT has been prepared and the display function has been set, vertices can continuously be drawn onto the canvas (the area inside the FreeGLUT window). Details on the implementation of FreeGLUT are covered in the OpenGL Programming Guide [SG09]. To start the interactive OpenGL animation the glutMainLoop() has to be executed. It repeatedly calls the display function which defines the geometric objects drawn by OpenGL.
The basic geometric primitives in OpenGL 4.6 are constructed out of multiple vertices in between glBegin() and glEnd() [SG09]. To specify the type of primitive that will be constructed out of the vertices, glBegin() expects an argument of type GLenum[3]. The possible GLenum values and therefore types of geometric primitives include, among others, the following:

[3]Information found at `http://docs.gl/gl3/glBegin`



(a) GL_POINTS (b) GL_LINES (c) GL_LINE_STRIP (d) GL_TRIANGLES

Figure 3.5.: Examples for basic geometric primitives in OpenGL

GL_POINTS: Vertices are visualized as a dot or point on the canvas (see fig. 3.5a). The size of these points can be altered with the function glPointSize() which takes a GLfloat type as an argument to replace the default value of 1. GL_POINTS is ideal to visualize raw point clouds.

GL_LINES: An even number of vertices is visualized pairwise (see fig. 3.5b). Every vertex is depending on its predecessor interpreted as either the beginning or the end of a line. The first vertex always defines the starting point of the first line.

GL_LINE_STRIP: Draws a line connecting the first vertex to the last. All of the intermediate vertices define the way according to their order (see fig. 3.5c).

GL_TRIANGLES: Groups of three vertices are interpreted as triangles similar to the interpretation of vertex pairs in GL_LINES. OpenGL colors the whole area of the triangle (see fig. 3.5d). Such triangles may be combined to create complex geometry.

```
1  void drawExample()
2  {
3    glLineWidth(10); glPointSize(10);
4    float v[18] = {-0.5,-0.5,0.0,   -0.5,0.5,0.0,   0.5,0.5,0.0,
5                   0.5,0.0,0.0,   -0.3,-0.5,0.0,   0.5,-0.5,0.0};
6    glBegin( GL_POINTS | GL_LINES | GL_LINE_STRIP | GL_TRIANGLES );
7    for (int i=0; i<18; i+=3){
8      glVertex3f(v[i+0],v[i+1],v[i+2]); // output each vertex
9    }
10   glEnd();
11 }
```

This algorithm produces the visuals shown in fig. 3.5 depending on the primitive that is selected (in the form of the according GLenum value in line 6) for the *glBegin* function.

3.5. The Representation of 3D Structures in Files

Many file formats for the representation of 3D-models and structures exist. However, when searching for a wide spread, easy to implement, exchange format, many such 3D file formats disqualify themselves. X3D contains unnecessary additional information, such as sounds, lights or web links. DXF supports triangle structures in 3D, but is mainly for 2D-drawings. 3DS has a limited number of vertices and polygons and contains unnecessary additional information as well. SLC contains only several 2D-slices of 3D-objects. SAT has a complicated topological data structure. And also STEP is a rather complex format. PLY was created to handle data originating form 3D-scanners. And lastly, with a simpler file structure than PLY, OBJ is a wide spread and easy to implement 3D-model file format. Unlike the other formats it does not contain unneeded additional data. OBJ suits the needs for an exchange format. [HL09]

The OBJ-file-format is an open ASCII-based format for representing 3D-geometry. It was initially developed for the program Advanced Visualizer by the company Wavefront Technologies. OBJ is a popular format, because it is easy to handle, since the ASCII form is human readable, and it has been adopted as a geometry import and export file type by multiple 3D-geometry editors, such as Blender[4] or Maya[5] Blender is an open source application with a very high range of capabilities, from serving as a modeling tool for simple research tasks to the creation of highly realistic virtual environments [KKK17]. Information inside of OBJ-files is stored in the form of several types of elements. Although the format is capable of storing textures, curves and other complex information in such elements, its structure allows to read and write files indifferent of many types elements, if they are unknown to the parser. This is possible because the elements of an OBJ-File are separated line-wise, with the first characters on each line identifying the type of element. Lines beginning with characters which can not be mapped to any element type are ignored. The information stored within elements is separated from the identifying characters by a space character; multiple parts of a single element are likewise separated by space characters. Basic elements of OBJ-files are identified by a single character. Such elements include, vertices, lines, faces, object elements and comments. [Pip03, p. 61-65]

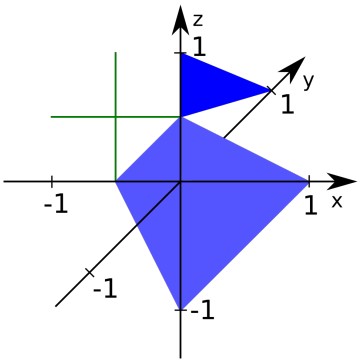
[4] Blender is an open source 3D creation suite. See `https://www.blender.org/`

[5] Find more about the 3D computer graphics application *Autodesk Maya* at `https://www.autodesk.eu/products/maya/overview`

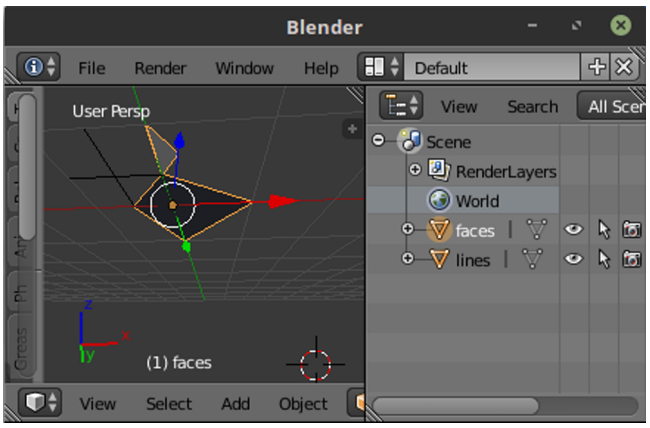
```
1  # Example OBJ
2  v -0.5 0 1
3  v -0.5 0 0
4  v -1 0 0.5
5  v 0 0 0.5
6  v 0 0 1
7  v 0 1 0
8  v 1 0 0
9  v 0 0 -1
10 o lines
11 l 1 2
12 l 3 4
13 o faces
14 f 4 5 6
15 f 2 4 7 8
```

The example of an OBJ-file, given above, includes five types of elements. The first line starting with the "#"-symbol is a comment element, its content is meant for human readers and is ignored by OBJ-file-parsers. Vertex elements (lines 2 to 9) begin with the character "v" and are followed by the x, y, and z coordinates of the vertices they store. Such coordinate values may be in floating point format. The elements line and face do not contain coordinates directly, they reference the vertex elements by index. The indices of vertex elements are assigned to them by counting vertices linewise from the first to the last line of the file. Consequently in this example the vertex element on line 2 is assigned the vertex index 1. The line element (lines 11 and 12) is identified by the "l" character and followed by a minimum of two vertex indices, which determine the coordinates between which a line is defined. Fig. 3.6a illustrates two such lines in green. Face elements (lines 14 and 15) are identified by the "f" character and require a minimum of three vertex indices to define a polygon. All points of such polygons must be members of a common plain within 3D-space. In fig. 3.6a the two polygons defined within the example OBJ-file are illustrated in blue. Lastly the example OBJ-file includes object elements (lines 10 and 13), these elements group other elements, such as lines and faces, into objects. An object begins with the object element line, identified with the character "o", and ends with the next object element or the end of the OBJ-file. Object elements also assign a name to each object. A benefit of sorting geometric elements inside an OBJ-file into objects, is that it allows programs to treat objects as separate entities. Fig. 3.6b shows how OBJ-objects are displayed inside Blender, once the OBJ-file has been imported.

To simplify the interpretation of OBJ-files for a basic OBJ-parser, triangulated faces should be required for all files excepted by the parser. Inside a triangulated OBJ-file only the minimal number of three vertex indices is referenced by each face element. This ensures an unambiguous interpretation of the face, since the order of vertices which connect into a triangle is interchangeable. Potential orders of more



(a) The two objects defined inside the OBJ-file are colored differently. The object "lines" is green and consists of two lines crossing each other in the x-z-plain. In blue the object "faces", consisting of a quadrangle in the x-z-plain and a triangle in the y-z-plain, is illustrated.



(b) On the left side a perspective view of the OBJ-file is visible. The list on the right side includes the objects, which have been specified inside the OBJ-file. The object "faces" is selected in and can be manipulated independent of the object "lines". Blender reinterprets the coordinates such that the y and z-axis are swapped.

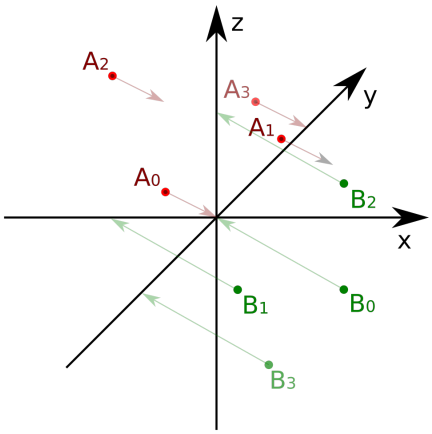
Figure 3.6.: The geometric content of a simple OBJ-file: The file includes two objects, one consisting of two lines and one consisting of two faces. Within the cartesian coordinate system in fig. 3.6a is illustrated how the OBJ-file is displayed in theory, while fig. 3.6b shows how the 3D-modeling software Blender interprets it.

than three vertices may lead to non convex polygons, which would require further processing in order to ensure proper visualization of them with graphic libraries such as OpenGL. Non triangulated OBJ-file can be converted into triangulated files with export options in 3D-modeling software like Blender. Displaying a warning to users, who try to read files including non implemented elements, is advisable. Advanced OBJ-File structures (with texture elements and likewise) include additional information into elements such as lines and faces, which may cause a basic parser to malfunction.

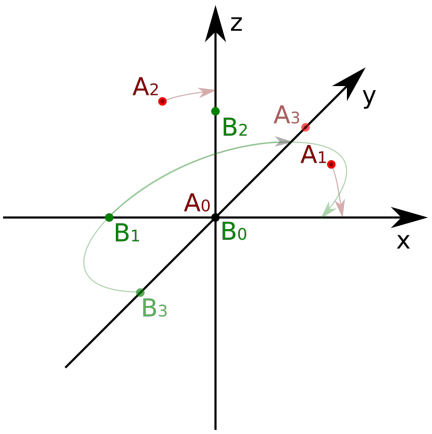
3.6. Transforming Point Cloud into Reference Model Coordinate System

When objects are successfully localized within a point cloud, a model containing these objects can be created. However, without any reference to the environment containing the object, the position of a single object reveals no information, because the origin of the coordinate system is arbitrarily positioned. The origin may not represent any relevant point in reality. A reference model of the environment must be created. Such a model must share axes and size with the laboratory specifications presented in fig. 2.4. Its position, rotation and size relative to the point cloud is crucial to all measurements recorded inside the point cloud based model. Because of the difference between size and orientation of the point cloud, when passed into the model without any adjustments, measurements initially appear randomly set. Thus, reference model and the model consisting of objects found within the point cloud, have to be lined up.

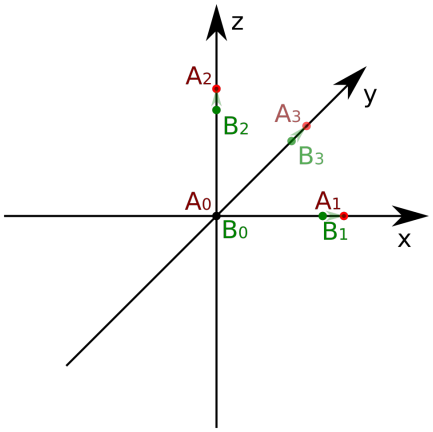
In order to define a 3D-coordinate system four points are needed such that three independent coordinate axes are constructible, represented as vectors. Inside existing 3D-coordinate systems any plain is defined by only two vectors. The size and orientation of these vectors unambiguously define not only positions within the plain, but also in the third direction. A common example of this is the normal vector. Thus, to synchronize the position of related existing 3D-Models, only three common points are sufficient, if no straight line connecting all three points can be constructed. Fig. 3.7 illustrates the theoretical steps of synchronization based on three common points. The points with the numbers 0, 1 and 2 are used to perform transformations, while both points with the number 3 illustrate how any additional point behaves when transformations to the calculated on the basis of only 3 points are executed. The first point (A_0 for group A and B_0 for group B in fig. 3.7a) is translated towards the origins of both common model coordinate systems. This is achieved by moving all points of each 3D-model (represented by groups A and B in 3.7) along the vector from A_0 or B_0 to the coordinate origin.



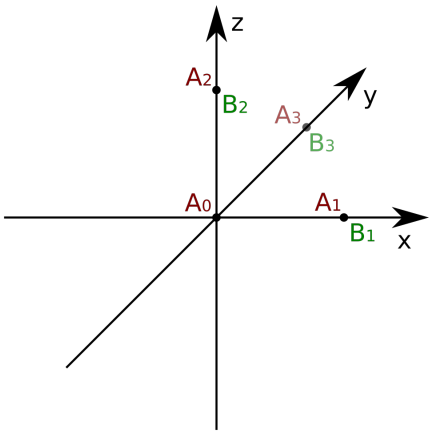
(a) Both groups A and B are translated to share the position origin. Their corresponding points A_0 and B_0 are illustrated to move towards the origin of the coordinate system.



(b) Group A and group B are individually rotated towards a similar position. Points located on the rotational axis (through the coordinate system origin) appear unaffected.



(c) Group B is illustrated to be scaled to the size of group A.



(d) The result of the previous operations; translation (subfig. 3.7a), rotation (subfig. 3.7b) and scaling (subfig. 3.7c), shows both groups at a congruent position.

Figure 3.7.: Three steps of synchronizing the position of identical but displaced and rescaled point sets: A group of red points (A_0 to A_3) and a group of green points (B_0 to B_3) are manipulated using a set of operations (subfig. 3.7a; 3.7b; 3.7c), which effect each group as a whole. The result is illustrated in subfig. 3.7d

In fig. 3.7b the translation of the groups A and B have been performed with the result, that A_0 and B_0 are now both at the origin of the coordinate system. To line up the positions of the points A_1 and B_1 both groups must at first be rotated in a fashion that results in the satisfiability of the equation $\overrightarrow{A_0A_1} \cdot k = \overrightarrow{B_0B_1}$ for $k > 0$. Conceptually one of the easiest ways to achieve that, is by rotating both, A_1 and B_1 onto the positive side of the same coordinate axis. However the same rotation must also fulfill the satisfiability condition for $\overrightarrow{A_0A_2} \cdot k = \overrightarrow{B_0B_2}$ for $k > 0$. Similarly this can be simplified. To keep the relations between all members within each group, the points A_2 and B_2 are not to be forced onto another axis. The rotation should put both points onto the same plain, such that within each group the distances between all points remain the same. This plain must still contain the axis onto which A_1 and B_1 are rotated and a second coordinate axis. Only rotations that move A_2 and B_2 onto the positive part of the second coordinate are allowed. In fig. 3.7b the plain, onto which the points are rotated, is the x-z-plain.

Finally the remaining positional differences between the points are resolved by scaling both or just one group of points. In fig. 3.7c group B is scaled up to the size of group A. The transformations result in congruent groups A and B as illustrated in fig. 3.7d.

All of the transformations to archive such results can be implemented using OpenGL if the values needed to set up the transformation matrices (see 3.4.1) can be computed. The first step, translating the groups such that A_0 and B_0 are moved to the common origin, requires only negating the values of the coordinates of A_0 and B_0. Translating the according groups by these values leads to the desired result. In the following paragraphs two methods, of computing turn angles and axes for the rotation during point synchronization, are covered.

Defining The Rotation Target

With arbitrarily oriented groups of points, only sharing a common origin, both of the later methods for calculating rotation angles and axes require setting a target. It defines where to the reference points are rotated along with the rest of the groups they belong to, when the desired transformation is applied. In fig. 3.8 the target positions are the green points d, e and f. The positions of the points a, b and c in the same fig. represent three points of a non synchronized group before rotation. As shown before, if a rotation in 3D-space is defined such that two points reach their target positions (and there is no straight line connecting them both to the pivot point), the rotation results in an unambiguous position of all points transformed with them.

To set the target position e for point b, the length of $\overrightarrow{ab}$ is used as a coordinate value of e while the remaining coordinates are set to zero. Since $a = d = (0; 0; 0)$, the distance of b to the coordinate origin is $||b|| = ||\overrightarrow{ab}||$. The position of e is defined:

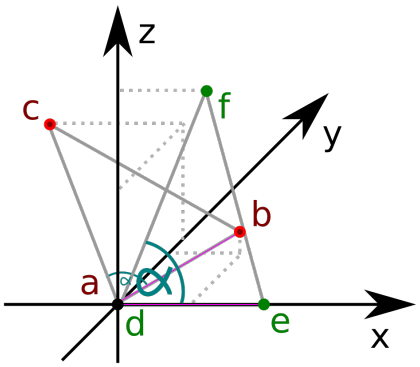


Figure 3.8.: Two groups of 3 points are depicted inside a 3D-coordinate system (group 1: a, b, c; group 2: d, e, f): The distances and angular relations between the points within each group are identical. The points a and d are in the same position and if they are used as a pivot for c and b, they could rotate them into the position of e and f.

$$e = (||b||; 0; 0)$$

The rotation of b to the position of e can not change the distance of the point towards the coordinate origin. The definition of e above ensures:

$$||e|| = \sqrt{\sqrt{{b_x}^2 + {b_y}^2 + {b_z}^2}^2 + 0 + 0} = \sqrt{{b_x}^2 + {b_y}^2 + {b_z}^2} = ||b||$$

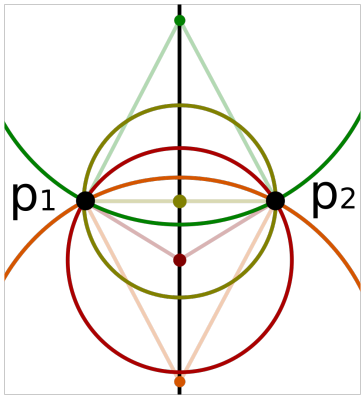
The target position f for point c must satisfy two conditions to ensure the point relations are preserved. The fist condition is again, $||f|| = ||c||$. The second condition is to preserve the angle α as illustrated in fig. 3.8. Thus, the angle between $\overrightarrow{ab}$ and $\overrightarrow{ac}$ must be equal to the angle between $\overrightarrow{de}$ and $\overrightarrow{df}$:

$$\arccos\left(\frac{\overrightarrow{ab}\cdot\overrightarrow{ac}}{||\overrightarrow{ab}||\cdot||\overrightarrow{ac}||}\right) = \alpha = \arccos\left(\frac{\overrightarrow{de}\cdot\overrightarrow{df}}{||\overrightarrow{de}||\cdot||\overrightarrow{df}||}\right)$$

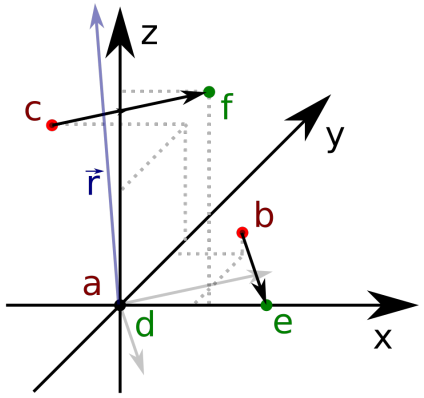
The angle α is calculated using $\overrightarrow{ab}$ and $\overrightarrow{ac}$ so it can be utilized for further computations. Since e is a point on the x-axis, the angle between $\overrightarrow{df}$ and the x-aches must be equal to α. As defined by the unit circle (adopted for 3D-space), a point g such that, $\overrightarrow{dg}$ has an angle of α towards the x-axis can be described by:

$$g_x = \cos(\alpha) \quad \text{and} \quad \sqrt{{g_y}^2 + {g_z}^2} = \sin(\alpha)$$

Any such point g would have a distance $||g|| = 1$ to the origin of the coordinate system (d) since it is based on the position of a point on the unit circle. To take the



(a) The pivot line between two points p_1 and p_2 in 2D-space. Several example rotations around points of the pivot line are illustrated with different colors.



(b) Visualized here are $\overrightarrow{be}$ and $\overrightarrow{cf}$ as they can be constructed between group 1 and group 2, which have been adopted from fig. 3.8. The vector $\vec{r}$ illustrates the cross product of $\overrightarrow{be}$ and $\overrightarrow{cf}$.

Figure 3.9.: Two concepts: 3.9a shows the position of pivot points in a 2D-system. 3.9b illustrates how a rotational axis is (symbolized by $\vec{r}$) is found in 3D-space.

first condition ($||f|| = ||c||$) into account for the definition of f, the sine and cosine function must be multiplied by the distance f is required to have to d. Consequently f must be defined such that:

$$f_x = \cos(\alpha) \cdot ||c|| \quad \text{and} \quad \sqrt{{f_y}^2 + {f_z}^2} = \sin(\alpha) \cdot ||c||$$

To position f on the x-z-plain as illustrated in fig. 3.8, the point can be defined such that:

$$f = (\cos(\alpha) \cdot ||c||; 0; \sin(\alpha) \cdot ||c||)$$

The One Step Rotation

To perform a rotation, the rotational axis and the angle of the rotation have to be specified. The rotational axis is represented by a vector $\vec{r}$, such that $\vec{r} \cdot k$ with $k \in \mathbb{R}$ determines every possible point on the rotational axis.

In order to find $\vec{r}$, two dimensional space is considered. Figure 3.9a visualizes the concept of a pivot line. Such a line is infinite and includes every potential pivot point around which either of the points p_1 or p_2 can be rotated in order to reach the

position of the other. The distance of any point on the pivot line p_{pivot} to the points p_1 and p_2 is equal.

$$||\overrightarrow{p_{pivot}p_1}|| = ||\overrightarrow{p_{pivot}p_2}||$$

Thus, the pivot line is perpendicular to the vector $\overrightarrow{p_1p_2}$. If the concept is adapted into 3D-space, all points of equal distance to p_1 and p_2 form a pivot plain perpendicular to $\overrightarrow{p_1p_2}$. Since during the construction of the reference points d, e and f (see 3.8) the conditions $||e|| = ||b||$ and $||f|| = ||c||$ where considered, the coordinate origin is among the points of the pivot plain. Thus, no support vector for the construction of the pivot plain perpendicular to $\overrightarrow{eb}$ and the other pivot plain perpendicular to $\overrightarrow{fc}$ is necessary. The plains are constructed using the normal form:

$$E_1 : \overrightarrow{eb} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} \quad \text{and} \quad E_2 : \overrightarrow{fc} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

The intersection of E_1 and E_2 is the rotational axis, because it contains the pivot points shared by both plains. A vector $\vec{r}$ on the axis is calculated as the cross product of both normal vectors:

$$\vec{r} = \overrightarrow{eb} \times \overrightarrow{fc}$$

Fig. 3.9b illustrates the vectors of the equation above. The angle of the rotation around $\vec{r}$ is computed using additional vector calculations.

When a point is rotated around an axis, its possible positions lie on a circular path, which is located on a plain within 3D-space. Conceptually angles between points on this path can be derived within the two dimensions of the plain. In fig. 3.9a such circular paths are illustrated. At the center of each circle lies the pivot point, which marks the intersection of plain and rotational axis. An isosceles triangle, connecting pivot point and the points on the circular path, is constructed. To compute θ the angle between $\overrightarrow{p_{pivot}p_1}$ and $\overrightarrow{p_{pivot}p_2}$, first the normal form of the plain is constructed:

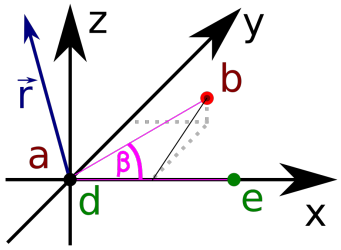
$$E_3 : \vec{r} \cdot \left(\begin{pmatrix} x \\ y \\ z \end{pmatrix} - \overrightarrow{p_1} \right)$$

To find the pivot point on this plain, the corresponding multiple of $\vec{r}$ is found by solving this equation for t:

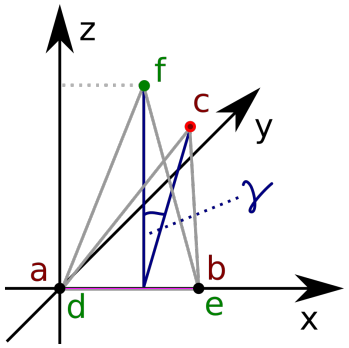
$$\vec{r} \cdot \left(\left(t \cdot \vec{r} \right) - \overrightarrow{p_1} \right) = 0 \qquad p_{pivot} = (t \cdot r_x; t \cdot r_y; t \cdot r_z)$$

The angle θ, for the appropriate rotation around $\vec{r}$ is computed:

$$\theta = \arccos \left(\frac{\overrightarrow{p_{pivot}p_1} \cdot \overrightarrow{p_{pivot}p_2}}{||\overrightarrow{p_{pivot}p_1}|| \cdot ||\overrightarrow{p_{pivot}p_2}||} \right)$$



(a) The rotation around $\vec{r}$ in a cartesian coordinate system. Points have been adopted from fig. 3.8.



(b) The angle γ, which must be determined when the rotation in fig. 3.10a is completed. γ is the angle for a rotation around the x-axis.

Figure 3.10.: The two step rotation method for point synchronization: Fig. 3.10a shows angle β for the first rotation and fig. 3.10b shows γ, the angle for the second rotation.

The two Step Rotation

A different approach to rotating two points into their target position, is transforming in two steps. Step 1 transforms all points such that the first point is rotated into its target position. Step 2 rotates the second point around the axis defined by the coordinate origin and the first point. For this approach the rotational matrix as defined for OpenGL (see 3.4.1) is used.

Step 1: As defined before and illustrated in fig. 3.8, the points a and b are considered to be members of the group that is rotated and the points d and e are target positions for the rotation. a and d are equal to the origin of the coordinate system. At first $\vec{r}$ is computed for the rotation of b to the position of e. The cross product between $\overrightarrow{ab}$ and $\overrightarrow{de}$ could be computed. However, to minimize point errors during the implementation $\overrightarrow{de}$ is substituted with a unit point on the axis where $\overrightarrow{de}$ is located. Thus, for e on the x-axis the calculation is:

$$\vec{r} = \overrightarrow{ab} \times \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

As the angle for the rotation around $\vec{r}$, β is defined (see fig. 3.9) , it could be calculated as the angle between $\overrightarrow{ab}$ and $\overrightarrow{de}$, but to minimize point errors $\overrightarrow{de}$ is likewise substituted:

$$\beta = \arccos\left(\frac{\overrightarrow{ab} \cdot \begin{pmatrix}1\\0\\0\end{pmatrix}}{||\overrightarrow{ab}|| \cdot || \begin{pmatrix}1\\0\\0\end{pmatrix} ||}\right)$$

Step 2: The rotational axis for the second rotation is the coordinate axis on which the first point is located after the first rotation was applied (see fig. 3.10b). To apply the first rotation the values calculated in step 1 are utilized to created a rotational matrix. The second point to rotate (c in fig. 3.10b) is brought into its starting position for the second rotation using the matrix vector product as specified for OpenGL (see 3.4.1).

Angle γ for the rotation around the x-axis is calculated:

$$\gamma = \arctan\left(\frac{c_y}{c_z}\right)$$

Matching the Scale of Models

When position and orientation of both models match each other (see fig. 3.7c), the size of the models has to be adjusted to fit each other. This ensures, all points, which represent the same position in reality, will match between the synchronized models of reality.

The distances to the coordinate origin of two points located on the same axis through the origin are sufficient to equalize the models they belong to in size. One such distance is calculated in the number of instances the other distance can fit inside it. This value can than be used to rescale a whole group of points (or model). The scaling value s is calculated:

$$s = \frac{||\overrightarrow{A_0 A_1}||}{||\overrightarrow{B_0 B_1}||}$$

This calculation uses the example points as visible in fig. 3.7c and the calculated value s can be applied to all three axes of group B to scale it into equal size of group A as visible in fig. 3.7d.

In order for the method of synchronization above to work, it is assumed, all models which have already been synchronized, match the relations between spatial position of points as defined by realty. If model show a distorted version of reality, the above defined method of synchronization may not work.

4. Implementation of a Prototype Software for Object Localization from 3D Point Clouds

This chapter describes the architecture and implementation of the prototype software for object localization from 3D point clouds. At first a general overview of the architecture and its context is given in 4.1. The visualization of the point cloud, including code examples, is explained in 4.2. The files containing 3D-coordinates are handled as described in 4.3. The sections 4.4 and 4.6 explain localization algorithm implementations. 4.5 describes the implementation of the calibration algorithm based on localizations. A summary of all implementations made during this thesis is given in 4.7.

4.1. Architecture of the Localization Prototype Software

The prototype software is required to have a user interface, create a model of the frame cage inside the audiovisual laboratory and its content, save an image of the object setting, and export localized object positions as floating point meter values. As described in 3.4, OpenGL is capable of visualizing and transforming 3D-models. Thus, the current version of OpenGL (4.6.0) is used to visualize a 3D-model of the frame cage, point cloud and highlighted object positions. To locate objects within the point cloud, the methods described in 3.2 are implemented. The point cloud texture image (providing the brightness values for the vertices) is utilized to locate ArUco markers as introduced in 3.3.2. To enable measuring positions according to the coordinate system illustrated in fig. 2.4, a synchronization of size and orientation of point cloud model and frame cage model is implemented based on 3.6. Calibration settings and point cloud masks are saved to files to reduce the efforts of working with the prototype software. A reduced version of the model displayed with OpenGL is exported using the OBJ-Format as introduced in 3.5. It contains only the frame cage model and object position markers. OBJ-files for both, frame cage model and position markers, must be imported into the prototype software. Thus, they are interchangeable. The raw positions in floating point meter values are displayed in the OpenGL UI and exported into exchange-files of the Comma Separated Value Format, CSV. The point cloud texture image is saved as a JPEG-file using a standard OpenCV function.

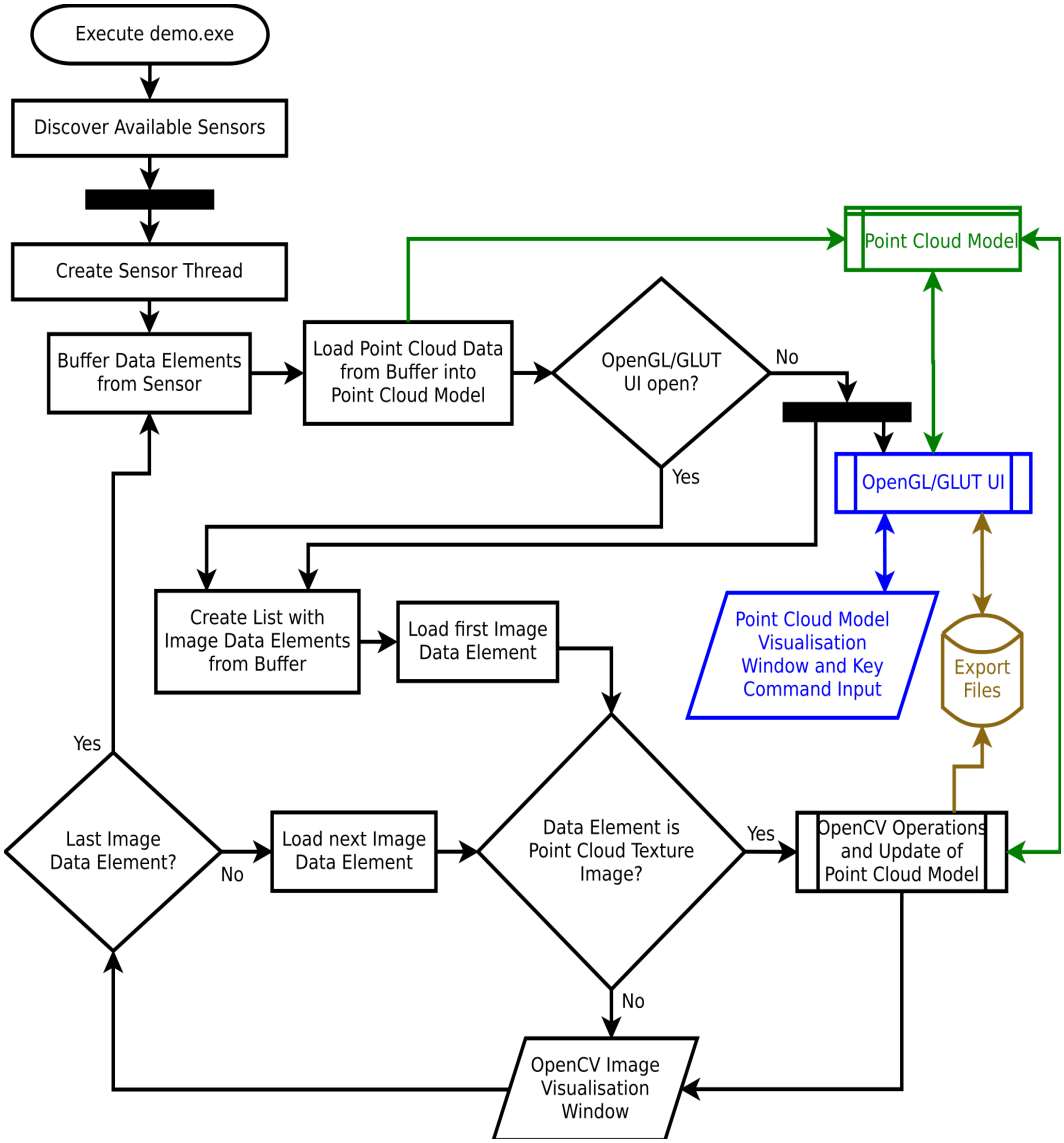


Figure 4.1.: This flowchart presents the structure of the prototype software for localizing objects from point clouds. On the left side the adopted structure from the sensor connection example (see 2.3) is visible. On the right side the *Point Cloud Model*, in green, and the OpenGL point cloud visualization, in blue, have been added. Between *Buffer Data Elements from Sensor* and *Create List with Image Data Elements from Buffer* (center left), the example code has been modified such that it updates the *Point Cloud Model* and starts the OpenGL visualization if it is not running. Before the *OpenCV Image Visualization Window* (at the bottom), another change to the example code has been made. If the loaded image is the point cloud texture, it is further analyzed with OpenCV to supply the *Point Cloud Model* with locations.

In order to access a constant stream of data from the S2000 sensor, the sensor connection example program introduce in 2.2 is utilized as the foundation of the prototype software for object localization from point clouds. Its structure, as visualized in fig. 2.3, is extended to incorporate the above described functionality into it. The resulting flowchart and cure structure of the prototype software for object localization from point clouds is illustrated in fig. 4.1. Most of the additional functionality is implemented into the *Point Cloud Model* class (green in 4.1) and in the *OpenGL/GLUT UI* section which is responsible for handling the *Point Cloud Model Visualization, Window and Key Command Input* user interface as its created by GLUT (blue in 4.1). These parts of the prototype software are entirely new code. To connect the Intenta sensor connection example code to the new point cloud visualization and export tools, and to add functionality in OpenCV, the example code is most importantly modified by inserting a section which updates the *Point Cloud Model* with streamed point cloud data. Since only one GLUT-Window is needed to display the data of multiple sensors, an *OpenGL/GLUT UI* thread is only started if it has not been started before.

To explain the structure behind the main threads, the flowchart (fig. 4.1) is compared to the sequence diagram in fig. 4.2. The components of both diagrams are colored corresponding to each other. The sequence diagram (fig. 4.2) gives an example of a connection to multiple sensors. Once a user has started the computer and the *File System* is active, executes the prototype software and thus starts the main thread. The main thread begins the step *Discover Available Sensors* before creating a new thread for every detected sensor (see top left in fig. 4.1). The sequence diagram (fig. 4.2) illustrates how the connections *Sensor Connection 1* and *Sensor Connection x* are created as a separate thread one by one. The fist thread initiates the execution of the *OpenGL/GLUT UI* thread. All active sensor connection threads supply the running *OpenGL/GLUT UI* thread with sensor data updates into the *Point Cloud Model*.

Localizing and transmitting object positions from the OpenCV image of the point cloud texture into the *Point Cloud Model* is realized by inserting the *OpenCV Operations and Update of Point Cloud Model* section into the code. As illustrated in fig 4.1 the additional code is only executed for the point cloud texture. It is part of the streaming routine running inside every single sensor connection. In fig. 4.2 the positions retrieved from the point cloud texture are part of the continues *Update Data* stream from each sensor to the *OpenGL/GLUT UI* thread.

After automatically loading previously stored settings once it is started, the *OpenGL/GLUT UI* thread visualizes the point cloud data from the sensor thread which initiated it. Pressing the I-key allows users to change the sensor on display. The currently displayed sensor is also the source for all exported files. Fig. 4.2 illustrates how once the user requests to export files, the *OpenGL/GLUT UI* thread

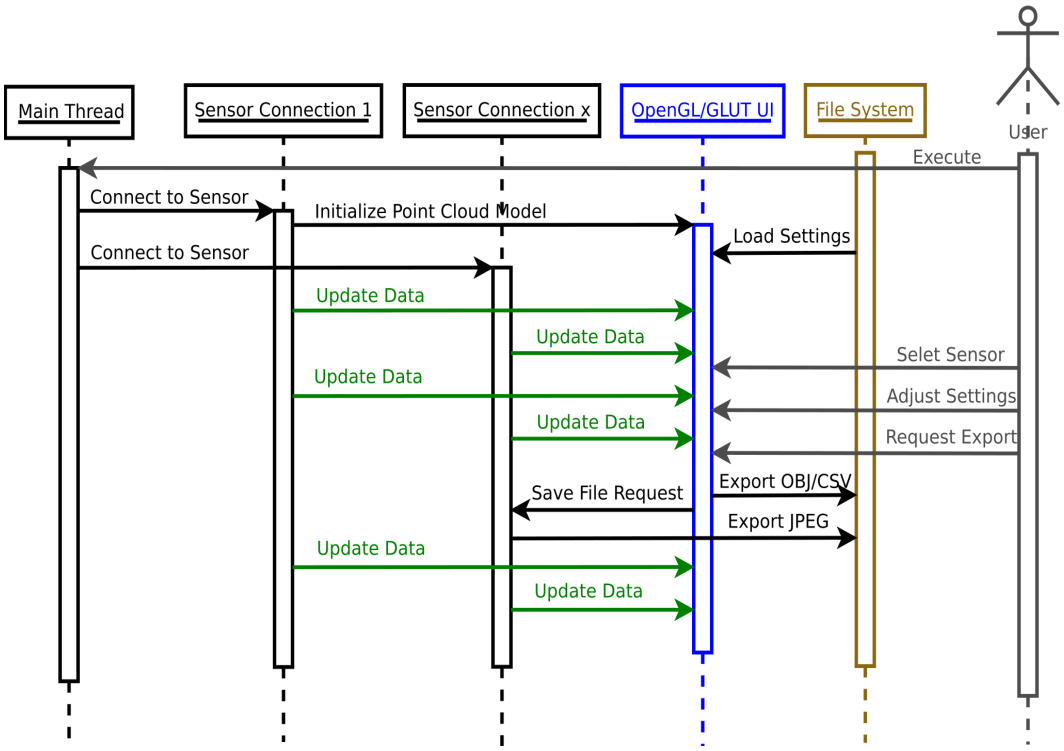


Figure 4.2.: This sequence diagram illustrates the communication of user and main threads of the prototype software for object localization from 3D point clouds. The three left most threads (illustrated in black) have been adopted from Intentas sensor connection example (see 2.2). The blue OpenGL/GLUT UI thread is the main addition to the Intenta example. It reserves a constant stream of sensor data updates (symbolized in green), interacts with the file system (in brown) and the user. The constant visual output to the user is not represented.

replies by exporting the OBJ and CSV-file into the file system. The JPEG-file is requested from the sensor connection of the currently displayed sensor which in turn exports the JPEG-file to the *File System* itself. To simplify the prototype software all possible types of export data are always exported simultaneously as soon as possible once the user requests an export. Due to underling structure of the server connection example provided sensor images from all connected sensors are visible in individual output windows while the main window and UI, *OpenGL/GLUT UI*, displays only the point cloud of one selected sensor. In fig. 4.1 the rhombus shapes symbolize visual output with the inbuilt parallelism to keep multiple output windows open.

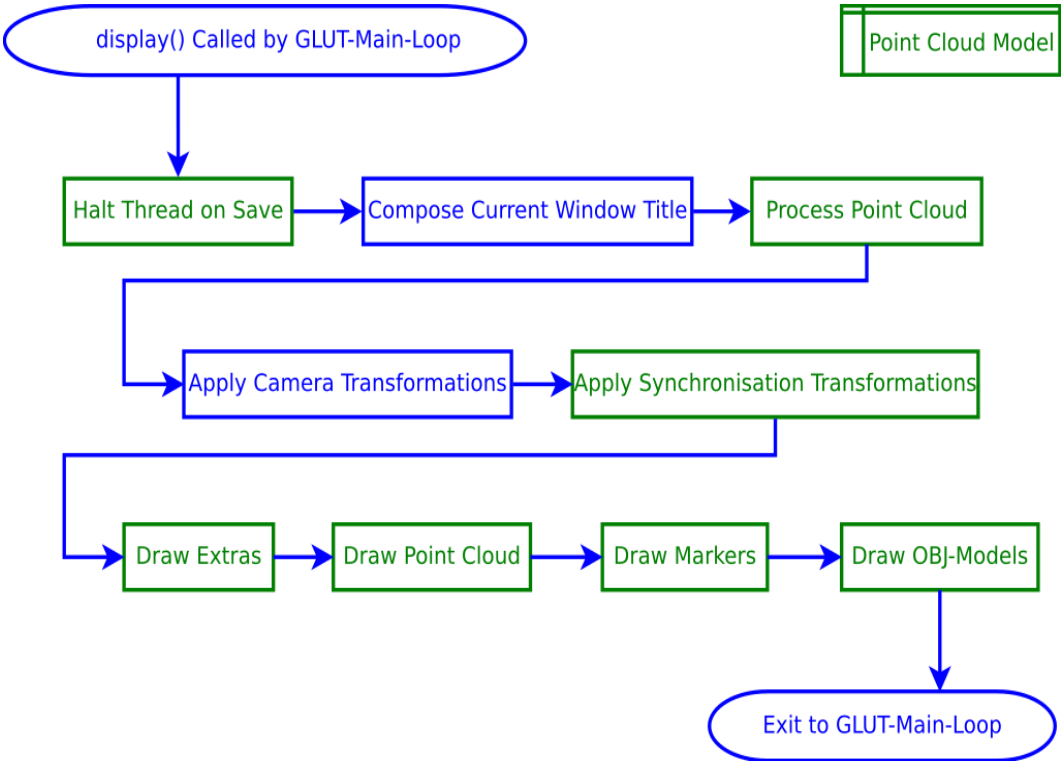


Figure 4.3.: This flowchart presents the structure of the *display()* function within the *OpenGL/GLUT UI* codes *GLUT-Main-Loop*. Represented in green are member functions of the class *Point Cloud Model*. Marked in blue are functions which belong to the *OpenGL/GLUT UI*. As visible the functionality is mostly included into the *Point Cloud Model* class, but it is executed by the *display()* function.

The display() Function The most essential part of the *OpenGL/GLUT UI* thread is the *display()* function. Everything visualized through OpenGL is selected, transformed and drawn through the execution of this function. The *OpenGL/GLUT UI* thread begins by calling the *glmain()* function, which initializes the GLUT window, inside of which an OpenGL scene is visualized. Most importantly *glmain()* starts the GLUT-Main-Loop, which constantly updates the visualization by calling the *display()* function.

Fig. 4.3 illustrates the importance of the *Point Cloud Model* class for the *OpenGL/GLUT UI* code. It incorporates almost every aspect about the model visualization. Only window title and the view towards the complex *Point Cloud Model* object are not connected to the class. When the *display()* function is executed it first waits for export files to be saved, if exporting is requested. Next the title of the OpenGL point cloud viewer window is set such that the currently

selected sensor is displayed in the title bar. Then the point cloud is processed. This means localization operations and preparations model preparations are performed according to current settings and point cloud data. After the point cloud has been processed transformations are applied to prepare drawing. As visible in fig. 4.3 transformations are categorized into the group of camera transformations and the group of synchronization transformations. While camera transformations enable the user of the localization software to view different the model from different angles and distances, the synchronization transformations apply primarily to the point cloud component of the model and enable calibrations. Size and position of the exportable OBJ parts of the model are not effected by the synchronization transformations. The positions of the object position markers (imported from OBJ-files) are computed externally of OpenGL based on the model view matrix (see 3.4.1) of the synchronization transformation during point cloud processing. This allows to export positions to OBJ and CSV-files regardless of the camera transformation. Accept for the OBJ model elements, other model elements like graphic extras, the point cloud and additional markers are drawn transformed by OpenGL. Rather graphic extras, position markers or even just the point cloud brightness values are drawn depends on the current user settings. The OBJ parts of the model are exported only if they are visible in the point cloud viewer window of the prototype software.

The keyboard() Function The GLUT windowing environment provides the *keyboard()* function to handle key input events. A switch case construct is utilized to differentiate between several input keys. Keys for the execution of several functions, provided by the *Point Cloud Model* class, are defined among keys influencing other settings for the point cloud viewer. Switching between multiple sensors is implemented through a key which triggers incrimination of the current sensor index or resets the index to zero, if no sensor is available at the next index. Every time an element of the *pointCloudModel* vector is utilized within the *OpenGL/GLUT UI* code, the current sensor index determines which point cloud model is accessed. Similar to the sensor index other numeric values or Boolean variable settings are stored globally and utilized to save key settings, such that they can effect visualizing options inside the *display()* function or similar. The key-map of the prototype software for object localization from point clouds is found in the appendix.

4.2. Visualizing the S2000 Point Cloud with OpenGL

In order to visualize the point cloud from the S2000, the 2D-RGB-image representation (see fig. 2.2d) available to the sensor connection example (see 2.2) must be converted into the coordinate values. Along with vertex coordinate the vertex brightness must be extracted similarly from the calibrated image of the left sensor

camera (see fig. 2.2b). The extracted numeric values are then feed into an instance of the *Point Cloud Model* class. Lastly the *OpenGL/GLUT UI* thread is initiated.

The showPntCloud() Function The tasks, extraction, *Point Cloud Model* updating and *OpenGL/GLUT UI* thread initiating, are implemented in the *showPntCloud()* function, which is equivalent to the black elements to the left of *Buffer Data Elements from Sensor* in fig. 4.1.

```
1  void showPntCloud(svp2_api::Sensor *sensor, bool &firstRun) {
2      svp2_api::Image pointCloudImage;
3      svp2_api::Image textureImage;
4
5      auto const pointCloudData
6      = Scoped(sensor->GetData(svp2_api::PointCloud));
7      if (pointCloudData != nullptr)
8          pointCloudImage = pointCloudData->GetImage();
9
10     auto const textureData
11     = Scoped(sensor->GetData(svp2_api::RectLeft));
12     if (textureData != nullptr)
13         textureImage = textureData->GetImage();
14
15     if (firstRun) {
16         int16_t* points
17         = reinterpret_cast<int16_t*>(pointCloudImage.data);
18         uint8_t* pntColor
19         = reinterpret_cast<uint8_t*>(textureImage.data);
20
21         pointCloudModel.push_back(
22             PointCloudModel(points, pntColor)
23             );
24         firstRun = false;
25     }
26
27     if (!openGLstarted) {
28         std::thread glThread(glmain, ref(pointCloudModel));
29         glThread.detach();
30         openGLstarted = true;
31     }
32 }
```

The function definition in line 1 defines, that a pointer to a SVP2_API Sensor variable must be passed to the *showPntCloud()* function and reference to the *firstRun* variable is provided. Both variables contain information about the current sensor connection thread. The variables *pointCloudModel* and *openGLstarted* are globally defined. To enable the creation of models for multiple sensor connections, the *pointCloudModel* variable is a vector of multiple instances of the *Point Cloud Model* class.

The variable *openGLstarted* is of type Boolean and initialized with false. In the lines 2 and 3 local *svp2_api::Image* variables for point cloud and texture are initiated. The lines 5, 6, 10 and 11 initiate pointer variables, through which data of the point cloud and its texture is accessible. The lines 8 and 13 use these pointers to access the images representing point cloud and texture and store them into the variables initiated in line 2 and 3. In lines 16 to 19 these variables are reinterpreted to extract the numeric values describing the image pixels. The *pointCloudModel* vector is then appended by a new *Point Cloud Model* instance, initialized with the numeric pixel values of point cloud and texture. To ensure only one *Point Cloud Model* instance is created per sensor connection thread the lines 16 to 24 are only executed the first time the thread runs the *showPntCloud()* function. Due to the use of pointers *pointCloudModel* kept updated with current sensor values. To visualize the *Point Cloud Model* in line 28 the *OpenGL/GLUT UI* thread is started with the *glmain()* function if it is not already running.

The *showPntCloud()* function, as presented above, is the main connection between the data stream from S2000 sensor to the *OpenGL/GLUT UI* thread. In principle the *glmain()* function could be executed with point cloud data from sources different to the SVP2_API for the S2000. To realize full functionality of the prototype software (beyond simply displaying point clouds) the *showPntCloud()* function has been modified to properly supply the final *Point Cloud Model* class implementation. Additional arguments are required by the final *Point Cloud Model* constructor.

The drawPointCloud() Function In order to visualize vertex of the point cloud based on its numeric values the *GL_POINTS* primitive type of OpenGL 3.4 is utilized. An iteration over all vertices within the point cloud is performed. While the prototype software assumes a fixed point cloud size to simplify, the point cloud image structure available through the SVP2_API provides the meta data to dynamically compute point cloud array size. It is calculated $size = rows \times columns \times channels$.

```
1  void PointCloudModel::drawPointCloud() {
2      glBegin(GL_POINTS);
3      for (int i = 0; i < 589824; i += 3) {
4          float brightness = (pntCltColor[int(i / 3)] / 255.0f);
5          glColor3f(brightness,
6                    brightness,
7                    1 - brightness);
8          glVertex3f(cloudPnts[i + 0],
9                     cloudPnts[i + 1],
10                    cloudPnts[i + 2]);
11     }
12     glEnd();
13 }
```

The illustrated *drawPointCloud()* function is a minimal simplified version, which supports drawing a point cloud with color values. In line 2 the interpretation of vertices as points is defined. Within the for loop the brightness of every vertex is interpreted as level of yellow color in lines 4 to 6. Dark colors appear blue due to line 7. Using such colors increases the visibility of blank spots within the point cloud, because the contrast to the dark background is higher. The lines 8 to 10 create the vertex at the position retrieved from the sensor. To realize full functionality of the prototype software the *drawPointCloud()* function has been extended with a subarea filter functionality.

4.3. Import and Export of 3D Objects and Positions

To create the exportable model, 3D-geometry has to be imported into the localization software prototype. For this purpose the *Model_OBJ* class is used[1]. This class type is utilized to represent geometry objects within *Point Cloud Model* class. The *Model_OBJ* class supports loading geometry from OBJ-files, drawing the geometry, transforming it and saving the transformed geometry of multiple objects into one file. The position of objects inside the export OBJ are meter values and, as such, extracted, displayed and exported into CSV-files.

Fig. 4.4 presents the relation between *Point Cloud Model* class and *Model_OBJ* class to illustrate how OBJ-files are handled within the prototype software. All objects loaded from OBJ-files are stored into a single array variable of type *Model_OBJ* inside of the *Point Cloud Model* class. This allows to implement the functions for drawing and saving all OBJ geometry by iterating through the array and executing the object specific functions. The *Point Cloud Model* class requires the definition of two OBJ-files inside its constructor. A file name for the environment model (called *roomObjName*) and a file name for an object model (called *objectObjName*) used to represent object locations as position marker. The constructor then calls the *loadOBJs()* function to load the *Model_OBJ* geometry using the load function of the *Model_OBJ* objects for each file name. This is illustrated with green arrows in fig. 4.4. All OBJ-files for import must be triangulated and are expected to carry only information about vertex positions and faces (see 3.5).

When loading a file, the *Model_OBJ* class stores not only vertices and faces into lists, but furthermore stores the biggest and smallest coordinate values from the vertices, calculates the size of the model and computes normals for light representation inside of the point cloud viewer. This information about OBJ geometry size is utilized by the *Point Cloud Model* class to calculate the center of the environment model

[1] *Model_OBJ* is based on the open source example glut_obj.cpp available at `http://openglsamples.sourceforge.net/`

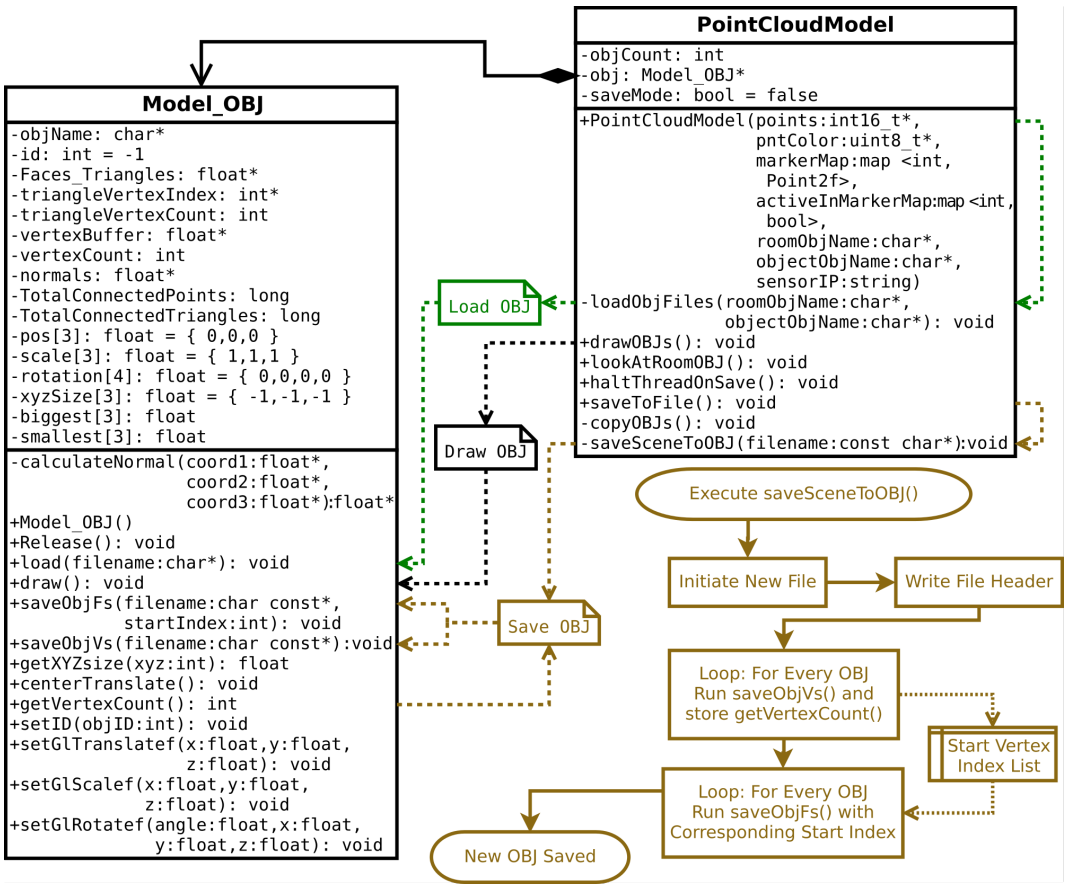


Figure 4.4.: In this UML representation the functionality of the *Model_OBJ* class is schematically explained. The *Point Cloud Model* class is reduced to the functions directly related to the OBJ-class. Dashed arrows highlight essential functions calls to illustrate the interaction of the classes. The green arrows connect functions concerned with importing OBJ-files. Dashed arrows in the center show functions for drawing imported OBJ geometry. The export of multiple OBJ geometry elements into one file is illustrated in brown.

inside the *lookAtRoomOBJ()* function, which is part of the camera transformations applied by the *display()* function mentioned in 4.1 fig. 4.3. To represent multiple object position with one loaded OBJ it is copied. During point cloud processing (see *display()* function in 4.1) the function *copyOBJs()* is called and creates marker geometry instances for every position localized within the point cloud. Both, *Point Cloud Model* and *Model_OBJ* class, use matrix representations of the transformations needed to reposition the marker geometry (see 3.4.1). As described in 4.1 for the *dispaly()* function, the OBJ geometries vertices are transformed on the CPU to make it exportable.

Once the geometry is loaded and markers are transformed to object positions, they are visualized by OpenGL or exported into a combining OBJ-file. The visualization is triggered by the *Point Cloud Model* class function *drawOBJs()* which iterates though the available *Model_OBJ* instances and calls each ones *draw()* function. The *drawOBJs()* function also displays the position of each object by drawing distance lines parallel to the coordinate axes and labeling them with the coordinate values of the geometry object. To prepare export of the object marker position coordinates into a CSV-file, a list with all values is stored inside *drawOBJs()* as well. While CSV-files are exported by alternating the stored values and semicolon chars, exporting the combined OBJ-file is more complex. The interactions required to export a single OBJ-file with the geometry from multiple *Model_OBJ* instances are highlighted with brown color in Fig. 4.4. All file exports are triggered through the *saveToFile()* function of the *Point Cloud Model* class. The same class incorporates the *saveSceneToOBJ()* function, which directs the merge of multiple *Model_OBJ* instances into one file. To accomplish this task the function uses the capability of the *Model_OBJ* class to individually store vertices and faces into a file by appending it. *saveSceneToOBJ()* at first creates a new file and writes a file header into it. Next for every *Model_OBJ* instances vertices are saved by utilizing its *saveObjVs()* function and the start vertex position for ever object is stored using the *getVertexCount()* function. In another loop, to properly save the faces of each *Model_OBJ* instance (see 3.5), the indices which describe faces are incremented by the previously stored start vertex position. This is done by the *saveObjFs()* function of the *Model_OBJ* class which additionally creates an OBJ object element, to identify the source of OBJ parts (see 3.5). The OBJ-file is complied.

To organize *Model_OBJ* instances, the origin file name is stored into the *objName* variable and an *id* variable is set to $-$ to indicate the absence of an id. In case ids are available for the locations represented by instances of *Model_OBJ*, they are assigned to the instance by the *copyOBJs()* function. Next to the origin file name the ids are used as part of the object names in exported OBJ-files. For instances without an id the starting vertex inside the OBJ-file substitutes the id in the object name. Also inside CSV-files the id values are listed last, after the three coordinate values.

4.4. Localizing Marker Positions with OpenCV

In this section the structure of the *OpenCV Operations and Update of Point Cloud Model* algorithm in the context of 4.1 (see fig. 4.1) is described. The algorithms primary purpose is to communicate the position and ID of an ArUco marker in the point cloud texture to the *Point Cloud Model*. An imported secondary function of the process is to draw found markers into the OpenCV image, which is saved as an JPEG-file if an export is requested. Lastly the conversion of 2D-coordinates to a 3D point cloud representation is explained.

Fig. 4.5 shows, that the first and most complex step in marker localization is handled by the ArUco library module for OpenCV (see 3.3.2). The marker detection algorithm searches the point cloud texture image for every marker of the *DICT_4X4_50* dictionary. If markers are located their corner coordinates and their IDs are stored into separate C++ vector elements. For all found markers their center coordinates are calculated as the intersection point between the diagonals constructed out of the marker corners. Only the center coordinates of the marker are utilized by the software prototype for object localization from point clouds. To avoid race condition problems due to inter thread communication through the point cloud model marker coordinates are not directly synchronized with the *Point Cloud Model*. When a previously visible marker is not located anymore and thus not present in the list of markers, the attempt to visualize it could crash the software. Consequently once a marker has been located and stored inside of the *Point Cloud Model* it is never removed from the model. To hide markers which are no longer detected an additional marker visibility variable controls rather a marker is visible. The *Update List of Visible Markers* element in fig. 4.5 symbolizes changing the value of visibility variables for currently not detected IDs to invisible, the value for visible markers is set to visible and previously unknown IDs are initialized as visible. Before updating the marker coordinates inside the *Point Cloud Model* to the current marker position, the marker IDs are drawn into the, not yet visualized, OpenCV image of the point cloud texture. After the coordinates have been updated the algorithm checks if it is ordered to export the current OpenCV image as a JPEG-file. To do so the *Point Cloud Model* function *hasSavingOrder()* is utilized. In case the export is requested, the JPEG-file is saved and to create a new file name for each image the number, representing the current count of exported files, is added to the file name. A *Point Cloud Model* function called *takeSavingOrderNumber()* provides this number and turns off the save request inside the *Point Cloud Model*. As illustrated in fig. 4.5 and 4.1 the now altered image is visualized by OpenCV in the end.

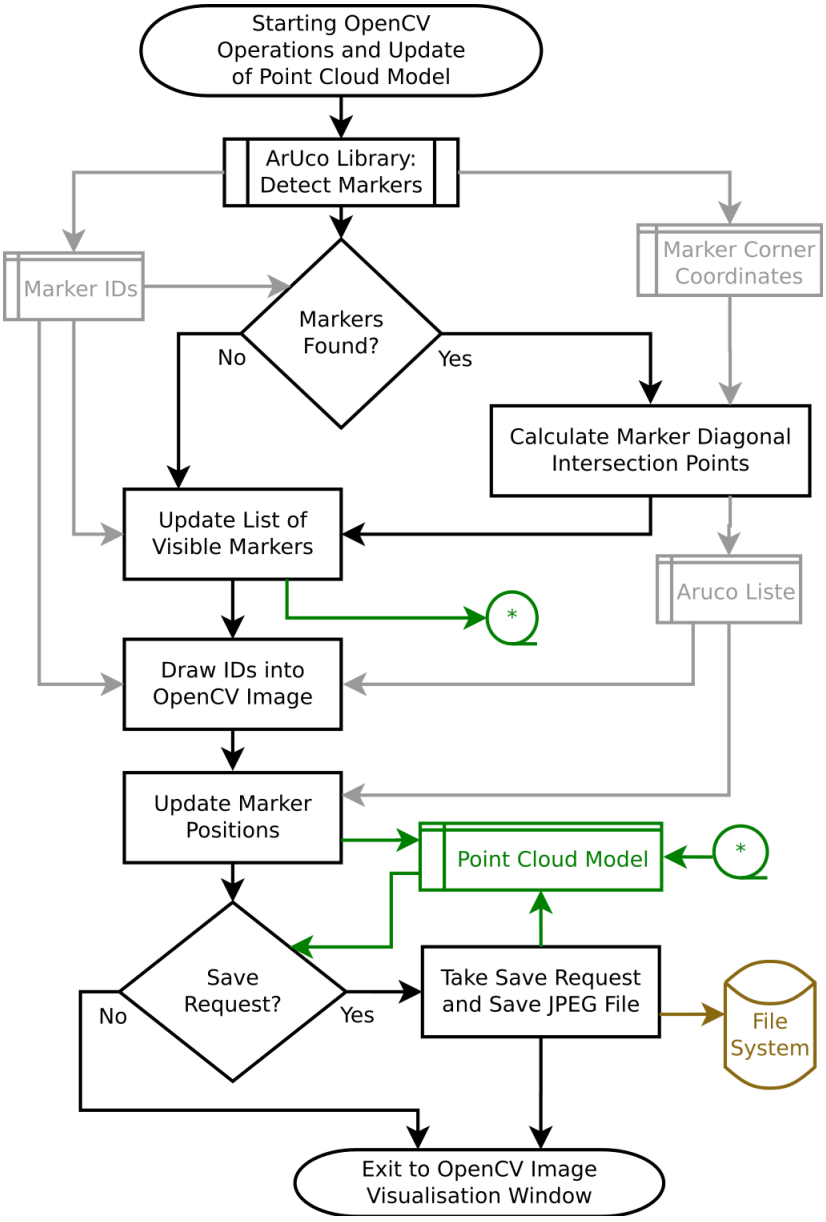


Figure 4.5.: This flowchart illustrates the algorithm *OpenCV Operations and Update of Point Cloud Model* in the context of fig. 4.1). Temporary local variables are colored gray, the *Point Cloud Model* communication is green and export related items are brown. The main flow is colored black and contains elements which summaries loops over multiple markers to simplify comprehension.

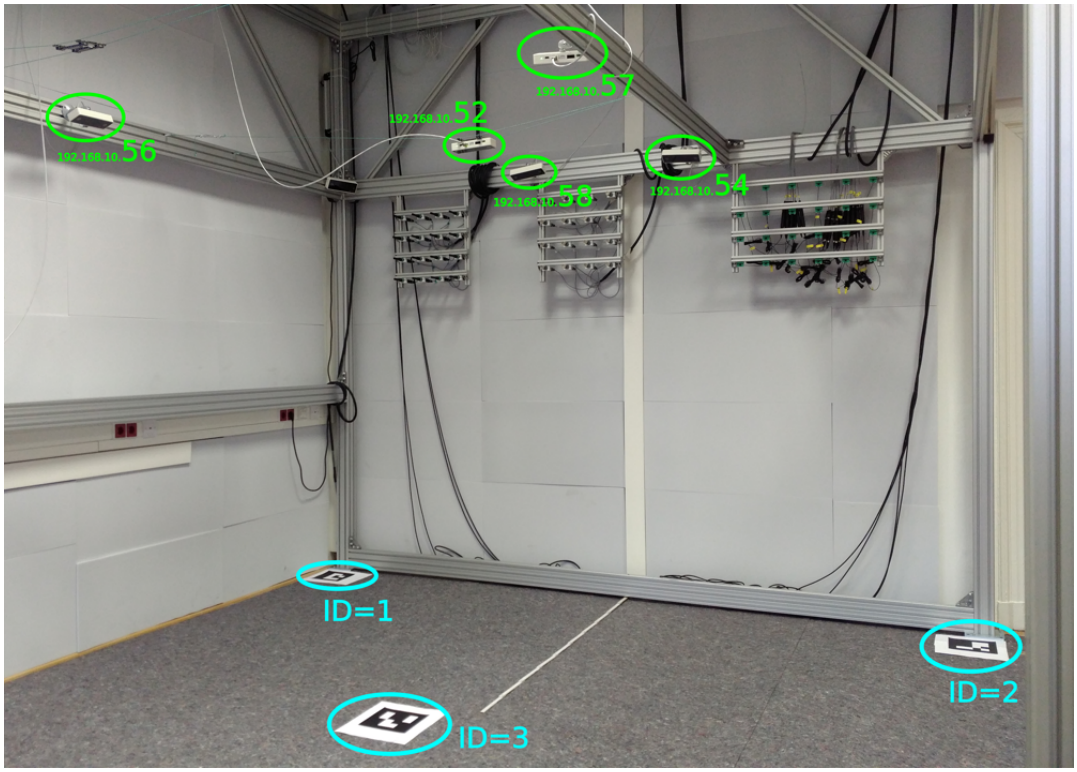
The pnt2fToPCIndex() Function The *Point Cloud Model* class is constantly updated with marker coordinates from OpenCV. However, these coordinates are only two dimensional. In order to utilize the locations found, in the 3D-model, the 2D-coordinates must be mapped into 3D-space. As described in 2.1, the S2000 provides its point cloud in an image structure derived from the stereo images, such that the point cloud texture image (from left camera) maps its brightness values to the point cloud image through equal 2D-coordinates. The *pnt2fToPCIndex()* function provides this mapping from texture image coordinate into the 3D vertex position.

```
1 int PointCloudModel::pnt2fToPCIndex(Point2f cvPnt, int widthImage) {
2         return ((int)cvPnt.y*widthImage + (int)cvPnt.x) * 3;
3 }
```

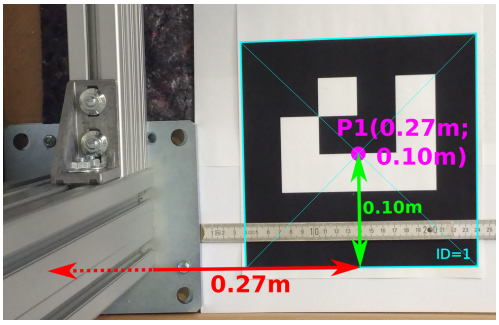
Line 1 defines the function and its two input arguments, the *cvPnt* 2D-coordinate from OpenCV and the width of the image (512 pixels is the common value). Because the point cloud has been converted into an array with one dimension, line 2 calculates the number of pixels in rows above *cvPnt* and adds it to the position of *cvPnt* within its row. The one dimensional index for *cvPnt* is then multiplied by the 3 coordinate channels, before its returned as the point cloud index. Any such calculated index is utilized similarly to the loop variable i in the *drawPointCloud()* function described in 4.2. Through a map type variable all marker coordinates are linked to their ID inside of the *Point Cloud Model*. Before the coordinates are utilized, the visibility of the point marker is checked by ID. Currently visible markers are highlieded inside the point cloud using the *drawMarkers()* function, which draws the ID numbers into the marker positions.The *processPointCloud()* function (see 4.1, fig. 4.3) provides options to located points and areas within the point cloud based on marker coordinates. The 3D location marker geometry described in 4.3 is also translated based on OpenCV coordinates if the option is selected by the user. The marker IDs are then also set as IDs for the *Model_OBJ* instances at their position.

4.5. Transforming Point Cloud into Reference Model Coordinate System

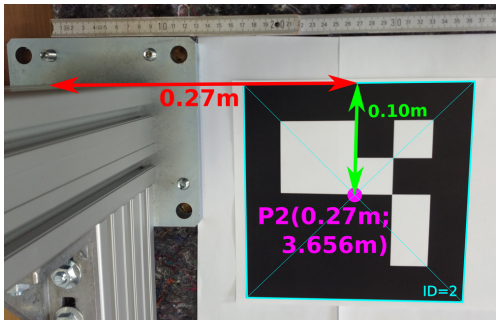
The environment model which has been loaded from OBJ is defined such that it is equivalent to scale and orientation of the frame cage in fig. 2.4b. Now the point cloud must be transformed such that it is synchronized with the frame cage model. As mathematically described in 3.6, the two step rotation is utilized to transform the point cloud into the reference model orientation. The one step rotation method is not utilized because it is less optimized for avoiding point errors in computer implementation. Scaling of the point cloud is also implemented as described in 3.6.



(a) Overview of thee ArUco markers highlighting reference points according to their IDs. The highlighted sensors have a good view of all markers.

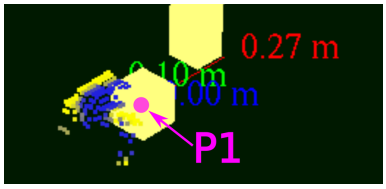


(b) Reference point 1 at the center of the ArUco marker with ID 1.

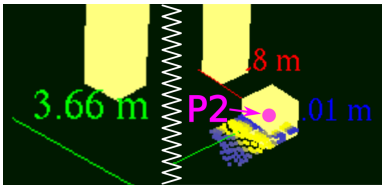


(c) Reference point 2 at the center of the ArUco marker with ID 2.

Figure 4.6.: The positions in which the reference points for calibration of the point cloud model must be positioned: Distances in x direction are illustrated in red, green marks distances into the y direction. ArUco codes and IDs are highlighted in blue. Five S2000 sensors and their IPs are marked green.



(a) Reference point 1 has manually been moved into a position that matches the real world (see fig. 4.6b). The location marker itself is visible and the distances to all 3 axes.



(b) Reference point 2 has manually been moved into a position that matches the real world (see fig. 4.6c). The image is split into left, Y-distance of marker, and right, the marker and remaining distances.

Figure 4.7.: In two screen shots from the prototype software for object localization from point clouds, the manual calibration setting after the automatized transformation into the coordinate system is shown. Visible are two foots of frame cage pillars and a cube object marking point locations. X-Y-Z-coordinates are visible in red, green and blue. The pink labels have been added on top of the screen shot.

The practical implementation of the calibration method begins with a definition of three reference points in the laboratory. Ideally the first two points should be equal to the corners of the frame cage on the y-axis. However, these corners are not visible to laboratory sensors because of the pillars of the frame cage. Thus, the reference points are defined at a close position to these corners. In order to make the points visible to the sensors and localization software $20 \times 20cm$ ArUco codes are placed in the laboratory. Fig. 4.6 illustrates a proper configuration of the markers inside the frame cage. The markers are placed inside on the ground such that both are at the same distance to the y-axis (0.27 m). Because of the marker size reference point 1 (see fig. 4.6b) is 0.1 m away from the x-axis and reference point 2 (see fig. 4.6c) is 3.656 m away from it. The third reference point does not have to be at a specific distance to x and y-axis, but it must be visible to the utilized sensors and not between the first two reference points. The position in fig. 4.6a is more than 2 meters away from the y-axis. The 3 reference points define the ground plain. The reference points 2 are used for scaling and rotating around reference point 1.

The prototype software localizes the markers as described in 4.4. When calibration is triggered by software users the *setRefPntsFromMarkers()* function stores the current 3D-coordinates of the marker codes as reference points into the *Point Cloud Model* for the transformation. The mathematical formulas of 3.6 are implemented in the *calculateSyncRotation()* function. At first all three points are translated such that reference point 1 is at the coordinate origin. Then the rotation axis and angle

for the rotation, which transform the second reference point into its position on the y-axis, are computed. These calculations do not utilize matrix algebra and are implement using standard C++ math functionality. However, to calculate the second rotation angle (see two step rotation in 3.6), the position of the third reference point after an OpenGL style transformation with the so far calculated values is required. Hence, a model view matrix is generated by calling the so far possible synchronization transformations. This matrix is then extracted to calculate the temporary position of the third reference point (see 3.4.1). Then the second angle and the scaling value are computed. All the calculated transformation values are stored in the *Point Cloud Model* in order to apply them to every frame computed with the *display()* function (see fig. 4.3 in 4.1) and to the OBJ geometry during the *Process Point Cloud* phase.

When the automatized transformation into the frame cage model coordinate system was performed successfully the position of reference point 1 is equal to the coordinate origin. Consequently it is manually translated to its real position with the software prototype. The distances to coordinate axes are visualized in the prototype software to support such and similar operations (see 4.7). The second reference point position is corrected once the first reference point has been properly transformed. To correct the second point position only the scaling option is utilized, because it does not effect the previously set position of reference point 1. To avoid overlapping of distance values from multiple reference points, the ArUco markers associated with the unused reference point are simply covered in reality. Consequently they are no longer visible in user interface of the software. Fig. 4.7a presents the OpenGL visualisation of reference point 1 after automatized and manual calibration, while the other reference points are hidden. Fig. 4.7b presents the equivalent visualisation of reference point 2 and its calibration. The third reference point is not manually adjusted, it should, however, have a Z-coordinate-value of $0.00m$. Precise calibration leads to more accurate model representations. Once the calibration is completed a configuration-file for the selected sensor is saved by the software prototype user. This configuration will be applied to the sensor every time it is connected to by the object localization software. To save calibration changes afterwards the calibration-file is overwritten.

4.6. Implementation of Point Cloud Mask & Filter

Masks, Filters and the concept of subareas, as implemented for the prototype software, are described in 3.2. The point cloud is split into subareas by rounding the coordinate values of each vertex. Multiple vertices in the same subarea share the same rounded coordinate values. The *createCloudMask()* function, illustrated in 4.8, implements this concept. The function returns a vector of *maskArea* instances, identifying all subareas which contain at least a single point cloud vertex. The

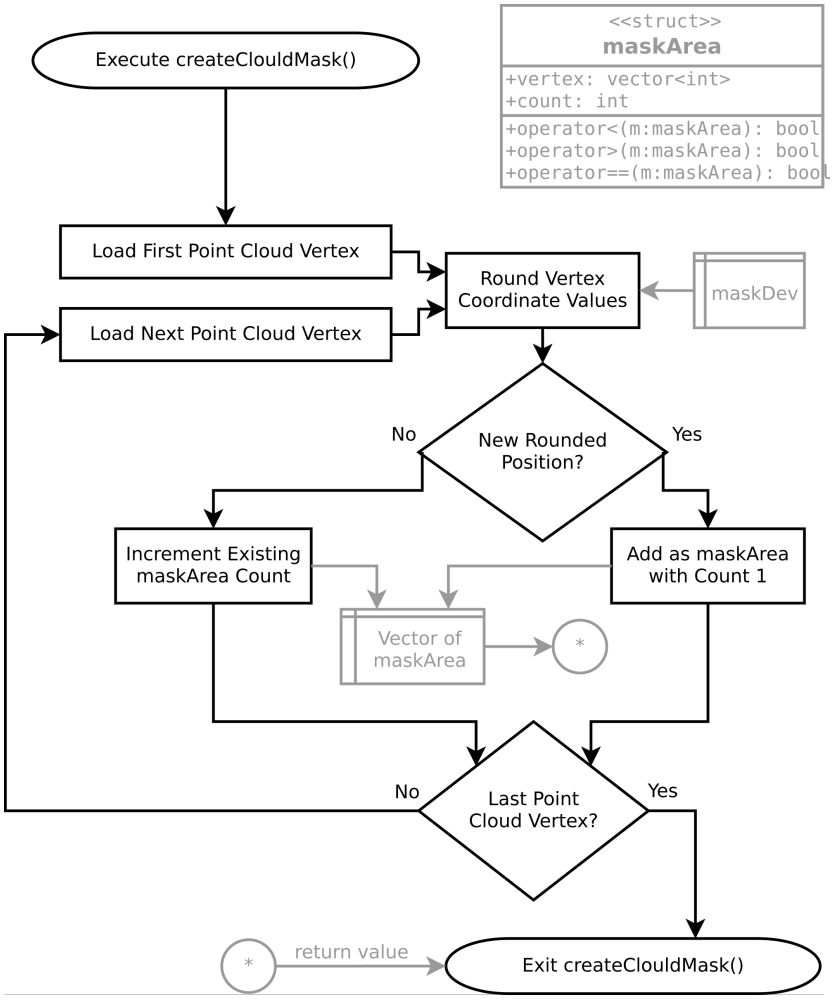


Figure 4.8.: This flowchart illustrates the algorithm for the generation of *maskArea* vectors, the *createCloudMask()* function. Such vectors contain not only a number of masked point cloud subareas, but also count the number of point cloud vertices within the subarea. Variables are colored gray and the main flow is black.

maskArea struct is defined to count the vertices found within each subarea of the point cloud. Thus, it can be interpreted as a point cloud mask, hiding all subareas inside of which vertices were found when the mask was recorded by executing the *createCloudMask()* function. If the number of vertices per subarea (or *maskArea*) is interpreted as the density value for the density filter, the same vector of *maskArea* objects may be interpreted as a filter.

In order to create a mask for the *Point Cloud Model*, its member function *useAsCloudMask()* is called by the *keyboard()* function (see 4.1). The newly generated mask is stored as a variable of the *Point Cloud Model* and can be utilized immediately. Another key option of the prototype software allows the user to back up (save) the point cloud mask. If such a mask-file exists for a sensor it is automatically loaded whenever to prototype software is started and connected to the same sensor again. To ensure all computations utilizing the mask are executed properly, the *maskDev* value, which defines the size of mask areas, is stored inside the mask-files. Interactively changing *maskDev* is not supported in the prototype software, but through changing the mask-file externally experiments may be conducted. To generate a filter, for every point cloud frame a vector of *maskArea* objects is created using the *createCloudMask()* function. Subareas are than filtered, if the current *count* value of the *maskArea* is less than a filter value. The point cloud filter value is alterable interactively. Filter values are stored into and loaded from the sensor specific configuration-file.

The filtered *maskArea* vertex positions are utilized as an alternative coordinate source for OBJ marker objects. Although the rounded coordinate values are not as accurate as the marker positions provided by OpenCV, it is a useful alternative, since no marker preparation is required. The *drawPointCloud()* function introduced in 4.2, is improved by inserting if conditions to check rather a vertex is masked or filtered before drawing it. The vectors of *maskArea* objects are constantly searched by vertex positions. Every single point cloud vertex is converted into a *maskArea* object and then searched for in the mask & filter vectors. To enable fast computation of this high number of search operations, the *std::binary_search* function is utilized. Comparison operators for *maskArea* are defined such that comparing the struct instances to each other is equal to comparing the member variable *vertex* of the instances. This allows the *std::binary_search* function to process *maskArea* objects.

4.7. Implementation Summary

For the implementation of the software prototype for loudspeaker localization from point clouds, 56 functions for 3 classes and multiple code units have been newly created. All new and modified functions have descriptive names and comments explaining the implemented functionality. Based on the *SVP2_API Multisensorconnection Demo* (see 2.2) by the Intenta GmbH and an example implementation of an OBJ-

File Type	File Name	Contained Data
Comma Separated Value	Number + IP + ".csv"	located coordinates and IDs
Wavefront OBJ	Number + IP + ".obj"	3D-model of the laboratory
JPEG-Image	Number + IP + ".jpg"	512×384 pixels image
(Internal) Mask File	IP + ".msk"	a list of masked subareas
(Internal) Settings	IP	calibration and adjustments

Table 4.1.: The files saved by the software prototype for localization. The last two files listed are created for the software prototype only, while others are exchange formats.

file-loader[2], the prototype was designed. The example by Intenta was modified such that it triggers a newly implemented OpenGL visualization of the S2000 point clouds. To control the software prototype centered with the OpenGL/GLUT UI more than 30 key commands (see attachment A) have been created. The core of the software prototype implementation is the *Point Cloud Model* class containing 33 functions alone. It was created to at as a connector between threads, a tool for point cloud analysis and a visualization tool. As such the *Point Cloud Model* class also visualizes OBJ-geometry. The *Model_OBJ* class from the OBJ-file-loader was extended by 4 functions which now enable exporting altered OBJ-geometry. Additionally two existing functions of the *Model_OBJ* class have been modified to add functionality. An essential part of the implementation is the localization of optical markers in the point cloud (see 4.4) and the model calibration (see 4.5). Model calibration is a part of the *Point Cloud Model* class, while the localization of optical markers relies on the modification of the *SVP2_API Multisensorconnection Demo* to which 5 functions with the purpose of connecting an OpenCV ArUco detection (see 4.4) to the *Point Cloud Model* class have been added.

To run the loudspeaker localization software at least one S2000 sensor must be connected other Ethernet and the files *lab.obj* and *object.obj* must be in the directory from which the program is executed. A guide on how to use the localization software is given in attachment B. The software prototype for loudspeaker localization from point clouds exports three types of files representing the current laboratory setting, a masking file and a file containing sensor configurations. Tab. 4.1 gives an overview of all file formats saved by the software prototype.

[2]Find the open source example glut_obj.cpp at `http://openglsamples.sourceforge.net/`

5. Evaluation of Localization Methods

The evaluations of the software prototype implemented in the course of this thesis is presented in this chapter. At first in 5.1 general considerations in preparation of the tests are made. The first test conducted in 5.2 evaluates the stability of the reference points utilized calibration of the sensors. 5.3 documents the tests conducted in order to analyze the point cloud precision and accuracy. Finally, the tests conducted in 5.4 compare different methods of loudspeaker localization.

5.1. Preliminary Considerations

The prototype software for the localization of loudspeakers inside the audiovisual laboratory utilizes the point clouds of S2000 sensors. As mentioned in 2.1, the sensors have a 97^o field of view and their point cloud is represented inside an image structure of 512×384 pixels, similar to the point cloud texture image. Based on the sensors field of view and simple a triangulation the area captured by the sensor in 1 m distance to it, is ca. 2.26 m across. Since this range is represented by $384 \rightarrow 512$ pixels, the distance between points represented through pixels is about 5 mm at 1 m distance to the sensor. It is concluded that the general distance between the points represented by pixels is roughly $5mm \times d$, where d is the distance of point to the sensor. Consequently, with the sensors installed at about 2.45 m above the ground, a precision of no more than a centimeter is aspected in testing. The conversion the from 2D-coordinates to 3D vertex positions may additionally impair the precision and accuracy of measurements.

In order to be able to evaluate the point cloud accuracy of a sensor it must be calibrated as described in 4.5. Five sensors with a good view over the markers (a requirement for the calibration) have been chosen for evaluation. They are positioned as illustrated in 4.6a. The last two digits of each S2000 IP will be utilized to refer to the sensors in the evaluation. The calibration transformation itself is evaluated in 5.2 for each sensor. Since the ArUco markers (see 3.3.2) are utilized to determine stable positions in 2D[1], the same markers[2] as used for calibration of the software are reused to test accuracy and precision of the point cloud in 5.3. It is not tested under which conditions the ArUco markers are detectable, since their detection depends,

[1] 2D-coordinates of the point cloud texture determine 3D-coordinates inside the point cloud as described in 2.1

[2] ArUco markers of the *DICT_4X4_50* dictionary with a size of $20 \times 20cm$.

among other factors, on the lighting conditions which cannot be documented and reproduced in the audiovisual laboratory.

Reference coordinates are independently measured using traditional measuring instruments and a laser measuring device. Based on the point cloud test, the mask & filter implementation (see 4.6) of localizing loudspeakers is evaluated against the method of using markers. The subareas, which represent a positions of objects without markers when located, are evaluated in size and position.

5.2. Stability of Calibration and Reference Points

The translation, two step rotation method and scaling approach introduced in 3.6 and implemented in 4.5 are evaluated alongside the stability of the point cloud by recording the coordinates of the reference points from multiple sensors after calibration. Fig. 5.1 and 5.2 illustrate the fluctuations measured in 3D-coordinates of the reference points over 25 recordings with the prototype software for localizing from 3D point clouds. Within the point cloud texture image, localized 2D-coordinates of the markers do not change with exception of a single change by one pixel of reference point 3 in the recordings of sensor 57. Thus, changes in the 3D-coordinates of every other sensor and reference point must be caused by the fluctuations of point cloud vertex positions. Tab. 5.1 compares the same coordinate recordings. All reference marker positions remain within one up to two centimeters of the ground, accept for rare extreme values. In the test recordings such values did not occur more than three times per 25 recordings. Point 1, the center of the calibration transformations (see 3.6) is the most consistent in its location, while point 2 (the furthest away the center of the calibration transformations), is slightly more inaccurate. However, considering the sensors resolution (see 5.1), both points are in the positions targeted by a calibration transformation (see 3.6). Also the z-coordinate of the third reference point is zero or within centimeters of it. Thus, transforming, rotating and scaling of the point cloud does not fail. The x and y coordinates of the third reference point (see tab. 5.1) are very stable within the individual sensors. However, a difference of up to half a meter between the positions, the sensors identify, appears to demonstrate point cloud distortions. The existence and nature of such distortions is investigated in 5.3.

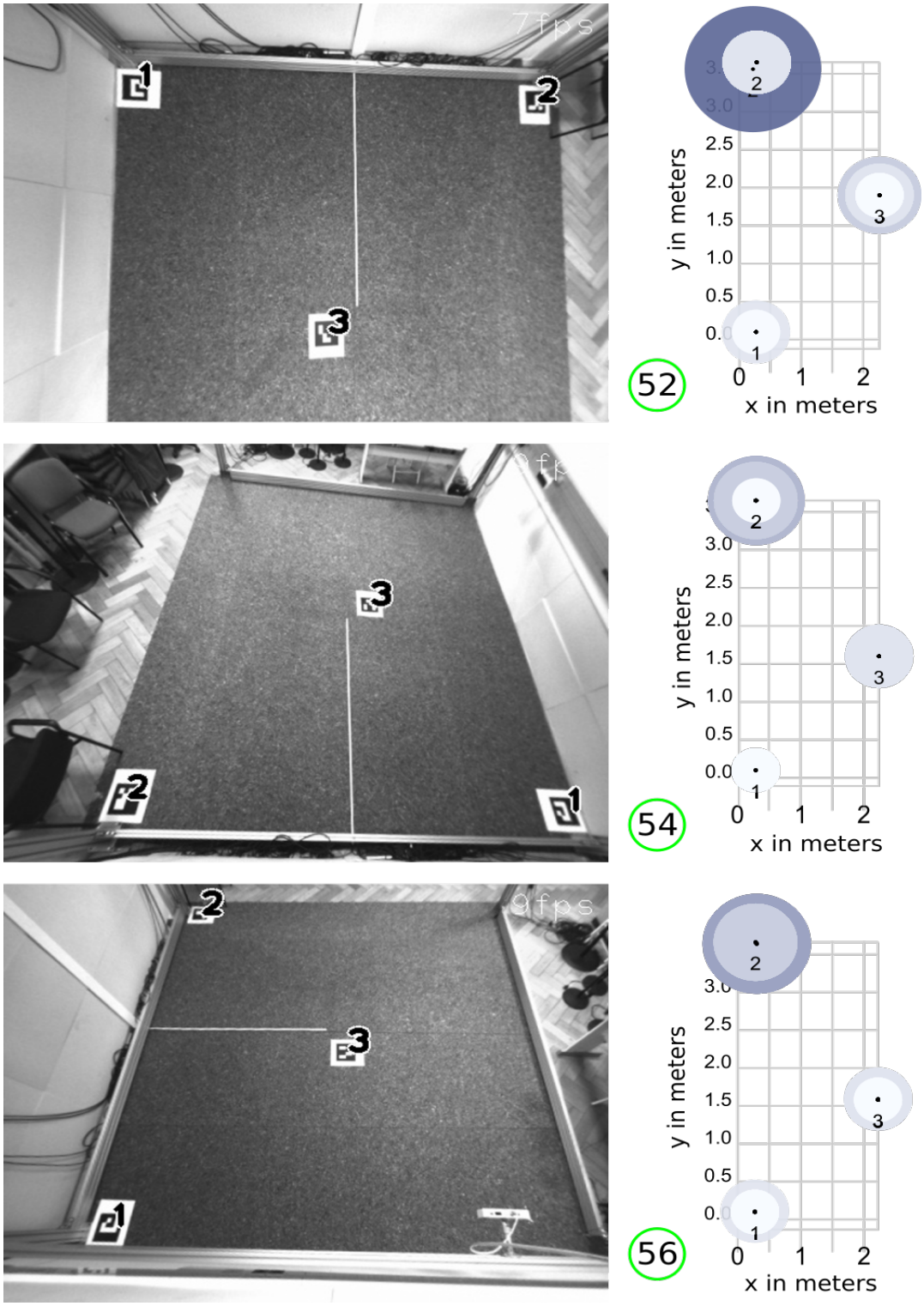


Figure 5.1.: Part 1 of the calibration test visualization. The results from sensor 52, 54 and 56 are presented in an image and a scatter plot. For the results of sensor 57 and 58 or more about the visualization see 5.2.

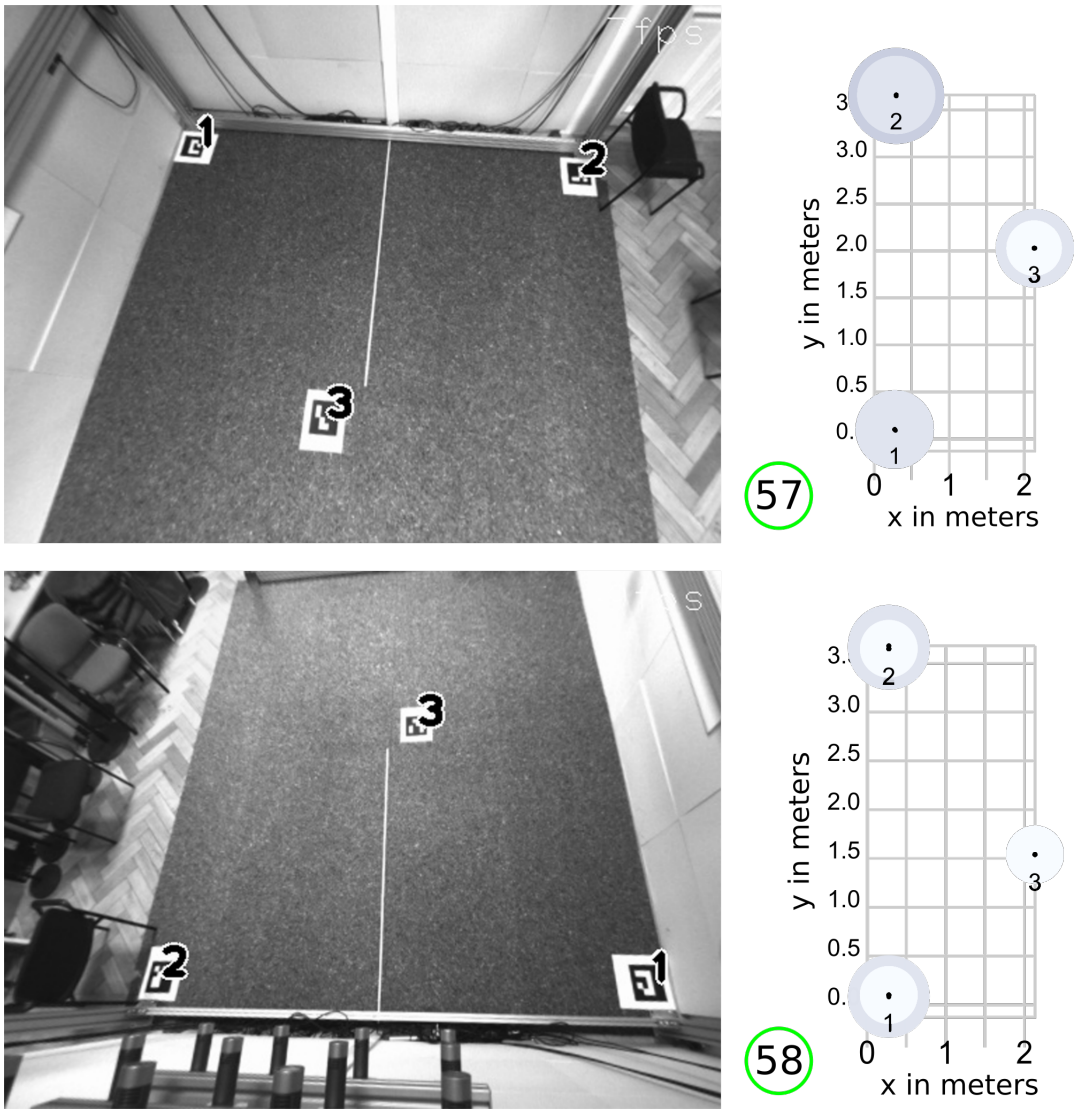


Figure 5.2.: Part 2 of the calibration test visualization. The point cloud texture images of sensor 57 and 58 are presented. In both parts of the calibration test visualization, the images with IDs of detected reference markers were exported along with one of the 25 (mostly equal) 3D-coordinate measurements visualized in the scatter plot on the right. The distance to zero on the z-axis is indicated by increasing size and blue color of the circles in the plots. The bluest and biggest circle (see sensor 52 in fig. 5.1 of part 1) indicates a computed distance of $7cm$ to the ground.

Point	Sensor	X	Y	Z
1	52	$27 \pm 0cm$	$10 \pm 0cm$	$0 \pm 1cm$
	54	$27 \pm 0cm$	$10 \pm 0cm$	$0 \pm 0cm$
	56	$27 \pm 1cm$	$10 \pm 1cm$	$0 \pm 1cm$
	57	$27 \pm 1cm$	$10 \pm 1cm$	$1 \pm 0cm$
	58	$27 \pm 0cm$	$9 \pm 1cm$	$1 \pm 1cm$
2	52	$26 \pm 4cm$	$360 \pm 5cm$	$1 \pm 6cm$
	54	$27 \pm 1cm$	$365 \pm 0cm$	$0 \pm 3cm$
	56	$28 \pm 2cm$	$365 \pm 1cm$	$2 \pm 2cm$
	57	$29 \pm 0cm$	$365 \pm 1cm$	$1 \pm 1cm$
	58	$27 \pm 0cm$	$365 \pm 3cm$	$0 \pm 1cm$
3	52	$226 \pm 0cm$	$190 \pm 0cm$	$1 \pm 1cm$
	54	$224 \pm 0cm$	$160 \pm 0cm$	$1 \pm 0cm$
	56	$222 \pm 0cm$	$159 \pm 1cm$	$0 \pm 1cm$
	57	$214 \pm 1cm$	$203 \pm 0cm$	$0 \pm 1cm$
	58	$213 \pm 0cm$	$154 \pm 0cm$	$0 \pm 0cm$

Table 5.1.: Reference point positions and fluctuations after calibration. The coordinate values consist of most frequent position and maximum fluctuation over 25 test recordings per sensor.

5.3. Point Cloud Precision and Accuracy

In order to evaluate the point cloud measurements of different sensors to manually measured references, a test pattern with 4×4 ArUco markers is created. The markers, of the *DICT_4X4_50* dictionary with a size of $20 \times 20cm$, are placed with a distance of $30cm$ to each other as visualized in 5.3a. The test pattern is recorded 20 times with all five sensors calibrated in 5.2. This process is repeated at five different heights. Thus, generating a total of more than 1500 localized coordinates per sensor. Fig. 5.3 presents all five recording heights from the perspective of sensor 57. Due to technical limitations the height of the test pattern may differ partially by a maximum of $\pm 5mm$, which is adaptable since the results of 5.2 show a common position fluctuation by $\pm 1cm$.

For each sensor all measurements are now concatenated from the CSV-files recorded with the prototype software. To help analyze the test results they are visualized with the RAWGraphs visualization tool[MEC+17], which is optimized for plotting data from CSV-files. To further customize the visualization Inkscape[3] is utilized. Using these tools the diagrams in fig. 5.4, 5.5, 5.6, 5.7 and 5.8 are created. Diagrams

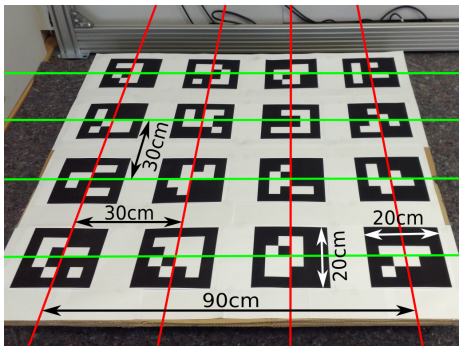
[3]Inkscape is a vector graphics editor which is open source. See `https://inkscape.org/en/`

for each sensor are accompanied by a screenshot of the prototype software's UI (e.g. 5.4a), where the point cloud reviles the sensor position due to the shadow like artifact described in 3.1.

Sensor precision is tested by comparing multiple measurements from one marker (differed by ID) and one sensor. The closer the located coordinates are to each other, the better is the precision of the sensor. Although notable irregularities occur in the tests performed with sensor 54 (see fig. 5.5) the majority of maker localizations of all sensors are within $\pm 5cm$ coordinate distance to each other. However, fig. 5.4c, 5.6d, 5.7c and 5.8c illustrate very well, displacements of x-y-coordinate values appear in z-direction. About $15cm$ of difference in the otherwise precise x-y-coordinate values appear when the measurements over the full test range of $1.28m$ are compared (see fig. 5.4b, 5.6b, 5.7b and 5.8b).

The sensor accuracy is tested by comparing the ground truth coordinates to the measurements recorded with the prototype program. Some of the coordinate values recorded by the sensors 52 and 58 deffer by $\pm 30cm$ from the ground truth position. As visible in fig. 5.4 and 5.8, this creates misrepresentations where detected markers could be mistaken for their neighbours if they had no ID assigned to them. An extreme case is sensor 57 (as shown in fig. 5.7), which has y-coordinate values of up to $50cm$ bigger than expected.

The over all best test results in accuracy and precision are achieved by the sensors 52 (see 5.4) and 56 (see 5.6) at current calibration. Altered calibration methods may improve accuracy.



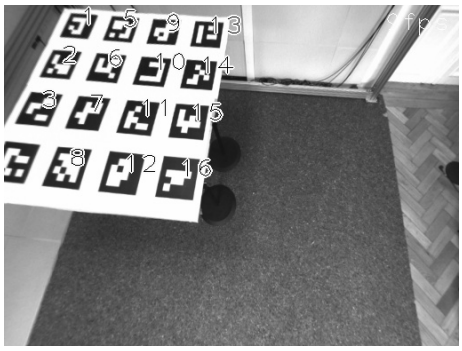
(a) The test pattern with 4×4 ArUco codes.

(b) Test pattern at the height of $0.15m$

(c) Test pattern at the height of $0.83m$

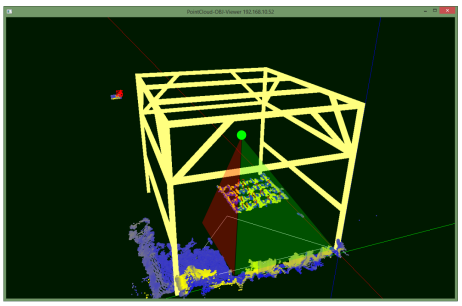
(d) Test pattern at the height of $1.13m$

(e) Test pattern at the height of $1.28m$

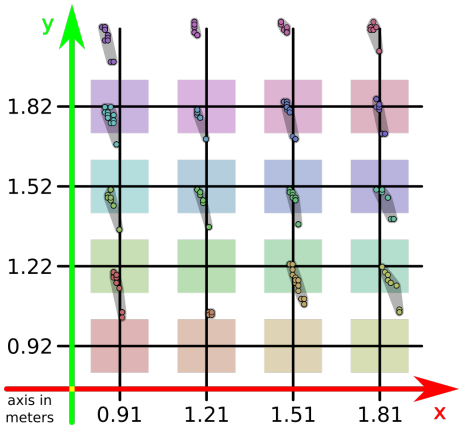


(f) Test pattern at the height of $1.43m$

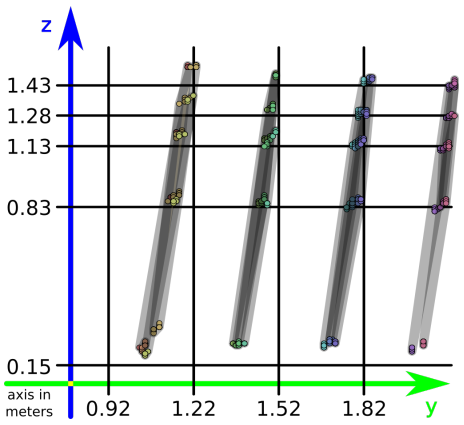
Figure 5.3.: The test for point cloud precision and accuracy: Fig. 5.3a presents the test pattern made of 16 ArUco codes $20 \times 20cm$ in a distance from $10cm$ to each other. The different testing heights are presented in fig. 5.3b, 5.3c, 5.3d, 5.3e and 5.3f from the view of sensor 57.



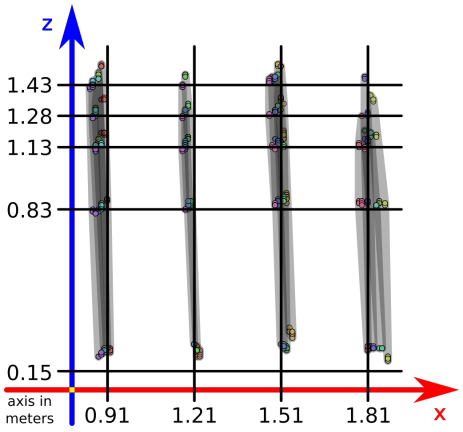
(a) The position of sensor 52 (symbolized by a green dot) and its view towards the test pattern in a screen shot of the localization software prototype UI. The visualisation shows the point cloud of sensor 52 for the test setting of height $0.83m$ inside the frame cage model.



(b) Diagram of the x-y-coordinates of test markers on the test pattern and recorded coordinates for each markers position localized by sensor 52.

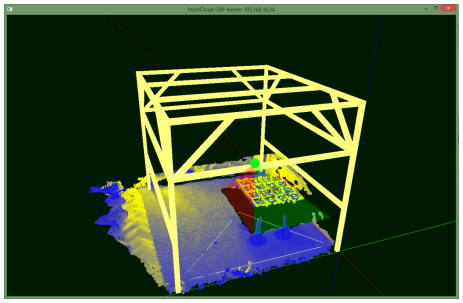


(c) Diagram of the y-z-coordinates recorded by sensor 52. The different heights of test recordings are illustrated by black lines through the z-axis.

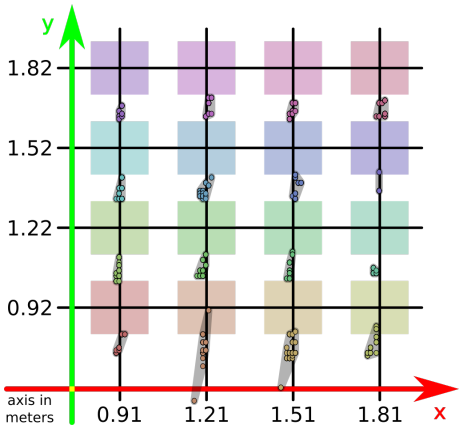


(d) Diagram of the x-z-coordinates recorded by sensor 52. The different heights of test recordings are illustrated by black lines through the z-axis.

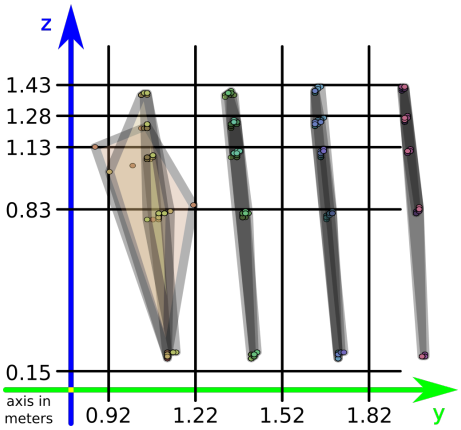
Figure 5.4.: Point cloud position localization test of sensor 52: 5.4a shows the software prototype UI during testing and the camera position. Over 1500 localized positions are plotted into the diagrams 5.4b, 5.4c and 5.4d (the less than 1 % failed localizations have been excluded). The axis scales (in meters) illustrate the ground truth coordinates of all markers. Different marker IDs of the utilized test pattern are symbolized by color.



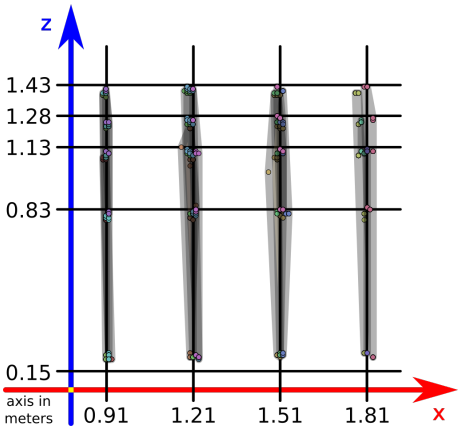
(a) The position of sensor 54 (symbolized by a green dot) and its view towards the test pattern in a screen shot of the localization software prototype UI. The visualisation shows the point cloud of sensor 54 for the test setting of height $0.83m$ inside the frame cage model.



(b) Diagram of the x-y-coordinates of test markers on the test pattern and recorded coordinates for each markers position localized by sensor 54.

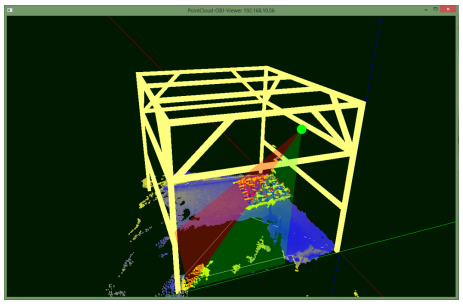


(c) Diagram of the y-z-coordinates recorded by sensor 54. The different heights of test recordings are illustrated by black lines through the z-axis.

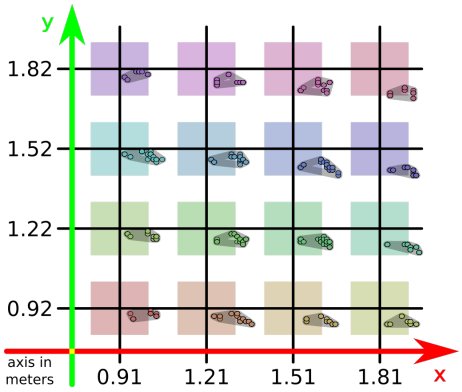


(d) Diagram of the x-z-coordinates recorded by sensor 54. The different heights of test recordings are illustrated by black lines through the z-axis.

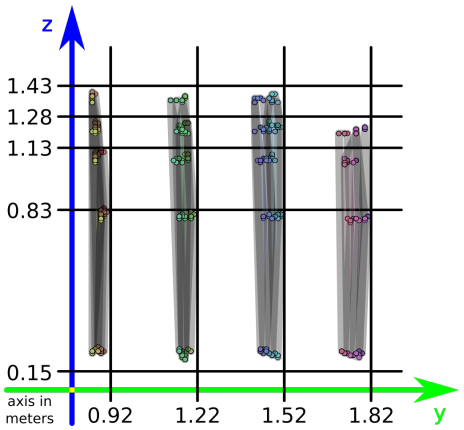
Figure 5.5.: Point cloud position localization test of sensor 54: 5.5a shows the software prototype UI during testing and the camera position. Over 1500 localized positions are plotted into the diagrams 5.5b, 5.5c and 5.5d (the less than 1 % failed localizations have been excluded). The axis scales (in meters) illustrate the ground truth coordinates of all markers. Different marker IDs of the utilized test pattern are symbolized by color.



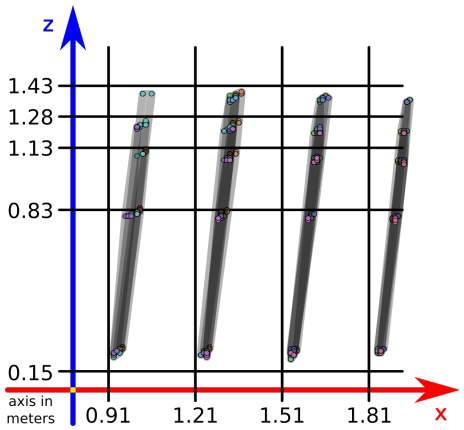
(a) The position of sensor 56 (symbolized by a green dot) and its view towards the test pattern in a screen shot of the localization software prototype UI. The visualisation shows the point cloud of sensor 56 for the test setting of height $0.83m$ inside the frame cage model.



(b) Diagram of the x-y-coordinates of test markers on the test pattern and recorded coordinates for each markers position localized by sensor 56.

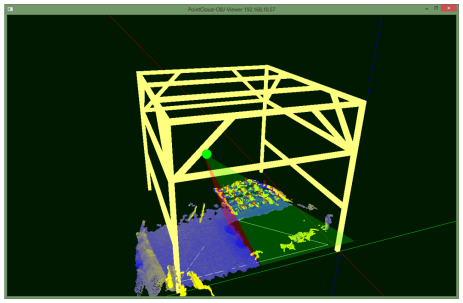


(c) Diagram of the y-z-coordinates recorded by sensor 56. The different heights of test recordings are illustrated by black lines through the z-axis.

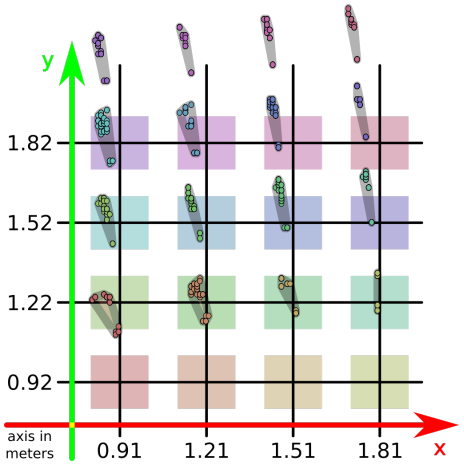


(d) Diagram of the x-z-coordinates recorded by sensor 56. The different heights of test recordings are illustrated by black lines through the z-axis.

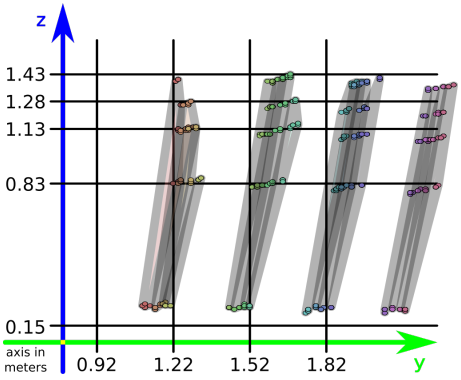
Figure 5.6.: Point cloud position localization test of sensor 56: 5.6a shows the software prototype UI during testing and the camera position. Over 1500 localized positions are plotted into the diagrams 5.6b, 5.6c and 5.6d (the less than 1 % failed localizations have been excluded). The axis scales (in meters) illustrate the ground truth coordinates of all markers. Different marker IDs of the utilized test pattern are symbolized by color.



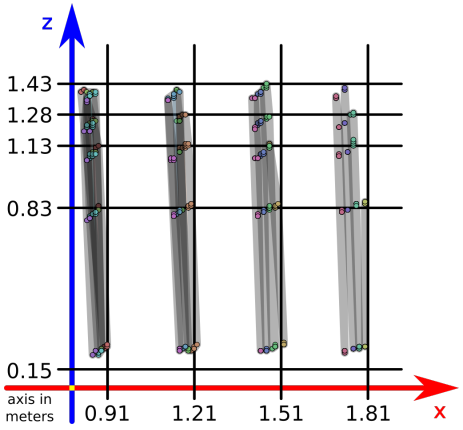
(a) The position of sensor 57 (symbolized by a green dot) and its view towards the test pattern in a screen shot of the localization software prototype UI. The visualisation shows the point cloud of sensor 57 for the test setting of height $0.83m$ inside the frame cage model.



(b) Diagram of the x-y-coordinates of test markers on the test pattern and recorded coordinates for each markers position localized by sensor 57.

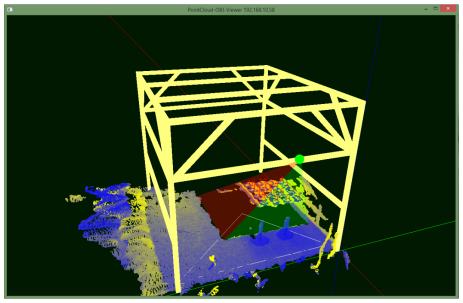


(c) Diagram of the y-z-coordinates recorded by sensor 57. The different heights of test recordings are illustrated by black lines through the z-axis.

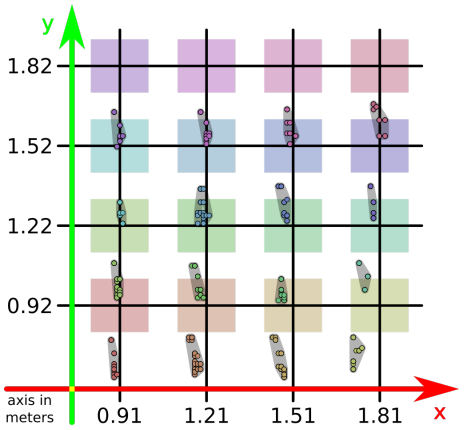


(d) Diagram of the x-z-coordinates recorded by sensor 57. The different heights of test recordings are illustrated by black lines through the z-axis.

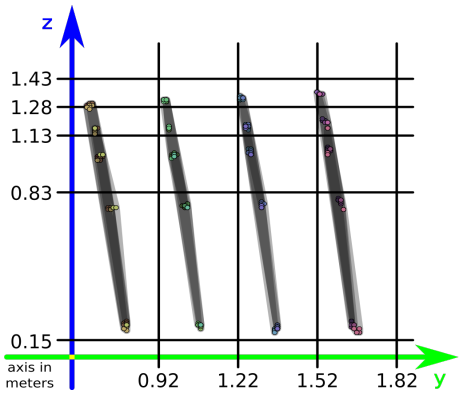
Figure 5.7.: Point cloud position localization test of sensor 57: 5.7a shows the software prototype UI during testing and the camera position. Over 1500 localized positions are plotted into the diagrams 5.7b, 5.7c and 5.7d (the less than 1 % failed localizations have been excluded). The axis scales (in meters) illustrate the ground truth coordinates of all markers. Different marker IDs of the utilized test pattern are symbolized by color.



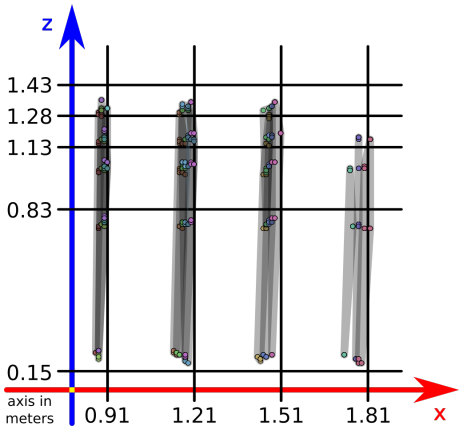
(a) The position of sensor 58 (symbolized by a green dot) and its view towards the test pattern in a screen shot of the localization software prototype UI. The visualisation shows the point cloud of sensor 58 for the test setting of height $0.83m$ inside the frame cage model.



(b) Diagram of the x-y-coordinates of test markers on the test pattern and recorded coordinates for each markers position localized by sensor 58.



(c) Diagram of the y-z-coordinates recorded by sensor 58. The different heights of test recordings are illustrated by black lines through the z-axis.



(d) Diagram of the x-z-coordinates recorded by sensor 58. The different heights of test recordings are illustrated by black lines through the z-axis.

Figure 5.8.: Point cloud position localization test of sensor 58: 5.8a shows the software prototype UI during testing and the camera position. Over 1500 localized positions are plotted into the diagrams 5.8b, 5.8c and 5.8d (the less than 1 % failed localizations have been excluded). The axis scales (in meters) illustrate the ground truth coordinates of all markers. Different marker IDs of the utilized test pattern are symbolized by color.

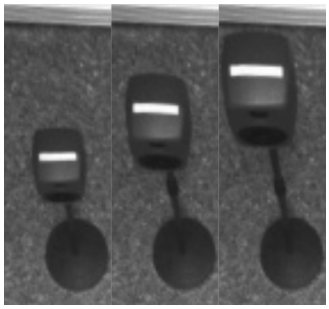
5.4. Loudspeaker Localization from Point Clouds

Based on the previous tests the location of loudspeaker is evaluated. By utilizing a point cloud mask & filter, as described in 4.6. The position of subareas, which remain in the visualized point cloud, is exported as object coordinates if done so. The size and position of subareas is tested by recording loudspeakers in $2cm$-wise steps through a minimum distance of half a meter. Fig 5.9 presents the localization test, performed with the smallest and biggest laboratory loudspeaker. Find loudspeaker specifications in tab. 2.1. Both tests are performed with sensor 52 and the default filter value of 170. Similar to 5.3, the test results are visualized with the RAWGraphs visualization tool[MEC+17] and further modified.

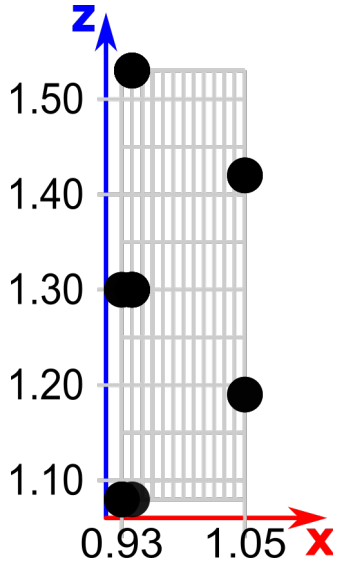
The Genelec 8010 AP is utilized to test the resolution with which changes in z-direction are recorded when the mask & filter method is utilized. The test is illustrated in fig. 5.9a. The test results in fig. 5.9b and 5.9c show characteristic coordinate jumps in not only the tested z-direction, but also the other two coordinates. In the test new subareas in z-direction are entered every 11 to $12cm$. Jumps of the same size occur in x-direction, jumping back and forth between two coordinate values surrounding the actual x-coordinate of the loudspeaker (see 5.9b). With similar jumps in y-direction the effect is even more drastic, as the back and forth alterations visualized in 5.9c have a distance of ca. $16cm$. Due to its small size the 8010 AP is detected as in one subarea position at a time during test recordings. Only at the bordering areas of two subareas (about $1,5cm$ wide) two subarea coordinates represented the single loudspeaker in recorded values.

The TANNOY Reveal 502 is utilized to test the resolution with which changes in y-direction are recorded when the mask & filter method is utilized. The test is illustrated in fig. 5.9d. The test results in fig. 5.9e and 5.9f, as in the previous test, show characteristic coordinate jumps in not only the tested y-direction, but also jumps in the other two coordinates. In the test new subareas in y-direction are entered every 15 to $16cm$. In x-direction jumps of only $9cm$ back and forth between three coordinate values surrounding the actual x-coordinate of the loudspeaker (see 5.9e) occur. This seems to be due to the large size of the loudspeaker, $23.8cm$ length in x-direction. Alterations of $11cm$ on the z-coordinate also occur (visualized in 5.9c). Due to its large size TANNOY Reveal 502 is detected as in two to three subarea position at a time during test recordings.

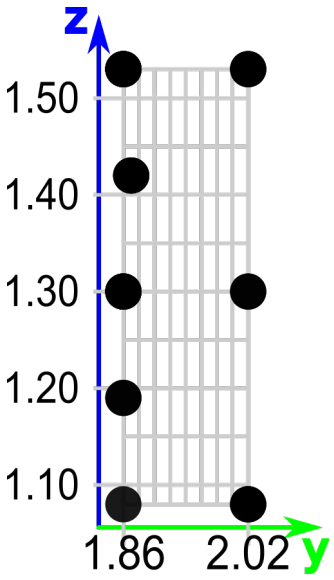
Over all up to $\pm 7.5cm$ error, due to the subarea coordinate rounding, are to be expected additionally to the errors occurring in 5.3 for each sensor. This is true, if the subarea coordinates properly center around the object they represent. A final test, comparing the subarea and marker based localizations of Genelec 8030 BPM loudspeakers is conducted with multiple sensors. To place the previously used $20 \times 20cm$ ArUco codes on the loudspeakers, double sided tape is sufficient. The



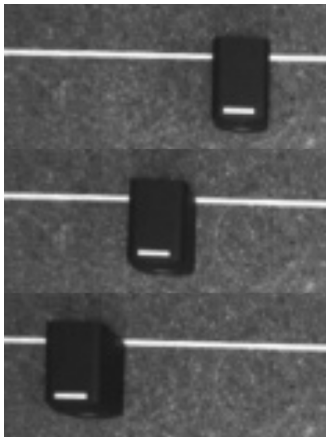
(a) The Genelec 8010 AP in three representative positions during the test. Only the height (z) is changed.



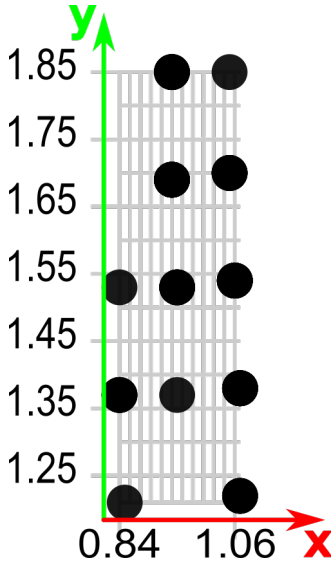
(b) 8010 AP coordinates derived from subarea in meters. The x-coordinate is falsely changing.



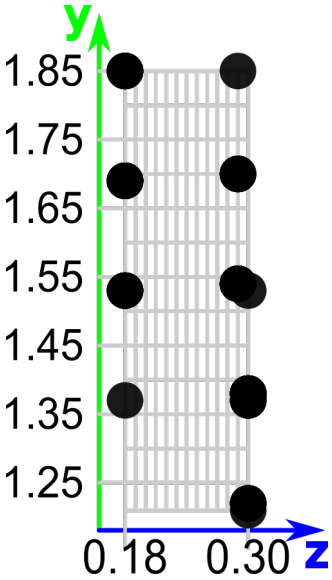
(c) 8010 AP coordinates derived from subarea in meters. The y-coordinate is falsely changing.



(d) The TANNOY Reveal 502 in three representative positions during the test. Only the y-coordinate is changed.



(e) Reveal 502 coordinates derived from subarea in meters. The x-coordinate is falsely changing.



(f) Reveal 502 coordinates derived from subarea in meters. The z-coordinate is falsely changing.

Figure 5.9.: The mask & filter subarea size tests: The small Genelec 8010 AP in fig. 5.9a is moved up in z-direction in 2cm steps resulting in the subarea coordinates visualized in fig. 5.9b and 5.9c. Similarly the bigger TANNOY Reveal 502 in 5.9d is moved sideways resulting in the subarea coordinates presented in fig. 5.9e and 5.9f.

markers are attached to the speakers, such that their center (localized by the software prototype) is equivalent to the center of the top most point of the loudspeakers. The markers must be visible to the sensor used for recording. Fig. 5.10 presents the view of sensor 56 towards the perpetrated 8030 BPM loudspeakers. The representative results achieved with sensor 56 are presented in fig. 5.11. The ground truth coordinates of the loudspeakers during this test are $1.80m$ and $1m$ distance to the x- and y-axis with one 8030 BPM closer to the x-axis (red in fig. 5.11) and the other closer to the y-axis (green in fig. 5.11). During the test the loudspeakers are moved in z-direction such that multiple neighbouring subareas are passed. A displacement of the subarea coordinates from the marker based coordinates in up to $30cm$ towards the x-direction is visible in fig. 5.11. As the marker based loudspeaker localizations are not in-between the subarea localizations (concerning the x-axis), the subarea coordinates are not properly centered around the object they represent. This is likely due to the method of rounding coordinate values in order to summarize vertex coordinates into a subarea (see 4.6). Tests with other sensors conform a displacement of up to $+30cm$ when subarea based localizations are utilized. These displacements must always be taken into account additionally to the point cloud distortion investigated in 5.3. Thus, the extra effort of attaching markers to the localized loudspeakers greatly improves the accuracy and precision of the localizations performed with the software prototype for the localization of loudspeakers from point clouds.

Figure 5.10.: The view of sensor 56 towards the test setting for the comparison between localizations conducted with ArUco markers and mask & filter subareas. two Genelec 8030 BPM loudspeakers with markers, attached to their top most point, are visible.

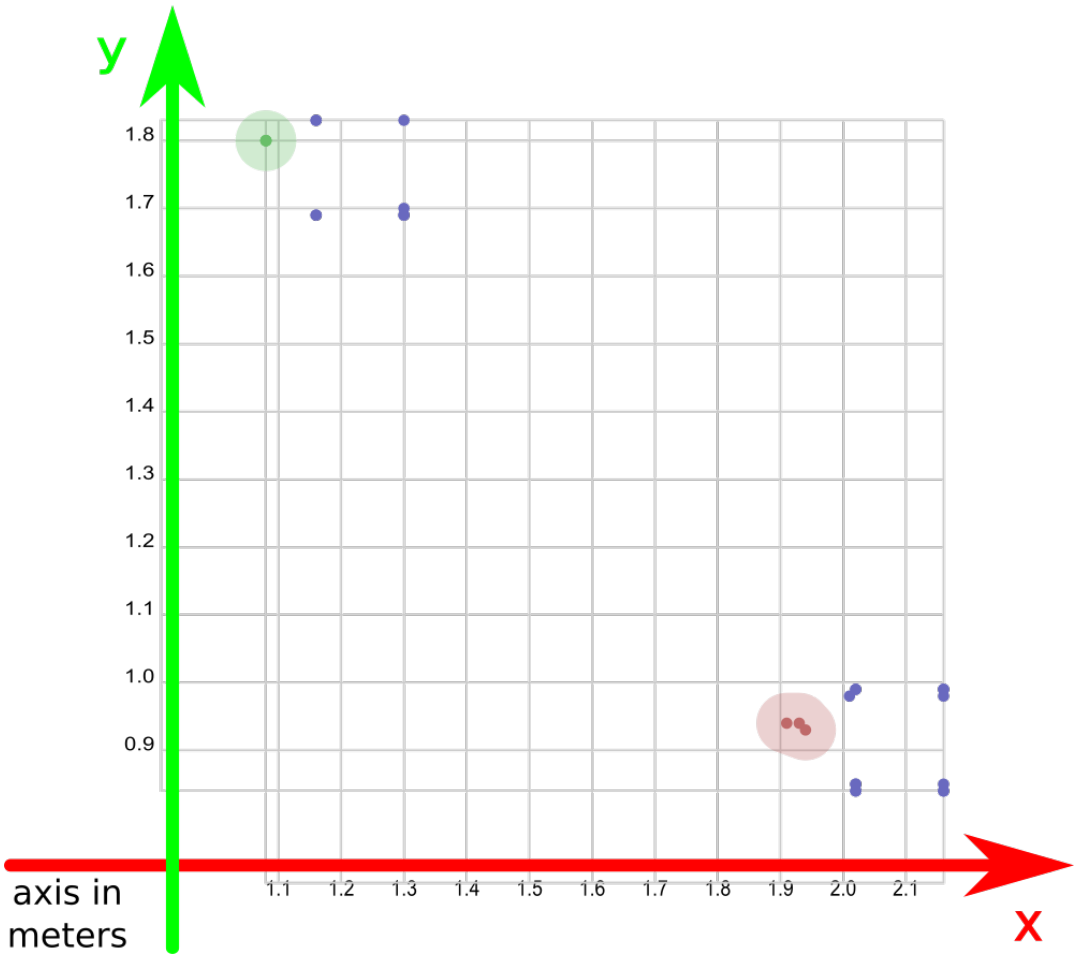


Figure 5.11.: Comparison between localizations conducted with ArUco markers (in green and red) and mask & filter subareas (in blue) are visualized in this diagram. two Genelec 8030 BPM loudspeakers with markers attached are used in the illustrated test. Several heights, combined in the plot, are recorded with both localization techniques, from sensor 56.

6. Summary and Conclusion

This thesis has introduced a software prototype for the localization of loudspeakers and other objects. It is based on optical sensor data, within the audiovisual laboratory of the junior professorship Media Computing at Chemnitz University of Technology. The requirements for the software prototype included utilizing the already installed S2000 smart sensors, produced by the Intenta GmbH. Two methods of localizing loudspeakers and other objects have been implemented and evaluated.

The mask & filter method has a low precision of $16cm$ distance between measurements of the same coordinates and lower accuracy of ca. $40cm$ to $80cm$ distance towards the actual object position in the worse case, depending on the utilized sensor. The benefit of the mask & filter method is, that it does not require additional preparations of the localized objects.

The optical marker method requires the preparation of objects to be localized with markers. However, its precision is as good as the precision of the point cloud itself, $0cm$ to $1cm$ distance between measurements of the same coordinate and up to $6cm$ in extreme, rare, cases. Accuracy of the optical marker method varies from less than $20cm$ distance, towards the actual object position on the calibrated ground, to more than $50cm$ in heights of more than $1.30m$. Test results depend on the utilized sensors. Another benefit of the optical marker method is that located objects are distinguishable by marker ID.

The main enhancement for the audiovisual laboratory, provided by the software prototype created in this thesis, is the file export functionality. As described in 4.3, 3D-models of test settings, inside the laboratory's frame cage, and CSV-files, containing 3D-coordinates recorded from the S2000 sensor point cloud, are exportable. The development and evaluation of acoustic localization methods in the audiovisual laboratory is simplified, because time consuming manual measurements must only be conducted for acoustic localizations of higher accuracy than provided by the software prototype for the localization from 3D point clouds.

7. Outlook

This thesis presented two methods for object localization based on the S2000 sensor data. As shown in 5.3, distortions of the point cloud lead to bad accuracy when only three points are used for calibration as implemented in 4.5. Such distortions influence both localization methods. The calibration method could be further improved by utilizing additional reference points. The mask & filter method is depended on the *maskDev* value, which regulates the size of subareas for masking, filtering and localization as described in 4.6. Systematic research could be conducted to find ideal *maskDev* values and thus, improve the mask & filter localization method. Furthermore dynamic localizations, of objects in motion, could be investigated.

Bibliography

[BK08] Gary Bradski & Adrian Kaehler: *Learning OpenCV: Computer Vision with the OpenCV Library*, "O'Reilly Media, Inc.", 2008, ISBN 978-0-596-55404-0, google-Books-ID: seAgiOfu2EIC.

[BPS17] Mohamad Bdiwi, Marko Pfeifer & Andreas Sterzing: *A new strategy for ensuring human safety during various levels of interaction with industrial robots*, *CIRP Annals*, volume 66(1):pages 453–456, 2017, ISSN 0007-8506.
URL `http://www.sciencedirect.com/science/article/pii/S0007850617300094`

[Bus03] Samuel R. Buss: *3D computer graphics: a mathematical introduction with OpenGL*, Cambridge University Press, 2003.

[Can83] John Francis Canny: *Finding edges and lines in images*, 1983.

[Che08] Jim X. Chen: *Guide to Graphics Software Tools*, Springer Science & Business Media, 2008, ISBN 978-1-84800-901-1, google-Books-ID: Rv2gncprg3gC.

[GDP10] David Gossow, Peter Decker & Dietrich Paulus: *An Evaluation of Open Source SURF Implementations*, in *RoboCup 2010: Robot Soccer World Cup XIV*, Lecture Notes in Computer Science, pages 169–179, Springer, Berlin, Heidelberg, 2010, ISBN 978-3-642-20216-2 978-3-642-20217-9.
URL `https://link.springer.com/chapter/10.1007/978-3-642-20217-9_15`

[GJMSMCMJ14] S. Garrido-Jurado, R. Muñoz-Salinas, F. J. Madrid-Cuevas & M. J. Marín-Jiménez: *Automatic generation and detection of highly reliable fiducial markers under occlusion*, *Pattern Recognition*, volume 47(6):pages 2280 – 2292, 2014, ISSN 0031-3203.
URL `http://www.sciencedirect.com/science/article/pii/S0031320314000235`

[Gra99] Rudolf F. Graf: *Modern Dictionary of Electronics*, Elsevier, 7th edition edition, 1999, ISBN 978-0-08-051198-6, google-Books-ID: AYEKAQAAQBAJ.

[HL09] Jonathan D. Hiller & Hod Lipson: *STL 2.0: a proposal for a universal multi-material Additive Manufacturing File format*, in *Proceedings of the Solid Freeform Fabrication Symposium*, pages 266–278, Citeseer, 2009.

[HMH+] Hussein Hussein, Robert Manthey, Abul Hasan, Manuel Heinzig, Marc Ritter, Danny Kowerko & Maximilian Eibl: *Design of a Laboratory for Audio and Video Based Object Localization and Tracking*, in *Internationale Summer School für Computer Sciences, Computer Engineering und Bildungstechnologien (ISCSET)*.
URL `http://www.ibs-laubusch.de/de/2016/12/29/iscset-international-summer-school/`

[Jä13] Bernd Jähne: *Digital Image Processing*, Springer Science & Business Media, 5th revised and extended edition edition, 2013, ISBN 978-3-662-04781-1, google-Books-ID: XczuCAAAQBAJ.

[Kho11] Kourosh Khoshelham: *Accuracy analysis of kinect depth data*, in *ISPRS workshop laser scanning*, volume 38, page W12, 2011.

[KKK17] Peter Kuna, Tomá Kozík, Silvia Kunová & Miroslav ebo: *Software Tools for Creating and Presenting Virtual 3D Models*, in *International Conference on Interactive Collaborative Learning*, pages 17–26, Springer, 2017.

[Lag14] Robert Laganière: *OpenCV Computer Vision Application Programming Cookbook Second Edition*, Packt Publishing Ltd, 2014, ISBN 978-1-78216-149-3, google-Books-ID: pv9bBAAAQBAJ.

[MEC+17] Michele Mauri, Tommaso Elli, Giorgio Caviglia, Giorgio Uboldi & Matteo Azzi: *RAWGraphs: A Visualisation Platform to Create Open Outputs*, in *Proceedings of the 12th Biannual Conference on Italian SIGCHI Chapter*, CHItaly '17, pages 28:1–28:5, ACM, New York, NY, USA, 2017, ISBN 978-1-4503-5237-6.
URL `http://doi.acm.org/10.1145/3125571.3125585`

[MV08] Jernej Mrovlje & Damir Vrancic: *Distance measuring based on stereoscopic pictures*, in *9th international PhD workshop on systems and control: young generation viewpoint*, volume 6, 2008.

[PBKE12] Kari Pulli, Anatoly Baksheev, Kirill Kornyakov & Victor Eruhimov: *Realtime Computer Vision with OpenCV*, *Queue*, volume 10(4):pages 40:40–40:56, 2012, ISSN 1542-7730.
URL `http://doi.acm.org/10.1145/2181796.2206309`

[Pip03] Evan Pipho: *Focus on 3D Models*, Cengage Learning, 2003, ISBN 978-1-59200-033-3, google-Books-ID: ztnZcbo41oAC.

[RC11] R. B. Rusu & S. Cousins: *3D is here: Point Cloud Library (PCL)*, in *2011 IEEE International Conference on Robotics and Automation*, pages 1–4, 2011.

[RDGF16] Joseph Redmon, Santosh Divvala, Ross Girshick & Ali Farhadi: *You only look once: Unified, real-time object detection*, in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[REK18] H. Hussein R. Siegel R. Erler, R. Manthey & D. Kowerko: *Realisation of an Audio Video Laboratory for Precise Object Localisation and Tracking*, in *Elektronische Sprachsignalverarbeitung 2018 (ESSV 2018)*, 2018.

[SG09] Dave Shreiner & Bill The Khronos OpenGL ARB Working Group: *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*, Pearson Education, 2009, ISBN 978-0-321-66927-8, google-Books-ID: xPu3mN2FPl4C.

[SP00] Jaakko Sauvola & Matti Pietikäinen: *Adaptive document image binarization*, *Pattern recognition*, volume 33(2):pages 225–236, 2000.

[ZHK17] Timon Zietlow, Hussein Hussein & Danny Kowerko: *Acoustic Source Localization in Home Environments - The Effect of Microphone Array Geometry*, in *Elektronische Sprachsignalverarbeitung 2017 (ESSV 2017)*, number 86 in Studientexte zur Sprachkommunikation, pages 219–226, TUDpress, Saarbrücken, 2017, ISBN 978-3-95908-094-1.
URL `http://essv2017.coli.uni-saarland.de/index.html`

Declaration of Authorship

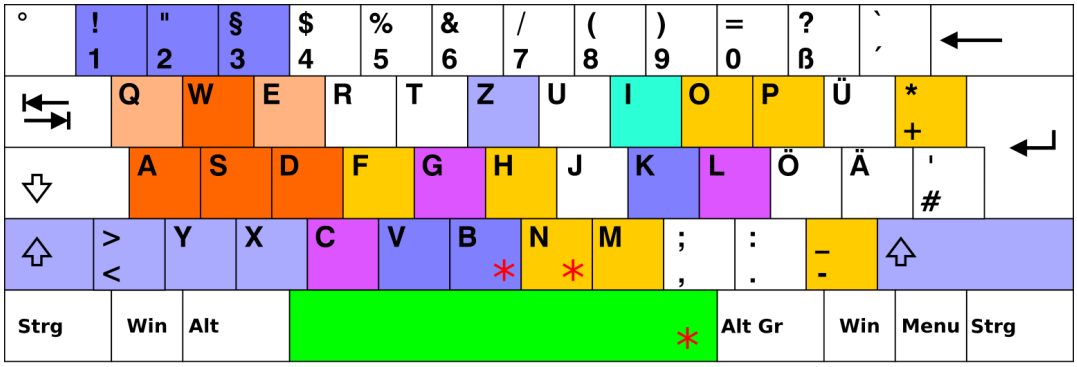
I hereby certify that this thesis has been composed by me and is based on my own work, unless stated otherwise. No other persons work has been used without due acknowledgement in this thesis. All references and verbatim extracts have been quoted, and all sources of information, including graphs and data sets, have been specifically acknowledged.

Chemnitz, March 19, 2018

Richard Siegel

Attachment

A. The Keymap for the Localization Software



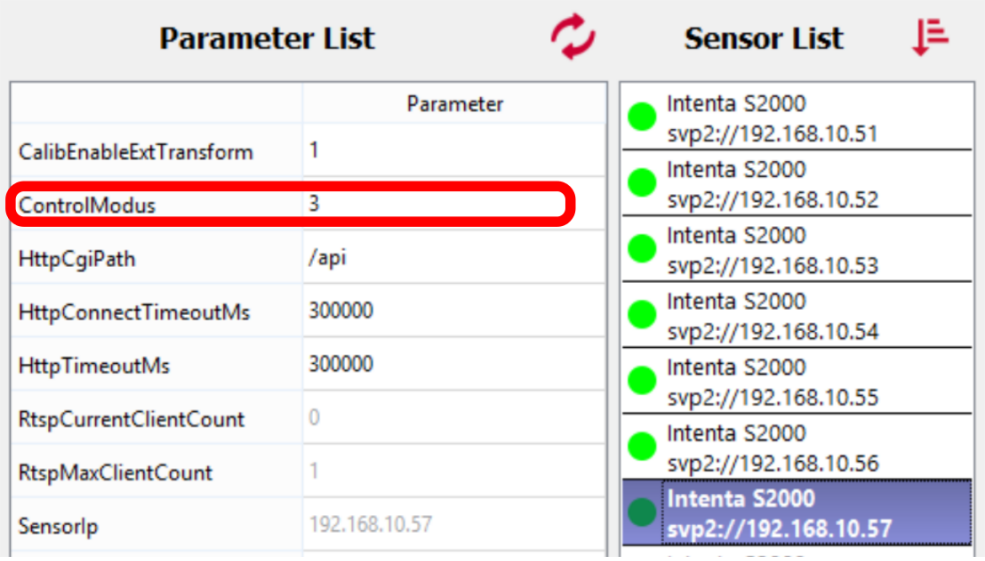
*This key writes or overwrites files!

[Esc] Exit
[Space] Save all files documenting the current state
[i] Next sensor
Move and Zoom Camera
Mask and Filter Options
Rendering Options
Moving PointCloud

```
[Esc] Exit
[Space] Save all files documenting the current state
[i]  Next sensor

Move and Zoom Camera
[a]  Rotate cammera around model
[d]  Counter rotate cammera around model
[w]  Move camera up
[s]  Move camera down
[q]  Zoom in
[e]  Zoom out

Mask and Filter Options
[p]  Toggle (On/Off) position by object markers
[m]  Toggle (On/Off) point cloud mask
[n]  Save new mask (and override mask backup with the old one)
[f]  Toggle (On/Off) point cloud filter
[+]  Increese filter value
[-]  Decreese filter value
[h]  Toggle (On/Off) hide unmarked point cloud areas
[o]  Toggle (On/Off) object boxes (for export of positions)

Rendering Options
[c]  Toggle (On/Off) point cloud color texture
[l]  Toggle (On/Off) GL-lighting
[g]  Toggle (On/Off) graphic support lines
     (Coord. sys. and referance point triangle)

Moving PointCloud
[k]  Auto calibrate with markers (press several times)
[1]  Rotate 90 degrees around x-axis
[2]  Rotate 90 degrees around y-axis
[3]  Rotate 90 degrees around z-axis
[<]  Scale down
[>]  Scale up
[x]  Move in x-direction (red)
[y]  Move in y-direction (green)
[z]  Move in z-direction (blue)
[X]  Move agains x-direction
[Y]  Move agains y-direction
[Z]  Move agains z-direction
[b]  backup sensor settings
[v]  Load backuped sensor settings
```

B. Guide to Using the Localization Software

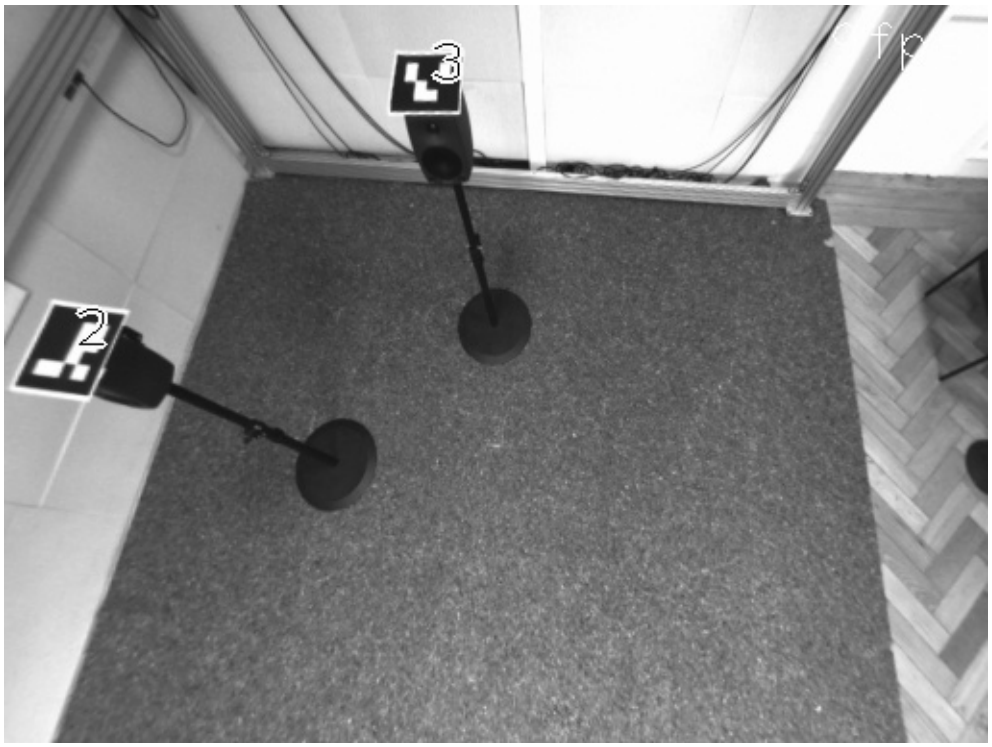
1. Make sure at least one S2000 sensor is properly connected via Ethernet.
2. Start the Intenta software *SVPManager* and set *ControlModus* to 3 for all sensors you want to use.



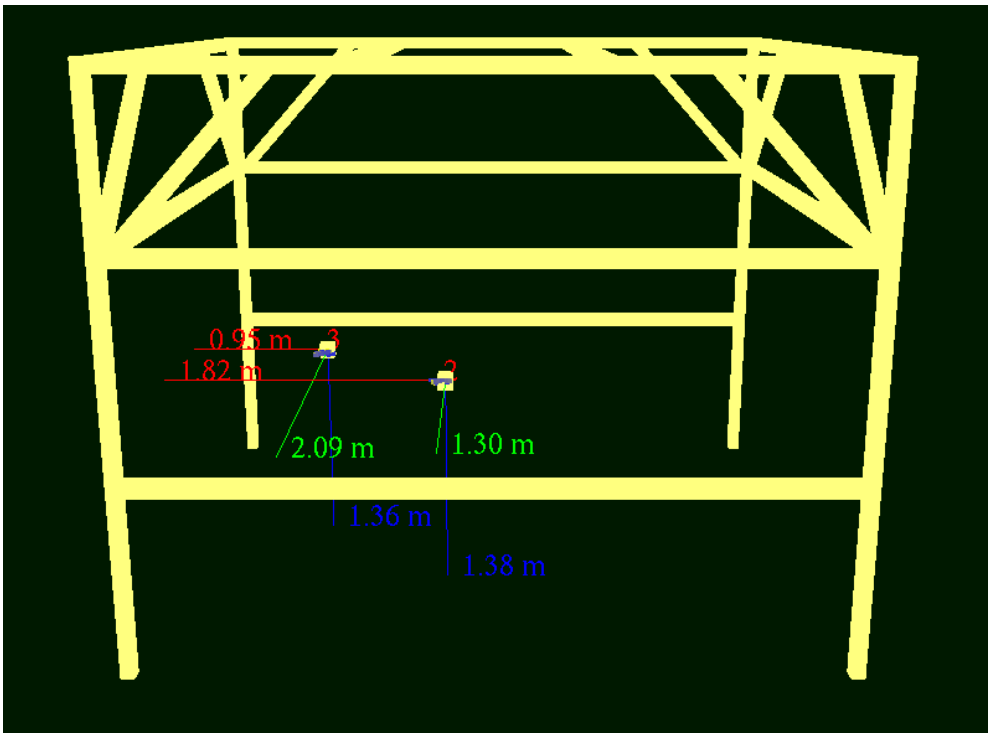
3. Make sure the files **lab.obj** and **object.obj** are in the same directory as the **prototype.exe** executable. Configuration or mask files should also be present in the same directory if you want to use them.
4. Start the localization prototype software. A double click on the **prototype.exe** executable will start the localization prototype software, connected to all available sensors. To select a sensor run **prototype.exe svp2://192.168.10.57** (with the IP of the sensor you want to connect to) in the command line.
5. Calibrate the the sensor as described in 4.5 if not already done so. Once the ArUco markers are placed, use the key options for calibration as described in attachment A (see *Moving PointCloud*). For manual calibration press the O-key and the H-key to see the positions of the markers without distraction by the point cloud.

6. Record a mask of the empty room by pressing the N-key. Use it by pressing the M-key. Notice that the keys F, H and M regulate the visibility of the point cloud. Make sure to combine only the options you want.

7. With a calibrated sensor you can now start to measure the locations you are interested in. Place the objects you want to record in the sensor view and make sure the markers on the objects are detected by the sensor. An image, similar to the example shown here, will be visible to you in one of the windows that started with the program.



8. Go to the main window of the software prototype and look at the detections the program is making. Use the *Mask and Filter Options* (see attachment A) to find the setting that suites you best. The P-key is very important as it switches between placing objects by mask & filter subareas or by ArUco markers. ArUco markers are the default option as used for the example screenshot.



9. To export the an image, CSV-file and OBJ 3D-model of the scene, press the space-key. The exported files can be used in other programs such as Blender (imports the OBJ-Model) and spreadsheet software (usually imports CSV files). See in this example screenshot how files may be viewed once they have been exported from the localization prototype software.