BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF MECHANICAL ENGINEERING
FAKULTA STROJNÍHO INŽENÝRSTVÍ

INSTITUTE OF AUTOMOTIVE ENGINEERING
ÚSTAV AUTOMOBILNÍHO A DOPRAVNÍHO INŽENÝRSTVÍ

OBSTACLE DETECTION USING AN IMAGE-BASED 3D SCANNER

DETEKCE PŘEKÁŽEK ZA POUŽITÍ KAMEROVÉHO 3D SKENERU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE

Andrei Rybin

SUPERVISOR
VEDOUCÍ PRÁCE

Ing. Jan Hrbáček

BRNO 2018

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF MECHANICAL ENGINEERING

Bachelor's Thesis Assignment

Institut:	Institute of Automotive Engineering
Student:	**Andrei Rybin**
Degree programm:	Engineering
Branch:	Machine and Equipment Construction
Supervisor:	**Ing. Jan Hrbáček**
Academic year:	2017/18

As provided for by the Act No. 111/98 Coll. on higher education institutions and the BUT Study and Examination Regulations, the director of the Institute hereby assigns the following topic of Bachelor's Thesis:

Obstacle detection using an image-based 3D scanner

Brief description:

Design and implementation of a simple obstacle detector for a mobile robot, based on depth image from a camera 3D scanner (Microsoft Kinect, Asus Xtion).
Basic programming skills in one of the languages supported by driver libraries (oficial SDK, libfreenect, OpenNI) are prerequisite.

Bachelor's Thesis goals:

Obstacle detection system design
- suitable sensor selection with respect to driver libraries,
- functionality design, image processing methods selection.

System implementation.

Recommended bibliography:

GIROD, Bernd, GREINER, Günther a Heinrich NIEMANN, ed. Principles of 3D Image Analysis and Synthesis. Springer US, 2002. ISBN 978-1-4757-3186-6.

Students are required to submit the thesis within the deadlines stated in the schedule of the academic year 2017/18.

In Brno, 30. 10. 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA STROJNÍHO INŽENÝRSTVÍ
DĚKANÁT
11

L. S.

[illegible]

prof. Ing. Václav Píštěk, DrSc.
Director of the Institute

[illegible]

doc. Ing. Jaroslav Katolický, Ph.D.
FME dean

ABSTRAKT

Tato bakalářská práce se zabývá implementaci softwaru pro systém detekci překážek pomoci kamerového 3D skeneru Kinect V2. V rešeršní části jsou detailně popsány existující druhy kamerových 3D skenerů, jsou analyzovány existujicí řešení, a algoritmy v oblasti detekci překážek a je uvedena potřebná teorie. Praktická část se skládá ze tři části: popis použité robotické platformy, rozbor implementaci softwaru pro detekci překážek, a diskuzní část s analýzou výsledků experimentů. Na závěr vyvínutý systém je zhodnocen, jsou navrženy možnosti jeho aplikace a dalšího vývoje.

ABSTRACT

This bachelor thesis is focused on the implementation of software for obstacle detection system based on image-based 3D scanner Kinect V2. The research part deals with the thorough description of existing types of image-based 3D scanners, the analysis of related solutions and algorithms, and the necessary theory is provided here. The practical part consists of three sections: the description of the chosen robotic platform, the examination of a software implementation, and the discussion with the analysis of conducted experiments' results. To conclude the developed system is evaluated, possibilities of application and further development are proposed.

KLÍČOVÁ SLOVA

Kinect V2, ToF kamera, kamerový 3D skener, hloubkový obraz, mračno bodů, segmentace roviny, C++, detekce překážek, počítačové vidění, ROS, PCL, libfreenect

KEYWORDS

Kinect V2, ToF camera, image-based 3D scanner, depth image, point cloud, plane segmentation, C++, obstacle detection, computer vision, ROS, PCL, libfreenect

BIBLIOGRAPHIC CITACION

RYBIN, A. *Detekce překážek za použití kamerového 3D skeneru*. Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2018. 86 s. Vedoucí bakalářské práce Ing. Jan Hrbáček.

ACKNOWLEDGEMENTS

I am deliriously thankful to my supervisor Ing. Jan Hrbáček and to Ing. Jan Meindl for their detailed guidance and assistance in the conduction of experiments, as well as for helpful discussions. I also owe my thanks to Bender Robotics s.r.o for providing required equipment, as well as a workplace. I am grateful to my family and girlfriend for substantial support and encouragement during my education.

AFFIRMATION

I declare that this bachelor thesis is based on my own work, led by my Bachelor's thesis supervisor Ing. Jan Hrbáček, and all utilized sources are properly listed in the bibliography. I proclaim that all presented information is accurate.

In Brno 22.05.2018 ………………………………………………

Andrei Rybin

CONTENTS

1 INTRODUCTION

Robotics has become an integral part of present technology. According to [1], there is a growing tendency in global sales of industrial robots and the rate of the growth is increasing year by year. This indicates an importance of robotics development and the research of that technical field.

Robot perception is one of the most important areas, the robotics research is focused on. This field includes navigation, mapping, obstacle detection and recognition. The former and latter are considered fundamental in order to perform more basic operations: obstacle avoidance and navigation. Originally, expensive 3D lasers were abundantly used for navigation, mapping, and obstacle detection tasks, and not all of the research facilities were able to afford such equipment [2]. Everything changed with the arrival of image-based 3D scanners: structured-light and time-of-flight (ToF) cameras. Since 2010 when Microsoft released sensor Kinect, 3D perception equipment has become much cheaper, and hence available for the most of researchers throughout the world [3].

The goal of this bachelor thesis is to propose and implement an obstacle detection system using an image-based 3D scanner. Microsoft Kinect V2 has been used as a capturing device, due to its relative cheapness and sufficient precision. Point cloud was chosen as a representation of Kinect’s depth map. The implemented algorithm uses a point cloud topic, which flows from the **kinect2_bridge** node available in **IAI Kinect2** package [4]. The actual detection and recognition algorithm is based on Euclidean Cluster Extraction with the usage of plane segmentation and normal estimation.

The research part deals with the description of related obstacle detection algorithms and solutions, briefly introduces ROS environment and key aspects of computer vision in robotics. Following the research part, the practical part introduces and thoroughly describes the proposed solution, which is followed by experiments evaluation. To sum up, the developed system has been analyzed in terms of application, efficiency and further possible modifications.

2 RELATED WORKS

There are plenty of existing solutions for obstacle detection and recognition problems. In fact, it would require a whole book to thoroughly describe each of them. Thus, only the most related ones were mentioned in this paper.

Since 2010 after the release of Microsoft Kinect, object recognition and detection methods with the usage of ToF cameras have been widely researched, however, there are still many possibilities to develop new methods and algorithms and to modify existing ones to meet new requirements. In 2010, Hinterstoisser S. et al. discovered a template based recognition approach called DOT (Dominant Orientation Templates), which, in essence, kept objects registered from different angles in a big RGB-D template database [5]. In a year later, PCL (Point Cloud Library) – a fully templated, C++ library for point cloud processing – was released [6]. This release boosted research conducted throughout the world and as a result, several other approaches to 3D obstacle detection were proposed.

In the same year when PCL was released, Rakprayoon P. et al. introduced a method to differ obstacles from manipulator in case they share the same workspace [7]. In 2012, Choi J. et al. proposed a color and depth fusion-based rear obstacle detection algorithm, which exceeded the limits of single view based algorithms [8]. In the same year, Bostanci E. et al. introduced a better approach for extracting planar features from the point cloud, which utilized the parallelism for better performance [9]. In the following year, Hulik R., Spanel M., Smrz P. and Materna Z. implemented robust plane detector based on 3D Hough transformation, which overcame RANSAC-based one implemented in PCL in the consistency of plane detection [10]. Also in the same year, Zhang Q. et al. proposed a real-time general object recognition for the indoor robot in complex scenes, which utilized Clustered Viewpoint Feature Histogram (CVFH) from PCL [11].

In 2014, Hongshan Y. et al. presented obstacle detection and classification method, which utilized ToF cameras for navigation in unstructured environments. Their method was based on the human knowledge of navigation and consisted of three steps: removing irrelevant objects in a scene, clustering of regions of interest, using multiple relevance vector machine (RVM) classifier to distinguish obstacles into four groups based on the terrain traversability and geometrical features of the obstacles [12].

Against the background of the recent success of deep and machine learning methods in the area of 2D image analysis, both concepts were subsequently introduced to obstacle categorization and recognition field. Reza F. proposed a plane-based object categorization with the usage of relational learning in 2014 [13], followed by the introduction of 3D convolutional neural networks (3D CNN) for object recognition by Maturana D. and Scherer S. in 2015 [14]. New deep learning-based approaches to object recognition in cluttered scenes [15] and to feature learning [16] surpassed most conventional ones. However, one must consider the fact that deep learning techniques require considerably larger datasets and significantly more powerful computer hardware along with highly sophisticated software, especially during the training phase. For example, the winners of Amazon Robotics Challenge 2017 (ARC) (previously Amazon Picking Challenge) designed relatively low-cost Cartesian manipulator[1] and its training phase was performed on an Intel Core i5-7600 along with four NVIDIA GTX1080Ti graphics cards [17][18]. Generally speaking, for cost-effective solutions conventional approach should still be the best fit, nevertheless if reaching the highest accuracy of recognition is the only goal, applying CNN will certainly be the best solution.

It is worth mentioning that on the Brno University of Technology several faculties are being involved in robotics research[2]. Obstacle detection and recognition itself has been paid a lot of attention in Bachelor and Master theses. In 2011, Dojava M. researched the application of RGB camera as a capturing device for assisting system of a car. The system was able to differ a roadway from obstacles [19]. In 2013, Najman J. developed an anti-collision system for experimental vehicle Car4 using Asus Xtion RGB-D sensor and Hokuyo laser scanner [20]. In the same year, Janás L. applied Microsoft Kinect with the usage of OpenCV library to detect people in a segmented image [21]. In 2015, Stříteský V. implemented a solution to obstacle detection problem based on waypoints changes [22].

[1] The estimated cost of assembled manipulator is $23,935
[2] Faculty of Mechanical Engineering, Faculty of Information Technology and Faculty of Electrical Engineering and Communication

3 COMPUTER VISION IN ROBOTICS

Perception is essential for an autonomous robot to know the structure of surrounding environment to move around without any collision and to manipulate with objects within it. That is why the development of robotics directly depends on the progress of computer vision.

This chapter deals with the description of key computer vision concepts utilized in robotics and gives a basic overview of 3D image-based scanners[3] to gain enough insight into techniques used in the implementation of the developed obstacle detection system. Whereas computer vision is a large field containing a lot of techniques and approaches to solve the mentioned problem, this chapter provides a brief overview of techniques, which are related to the implemented solution.

3.1 RGB-D sensors

3D perception is an acquisition of the distance from the capturing device to the scene, which is also known as depth or z-coordinate in RGB-D sensor's coordinate frame.

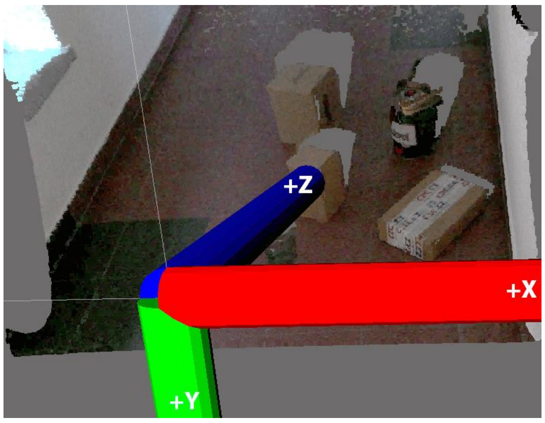


Figure. 1) Kinect v2 coordinate frame displayed in RVIZ

The RGB-D sensor is capable of mapping not only the color information but also the depth of a scene. The depth map is basically a matrix data structure, where each pixel contains the depth information. As a result, the output of RGB-D sensor consists of both color and depth map of the registered environment. The resulted depth and color map can be used to produce a point cloud – a special data structure, which represents the set of colored 3D points (RGB data and XYZ coordinates). RGB-D sensors can be distinguished by the method applied for obtaining the depth information into two types [23].

3.1.1 Structured-light cameras

That kind of sensor obtains a depth information by using laser source and an image sensor that both operate at IR (infrared) wavelengths. The laser beam coming from the IR laser source is projected in a pseudo-random dot pattern, which illuminates the entire scene. Subsequently, the

[3] Better-known as RGB-D sensors

IR image sensor attempts to seek the matching dot pattern in the projected pattern. After that, it is possible to triangulate the 3D position of each point in the projected pattern by knowing the position and orientation of the laser source relative to the IR image sensor [24].

A structured-light camera is categorized as an active type sensor, because the pattern is actively projected to the scene to be mapped [23].

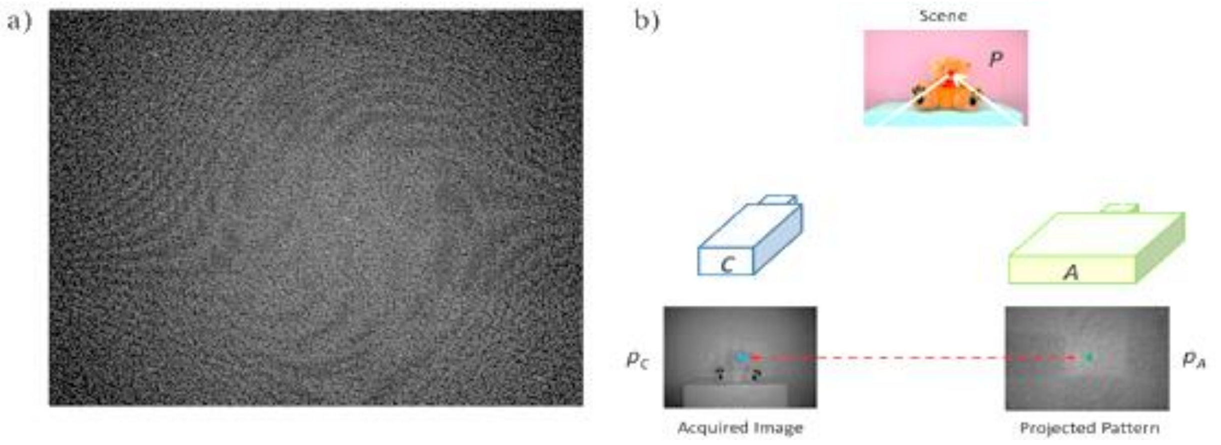


Figure. 2) A pseudo-random dot pattern projecting and processing [23]; a) The pattern registered by Kinect IR camera; b) The process of active triangulation: pixel p_A (green dot) is encoded in a pattern. The pattern is projected to the scene and registered by C (IR camera). The point in a scene associated with p_A is P and the conjugated point of p_A in an acquired image is p_C (blue dot). The correspondence estimation algorithm (red dashed arrow) calculates the conjugate points

Structured-light cameras are having a great success in today's mass market. They provide real-time 3D data streams. Their measurement errors are mostly produced due to local reflectivity and geometric configurations of a scene. The most widespread structured-light cameras are Intel RealSense F200, Intel RealSense R200, Asus Xtion Pro Live and Microsoft Kinect v1 [26].

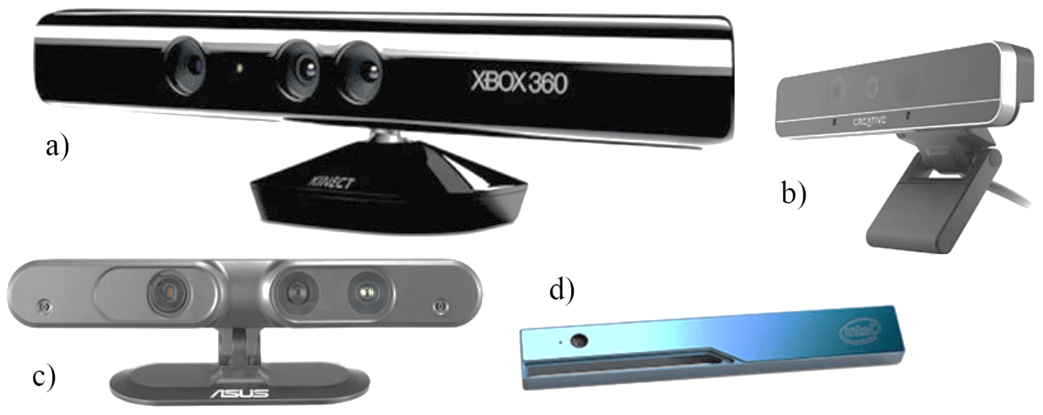


Figure. 3) The most widespread structured-light cameras [26]: a) Microsoft Kinect v1; b) Intel RealSense F200; c) Asus Xtion Pro Live; d) Intel RealSense R200

3.1.2 Time-of-Flight (ToF) cameras

The ToF technique is based on a principle of measuring the time that light beam emitted from the projector requires to travel to an object in the scene and back to the sensor. Most of ToF cameras utilize the Continuous Wave (CW) Intensity Modulation approach, which means that the scene is actively illuminated by near infrared (NIR) intensity-modulated periodic light. Firstly, IR wave is emitted to the target object. It is possible to calculate the distance to the object d by measuring the phase shift t_d between the emitted and reflected IR waves [25][27]:

$$d = \frac{c}{2f}\frac{t_d}{2\pi} \tag{1}$$

where c is the speed of light, f is the emitted signal frequency. The ratio $c/(2f)$ is the maximum distance that can be measured without ambiguity. The phase shift is estimated in each sensor pixel by mixing process.

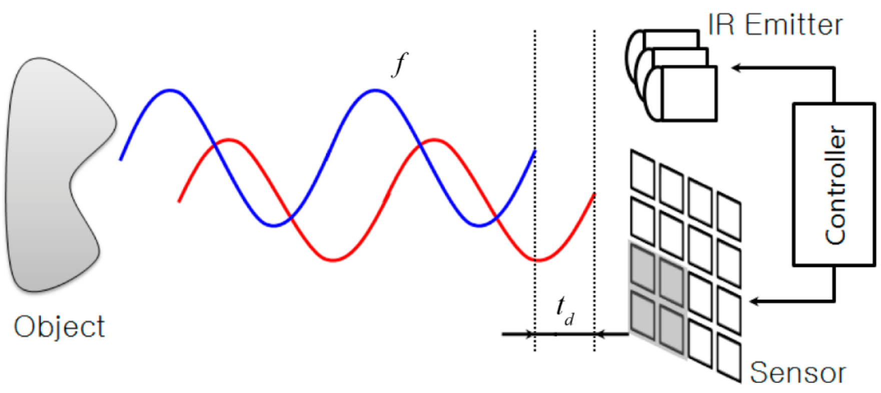


Figure. 4) The principle of operation of ToF camera [27]: The red curve represents the emitted NIR wave and the blue one represents the reflected NIR wave.

In ToF camera like Microsoft Kinect v2, there is an array of IR receivers that sample the reflected NIR light from a set of points in a scene. The IR emitter is constructed in a way that its virtual center is located in the center of the array of receivers and hence the sampling frequency of the receivers is consequently synchronized, which allows the number of points to be captured in a scene simultaneously. As a result, the sampled scene can be reconstructed in a point cloud [27].

Currently, there are several ToF cameras in different price ranges. One can find the budget ones: Microsoft Kinect v2, Creative Senz3D, SoftKinetic DS311 and DS325; as well as the high-cost industrial solution: SwissRanger SR4000[4] [28].

[4] The estimated costs are around $499, $199, $299, $249 and $4,295 respectively

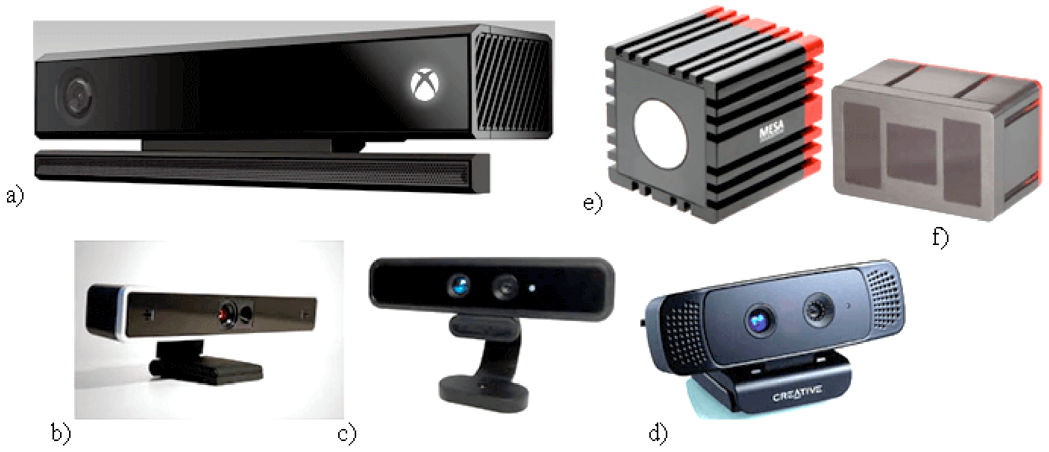


Figure. 5) ToF cameras [26][29]: a) Microsoft Kinect v2; b) SoftKinetic DS311; c) SoftKinetic DS325; d) Creative Senz3D; e) MESA Imaging SR4000; f) MESA Imaging SR4500

3.2 Point Cloud

In a nutshell, the point cloud is a set of points in XYZ coordinate system. Each point in a cloud represents a data structure with its own features: RGB values, hue value, normal values etc. The point cloud can be provided in the organized or unorganized form. It is worth mentioning that organizing the point cloud can be advantageous, due to the fact that organized one requires much less time to search through its points. The point cloud is one of the most applicable data structures to create a 3D representation of a scene.

Microsoft Kinect v2 provides depth, IR and RGB data streams. The resolution of depth image is 512 x 424 pixels, which comparing to the RGB one (1920 x 1080 pixels) is much lower, hence a great part of RGB information is lost during the process of matching both images [25][26]. Moreover, the depth mapping is not 100% accurate, that is why shadows (empty spaces in a point cloud behind objects) are visible.

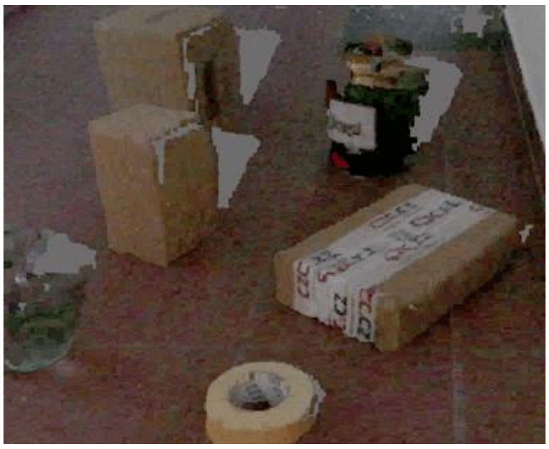


Figure. 6) Data streams available from Microsoft Kinect v2

3.3 Point Cloud Processing

Raw point cloud data obtained from depth sensor usually contain noise, due to the ambiguity of reflected IR amplitude, and other inaccurate information that needs to be removed [27]. Generally, it is necessary to segment the point cloud into some regions of interest to reduce program execution time by better organizing data or to exclude unnecessary points. That is why several algorithms were implemented for these applications.

This section briefly introduces some basic concepts about point cloud processing as well as some relevant algorithms and methods for that purpose are presented.

3.3.1 Point Cloud Library (PCL)

PCL is a powerful, modern C++ library for 3D point cloud processing. It implements plenty of algorithms and methods to cover every single part of 3D point cloud processing: segmentation, recognition, filtering, surface reconstruction, feature estimation, visualization, I/O communication etc. PCL was released as open source software, hence it is free for commercial and research use [6].

pcl

Figure. 7) The Point Cloud Library logo [31].

PCL is fully integrated with ROS. There is a message-based PCL interface for ROS, which allows PCL point clouds to be communicated and provides a set of functions to convert from native PCL types to ROS messages [30].

3.3.2 PassThrough filter

This filter simply passes points in an input cloud, which fall in an interval specified by **setFilterLimits()**, which applies on a field specified by **setFilterFieldName()**. For example, the filter is applied to the *z* coordinate, and accepts interval values from 0.0 to 1.0. The result is visualized in Figure. 8). The green points are the ones that remain after applying the filter and red ones are removed by the filter [32].

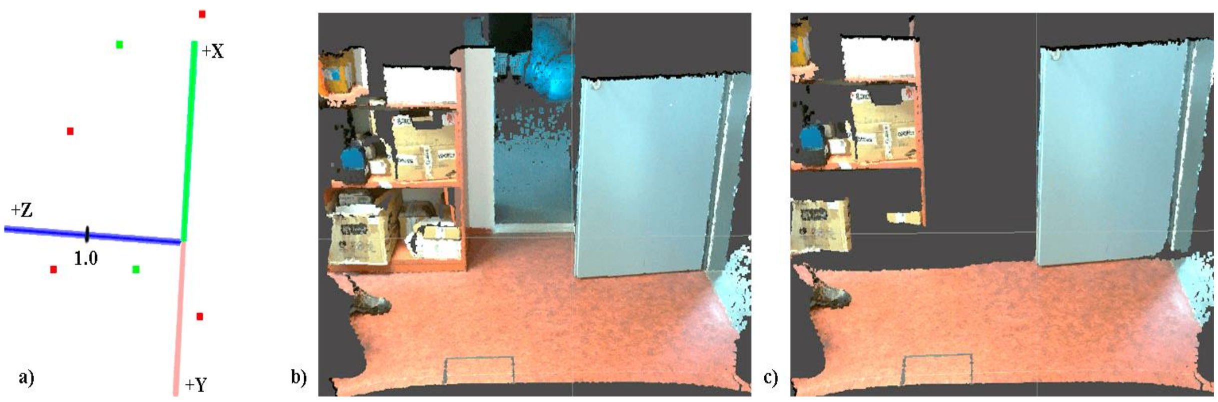


Figure. 8) PassThrough filter example: a) A schematic representation of PassThrough filter [31]; b) original point cloud; c) the point cloud after applying PassThrough filter

3.3.3 Downsampling

It is a method for reducing the size and complexity of a point cloud. As a result, the calculation load in a later processing is reduced and a resulted algorithm is running much faster. The most sophisticated downsampling algorithms preserve most of the original information that was in a cloud before applying the filter.

Voxel grid sampling is one of the most common downsampling methods. The respective **VoxelGrid** class in PCL creates a 3D voxel grid (in a nutshell, a set of small 3D elemental boxes) for an input point cloud data. Subsequently, the centroid of all points that lie inside each voxel is estimated, and new point with centroid coordinates substitutes the original points in each voxel. It is a bit slower than substituting them with the center of the voxel, but surfaces in a scene are represented more precisely. The smaller the size of voxels, the more accurate details in an input point cloud are preserved [31][32].

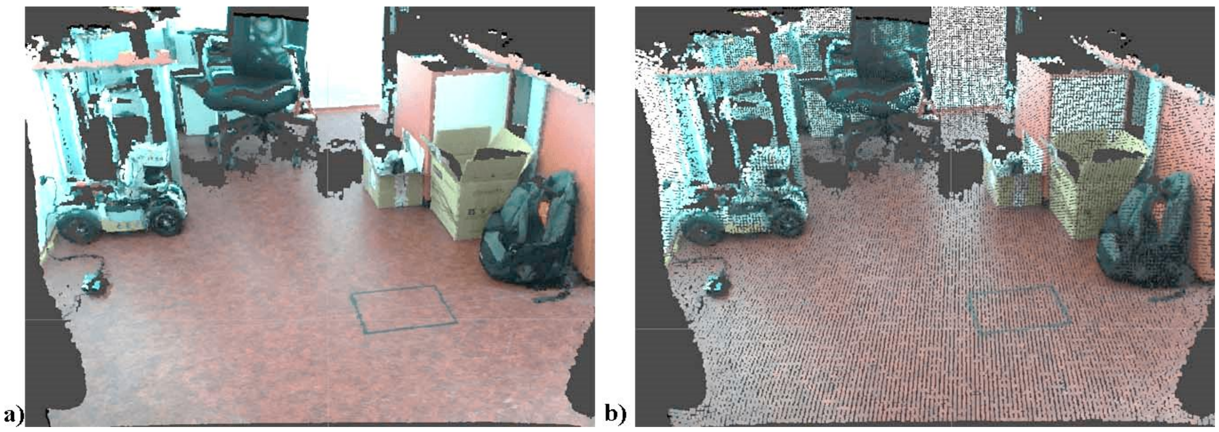


Figure. 9) The VoxelGrid downsampling method: a) original point cloud; b) point cloud after applying the VoxelGrid downsampling method with leaf size parameter 1.5 cm

3.3.4 Noise removal

As was stated earlier in 3.3, the point cloud obtained by RGB-D camera usually contain noisy data, which corrupts the resulting data. This complicates the estimation of point cloud features such as normal or curvature changes, which results in malfunctioning of methods that rely on such features[5]. There are currently three methods of outlier removal implemented in PCL: **statistical**, **conditional** and **radius** [31][32].

- The **RadiusOutlierRemoval** filter is the most primitive one, it just simply removes all points in an input point cloud that does not have the minimum number of neighbors within a certain range.
- The **ConditionalRemoval** filter removes all points in an input cloud that do not satisfy one or more custom conditions.
- The **StatisticalOutlierRemoval** filter is based on a statistical analysis on each point's neighborhood and computes the distribution of point to neighbors distances[6] for each

[5] For example: **SACSegmentationFromNormals**< **PointT, PointNT** >, which additionally requires a point cloud with estimated normals as an input[32]

[6] The number of neighbors to analyze the distance is set by **setMeanK()** method [32].

point in an input point cloud. As a result, all points whose mean distances do not lie inside an interval defined by the global distances mean and standard deviation[7] are considered as outliers and removed from an input dataset.

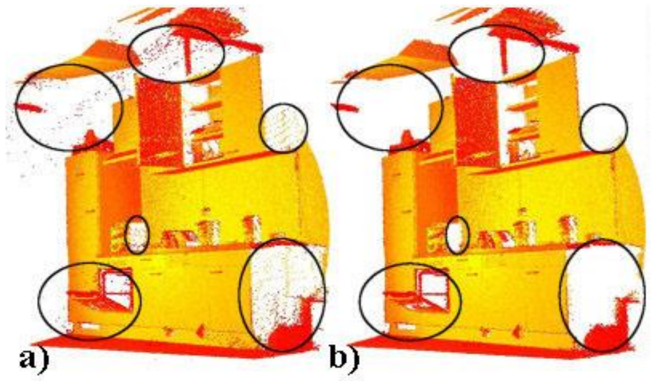


Figure. 10) Usage of **StatisticalOutlierRemoval** filter [31]: a) an input point cloud; b) a point cloud after applying the filter

3.3.5 The estimation of surface normals

Surface normals are quite important features of a point cloud. It is much easier and more accurate to segment planes and other primitive shapes in a point cloud with estimated normals.

NormalEstimation algorithm in PCL estimates a normal to a point on a surface by calculating the normal of a plane tangent to the surface, which basically relies on an analysis of the eigenvectors and eigenvalues of a covariance matrix created from the nearest neighbors of the point of interest. The scale factor k, which is set by **setKSearch()** method, defines the size of the smallest curvature that the algorithm is able to fully estimate its normals. The smaller the scale factor k, the more details on the edges are estimated as shown in Figure. 11) c) [31].

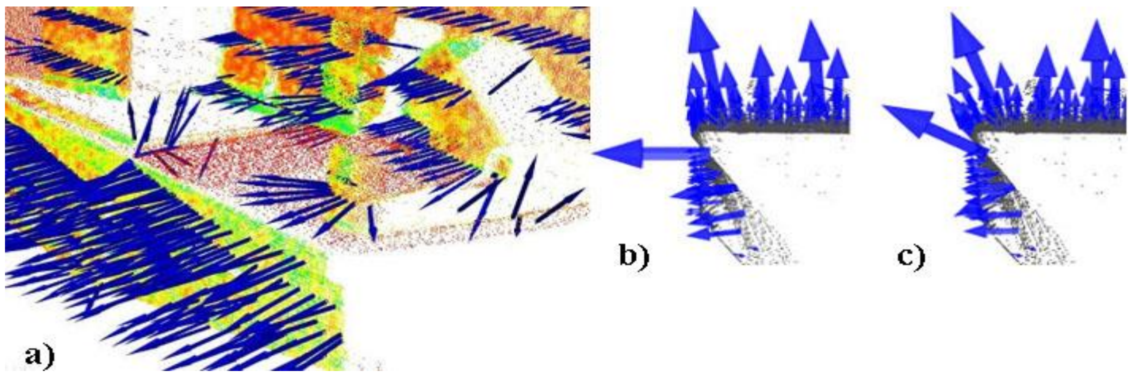


Figure. 11) The NormalEstimation algorithm results [31]: a) the orientation of normals in a scene (blue arrows are estimated normals); b), c) the estimated normals with different values of scale factor k

[7] Standard deviation can be changed by setting the standard deviation multiplier by **setStddevMulThresh()** method [32].

3.3.6 Data structures for managing point clouds

As was stated in 3.2, it is usually convenient to organize point clouds due to better performance. There are two main data structures for organizing point clouds implemented in PCL:

- An **Octree** – a tree-based data structure for managing scattered 3D data. Space is recursively divided into eight nodes (octants). Subsequently, each node can be also divided into other eight nodes and so forth. Two main applications of the octree are spatial partitioning and neighbor search within a point cloud [31].
- A **k-d tree** (k-dimensional tree) – a binary-search tree for organizing points in a space with k dimensions. It is extremely useful for range and nearest neighbor searches. Children in each level are separated along a specific dimension. At the root of the tree the children are separated by the first dimension: if the first dimension of a child is more than the root one it will be in the right sub-tree and on the contrary if the first dimension is less than the root one a child will be in a left sub-tree. Going each level down the tree divides on the next dimension. When all dimensions are exhausted, the division returns to the first one [31].

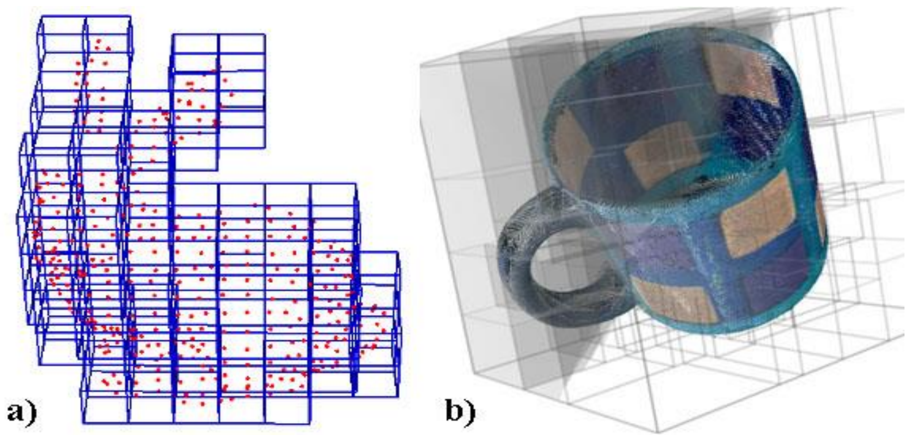


Figure. 12) Data structures for organizing point clouds [31]: a) an Octree representation of a point cloud; b) a K-d tree representation of a point cloud

3.3.7 Point Cloud projection and reconstruction of convex hull polygon

Sometimes it is required to acquire flat surface from an input point cloud by projection. There is an implemented method in PCL (**ProjectInliers** class), which projects a set of points onto a parametric model (plane, cylinder, sphere etc.) defined by model coefficients [31].

The projection can be further used as an input for reconstructing a 2D convex hull polygon (this approach is used for estimating object XY dimensions in the implemented solution). The method for reconstruction of convex hull polygon is implemented in PCL (**ConvexHull** class) with the usage of **libqhull** library [32].

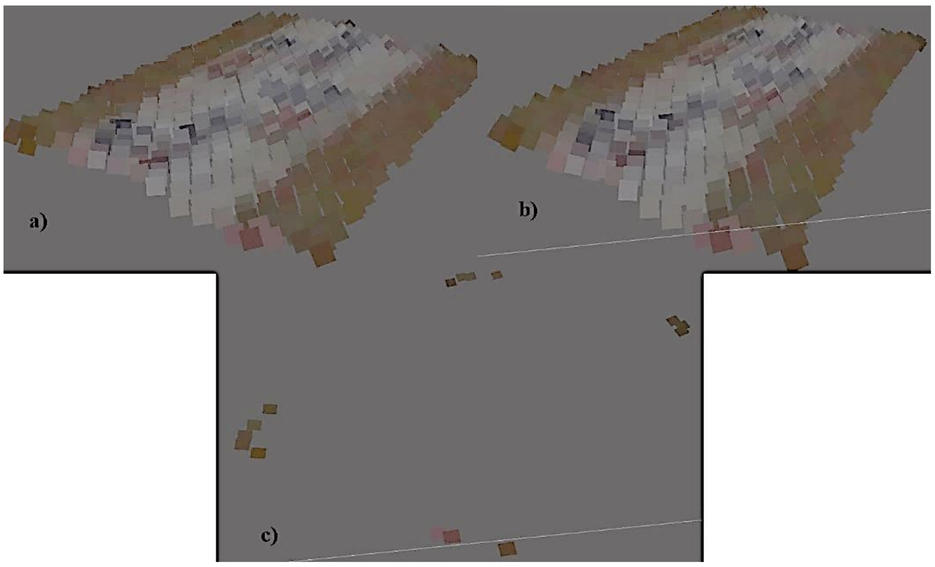


Figure. 13) The example of projection and convex hull reconstruction: a) an original cloud; b) a projection of the cloud to XY plane; c) 2D convex hull reconstructed from the projection.

3.4 Segmentation

Segmentation is a process of separating groups of points with related features based on some criteria: specific values of point coordinates, RGB values or values of estimated normals (see 3.3.5). As a result of the segmentation algorithm, the groups of segmented points can be easily analyzed, filtered or removed from an input point cloud.

A great part of implemented algorithms in PCL are related to the segmentation process. A big part of the implemented obstacle detection solution relies on segmentation algorithms implemented in PCL, thus a brief description of the most important segmentation methods will be provided in this section.

3.4.1 Module sample_consensus

One can find plenty of primitive shapes (like planes, cylinders, circles, spheres etc.) in a surrounding environment. Thus, all these shapes can be easily interpreted as a set of specific models with certain coefficients. Module sample_consensus in PCL library utilizes this concept by implementing such models. Currently, a plane has 5 implemented models in PCL, which use different constraints for detection algorithm:

- SACMODEL_PLANE,
- SACMODEL_PERPENDICULAR_PLANE,
- SACMODEL_PARALLEL_PLANE,
- SACMODEL_NORMAL_PLANE,
- SACMODEL_NORMAL_PARALLEL_PLANE

For example, SACMODEL_PLANE represents a plane model with four coefficients in Hessian Normal form: [normal_x, normal_y, normal_z, d]. In contrast, SACMODEL_NORMAL_PLANE has the same coefficients as SACMODEL_PLANE, but also uses an additional criterion: the estimated surface normals at each inlier point has to be parallel to the surface normal of the detected plane, within a certain maximum angular deviation. This often results in more accurate plane detection [32].

Module sample_consensus also provides implementations of robust Sample Consensus (SAC) estimators: SAC_RANSAC, SAC_LMEDS, SAC_MSAC, SAC_RRANSAC, SAC_RMSAC, SAC_MLESAC and SAC_PROSAC [32].

3.4.2 Random Sample Consensus (RANSAC) estimator

RANSAC is an iterative algorithm for fitting a precisely defined model into an input data set. The method implies that an observed input data consists only of both **inliers** and **outliers**. An **inlier** is a point that fits a defined model algorithm searches for, while an **outlier** is a point that does not fit a model under any circumstances. RANSAC iteratively selects s random subset of points in the input data. These selected points are hypothetical inliers and the hypothesis is checked by performing the following steps [31][33]:

1. The parameters of the defined model are reconstructed based on the hypothetical inliers;
2. The remaining points are subsequently tested against the parameters of the reconstructed model. The points, which fit good enough to the estimated model are also recognized as hypothetical inliers;
3. The estimated model is considered valid if the reasonable amount of points were recognized as hypothetical inliers;
4. The parameters of a model are reconstructed from all hypothetical inliers that were found;
5. In the end, the error of the inliers relative to the resulted model is estimated.

This algorithm then iterates a certain number of times. The resulted model of each iteration is either rejected because a small amount of points is considered as inliers or accepted with a corresponding error evaluated. If the error is lower compared to the last saved model, the new model overrides the last one [31][33].

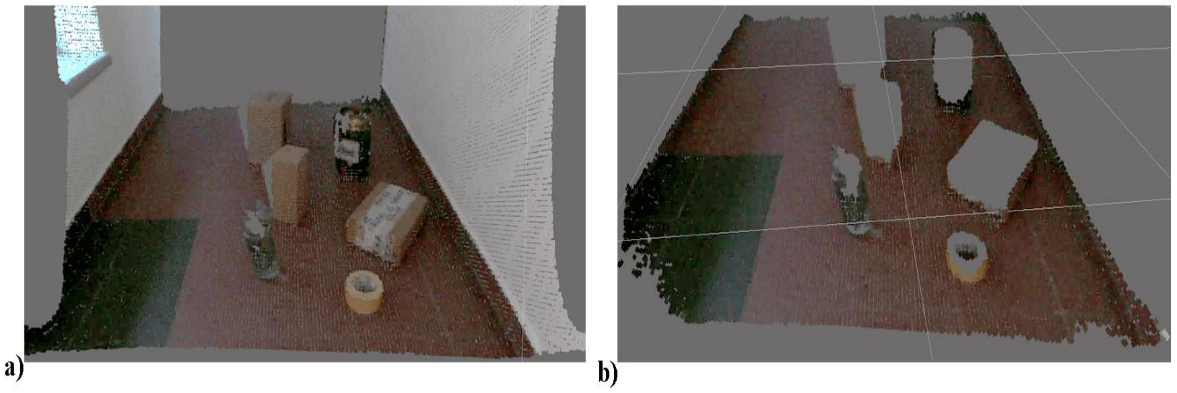


Figure. 14) The example of RANSAC plane segmentation: a) an input point cloud; b) the segmented ground plane

The preprocessing of an input point cloud, such as noise filtering (see 3.3.4), clustering etc. is often required, because RANSAC estimator can find only one model for an input data set [31]. That is why the basic clustering methods are introduced in next section.

3.4.3 Clustering

Clustering is a method that separates an input point cloud into smaller parts (clusters), which allows some algorithms to perform better (see 3.4.2) and reduces the overall processing time by filtering out unnecessary clusters. Currently, there are implemented several clustering methods in PCL: Euclidean clustering, Region Growing segmentation, Min-Cut Based segmentation, Difference of Normals Based segmentation, Conditional Euclidean clustering and Clustering into Voxels [31].

One of the simplest clustering algorithms is Euclidean Clustering, which is implemented in PCL as **EuclideanClusterExtraction** class. This algorithm simply groups points based on the Euclidean distance between them. The closeness threshold must be set (by **setClusterTolerance()** method), which defines the maximum distance between points to be considered as a part of the same cluster [31][32].

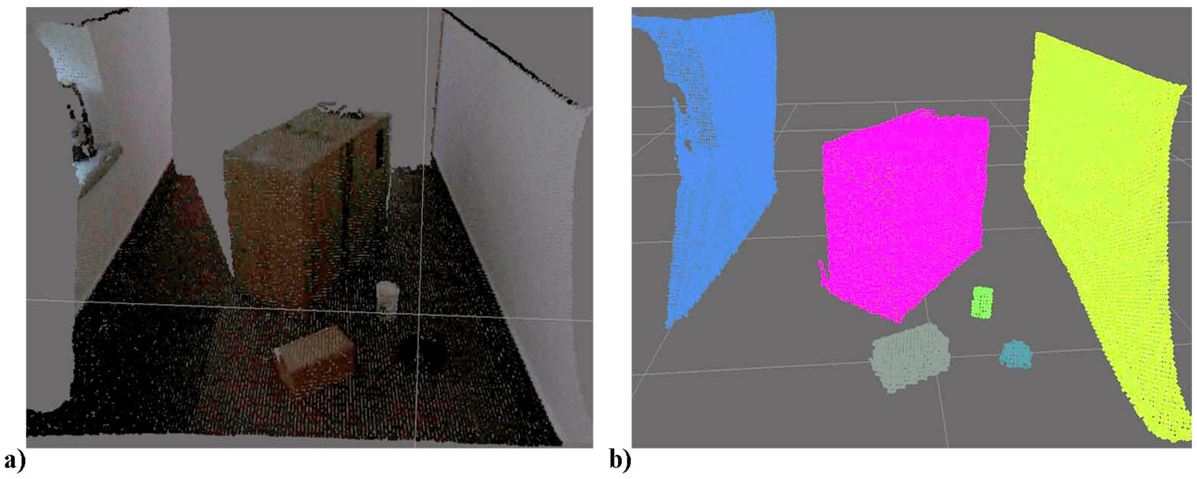


Figure. 15) The application of Euclidean Clustering algorithm: a) an input point cloud; b) the resulting clusters (different color indicates different clusters)

The PCL implementation of algorithm utilizes a Kd-tree data structure for finding the nearest neighbors, and contains the following steps[31]:

1. Create a Kd-tree data structure to organize an input point cloud P;
2. Create an empty list of clusters C and an empty queue Q, which will contain points that need to be checked;
3. Cycle through every point $p_i \in P$ do:
    a) Add p_i to the queue Q;
    b) For every point $p_j \in Q$ do:
        - Look for the set P_j^k of neighbors of a point p_j in a sphere with radius $r < d_{thresh}$, where d_{thresh} is a closeness threshold;

   - For each neighbor found $p_j^k \in P_j^k$, add point p_j^k to queue Q, if it has not been already processed;

   c) If all points in Q are processed, add Q to the list of clusters C, and reset Q to an empty list;

4. Repeat the previous step until all points $p_i \in P$ have been processed. As a result, all points $p_i \in P$ are now separated as items of the list of clusters C.

4 ROBOTIC PLATFORM

The proposed obstacle detection system includes not only an implemented detection algorithms but also a capturing device with a configured driver, which links a device against a software and allows their interaction. Moreover, a software core of the system was built under Xubuntu operating system with installed Robot Operating System (ROS) environment.

Therefore, this chapter deals with an introduction of ROS environment as well as gives a thorough specification of used capturing device and utilized drivers. Then, the chapter provides a description of a mobile robot platform the obstacle detection system has been tested on.

4.1 The Robot Operating System

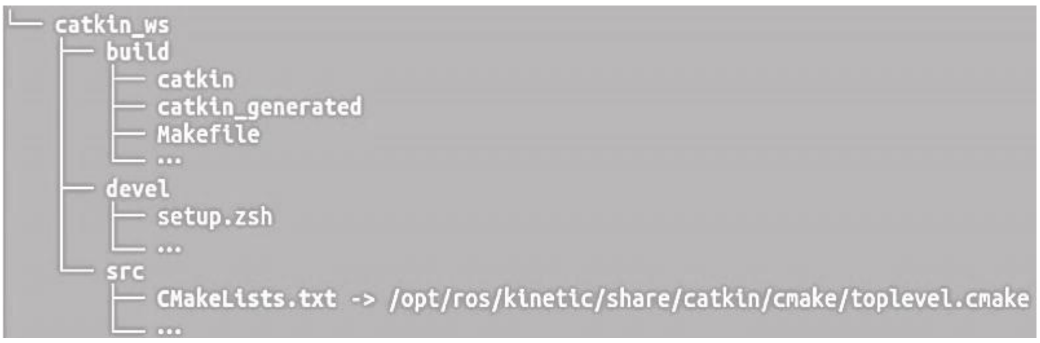
A robot itself is a highly complex system and usually consists of multiple hardware parts (sensors, actuators etc.), which are handled by complicated software. Building a real truly robust robotic system is so challenging that no single person, laboratory or institution can hope to do it easily on their own. To solve this issue, the collaborative robotics software development approach was proposed (the researchers throughout the world can collaborate and build upon each other's work) [34].

The Robot Operating System is an open source flexible framework for writing robot software that was created specifically to support the collaboration of robotics researchers. ROS consists of various tools, libraries, and conventions that help to simplify the process of writing a robust and complex robot software. ROS also utilizes a distributed approach, which means that it is composed of more than 3,000 user-contributed packages that can be easily integrated into every ROS compatible system [34].

4.1.1 The Workspace

ROS offers its own file system, which centralizes the building of a project, but at the same time is flexible enough to decouple dependencies [30].

The Workspace is a part of that file system, which centralizes various packages under one folder. It is divided into three directories and each of them has a different function [30]:



```
└── catkin_ws
    ├── build
    │   ├── catkin
    │   ├── catkin_generated
    │   ├── Makefile
    │   └── ...
    ├── devel
    │   ├── setup.zsh
    │   └── ...
    └── src
        ├── CMakeLists.txt -> /opt/ros/kinetic/share/catkin/cmake/toplevel.cmake
        └── ...
```

Figure. 16) The ROS workspace file structure [30]

- The **src** directory contains all packages used by the workspace;
- The **build** directory contains CMake and catkin cache information and configuration;
- The **devel** directory keeps all the compiled programs (used to test programs during the development step).

Packages inside the workspace are compiled by the bash command:

$$\$\ catkin\_make$$

As a result, executables are built inside the build space directory [30].

4.1.2 Packages

ROS software consists of ROS packages and each of them represents the minimum content to create a program within ROS. The package consists of ROS runtime processes (nodes), configuration and message files etc. The ROS community has thousands of publically available packages in open repositories [30][35].

As was stated in 4.1.1, ROS packages are placed inside **src** directory in a workspace. Every package must include **CMakeLists.txt** and **package.xml** files that describe the contents of the package and information for compilation and interaction with catkin. The structure of the package looks as follows [35]:

- The **src** directory – the source code of nodes (see 4.1.3) lies here;
- The **scripts** directory – executable scripts written in Bash, Python or other scripting language are located here;
- The **srv** directory – service files of the .srv extension are located here;
- The **msg** directory – the custom message files of the .msg extension lie here;
- The **launch** directory – the launch files of the .launch extension (see 4.1.3) lie here;
- The **include/package_name** directory – the headers of the libraries that package needs are included here;
- The **Package.xml** file – a package manifest file.
- The **CMakeLists.txt** file – a CMake build file;

4.1.3 Nodes and nodelets

A **node** is an executable file that is able to communicate with other nodes using topics and services. Nodes are written in either C++ using **roscpp** library or in Python using **rospy** library. The node's implementation is decoupled from other parts of a system, making the system simpler to maintain and easier to understand.

There is also a special type of node called **nodelet**, which allows running multiple nodes in a single thread. It allows nodes to communicate more efficiently, without overloading the network. This is crucial in the case for using 3D sensors, where there is a huge amount of data transferred [30].

Every node must have a unique name (identifier) in the ROS environment, which is used for communication without ambiguity. Nodes can use ROS Parameter server, which is a shared dictionary accessible via a network. This server provides the node with ROS parameters –

special parameters, which can be changed dynamically during the runtime of a node. This allows reconfiguring the node without recompiling the code, thus saving the development time and making the code more versatile [30][35].

Single ROS node is launched by shell command:

$$\$\ rosrun\ [package\_name]\ [node\_name]$$

Because most of the time ROS software needs several nodes to be launched, a typing into shell repeating commands can become quite tedious. That is why the **launch files** were added to ROS environment to allow launching multiple nodes at once. Moreover, ROS parameters can be set with initial values in the launch file. The **launch file** is basically a configuration file of the .launch extension, which is written in XML syntax. It is launched by shell command[30]:

$$\$\ roslaunch\ [package\_name]\ [launch\_file\_name]$$

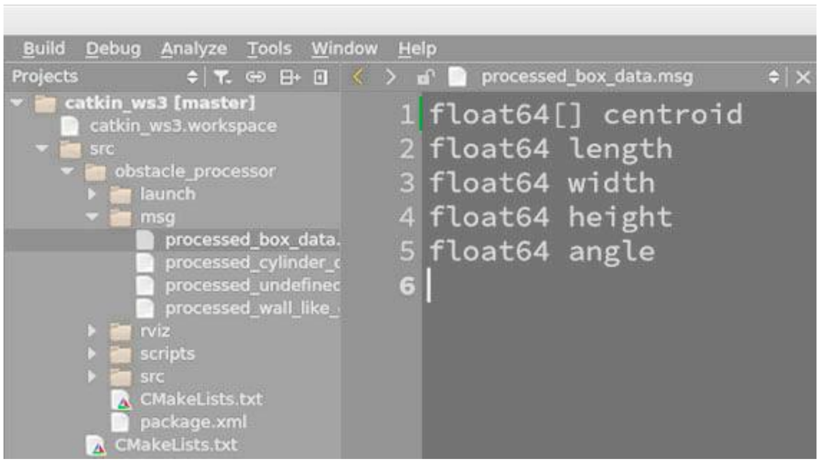
4.1.4 Topics

Nodes communicate with each other by transmitting data through **topics**. In a nutshell, a topic is a stream of ROS messages (see 4.1.5) with the defined data type. Topics allow ROS to utilize so-called **publish/subscribe communication** approach, which is a perfect solution for data exchanging in a distributed system. Basically, the distribution of data in a ROS program is done by performing the following steps[35]:

1. A node advertises the topic name and the type of messages that are going to be sent.
2. The node publishes the actual message on the topic.
3. Nodes that are interested in receiving messages on a specific topic must subscribe to it by making a request to **roscore** service.
4. All messages transferred to the topic are delivered to the node that subscribed to the respective topic.

4.1.5 Messages

ROS message is a simple data container that uses standard data types or user-defined ones. The message is a file with the .msg extension, which has been written using conventions established by ROS.



```
float64[] centroid
float64 length
float64 width
float64 height
float64 angle
```

Figure. 17) The example of custom message data

4.1.6 Roscore

ROS architecture is a hybrid version of classical client/server approach and fully distributed one. This is possible, because of a presence of a central **roscore** service, which allows **peer-to-peer** communication between nodes [35].

Roscore is basically a service, which allows nodes to find each other. When a node launches, **roscore** registers the information about message streams node publishes and the ones the node is interested in. After a new node appears, roscore gives it a necessary information (addresses of the relevant nodes) to make a peer-to-peer connection with nodes that operate on the same message topics. Each node periodically calls on **roscore** to update data about its peers (see Figure. 18) [30][35].

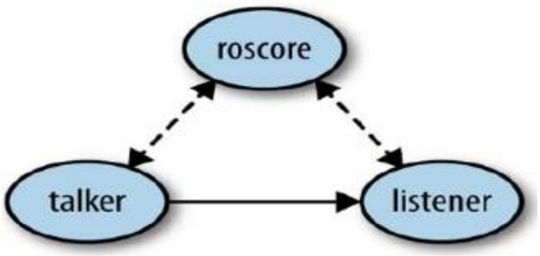


Figure. 18) The scheme of minimalist 2 node system [35]: the connection between roscore and nodes is short-lived and periodic, comparing to the continuous one between nodes

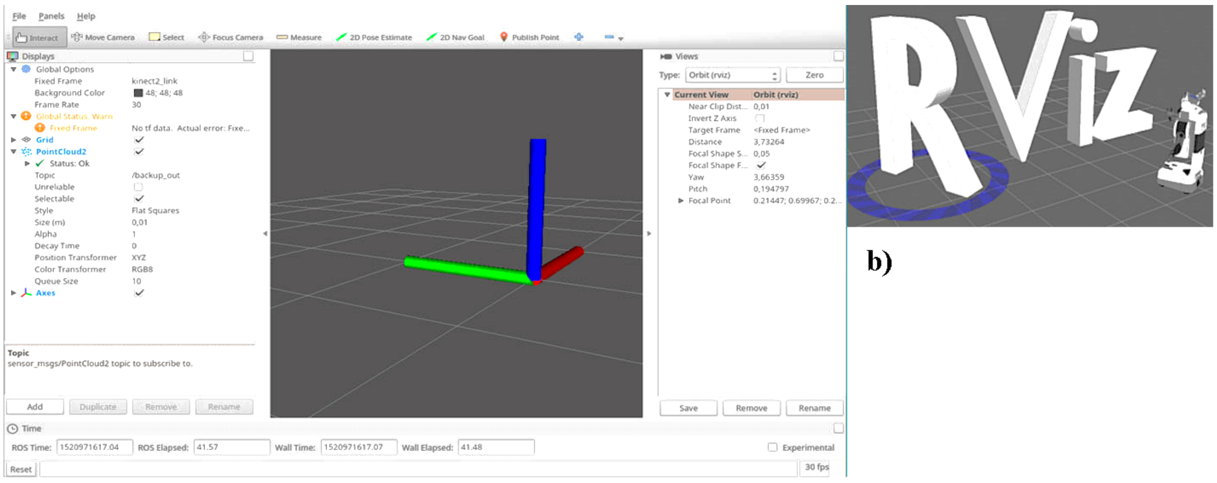
4.1.7 Bags

ROS has a useful feature to save all the information about messages, topics, services etc. This data is saved as a **bag** file with the .bag extension. The **bag** file can be later replayed to virtually simulate everything it contains. In other words, a real session that has been recorded can be restored.

Bag files allow simpler debugging and hence faster development of ROS software. The bag functionality was extensively used, during the project development, and undoubtedly it accelerated the process.

4.1.8 RVIZ

Data coming from sensors is needed to be visualized, due to various reasons: software development, program debugging or presentation of implemented solution. There is a 3D visualization tool (RVIZ) implemented for ROS. It allows to selectively visualize different kinds of data utilized in ROS environment [34][36].



```
Global Options
  Fixed Frame            kinect2_link
  Background Color       48; 48; 48
  Frame Rate             30
Global Status: Warn
  Fixed Frame            No tf data.  Actual error: Fixe...
Grid                     ✓
PointCloud2              ✓
  Status: Ok
  Topic                  /backup_out
  Unreliable             ☐
  Selectable             ✓
  Style                  Flat Squares
  Size (m)               0,01
  Alpha                  1
  Decay Time             0
  Position Transformer   XYZ
  Color Transformer      RGB8
  Queue Size             10
Axes                     ✓

Topic
sensor_msgs/PointCloud2 topic to subscribe to.

ROS Time: 1520971617.04   ROS Elapsed: 41.57   Wall Time: 1520971617.07   Wall Elapsed: 41.48

Type: Orbit (rviz)
Current View             Orbit (rviz)
  Near Clip Dist...      0,01
  Invert Z Axis          ☐
  Target Frame           <Fixed Frame>
  Distance               3,73264
  Focal Shape S...       0,05
  Focal Shape F...       ✓
  Yaw                    3,66359
  Pitch                  0,194797
  Focal Point            0.21447; 0.69967; 0.2...
```

b)

a)

Figure. 19) 3D Robot Visualizer: a) the GUI of RVIZ; b) the logo of RVIZ [36].

4.1.9 ROS plugin for Qt Creator

Qt Creator is a popular IDE for C++ development, and ROS community created an extension for it to simplify development of ROS packages and to centralize ROS tools in one place. Besides ROS tools it also provides users with syntax highlighting, code completion, version control (Git, Subversion), editors for C++ and Python, debugging tools (GDB, CDB, LLDB etc).

4.2 Microsoft Kinect v2

In 2013 with the release of an entertaining system Microsoft Xbox ONE, the second generation of Kinect cameras was released. The Microsoft Kinect v2 is a completely different RGB-D camera comparing to the Kinect v1, which utilizes the ToF technology instead of structured light one (see 3.1.1 and 3.1.2) [38].

The implemented obstacle detection system uses Microsoft Kinect v2 as a capturing device. That is why a complete description of this device is provided in this section.

4.2.1 The overview of key components

Kinect v2 consists of RGB camera, IR camera, and three IR light sources: each of them generates a modulated wave with different amplitude. The location of these components is demonstrated in Figure. 20) b).

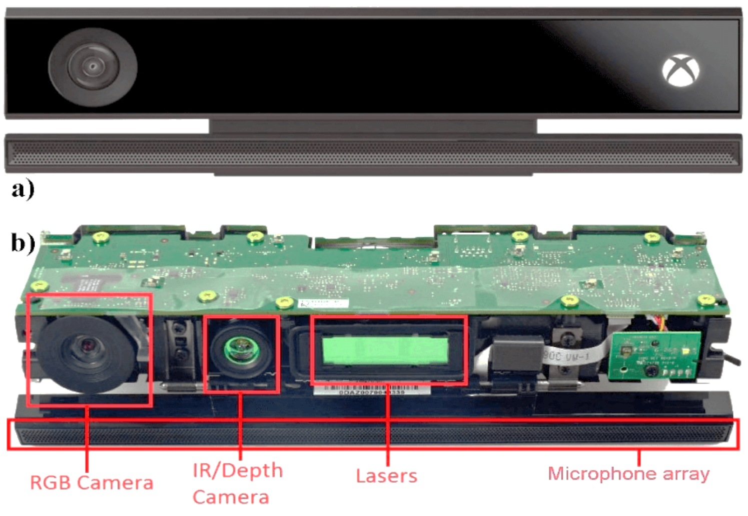


Figure. 20) Microsoft Kinect v2 RGB-D camera [39]: a) an anterior view of a device; b) an arrangement of camera's components.

The device captures a 1920 x 1080 color image and 512 x 424 depth map at maximum 30 fps. The fields of view (FOV) of Kinect v2's both RGB and depth cameras are considerably larger compared to Kinect v1's ones[8]. Because of utilization of ToF principle and higher depth map resolution, Kinect v2 is considerably more precise compared to Kinect v1. The minimum distance the camera is able to detect depth is 500 mm. Even though the Kinect v2 is able to measure depth values for distances longer than 9 meters, the official drivers do not allow to evaluate distances longer than 4.5 meters. That is why open source Kinect drivers were implemented to use the full potential of the device[25][38][40].

4.2.2 IAI Kinect2 and libfreenect2

Libfreenect2 is an open source cross-platform driver (supported by Windows, MacOS X, and Linux) designed for communication between Microsoft Kinect v2 and PC. The driver supports RGB image transfer, IR and depth image transfer, registration of RGB and depth images. The connection between PC and Microsoft Kinect v2 is ensured by USB 3.0 controller[41].

IAI Kinect2 is a set of ROS packages, which has a collection of tools and libraries to communicate between ROS and Kinect v2 [4]. It consists of:

- Tools for calibration, which allows calibrating the IR sensor against the RGB sensor, and the depth measurements;
- Depth registration library with **OpenCL** support;
- A viewer for the RGB, depth and point cloud data;
- A bridge to convert data between **libfreenect2** and ROS.

During the development of the obstacle detection system, both **libfreenect2** and **IAI Kinect2** were extensively used for different purposes:

[8] 84.1 x 53.8 against 62 x 48.6 for RGB camera and 70.6 x 60 against 58.5 x 46.6 for depth camera.

- At hardware level to gather a depth data from Kinect v2 and to convert it into point clouds;
- To represent data acquired on a hardware level as ROS topics and to publish them;
- To calibrate RGB and depth cameras to acquire more precise data;

4.3 Mobile platform setup

An implemented solution is flexible enough to run on different configurations of a mobile platform. The only requirement is that the mobile platform must have the configured ROS based embedded control system.

Obstacle detection system has been launched and tested on **Breach** robotic platform, which has been designed by Bender Robotics s.r.o. **Breach** is a compact modular robotic platform designed for populated indoor environments. It is provided with ROS based embedded control system, differential chassis, power supply system and on-board sensors (ultrasonic proximity sensors, laser rangefinder) to ensure basic functionality such as localization and mapping. The general specifications of Breach are shown in Appendix 1 a [42].

Breach platform has been extended with a sturdy construction to support Microsoft Kinect v2 as shown in Appendix 1 b. The camera is powered by on-board power supply system and is connected to the laptop. The ROS master is launched on the mobile platform and is connected with ROS host (laptop). After that, the obstacle detection system is ready to run and it only requires a proper sequence of shell commands to be launched. The basic sequences of such commands are provided in Appendix 1 c.

5 IMPLEMENTATION

This chapter provides an in-depth overview of software implementation of the obstacle detection system. The main algorithms are thoroughly described step by step to clarify their logic.

5.1 Structure of the system

The implemented solution has been built in ROS environment (see 4.1) and basically represents ROS workspace, which contains two ROS packages:

- **Iai_kinect2** (see 4.2.2): this package communicates with the **libfreenect2** driver to perform low-level tasks, such as retrieving a point cloud data (**kinect2_bridge**) and camera calibration (**kinect2_calibration**).
- **Obstacle_processor**: the actual system implementation lies here. It contains **calibration** and **obstacle_processor** nodes.

  The system also relies on external software such as:
- **tf_publisher** – a node which publishes transformations between the capturing device and so-called **base_link** – an approximated centroid of a robot mobile base.
- **Libfreenect2** (see 4.2.2) – a driver for low-level communication between PC and Microsoft Kinect v2.

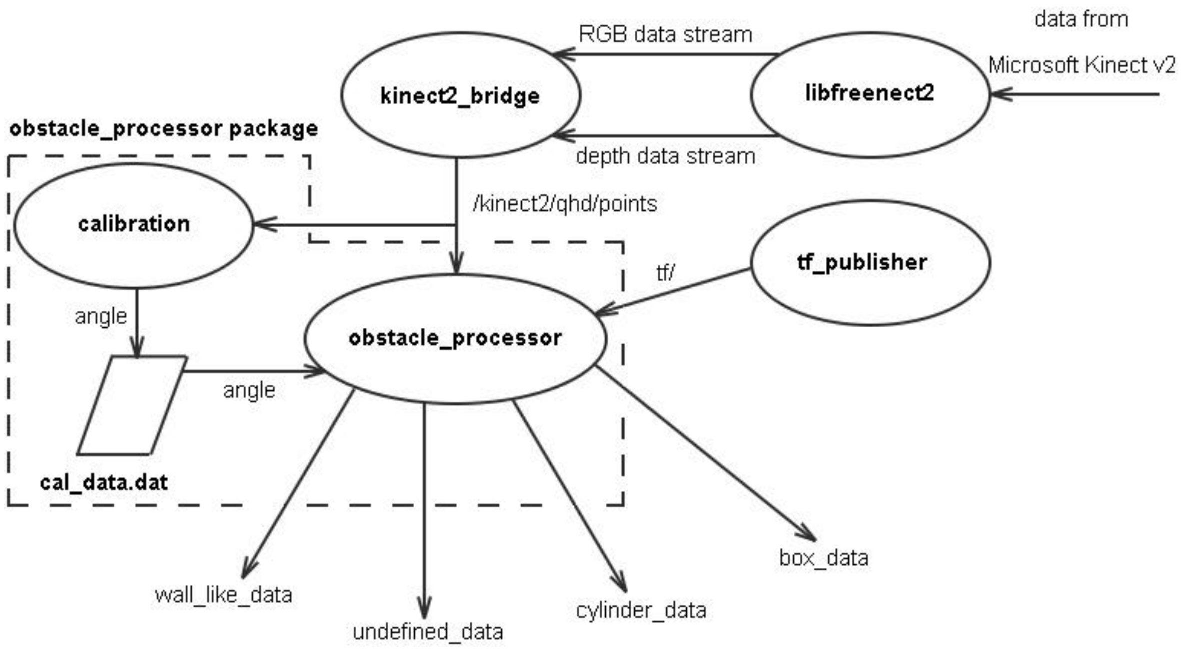
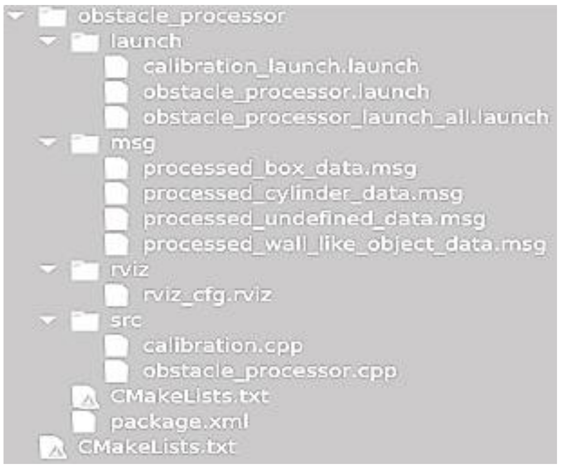


Figure. 21) The schematic structure of the system

The **obstacle_processor** package has a typical ROS package structure (see 4.1.2) and also contains one extra folder **rviz/** used to store an RVIZ configuration file as shown in Figure. 22).



```
obstacle_processor
    launch
        calibration_launch.launch
        obstacle_processor.launch
        obstacle_processor_launch_all.launch
    msg
        processed_box_data.msg
        processed_cylinder_data.msg
        processed_undefined_data.msg
        processed_wall_like_object_data.msg
    rviz
        rviz_cfg.rviz
    src
        calibration.cpp
        obstacle_processor.cpp
    CMakeLists.txt
    package.xml
CMakeLists.txt
```

Figure. 22) The structure of the obstacle_processor ROS package

5.2 The principle of operation in a nutshell

Since next sections are focused on an explanation of elementary pieces of the system, this section provides a brief description of how the system actually works and gives a basic overview of its data flow.

The system’s data flow is demonstrated in Figure. 21) and can be described by the following steps:

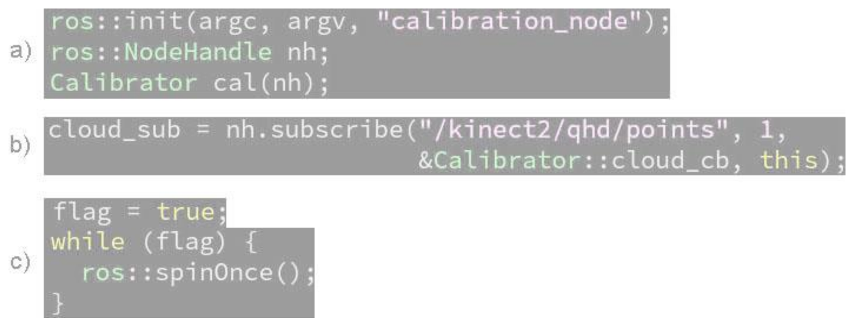
1. Firstly, **kinect2_bridge** is launched and it reads data captured by Kinect v2 with the usage of the **libfreenect2** driver;
2. Next, RGB and depth data are processed by **kinect2_bridg**e to acquire a point cloud data. The resulted data is published on a ROS topic **/kinect2/qhd/points**;
3. If launching a system for the first time or the angle of Kinect v2’s decline is changed, the calibration node is launched. This node subscribes to **/kinect2/qhd/points** topic, then acquires the angle and finally writes it to **cal_data.dat** calibration file or rewrites the existing one;
4. Obstacle processor node firstly tries to read an angle from a **cal_data.dat** file and, if fails, quits with an error. Then, if transform flag is set to true, the node subscribes to **tf/** topic to acquire transformation data. Subsequently, it processes input point cloud data to find obstacles, which are then recognized. If transform flag is set to true, the coordinates of a centroid of obstacles are transformed to the **base_link** coordinate frame;
5. Finally, obstacles data is published on respective topics and can be used by any ROS node for further processing.

5.3 Calibration node

A capturing device can be mounted on a robot mobile base with different decline angle. In order to avoid tedious measuring and manual setup, a calibration node has been implemented. Calibration node expects that the largest plane in an input point cloud data is a ground plane to successfully acquire a calibration data, hence a preparation step is often required. The entire implementation of the calibration node has been divided into elementary steps (see Appendix 2 A) and each of following subsections provides a thorough description of a corresponding step.

5.3.1 Initialization

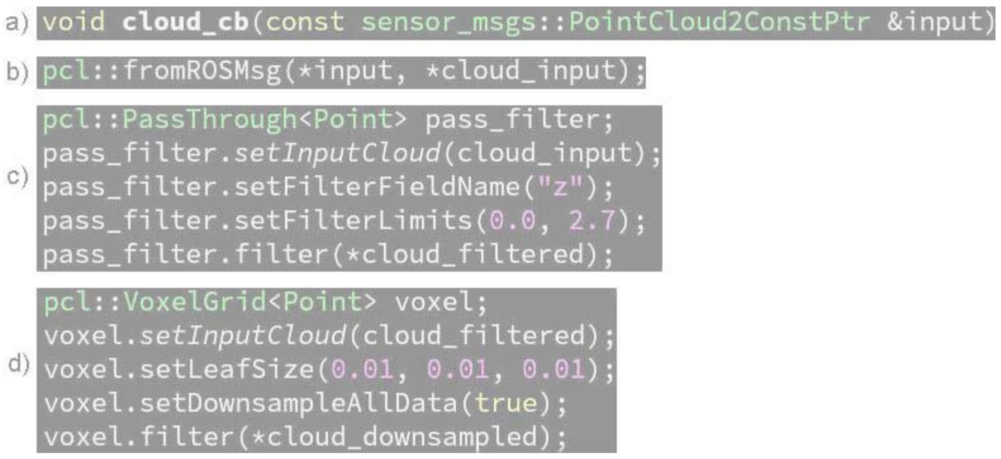
The initialization step is done inside the **main()** method. This is the first method, which is entered after launching the node. The necessary ROS setup is done here, an object of **Calibrator** class is constructed from passed **NodeHandle** object (see Figure. 23) a) and a subscription to point cloud topic is done inside **Calibrator** constructor (see Figure. 23) b). Also, the logical variable **flag** is initialized with **true** value. This variable ensures that **cloud_cb** method is entered only once (see Figure. 23) c.)


a)
```
ros::init(argc, argv, "calibration_node");
ros::NodeHandle nh;
Calibrator cal(nh);
```
b)
```
cloud_sub = nh.subscribe("/kinect2/qhd/points", 1,
                         &Calibrator::cloud_cb, this);
```
c)
```
flag = true;
while (flag) {
  ros::spinOnce();
}
```
Figure. 23) The source code of initialization of the calibration node

5.3.2 Preprocessing

The actual algorithm's logic lies inside a **cloud_cb()** method (see Figure. 24) a), which is invoked, when a point cloud topic is published. Preprocessing can be split into three steps:

1. The input point cloud is converted from ROS message to PCL compatible type, as shown in Figure. 24) b);
2. **PassThrough** filter (see 3.3.2) is applied to the z coordinate in camera's coordinate frame (see Figure. 1) to reduce the processed data's size, as shown in Figure. 24) c;
3. The **VoxelGrid** downsampling (see 3.3.3) is applied to a point cloud to improve a performance of the algorithm, as shown in Figure. 24) d;



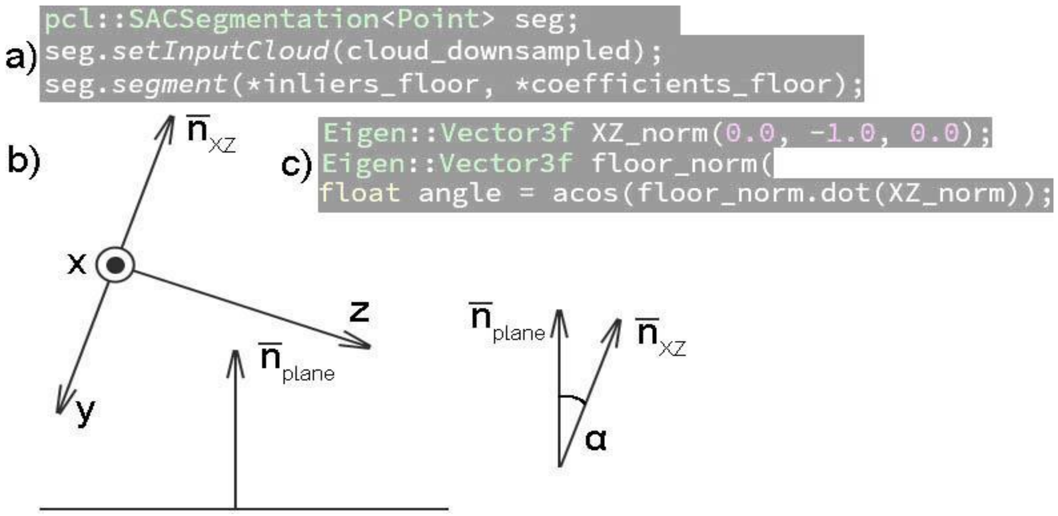
a)
```
void cloud_cb(const sensor_msgs::PointCloud2ConstPtr &input)
```
b)
```
pcl::fromROSMsg(*input, *cloud_input);
```
c)
```
pcl::PassThrough<Point> pass_filter;
pass_filter.setInputCloud(cloud_input);
pass_filter.setFilterFieldName("z");
pass_filter.setFilterLimits(0.0, 2.7);
pass_filter.filter(*cloud_filtered);
```
d)
```
pcl::VoxelGrid<Point> voxel;
voxel.setInputCloud(cloud_filtered);
voxel.setLeafSize(0.01, 0.01, 0.01);
voxel.setDownsampleAllData(true);
voxel.filter(*cloud_downsampled);
```

Figure. 24) The source code of preprocessing of the calibration node

5.3.3 Processing

During the processing step, the decline angle is determined and written to the file. This process can be split into the following steps:

1. A ground plane is segmented from an input cloud by using RANSAC-based segmentation method (see 3.4.2) as shown in Figure. 25) a;
2. The output **ModelCoefficients** object is used to create a **Vector3f** object, which represents a normal vector of the segmented ground plane as shown in Figure. 25) c;
3. The angle between the normal vector of the plane and a normal to XZ plane of Kinect's coordinate frame is calculated as shown in Figure. 25) c. A scheme of the angle calculation is demonstrated in Figure. 25) b;
4. The estimated angle is written to the **cal_data.dat** file;
5. The logical variable **flag** is set to false allowing to leave callback function and shut down the node.



a)
```
pcl::SACSegmentation<Point> seg;
seg.setInputCloud(cloud_downsampled);
seg.segment(*inliers_floor, *coefficients_floor);
```
c)
```
Eigen::Vector3f XZ_norm(0.0, -1.0, 0.0);
Eigen::Vector3f floor_norm(
float angle = acos(floor_norm.dot(XZ_norm));
```

Figure. 25) Processing step of the calibration node

5.4 Obstacle processor node

The main detection and recognition logic lies inside the **obstacle_processor** node. The node expects that the calibration of a capturing device has been conducted and the calibration file **cal_data.dat** exists with a valid decline angle written to it. The implemented algorithm has been split into elementary steps (see Appendix 2 b, c) and each of following subsections describes each step in a detail.

5.4.1 Initialization

An initialization step (see Appendix 2 b) is done inside the **main()** method. Firstly, ROS node is initialized and a decline angle is read from the **cal_data.dat** file. The method exits, if an error occurs during reading an angle. Next, **ObsctacleProcessor** object is constructed with the **NodeHandle** object. Inside **ObstacleProcessor** constructor, a subscription to point cloud topic is registered. Also, all necessary publishers are initialized and an algorithm execution continues inside the **cloud_cb()** method, which is invoked when new point cloud topic is sent.

5.4.2 Preprocessing

Firstly, an input point cloud has to be converted from ROS message type to PCL compatible one. Subsequently, **PassThrough** (see 3.3.2) and **VoxelGrid** (see 3.3.3) filters are applied to converted point cloud to reduce a size of data, hence this step improves the performance of the entire algorithm.

5.4.3 Ground plane processing

Following the preprocessing step, it is essential to properly segment a ground plane and use the data acquired during this segmentation for further data processing. This is done by the following steps:

1. Angle α, which has been read from a **cal_data.dat** file, is used to define a vector, which is an approximation of a ground plane's normal. This vector is used to find the largest plane perpendicular to it in an input point cloud. The RANSAC-based segmentation algorithm is used to segment the ground plane and a scheme of the process is shown in Figure. 26);

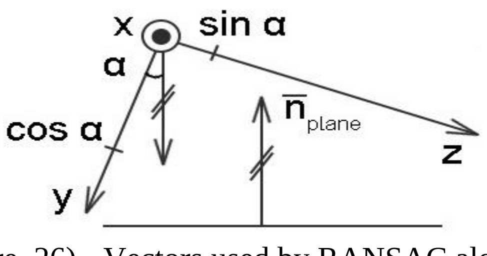


Figure. 26) Vectors used by RANSAC algorithm

2. The previously used algorithm also produces a **ModelCoefficients** object, which contains Hessian plane coefficients. First three coefficients are used to define a normal of the ground plane. After that, an angle between the normal and a vector perpendicular to XY plane is calculated;

3. The angle is used to rotate an input point cloud so that *z* coordinate axis is collinear with the normal of ground plane as shown in Figure. 27) b;
4. The arbitrary point, which lies on a ground plane, is selected and negative value of its z coordinate is used to define z component of a translation vector. Next, translation is applied to move an origin of the coordinate frame onto the ground plane as shown in Figure. 27) c;
5. The RANSAC-based segmentation algorithm is reapplied to transformed point cloud to refine Hessian coefficients of a ground plane.

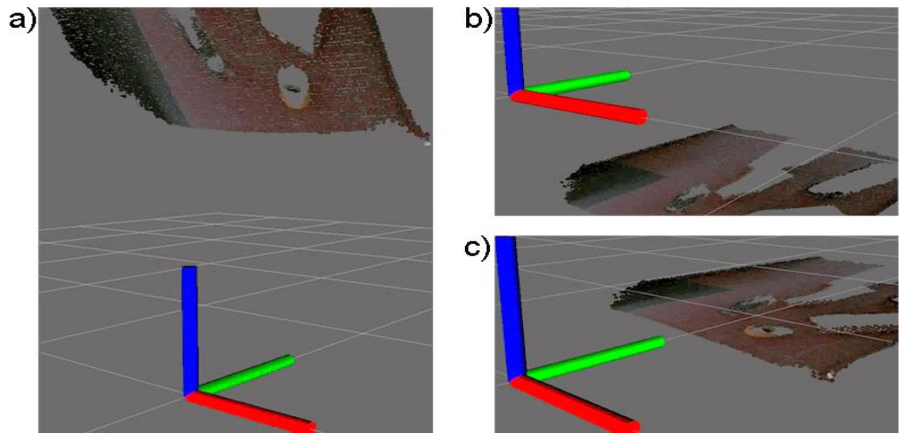


Figure. 27) Transformation applied to a ground plane: a) an original ground plane; b) ground plane after applying a rotation; c) ground plane after applying a translation

5.4.4 Extraction of clusters

Next, **EuclideanClusterExtraction** object (see 3.4.3) is utilized to separate an input point cloud into **clusters** i.e. potential obstacles as shown in Figure. 28). Three thresholds are used to configure this process:

1. **Min_cluster** defines the minimum number of points that a cluster must contain to be considered valid. This helps to sort potential outliers out.
2. **Max_cluster** is opposite to the previous parameter. This parameter can be set to a specific value to sort out too large clusters.
3. **Cluster_tol** defines the spatial cluster tolerance to differentiate individual clusters. The optimal value for provided configuration has been chosen, but it is also possible to dynamically change it.

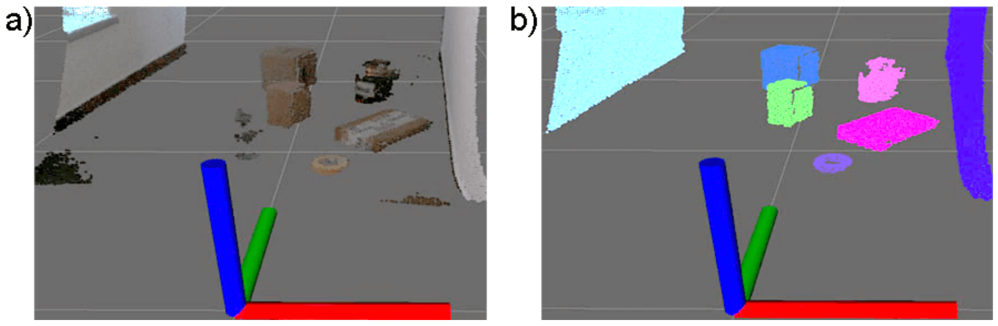


Figure. 28) Extraction of clusters example: a) an input cloud before applying extraction; b) an input cloud after applying extraction and filtering out invalid clusters.

5.4.5 Cluster preprocessing

The extraction of clusters returns a vector of **PointIndices** objects, where each object represents an individual cluster. Thus, a loop through this vector is used to process each cluster. Firstly, cluster preprocessing is applied inside this loop, which is done by the following steps:

1. The second loop is initialized to loop through individual points inside the **PointIndices** object to populate cluster's point cloud with points. Also, two counters (**pointsCount, pointCountHeight**) are incremented inside this loop.
2. The cluster is checked against the wall-like condition, and two counters are used to filter out invalid clusters: **pointsCount** is used to remove remaining ground plane's pieces, **pointsCountHeight** is used to remove levitating clusters. This is implemented as illustrated in the pseudocode below:

```
$IF\ size\ of\ a\ cluster > 12000$
   $SET\ isWallLike\ to\ true$
$ELSE$
   $IF\ \frac{pointsCount}{size\ of\ a\ cluster} < 0{,}25\ AND\ \frac{pointsCountHeight}{size\ of\ a\ cluster} > 0{,}025$
      $remove\ outliers\ from\ a\ cluster$
      $estimate\ normals\ of\ a\ cluster$
      $copy\ cluster's\ point\ cloud\ and\ normal's\ point\ cloud$
      $IF\ size\ of\ a\ cluster > 2500$
         $segment\ the\ largest\ plane\ in\ cluster's\ point\ cloud$
         $IF\ \frac{size\ of\ segmented\ plane}{size\ of\ a\ cluster} > 0{,}65\ AND\ segmented\ plane\ \perp XY\ plane$
            $SET\ isWallLike\ to\ true$
         $END\ IF$
      $END\ IF$
   $ELSE$
      $CONTINUE$
   $END\ IF$
$END\ IF$
```

5.4.6 Cluster processing

As has been mentioned before, a cluster has been tested against several conditions to determine whether it represents **wall-like object** i.e. huge object with the prevailing frontal plane, or another kind of object, or invalid cluster. Every invalid cluster skips a body of the loop, but in case of a valid cluster following steps are executed:

1. Cluster's approximate dimensions are estimated by utilizing custom method **findMinMax3D()**. This method returns a vector of size 6, first three values of which represent minimum x, y, z coordinates of points found in an input data set and last three values represent maximum x, y, z coordinates. Its implementation is represented in pseudocode provided below:

```
$create\ vector\ data\ of\ size\ 6$
$SET\ data[0]\ AND\ data[1]\ AND\ data[2]\ to\ 1000$
$SET\ data[3]\ AND\ data[4]\ AND\ data[5]\ to - 1000$
$FOR\ each\ point\ \boldsymbol{p}\ in\ input\ point\ cloud$
   $IF\ p.x < data[0]$
      $data[0] = p.x$
   $IF\ p.y < data[1]$
      $data[1] = p.y$
   $IF\ p.z < data[0]$
      $data[2] = p.z$
   $IF\ p.x < data[0]$
      $data[3] = p.x$
   $IF\ p.y < data[0]$
      $data[4] = p.y$
   $IF\ p.z < data[5]$
      $data[5] = p.z$
   $END\ IF$
$END\ FOR$
$return\ data$
```

2. The value of the **isWallLike** variable is checked, and if it equals true, the execution skips to **cluster postprocessing** step.
3. In case **isWallLike** is false, the algorithm firstly searches for a top plane in a cluster as illustrated in a pseudocode below:

```
$DO$
   $segment\ the\ largest\ plane\ in\ cluster's\ point\ cloud\ using\ normals\ point\ cloud$
   $IF\ \frac{size\ of\ segmented\ plane}{size\ of\ a\ cluster} > 0{,}05$
      $estimate\ approximate\ distance\ \boldsymbol{d}\ between\ segmented\ plane\ and\ ground\ plane$
      $IF\ CALL\ isTopParallelPlane\ with\ ground\ plane's\ coefficients\ AND$
            $segmented\ plane's\ coefficients\ AND\ \boldsymbol{d}\ AND\ z\ dimension\ of\ a\ cluster$
      $SET\ hasPlane\ to\ false$
      $SET\ hasParallelPlaneToFloor\ to\ true$
    $ELSE$
      $remove\ segmented\ plane\ from\ cluster's\ point\ cloud$
      $remove\ segmented\ normals\ from\ normals\ point\ cloud$
    $END\ IF$
  $ELSE$
    $SET\ hasPlane\ to\ false$
  $END\ IF$
$WHILE\ hasPlane$
```

The **isTopParallelPlane()** method returns true if two planes defined by **ModelCoefficients** objects are parallel, and a distance between these planes is bigger than a certain threshold. Its implementation is represented as pseudocode provided below:

```
$DEFINE\ normal\ vector\ of\ plane\ 1\ using\ ground\ plane's\ model\ coefficients$
$DEFINE\ normal\ vector\ of\ plane\ 2\ using\ segmented\ plane's\ model\ coefficients$
$calculate\ angle\ between\ these\ vectors\ (2)$
$\text{SET result to false}$
$\text{IF (|angle|} < \text{tolerance OR |angle} - 180| < \text{tolerance)}$
  $result = true$
  $IF\ \frac{approximate\ distance\ \boldsymbol{d}\ between\ segmented\ plane\ and\ ground\ plane}{z\ dimension\ of\ a\ cluster} < 0,5$
    $\text{result} = \text{false}$
  $\text{END IF}$
$\text{END IF}$
$RETURN\ result$
```

An angle between two normal vectors is calculated by the equation, provided below:

$$angle = \operatorname{acos}(v1 \cdot v2) \tag{2}$$

Where $v1, v2$ – normalized vectors so that $|v1| = |v2| = 1$; $v1 \cdot v2$ – dot product of vectors.

4. If a top plane has been found, **projectAndProcess()** method is called, it’s implementation is described by following steps:
   a) The top plane is projected onto a ground plane (see 3.3.7). The original top plane’s point cloud is shown in Figure. 29) a, and its projection is shown in Figure. 29) b;
   b) Top plane’s projection is rotated so that its normal is collinear with z coordinate axis as shown in Figure. 29) c;
   c) Convex hull polygon (see 3.3.7) is reconstructed from transformed top plane’s projection as shown in Figure. 29) d;

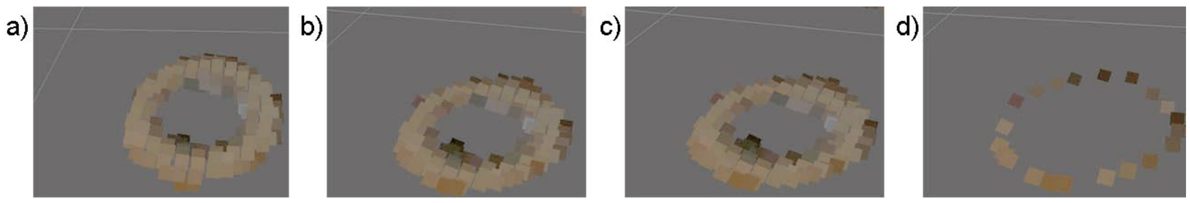


Figure. 29) Processing of a top plane

d) Rotating Calipers algorithm is a perfect tool to determine oriented minimum bounding box (OMBB) on the 2D surface. OMBB is used by the algorithm to estimate geometric dimensions of a top plane (width and length). For every point P in convex hull do [43]:

- Find an angle φ between x coordinate axis and a line crossing current point P_i and next point P_{i+1};
- Rotate the convex hull by φ;
- Find maximum and minimum values of x and y coordinates among all P in the transformed coordinate frame;
- Calculate x and y dimensions of the convex hull in the rotated coordinate frame, and a rectangular area a from these dimensions:

$$x_{dimension} = |x_{max} - x_{min}| \tag{3}$$

$$y_{dimension} = |y_{max} - y_{min}| \tag{4}$$

$$a = x_{dimension} \cdot y_{dimension} \tag{5}$$

- If a is less than **min_area** variable: assign width and length of an object with $x_{dimension}$ and $y_{dimension}$ (width with the smallest one), and assign **min_area** with a;
- If **hasTopPlaneParallel** is set to true, calculate a vector indicating a position of object's centroid, and apply a reverse rotation to it.

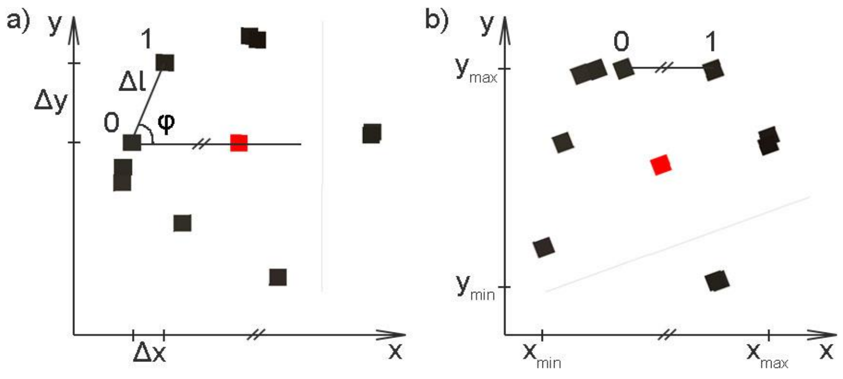


Figure. 30) Demonstration of convex hull processing: a) state before rotation of convex hull; b) state after rotation (red dot is an estimated centroid).

e) Define a circle with radius r:

$$r = \frac{x_{dimension} + y_{dimension}}{4} \tag{5}$$

f) For every point P in convex hull do:

- Calculate Euclidean distance d between point P_i and the centroid of a convex hull;

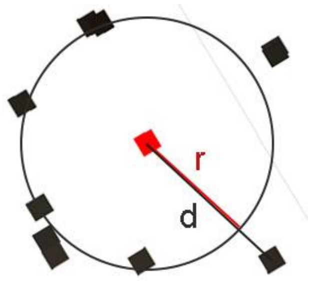


Figure. 31) Determination of a circular convex hull

- Calculate a deviation δ between circlular and convex hull shapes (see Figure. 31) and summarize deviations:

$$\delta = |r - d| \tag{6}$$

g) Calculate an average deviation and if it is lower than a certain threshold, set the **isCircle** flag to true;

h) Return a vector of a size 2, which contains width and length of an object.

5. The centroid is transformed back from transformed coordinate frame to original Kinect's coordinate frame by applying reversed translation and rotation respectively. After that, if flag **transform_to_base_link** is set to **true**, the centroid is transformed from Kinect's coordinate frame to **base_link** coordinate frame by applying **transformToBaseLink()** method.
6. If flag **isCircle** is set to true, apply additional filtering to filter out incorrectly detected cylinders. This algorithm is represented by the pseudocode below:

```
$calculate\ area\_ratio = \frac{area\ of\ a\ top\ plane}{total\ area\ of\ a\ box\ shape}$
$DO$
  $segment\ the\ largest\ plane\ in\ cluster's\ point\ cloud\ using\ normals\ point\ cloud$
  $calculate\ ratio = \frac{size\ of\ a\ segmented\ plane}{size\ of\ a\ cluster}$
  $IF\ ratio > 0{,}15 * area\_ratio$
    $DEFINE\ normal\ vector\ of\ a\ segmented\ plane$
    $IF\ CALL\ isPerpendicularToPlane\ with\ false\ AND\ normal\ vector\ AND$
        $z\ axis\ vector\ AND\ frontAngle$
      $increment\ count$
      $increment\ ratio\_med\ by\ ratio$
    $END\ IF$
```

```
        $remove\ segmented\ plane\ from\ cluster's\ point\ cloud$
        $remove\ segmented\ normals\ from\ normals\ point\ cloud$
    $ELSE$
        $SET\ hasPlane\ to\ false$
    $END\ IF$
$WHILE\ hasPlane$
$ratio\_med = \frac{ratio\_med}{count \cdot area\_ratio}$
$IF\ (ratio\_med > 0)\ AND\ (ratio\_med < 0{,}45)$
    $SET\ isCylinder\ to\ true$
$ELSE$
    $SET\ isCylinder\ to\ false$
$END\ IF$
```

Method **isPerpendicularToPlane()** return true if two planes defined by normal vectors are perpendicular to each other. An implementation of this method is represented by pseudocode below:

```
$DEFINE\ normal\ vector\ of\ plane\ 1\ using\ model\ coefficients$
$DEFINE\ normal\ vector\ of\ plane\ 2\ using\ model\ coefficients$
$calculate\ angle\ between\ these\ vectors\ (2)$
$IF\ (|angle| < 90 + tolerance)\ AND\ (|angle| > 90 - tolerance)$
    $return\ true$
$END\ IF$
$return\ false$
```

5.4.7 Cluster postprocessing

Implemented algorithm differentiates all obstacles into several groups: wall-like objects (ones having prevailing plane perpendicular to the ground), box-like objects, cylinder-like objects and objects of undefined shape. Differentiation itself has been done at cluster processing step. At the postprocessing step, each obstacle's cluster is processed based on its type:

- **Cylinder-like obstacle**: radius of the obstacle is calculated and message object is initialized with radius, height and centroid values. This message is then published on the respective ROS topic **/cylinder_data** and is ready to be processed further.
- **Box-like obstacle**: an algorithm firstly searches for obstacle's frontal plane (plane perpendicular to the ground). The implementation is represented by pseudocode below:

```
$DO$
    $segment\ the\ largest\ plane\ in\ cluster's\ point\ cloud\ using\ normals\ point\ cloud$
    $IF\ \frac{size\ of\ a\ segmented\ plane}{size\ of\ a\ cluster} > 0{,}1$
        $DEFINE\ normal\ vector\ of\ a\ segmented\ plane$
        $IF\ CALL\ isPerpendicularToPlane\ with\ false\ AND\ normal\ vector\ AND$
```

```
          $z\ axis\ vector\ AND\ frontAngle$
        $SET\ hasPlane\ to\ false$
      $ELSE$
        $remove\ segmented\ plane\ from\ cluster's\ point\ cloud$
        $remove\ segmented\ normals\ from\ normals\ point\ cloud$
      $END\ IF$
    $ELSE$
      $SET\ hasPlane\ to\ false$
    $END\ IF$
  $WHILE\ hasPlane$
```

At this step, a frontal angle is estimated (the smallest angle between robot's x coordinate and normal vector of a detected frontal plane). The message object is initialized with length, width, height, centroid and frontal angle values. This message is then published on the respective ROS topic **/box_data** and is ready to be processed further.

- **Wall-like obstacle**: obstacle dimensions are estimated by using the **findMinMax3D()** method. Length of an obstacle is calculated based on the value of its frontal angle and centroid is estimated by using built-in PCL method **computeCentroid()**. After that, a message object is initialized with length, width, height, centroid and frontal angle. This message is published on the respective ROS topic **/wall_like_object_data** and is ready to be processed further.
- **Undefined obstacle**: an obstacles profile is defined as its x and z dimensions. After that cluster's cloud is projected onto a ground plane, and length and width of a cluster are approximated by utilizing the **projectAndProcess()** method. It is worth to mention that these length and width values are only approximations due to uncertainty about obstacle's shape. Centroid is calculated by utilizing built-in PCL method **computeCentroid()**. If the **isCylinder** flag is false then a message object is initialized with a profile, length, width, and centroid, and is subsequently published on the respective ROS topic **/undefined_data**, otherwise, it assumes the obstacle has a cylindrical shape and the step for a cylinder-like obstacle is done (see above).

5.5 Launch files, dynamic reconfiguration, and debug options

The proposed system has been implemented with a concentration on its flexibility. One of the main features that allow flexibility is utilizing launch files and dynamic reconfiguration.

As has been mentioned above, launch files are extremely useful from several perspectives:

- They allow to run several ROS nodes with one shell command;
- They make implemented solution much cleaner and easier to run, especially for other researchers that are not familiar with it;

- Command line arguments allow to run the system with different configuration setups;
- Launch files support ROS parameters for so-called dynamic reconfiguration (ROS parameters set up in launch file can be used inside ROS node and be changed dynamically during runtime).

Launch files can be nested inside each other which prevents code duplication. There are 2 standalone launch files to run **calibration** and **obstacle_processor** nodes independently. The **obstacle_processor.launch** file contains a lot of input arguments, their values are then assigned to respective ROS parameters. The purpose of each ROS parameter is listed in Appendix 1 c.

Two boolean ROS parameters (**cmd_out** and **rviz_out**) are used for special use cases, as they activate debug options: command line and graphical ones.

6 EXPERIMENTS AND DISCUSSION

Every complex solution requires quality tests or experiments to be conducted. Thus, the implemented system has been tested to ensure its robustness, to analyze how accurately it operates and more importantly to identify its weaknesses and propose possible modifications to the system.

As has been mentioned earlier, the system has been tested on a Breach robotic platform and all conducted experiments have been split into three groups:

- **Static experiments** – a static scene was observed by Kinect v2 i.e. robotic platform didn't move during an experiment;
- **Dynamic experiments** – a dynamic scene was observed by Kinect v2 i.e. robotic platform was moving during an experiment;
- **Performance experiment** – a static scene was observed by Kinect v2 to analyze the performance of the implemented solution;

This chapter provides a description and an evaluation of conducted experiments. At the end of a chapter, discussion about the implemented system is given, which evaluates the system from all possible sides: performance, advantages, shortcomings, application and further development possibilities.

6.1 Static experiments

These experiments have been conducted to evaluate the overall precision of implemented solution and to determine main factors that influence the precision of measurement. Three static experiments have been conducted:

1. The static scene with 9 objects (cylinders, boxes, and objects of undefined shape) was observed by Kinect camera as shown in Figure. 32).

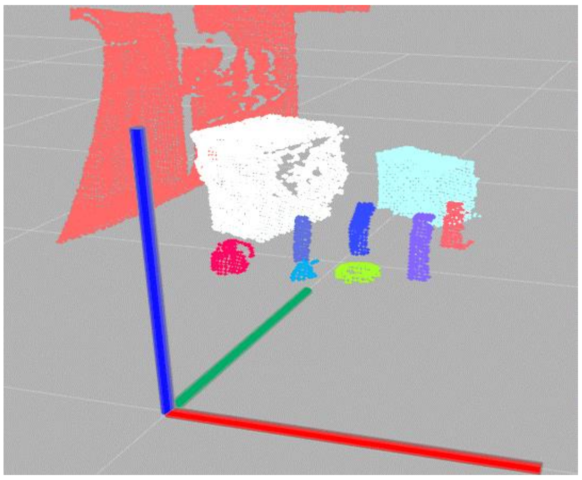


Figure. 32) First static experiment's scene

The camera's decline angle was changed before each experiment's phase, while the distance between camera and objects was being kept constant. The results of the conducted experiment are demonstrated in a form of tables and diagrams in Appendix 3 a.

2. The static scene with one box was observed by Kinect camera as shown in Figure. 33). The box was rotated between each experiment's phase, while the distance between Kinect camera and box was being kept constant. This experiment was conducted to determine an influence of box rotation on the precision of measured box's dimensions. The results of the conducted experiment are demonstrated in a form of tables and diagrams in Appendix 3 b.

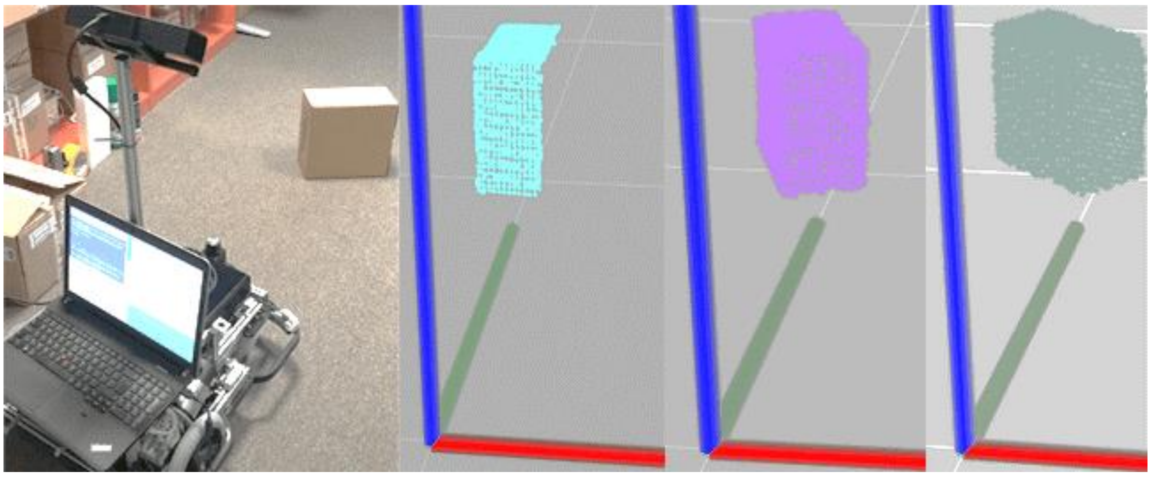


Figure. 33) Second static experiment's scene

3. The static scene with a single object was observed by Kinect camera as shown in Figure. 34). This object was chosen from a set of 4 boxes and 4 cylinders and being measured at 4 different distances from the Kinect camera. The experiment was conducted to determine an influence of object's distance from the camera on measured dimensions and also to compare the precision of measurements between boxes and cylinders. The results of the conducted experiment are demonstrated in a form of tables and diagrams in Appendix 3 c.

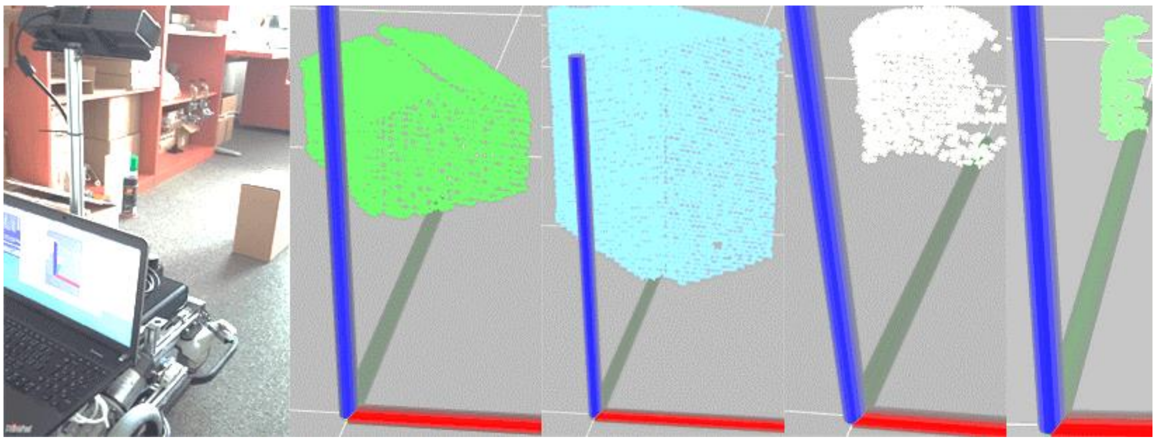


Figure. 34) Third static experiment's scene

The results of static experiments are analyzed and provided below in a structured way:

- The accuracy of measurement is influenced by decline angle of the camera: increasing decline angle results in a slightly better precision of measurement. However, bigger decline angle of camera restricts the maximum distance at which it is possible to properly detect obstacles. That is why it is recommended to use a preferable value (around 25°) of decline angle;
- Obstacles of box shape are measured more accurately comparing to ones of cylindrical shape;
- Rotation of box-like obstacle relatively to camera slightly influences the overall precision of measurement. It was empirically found that slightly more accurate measurements were obtained, when an angle between camera and normal vector of smaller obstacle's plane is bigger that one of the larger plane;
- The distance between the camera and an obstacle influences precision of measured dimensions: more mistakes in measurements have been observed for distant obstacles. Moreover, obstacles of cylindrical shape and box shape are prone to be determined as undefined shape at longer distances, due to the inability of the camera to properly detect top plane of an obstacle as shown in Figure. 35);

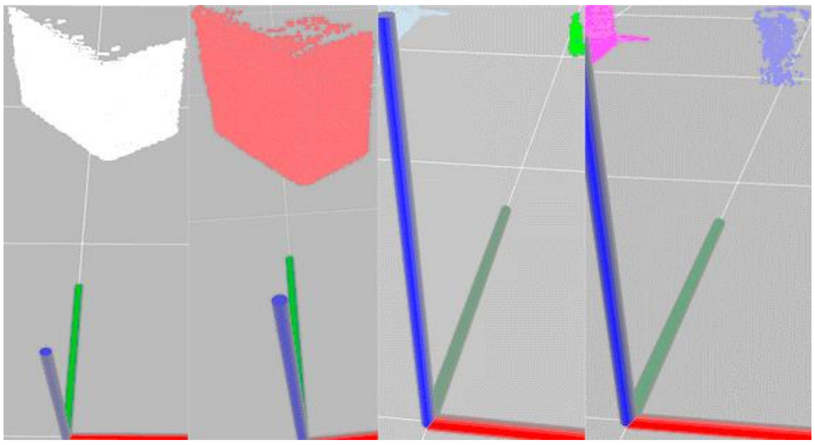


Figure. 35) Improper detection of top plane of an obstacle

- Larger objects are measured less accurately than smaller ones;
- The coordinates of centroid are measured by 2,5 % more accurately in case of obstacles of cylindrical shape comparing to ones of box shape.

6.2 Dynamic experiments

These experiments have been conducted to evaluate the precision of an implemented solution while the robotic platform is moving. A scene with 4 boxes and 3 cylinders was observed by Microsoft Kinect while the robotic platform was moving as shown in Figure. 36). The precision of estimated obstacles' dimensions was evaluated at 3 different speeds of the robotic platform. The results of conducted dynamic experiments are demonstrated in form of tables and diagrams in Appendix 3 d.

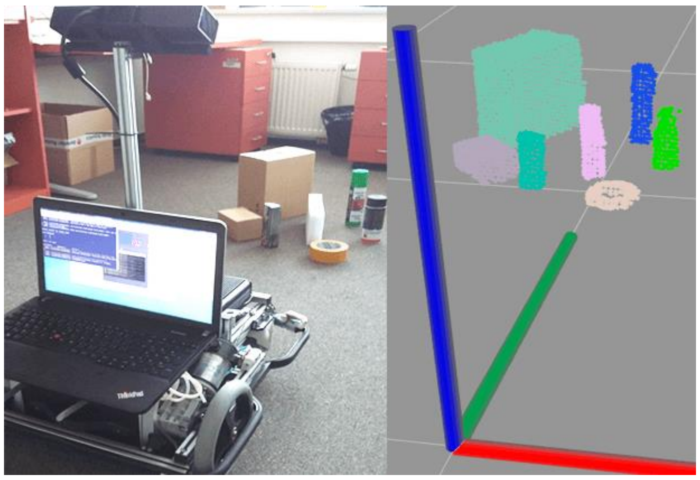


Figure. 36) Dynamic experiments‘ scene

The results of conducted dynamic experiments are analyzed and provided below in a structured way:

- The overall precision of estimated dimensions is higher for obstacles of box shape comparing to ones of cylindrical shape, however, this difference is not significant (about 2-2,5%);
- The precision of estimated dimensions decreases with an increase of robotic platform’s velocity, hence to meet precision demands it is recommended to use smaller velocities (0,1 – 0,3 m/s);
- It is recommended to use implemented solution for optimal velocity range (0,1 – 0,2 m/s), because at higher velocities there is a delay between provided measurements and robot position, due to time consumption of program’s cycle. As a result, some measurements appear to be outdated.

6.3 Performance experiment

The implemented solution must be sufficient in terms of performance for its dynamic applications, hence performance experiments were conducted to evaluate it. The performance of algorithm strictly depends on a specification of PC, on which the algorithm was launched, therefore the specification is provided in Appendix 3 e for the sake of completeness.

A static scene with single object was observed by Kinect camera. This object was chosen from a set of 5 boxes, 5 cylinders and 5 undefined shapes. Time needed to process cloud before applying clustering algoritm was calculated, as well as time needed to process the obstacle’s cluster. Then the overall processing time was calculated. The results of conducted experiment are provided in a form of table in Appendix 3 f.

The results of performance experiment are analyzed and provided below in a structured way:

- Even though, the implemented solution was tested on a lapton with relatively weak configuration, it was capable to process scene with one large object in less than 600 ms;

- There are fluctuations in calculated processing time, due to unpredictable task allocation by CPU and its changeable load;
- The processing time is increasing with the size of an obstacle, and most importantly with the size of an input point cloud;
- The preprocessing of input point cloud before actual clustering algorithm consumes the greater part of overall processing time;
- Obstacles of box shape require the least time to be processed, because the top plane is detected faster;
- Obstacles of undefined shape require the most time to be processed, because the top plane is not detected at all (more iterations are conducted to search for a top plane);

6.4 Discussion

This subsection deals with an evaluation of an implemented system from all relevant sides, hence the subsection is divided into several parts and each of them evaluates the solution from a respective side.

6.4.1 Performance

The implemented solution strictly depends on a complexity of an observed scene, i.e. more obstacles scene contains, more processing time is required. Moreover, as was stated in 6.3, the size of individual obstacles and the complexity of their shape are also relevant, hence larger obstacles and ones of undefined shape require more time to be processed, which increases overall processing time. Also, the processing time of the program depends on the hardware of the computer, on which the program runs.

Thus, if it is required to use the program for dynamic application, test runs must be conducted to determine whether the program is capable to process all obstacles in a scene in a required amount of time, which is given by velocity of the robotic platform. And if the program cannot provide actual information about obstacles, the system has to be used statically for such use case.

6.4.2 Application

The implemented solution has been developed for the purpose of being flexible, i.e. it can be launched under different configurations of the robotic platform. Also, it supports any depth sensors that are compatible with ROS environment, i.e. such sensors have ROS compatible driver and they produce point cloud ROS topic as an output.

There are two main applications of the implemented solution:

1. Application for static robotic platforms;
2. Application for dynamic robotic platforms.

In the first case, for example, the system can be used for object detection and recognition. Obtained information about objects can be used as an input for manipulation or statistical purposes.

However, it is needed to say that the application for dynamic purposes is limited to small velocities due to performance restrictions of the program as has been mentioned in 6.4.1. Moreover, the implemented solution only provides detection and recognition functionalities and does not deal with navigation and mapping tasks. This means, that for dynamic purposes the implemented system has to be used along with a configured package that deals with navigation and mapping tasks (for example AMCL (Autonomous Navigation and Mapping) package), to allow secure movement without any collisions.

6.4.3 Advantages

This section will provide main advantages of the implemented system based on thorough analysis:

1. System's flexibility (see 6.4.2);
2. Modularity of the solution, due to ROS nature;
3. Extensibility of **obstacle_processor** node. Main logic lies inside **ObstacleProcessor** class, which can be easily inherited from to extend the functionality of the system;
4. Various launch configurations allowing different output options via command line or RVIZ;
5. Support of ROS bag utility for testing purposes and for an offline run without capturing device;
6. Dynamic reconfiguration support allowing easily change input ROS parameters during runtime.

6.4.4 Shortcomings

This section will provide main disadvantages of the implemented system based on thorough analysis:

1. The processing time of the system strictly depends on the complexity of an observed scene (number of obstacles and their dimensions);
2. Inability to provide actual information at higher velocities of a robotic platform, due to performance restrictions of the system;
3. Dependence on a detection of top plane of an obstacle, which means that too large obstacles will always be considered as undefined or wall-like ones;
4. Inability to properly detect top plane of an obstacle at higher distances due to decline angle of a capturing device;

6.4.5 Further development

Even though, the implemented system is a complete and functional software, it has to be developed further to extend its functionality, to increase the efficiency of the algorithm and to support various use cases.

Possible sides of the program that may require modifications are listed below:

1. Integration of machine learning techniques to improve recognition ability of the program;
2. Concurrent processing of each cluster in an input point cloud to shorten the processing time of the algorithm;
3. Transformation of found centroids' coordinates to the global coordinate system (map's coordinate system);
4. Database support integration, which will allow saving all detected obstacles to the database, and at the same time skip already detected obstacles using modification from point 3;

The implemented system is inspired by ROS collaborative approach, hence it is an open source software and has a repository on GitHub [44], where every concerned person is able to contribute to the further development of the project.

7 CONCLUSION

The main goal of the bachelor thesis was to propose a design of an obstacle detection system. Firstly, the suitable image-based 3D scanner had to selected with respect to driver libraries, and as a result, Microsoft Kinect v2 was selected as a capturing device to the implemented system. Secondly, functionality design had to be done including a selection of image processing methods and followed by an implementation of the system. The system was implemented under ROS environment as ROS package and it utilized PCL library for image processing.

The thesis provided the research of related works, which also included an overview of related bachelor theses that were published at the Brno University of Technology. Subsequently, the necessary theoretical information related to the program implementation was provided. After that, an implementation of the whole system was thoroughly described step by step. In the end, conducted experiments were evaluated and the whole system was analyzed from all possible sides to determine its advantages, shortcomings, to identify possible areas of application, and to propose possible modifications that can be done for the further development of the system.

The result of this bachelor thesis is a complete and functional software solution for obstacle detection and recognition problem. The system was tested on Breach robotic platform to evaluate its precision and to determine the correctness of its function. Both static and dynamic experiments were conducted to analyze the behavior of the program under different conditions.

Based on the results of experiments and an analysis of implemented solution, it can be said that the implemented system functions properly and is precise enough to provide valid output. It is an open source software and is open on GitHub [44] for contributions from all concerned people for its further development.

8 BIBLIOGRAPHY

[1] ANONYMOUS. Industrial Robotics Market Growth is Driven by Electronics Manufacturing. In: *Robotic Industries Association* [online]. 2017 [cit. 2018-02-28]. Available from: https://www.robotics.org/blog-article.cfm/Industrial-Robotics-Market-Growth-is-Driven-by-Electronics-Manufacturing/70

[2] DAVIES, Alex. This palm-sized laser could make self-driving cars way cheaper. In: *Wired* [online]. 2014 [cit. 2018-03-06]. Available from: https://www.wired.com/2014/09/velodyne-lidar-self-driving-cars/

[3] MUNARO, Matteo, Radu B. RUSU and Emanuele MENEGATTI. 3D robot perception with Point Cloud Library. In: *Robotics and Autonomous Systems* [online]. Elsevier B.V, 2016, 78, 97-99 [cit. 2018-02-28]. DOI: 10.1016/j.robot.2015.12.008. ISSN 0921-8890. Available from: https://www-sciencedirect-com.ezproxy.lib.vutbr.cz/science/article/pii/S0921889015003176

[4] WIEDEMEYER, Thiemo. *IAI Kinect2* [online]. Institute for Artificial Intelligence, University Bremen, 2014 – 2015, accessed on 2018-03-6. Available from: https://github.com/code-iai/iai_kinect2

[5] HINTERSTOISSER, Stefan, Vincent LEPETIT, Slobodan ILIC, Pascal FUA and Nassir NAVAB. Dominant orientation templates for real-time detection of texture-less objects. In: *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference* [online]. IEEE Publishing, 2010, p. 2257-2264 [cit. 2018-03-06]. DOI: 10.1109/CVPR.2010.5539908. ISBN 978-1-4244-6984-0. ISSN 1063-6919. Available from: http://ieeexplore.ieee.org/document/5539908/

[6] RUSU, Radu Bogdan and Steve COUSINS. 3D is here: Point Cloud Library (PCL). In: *Robotics and Automation (ICRA), 2011 IEEE International Conference* [online]. IEEE Publishing, 2011, p. 1-4 [cit. 2018-03-06]. DOI: 10.1109/ICRA.2011.5980567. ISBN 978-1-61284-386-5. ISSN 1050-4729. Available from: http://ieeexplore.ieee.org/document/5980567/

[7] RAKPRAYOON, Panjawee, Miti RUCHANURUCKS and Ada COUNDOUL. Kinect-based obstacle detection for manipulator. In: *System Integration (SII), 2011 IEEE/SICE International Symposium* [online]. IEEE Publishing, 2011, p. 68-73 [cit. 2018-03-06]. DOI: 10.1109/SII.2011.6147421. ISBN 978-1-4577-1523-5. Available from: http://ieeexplore.ieee.org/document/6147421/

[8] JINWOOK, Choi, Kim DEUKHYEON, Yoo HUNJAE and Sohn KWANGHOON. Rear obstacle detection system based on depth from kinect. In: *Intelligent Transportation Systems (ITSC), 2012 15th International IEEE Conference* [online]. IEEE, 2012, p. 98-101 [cit. 2018-03-06]. DOI: 10.1109/ITSC.2012.6338794. ISBN 978-1-4673-3064-0. ISSN 2153-0009. Available from: http://ieeexplore.ieee.org/document/6338794/

[9] BOSTANCI, E., N. KANWAL and A. F. CLARK. Extracting planar features from Kinect sensor. In: *Computer Science and Electronic Engineering Conference (CEEC), 2012 4th* [online]. IEEE, 2012, p. 111-116 [cit. 2018-03-06]. DOI: 10.1109/CEEC.2012.6375388. ISBN 978-1-4673-2665-0. Available from: http://ieeexplore.ieee.org/document/6375388/

[10] HULIK, Rostislav, Michal SPANEL, Pavel SMRZ and Zdenek MATERNA. Continuous plane detection in point-cloud data based on 3D Hough Transform. *Journal*

of Visual Communication and Image Representation [online]. 2013 [cit. 2018-03-07]. DOI: 10.1016/j.jvcir.2013.04.001. ISSN 10473203. Available from: https://www-sciencedirect-com.ezproxy.lib.vutbr.cz/science/article/pii/S104732031300062X

[11] QIANG, Zhang, Kong LINGCHENG and Zhao JIANGHAI. Real-time general object recognition for indoor robot based on PCL. In: *Robotics and Biomimetics (ROBIO), 2013 IEEE International Conference* [online]. IEEE, 2013, p. 651-655 [cit. 2018-03-07]. DOI: 10.1109/ROBIO.2013.6739534. Available from: http://ieeexplore.ieee.org/document/6739534/

[12] YU, Hongshan, Jiang ZHU, Yaonan WANG, Wenyan JIA, Mingui SUN and Yandong TANG. Obstacle Classification and 3D Measurement in Unstructured Environments Based on ToF Cameras. *Sensors(Switzerland)* [online]. Basel: MDPI, 2014, 14(6), 10753-10782 [cit. 2018-03-07]. DOI: 10.3390/s140610753. ISSN 14248220. Available from: http://www.mdpi.com/1424-8220/14/6/10753

[13] FARID, Reza and Claude SAMMUT. Plane-based object categorisation using relational learning. *Machine Learning* [online]. New York: Springer US, 2014, 94(1), 3-23 [cit. 2018-03-07]. DOI: 10.1007/s10994-013-5352-9. ISSN 0885-6125. Available from: https://link-springer-com.ezproxy.lib.vutbr.cz/article/10.1007/s10994-013-5352-9

[14] MATURANA, D. and S. SCHERER. VoxNet: A 3D Convolutional Neural Network for real-time object recognition. In: *IEEE International Conference on Intelligent Robots and Systems* [online]. Institute of Electrical and Electronics Engineers, 2015, p. 922-928 [cit. 2018-03-07]. DOI: 10.1109/IROS.2015.7353481. ISBN 9781479999941. ISSN 21530858. Available from: http://ieeexplore.ieee.org/document/7353481/

[15] SONG, Shuran and Jianxiong XIAO. *Deep Sliding Shapes for Amodal 3D Object Detection in RGB-D Images* [online]. 2015 [cit. 2018-03-07]. Available from: https://arxiv.org/abs/1511.02300

[16] WANG, Anran, Jianfei CAI, Jiwen LU a Tat-Jen CHAM. MMSS: Multi-modal Sharable and Specific Feature Learning for RGB-D Object Recognition. In: *Computer Vision (ICCV), 2015 IEEE International Conference* [online]. IEEE, 2015, p. 1125-1133 [cit. 2018-03-07]. DOI: 10.1109/ICCV.2015.134. Available from: http://ieeexplore.ieee.org/document/7410491/

[17] MORRISON, D., A. W. TOW, M. MCTAGGART, et al. *Cartman: The low-cost Cartesian Manipulator that won the Amazon Robotics Challenge* [online]. 2017 [cit. 2018-03-07]. Available from: https://arxiv.org/pdf/1709.06283.pdf

[18] ANONYMOUS. Amazon Robotics Challenge 2017 won by Australian budget bot. In: *BBC News* [online]. 2017 [cit. 2018-03-07]. Available from: http://www.bbc.com/news/technology-40774385

[19] DOJAVA, Marian. *Detekce jízdních pruhů a překážek* [online]. Brno University of Technology. Faculty of Electrical Engineering and Communication, 2011 [cit. 2018-03-07]. 66 p. Available from: https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=38587

[20] NAJMAN, Jan. *Rozšíření robotu Car4 o palubní počítač a snímače Kinect a Hokuyo* [online]. Brno University of Technology. Faculty of Mechanical Engineering, 2013 [cit. 2018-03-07]. 41 p. Available from: https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=66207

[21] JANÁS, Lukáš. *Využití senzoru Kinect pro detekci osob* [online]. Brno University of Technology. Faculty of Electrical Engineering and Communication, 2013 [cit. 2018-03-

07]. 56 p. Available from: https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=69175

[22] STŘÍTESKÝ, Vladimír. *Bezkolizní navigace mobilního robotu* [online]. Brno University of Technology. Faculty of Electrical Engineering and Communication, 2015 [cit. 2018-03-07]. 77 p. Available from: https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=100272

[23] DAL MUTTO, Carlo, Pietro ZANUTTIGH, and Guido M CORTELAZZO. *Time-of-flight cameras and microsoft Kinect*. Springer Science & Business Media, 2012.

[24] LAU, Daniel. In: *The Science Behind Kinects or Kinect 1.0 versus 2.0* [online]. 2013-11-27 [cit. 2018-03-09]. Available from: https://www.gamasutra.com/blogs/DanielLau/20131127/205820/The_Science_Behind_Kinects_or_Kinect_10_versus_20.php

[25] SARBOLANDI, Hamed, Damien LEFLOCH and Andreas KOLB. Kinect range sensing: Structured-light versus Time-of-Flight Kinect. *Computer Vision and Image Understanding* [online]. Elsevier, 2015, 139, 1-20 [cit. 2018-03-09]. DOI: 10.1016/j.cviu.2015.05.006. ISSN 1077-3142. Available from: https://www-sciencedirect-com.ezproxy.lib.vutbr.cz/science/article/pii/S1077314215001071

[26] ZANUTTIGH, P. et al. *Time-of-Flight and Structured Light Depth Cameras*. Springer International Publishing Switzerland, 2016.

[27] HANSARD, Miles, Seungkyu LEE, Ouk CHOI and Radu HORAUD. *Time of Flight Cameras: Principles, Methods, and Applications*. Springer, pp.95, 2012, SpringerBriefs in Computer Science, ISBN 978-1-4471-4658-2.

[28] BECERIK-GERBER, Burcin and Sami F. MASRI. *An Inexpensive Vision-Based Approach for the Autonomous Detection, Localization, and Quantification of Pavement Defects* [online]. University of Southern California, 2015. Available from: http://onlinepubs.trb.org/onlinepubs/IDEA/FinalReports/Highway/NCHRP169_Final_Report.pdf

[29] ROMERO, Marco. In: *Kinect V2 Motion Sensor* [online]. 2015-06-30 [cit. 2018-03-09]. Available from: http://maromero3d.blogspot.cz/2015/06/kinect-v2-motion-sensor.html

[30] MAHTANI, Anil, Luis SANCHEZ, Enrique FERNANDEZ and Aaron MARTINEZ. *Effective Robotics Programming with ROS - Third Edition*. Birmingham: Packt Publishing, 2016.

[31] *Documentation – Point Cloud Library (PCL)*. [online]. [cit. 2018-03-10]. Available from: http://pointclouds.org/documentation/

[32] *PCL API Documentation*. [online]. [cit. 2018-03-10]. Available from: http://docs.pointclouds.org/trunk/index.html

[33] FISCHLER, Martin and Robert BOLLES. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM* [online]. ACM, 1981, 24(6), 381-395 [cit. 2018-03-11]. DOI: 10.1145/358669.358692. ISSN 0001-0782. Available from: https://dl-acm-org.ezproxy.lib.vutbr.cz/citation.cfm?doid=358669.358692

[34] *Documentation – Robot Operating System*. [online]. [cit. 2018-03-13]. Available from: http://wiki.ros.org/

[35] QUIGLEY, Morgan, Brian GERKEY and William D SMART. *Programming Robots with ROS: A Practical Introduction to the Robot Operating System*. O'Reilly Media, 2015. ISBN 9781449323899.

[36] GOSSOW, David, Chad ROCKEY, Kei OKADA, Julius KAMMERL and Acorn POOLEY. *ROS 3D Robot Visualizer* [online]. Accessed on 2018-03-13. Available from: https://github.com/ros-visualization/rviz

[37] ANONYMOUS. *ROS Qt Creator Plug-in* [online]. Accessed on 2018-03-13. Available from: https://github.com/ros-industrial/ros_qtc_plugin

[38] ZENNARO, S., M. MUNARO, S. MILANI, P. ZANUTTIGH, A. BERNARDI, S. GHIDONI and E. MENEGATTI. Performance evaluation of the 1st and 2nd generation Kinect for multimedia applications. In: *Multimedia and Expo (ICME), 2015 IEEE International Conference* [online]. IEEE, 2015, s. 1-6 [cit. 2018-03-14]. DOI: 10.1109/ICME.2015.7177380. Available from: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7177380

[39] VALGMA, Lembit. *3D reconstruction using Kinect v2 camera* [online]. University of Tartu. Faculty of Science and Technology, 2016 [cit. 2018-03-14]. 42 p. Available from: https://www.tuit.ut.ee/sites/default/files/tuit/atprog-courses-bakalaureuset55-loti.05.029-lembit-valgma-text-20160520.pdf

[40] SMEENK, Roland. *Kinect V1 and Kinect V2 fields of view compared* [online]. 2014-03-11 [cit. 2018-03-14]. Available from: https://smeenk.com/kinect-field-of-view-comparison/

[41] XIANG, Lingzhu, Florian ECHTLER, Christian KERL and Joshua BLAKE. *libfreenect2* [online]. Accessed on 2018-03-14. Available from: https://github.com/OpenKinect/libfreenect2

[42] *BREACH mobile robotic platform*. [online]. [cit. 2018-03-23]. Available from: http://www.benderrobotics.com/breach.html

[43] FREEMAN, H and R SHAPIRA. *Determining the minimum-area encasing rectangle for an arbitrary closed curve. Communications of the ACM* [online]. ACM, 1975, 18(7), 409-413 [cit. 2018-04-28]. DOI: 10.1145/360881.360919. ISSN 0001-0782. Available from: https://dl-acm-org.ezproxy.lib.vutbr.cz/citation.cfm?doid=360881.360919

[44] RYBIN A. *Obstacle_processor ROS package* [online]. Accessed on 2018-05-20. Available from: https://github.com/dragoon000320/obstacle_processor

9 LIST OF FIGURES

10 LIST OF ABBREVIATIONS AND SYMBOLS

2D	Two-Dimensional
3D	Three-Dimensional
AMCL	Autonomous Navigation and Mapping
ARC	Amazon Robotics Challenge
CNN	Convolutional Neural Networks
CPU	Central Processing Unit
CVFH	Clustered Viewpoint Feature Histogram
CW	Continuous Wave
DOT	Dominant Orientation Templates
FOV	Field of View
GUI	Graphical User Interface
I/O	Input/Output
IR	Infrared
NIR	Near Infrared
PC	Personal Computer
PCL	Point Cloud Library
RANSAC	Random Sample Consensus
RGB-D	Red Green Blue color model with depth
ROS	Robot Operating System
RVM	Relevance Vector Machine
ToF	Time-of-Flight
XML	Extensible Markup Language

$\delta\ [m]$	Deviation between circular and convex hull shapes
$angle\ [rad]$	Distance
$c\ [m/s]$	Speed of light
$d\ [m]$	Distance
$r\ [m]$	Radius

$t_d\ [rad]$	Phase shift
$x_{dimension}\ [m]$	X dimension of convex hull
$x_{max}\ [m]$	Maximum x coordinate of point in a convex hull
$x_{min}\ [m]$	Minimum x coordinate of point in a convex hull
$y_{dimension}\ [m]$	Y dimension of convex hull
$y_{max}\ [m]$	Maximum y coordinate of point in a convex hull
$y_{min}\ [m]$	Minimum y coordinate of point in a convex hull

11 LIST OF APPENDICES

Appendix 1: Robotic mobile platform;

Appendix 2: Schematic structure of nodes;

Appendix 3: The data of conducted experiments;

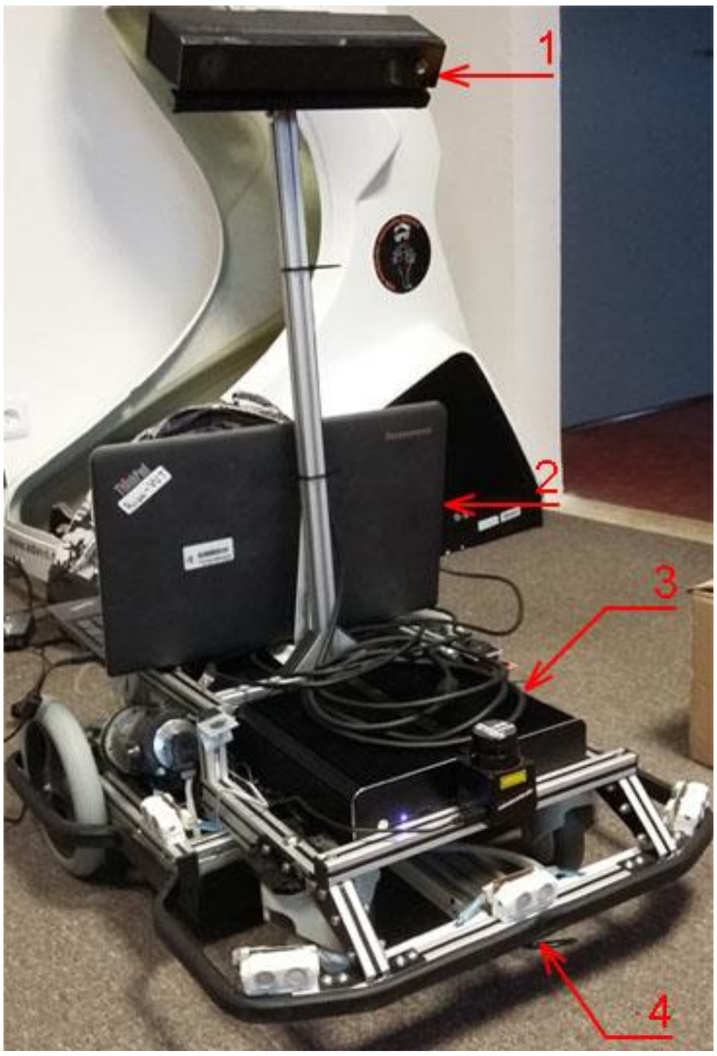
CD with source code of implemented solution in a form of catkin workspace, which includes two packages (**iai_kinect2** and **obstacle_processor**) under the **src** directory.

APPENDIX 1: ROBOTIC MOBILE PLATFORM

a) Specifications of Breach mobile platform[42].

Overall dimensions (L x W x H)	620 x 540 x 255 mm
Platform weight	32 kg
Maximum payload	20 kg
Operating speed	1.5 kph
Operating range	7.4 km
Battery run time	8 h
Environment type	Indoor
Chassis type	2WD, differential

b) Demonstration of an extended Breach platform with Microsoft Kinect v2: 1 – Microsoft Kinect v2; 2 – a laptop, which operates Microsoft Kinect v2 and is connected via Wi-Fi with on-board computer (ROS master); 3 – On-board computer with configured embedded control system; 4 – Breach robotic platform



c) Launch files, launch options and ROS parameters.

The implemented solution is run by typing a sequence of shell commands into terminal. In case of used Breach mobile platform with Microsoft Kinect as a capturing device, the sequence of those commands looks as follows:

1. Catkin workspace’s file **setup.bash** is sourced to make **obstacle_processor** and **kinect2_bridge** packages visible for ROS:

```
$source catkin_3ws/devel/setup.bash
```

2. Microsoft Kinect is activated by running **kinect2_bridge** node, default frame rate is set to 1 frame per second, due to performance restrictions:

```
$roslaunch kinect2_bridge kinect2_bridge.launch
```

3. **Calibration** node is launched to obtain decline angle of a camera relatively to ground plane, which is used by **obstacle_processor** node:

```
$roslaunch obstacle_processor calibration.launch
```

4. Obstacle processor is launched by typing the respective launch command. There are a few possible commands to run this node with different configurations:
   - Launch only **obstacle_processor** node:

   ```
   $roslaunch obstacle_processor.launch
   ```

   - Launch both **obstacle_processor** and **kinect2_bridge** simultaneously:

   ```
   $roslaunch obstacle_processor_launch_all.launch
   ```

   - Launch both **obstacle_processor** and **kinect2_bridge** simultaneously with **RVIZ**:

   ```
   $roslaunch obstacle_processor_launch_all_rviz.launch
   ```

   - Launch **obstacle_processor** node and RVIZ for visual output of an algorithm (see 4.1.8):

   ```
   $roslaunch obstacle_processor_rviz.launch
   ```

   - Launch **obstacle_processor** node, RVIZ and **rqt_bag** utility to simulate camera input using bag files (see 4.1.7):

   ```
   $roslaunch obstacle_processor_rviz_debug.launch
   ```

Obstacle_processor.launch file contains 26 input arguments, that set up ROS parameters defined inside this launch file and are used by **obstacle_processor** node itself. These parameters are dynamic and can be assigned with new value during runtime:

```
$rosparam set /obstacle_processor_node/[param_name] [param_value]
```

The definition of each ROS parameter is provided below:

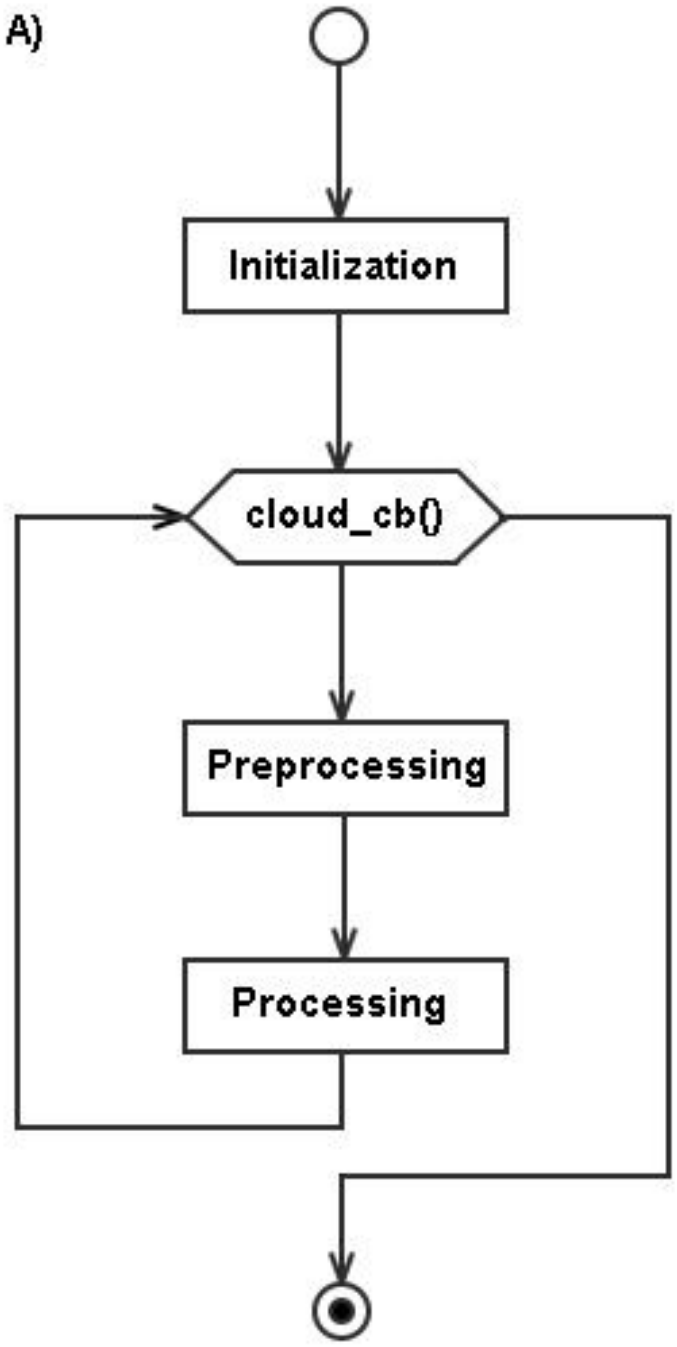
1) **pc_topic** – name of input ROS point cloud topic;
2) **min_z** – minimum distance from camera at which it is able to observe;
3) **max_z** – maximum distance from camera at which it is able to observe;

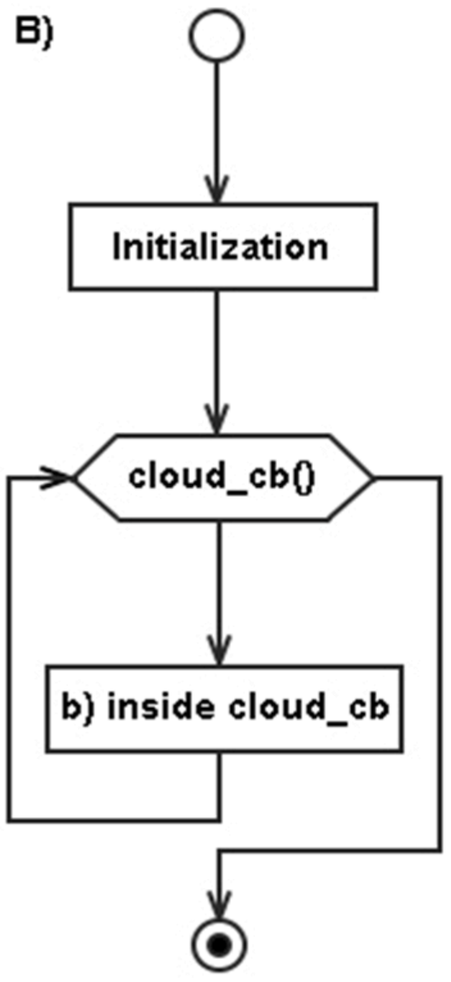
4) **iter** – maximum number of iterations used by groud plane's segmentation;
5) **voxel_size** – a size of a leaf, produced by voxel downsampling filter;
6) **cluster_tol** – parameter, which defines threshold to separate scene into clusters;
7) **min_cluster** – minimum number of points that cluster must have to be considered as cluster;
8) **max_cluster** – maximum number of points that cluster must have to be considered as cluster;
9) **plane_seg_thresh** – distance threshold, which determines how close a point must be to the model in order to be considered an inlier, used by groud plane's segmentation algorithm;
10) **meanK** – number of nearest neighbors to use for mean distance estimation used by **StatisticalOutlierRemoval** filter;
11) **outliers_thresh** – standard deviation multiplier for the distance threshold calculation used by **StatisticalOutlierRemoval** filter;
12) **point_min_z** – minimum value of z coordinate to determine if point is lying on a ground plane;
13) **point_max_z** – maximum value of z coordinate to determine if point is lying on a ground plane;
14) **height_thresh_min** – minimum value of z coordinate used to calculate number of points to filter out clusters that are considered invalid;
15) **height_thresh_max** – maximum value of z coordinate used to calculate number of points to filter out clusters that are considered invalid;
16) **dist_weight** – relative weight (between 0 and 1) to give to the angular distance (0 to $\pi/2$) between point normals and the plane normal;
17) **K_search** – k scale factor used by normal estimation algorithm;
18) **iter2** – maximum number of iterations used by other plane segmentation algorithms applied in **obstacle_processor** node;
19) **thresh2** – distance threshold, which determines how close a point must be to the model in order to be considered an inlier, used by other plane segmentation algorithms applied in **obstacle_processor** node;
20) **tolerance** – parameter that defines an error when estimating an angle between two planes;
21) **iter_circ** – maximum number of iterations used by segmentation algorithm, which helps to determine cylindrical shapes;
22) **dist_weight_circ** – relative weight (between 0 and 1) to give to the angular distance (0 to $\pi/2$) between point normals and the plane normal, used by segmentation algorithm, which helps to determine cylindrical shapes;

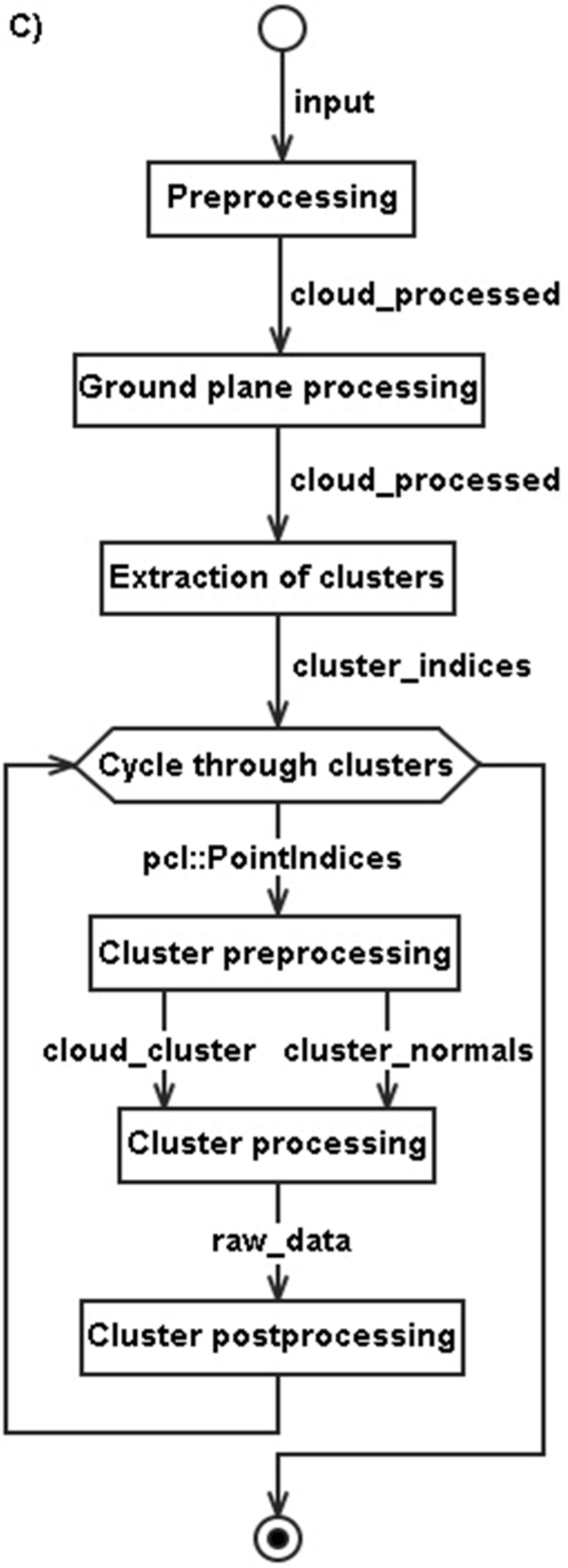
23) **dist_thresh_circ** – distance threshold, which determines how close a point must be to the model in order to be considered an inlier, used by segmentation algorithm, which helps to determine cylindrical shapes;
24) **rviz_out** – logical variable, which defines whether algorithm should output processing steps visually as ROS topics, that are accessible by RVIZ;
25) **cmd_out** – logical variable, which defines whether algorithm should output processing steps into command line;
26) **transform_to_base_link** – logical variable, which defines whether algorithm should transform estimated centroid’s coordinates to **base_link** coordinate frame;

APPENDIX 2: SCHEMATIC STRUCTURE OF NODES

A)




B)
Initialization
cloud_cb()
b) inside cloud_cb


C)
input
Preprocessing
cloud_processed
Ground plane processing
cloud_processed
Extraction of clusters
cluster_indices
Cycle through clusters
pcl::PointIndices
Cluster preprocessing
cloud_cluster
cluster_normals
Cluster processing
raw_data
Cluster postprocessing

APPENDIX 3: THE DATA OF CONDUCTED EXPERIMENTS

a) First static experiment

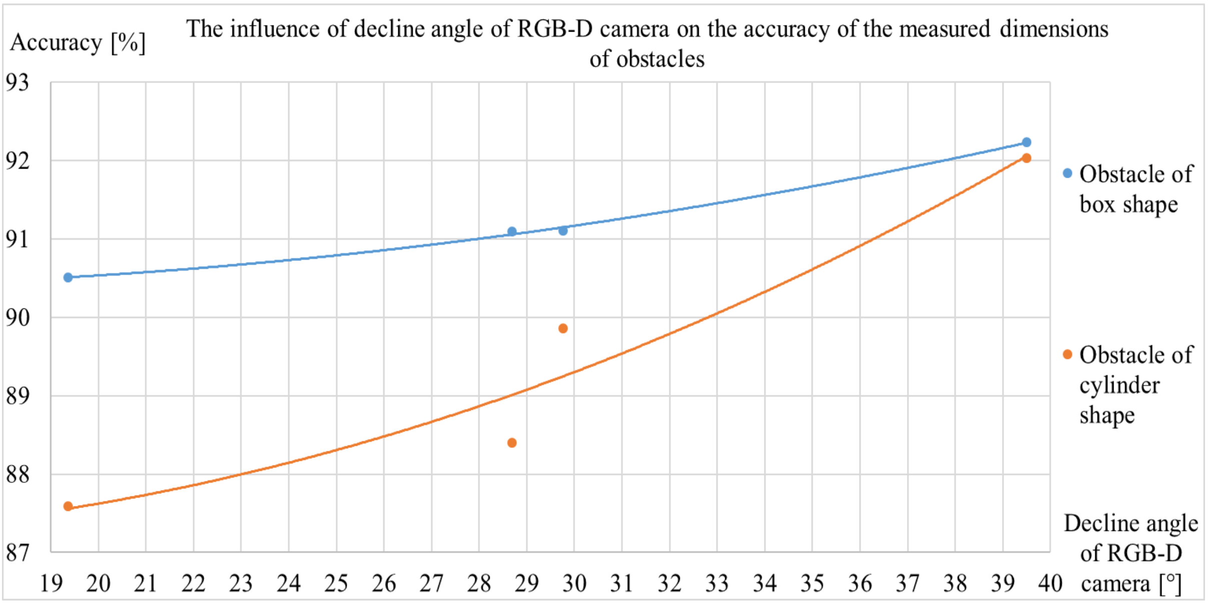
Results of measured obstacles' dimensions at 4 different decline angles of camera are shown in the following table:

decline angle [°]		28,68			19,365			39,5			29,76		
		dimensions [mm]		accuracy	dimensions [mm]		accuracy	dimensions [mm]		accuracy	dimensions [mm]		accuracy
		meas.	real	[%]	meas.	real	[%]	meas.	real	[%]	meas.	real	[%]
1	width	225	263	85,5513	237	263	90,1141	228	263	86,692	221	263	84,0304
	length	324	347	93,3718	325	347	93,6599	295	347	85,0144	302	347	87,0317
	height	403	422	95,4976	405	422	95,9716	404	422	95,7346	411	422	97,3934
	angle	18,58	17	91,4962	18,9	17	89,9471	22,6	25,2	89,6825	22,6	24	94,1667
2	width	129	148	87,1622	110	148	74,3243	129	148	87,1622	117	148	79,0541
	length	281	298	94,2953	287	298	96,3087	287	298	96,3087	289	298	96,9799
	height	271	262	96,679	264	262	99,2424	269	262	97,3978	270	262	97,037
	angle	32,5	33	98,4848	34,4	33	95,9302	36,57	39	93,7692	37,9	39	97,1795
3	width	48	49	97,9592	45	49	91,8367	48	49	97,9592	49	49	100
	length	291	288	98,9691	285	288	98,9583	291	288	98,9691	289	288	99,654
	height	119	138	86,2319	126	138	91,3043	118	138	85,5072	129	138	93,4783
	angle	13,362	14,2	94,0986	16,2	14,2	87,6543	13,2	12	90,9091	10,69	12	89,0833
4	width	26	38	68,4211	28	38	73,6842	34	38	89,4737	29	38	76,3158
	length	50	51	98,0392	46	51	90,1961	45	51	88,2353	42	51	82,3529
	height	161	177	90,9605	165	177	93,2203	174	177	98,3051	177	177	100
	angle	28,69	32,5	88,2769	26,69	29,5	90,4746	33,2	35	94,8571	34,2	36	95
5	radius	58	59	98,3051	60	59	98,3333	60	59	98,3333	58	59	98,3051
	height	44	52	84,6154	42	52	80,7692	42	52	80,7692	47	52	90,3846
6	radius	18	25	72	21	25	84	22	25	88	22	25	88
	height	242	251	96,4143	221	251	88,0478	246	251	98,008	231	251	92,0319
7	radius	34	31	91,1765	34	31	91,1765	29	31	93,5484	24	31	77,4194
	height	152	173	87,8613	144	173	83,237	185	173	93,5135	186	173	93,0108
total				90,7212			89,9269			92,1886			91,2686

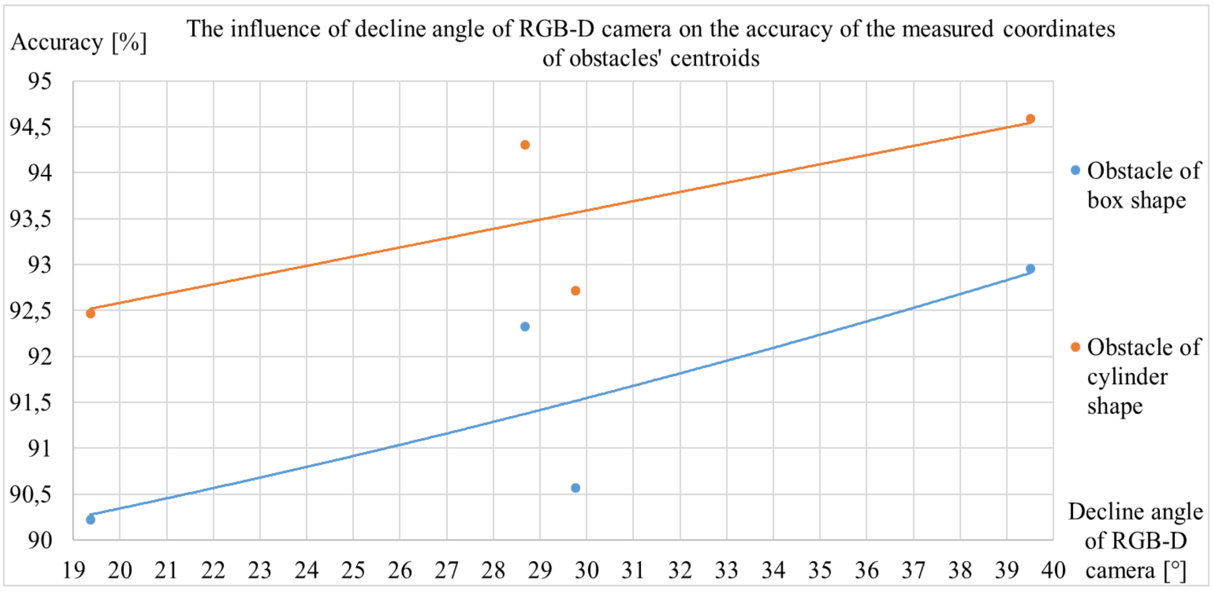
Results of measured obstacles' centroid coordinates at 4 different decline angles of camera are shown in the following table:

decline angle [°]		28,68			19,365			39,5			29,76		
		coordinates [mm]		accuracy	coordinates [mm]		accuracy	coordinates [mm]		accuracy	coordinates [mm]		accuracy
		meas.	real	[%]	meas.	real	[%]	meas.	real	[%]	meas.	real	[%]
1	x	1408	1500	93,8667	1731	1700	98,2091	1192	1172	98,3221	1468	1550	94,7097
	y	324	338	95,858	313	295	94,2492	275	262	95,2727	334	358	93,2961
	z	121	130	93,0769	108	130	83,0769	133	130	97,7444	120	130	92,3077
2	x	1781	1822	97,7497	1922	2022	95,0544	1487	1494	99,5315	1801	1872	96,2073
	y	-56	-52	92,8571	-105	-95	90,4762	-67	-62	92,5373	-29	-32	90,625
	z	74	80	92,5	64	80	80	72	80	90	74	80	92,5
3	x	1471	1506	97,676	1687	1706	98,8863	1183	1178	99,5773	1491	1556	95,8226
	y	25	28	89,2857	-17	-15	88,2353	16	18	88,8889	43	48	89,5833
	z	-6	-7	85,7143	-9	-7	77,7778	-8	-7	87,5	-6	-7	85,7143
4	x	1178	1245	94,6185	1465	1445	98,6348	941	917	97,4495	1278	1295	98,6873
	y	101	111	90,991	78	68	87,1795	96	101	95,0495	119	131	90,8397
	z	13	11	84,6154	10	11	90,9091	12	11	91,6667	13	11	84,6154
5	x	1109	1172	94,6246	1380	1372	99,4203	812	844	96,2085	1159	1222	94,8445
	y	-88	-78	88,6364	-110	-121	90,9091	-81	-88	92,0455	-52	-58	89,6552
	z	-41	-38	92,6829	-45	-38	84,4444	-39	-38	97,4359	-41	-38	92,6829
6	x	1162	1215	95,6379	1437	1415	98,469	893	887	99,3281	1192	1265	94,2292
	y	-276	-272	98,5507	-304	-315	96,5079	-291	-282	96,9072	-236	-252	93,6508
	z	67	70	95,7143	62	70	88,5714	64	70	91,4286	67	70	95,7143
7	x	1481	1517	97,6269	1702	1717	99,1264	1165	1189	97,9815	1481	1517	97,6269
	y	-222	-217	97,7477	-287	-260	90,5923	-211	-227	92,9515	-189	-197	95,9391
	z	42	38	90,4762	32	38	84,2105	41	38	92,6829	42	38	90,4762
total				93,3575			91,1876			94,7862			92,8442

The following diagram demonstrates the influence of decline angle of RGB-D camera on the accuracy of the measured dimensions of obstacles:



The following diagram demonstrates the influence of decline angle of RGB-D camera on the accuracy of the measured coordinates of obstacles' centroids:

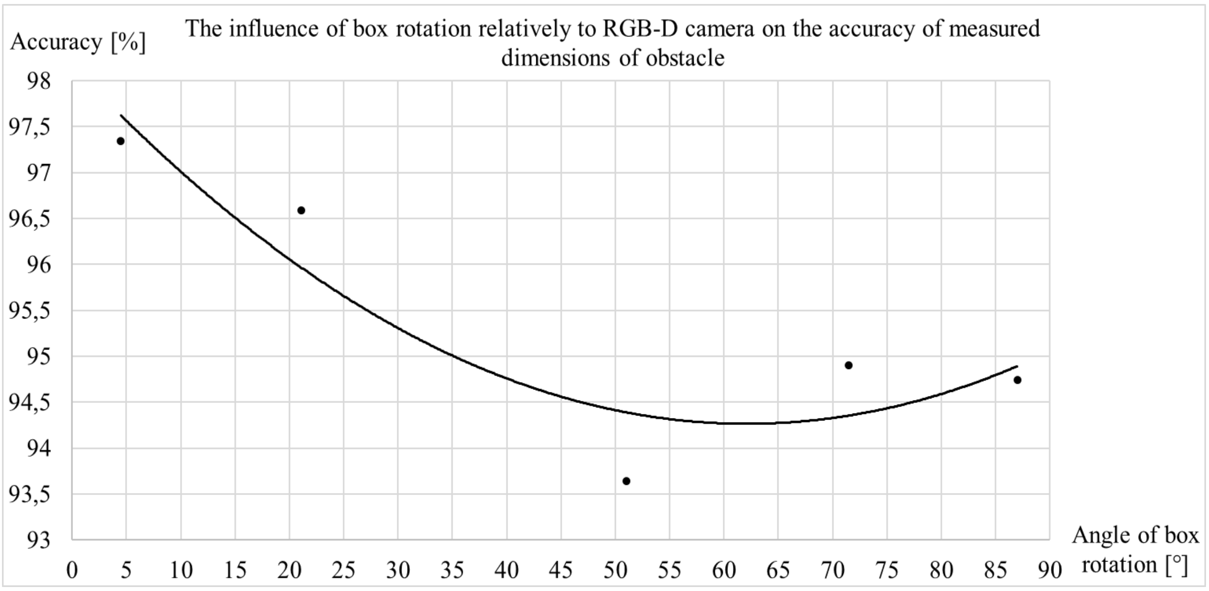


b) Second static experiment

Results of measured box's dimensions and camera's decline angle at 5 different angles of box rotation (an angle between the smallest area of box and camera) are shown in the following table:

Angle of box rotation [°]	Dimensions [mm]			Measured angle [°]	Accuracy [%]
	148	262	298		
	131	269	300	2,689	
	129	270	299	2,419	
	129	270	299	2,333	
87	122	267	299	2,327	
	134	268	299	2,293	
	129	268,8	299,2	87,5878	
	87,162162	97,470238	99,59893	99,328902	94,743777
	123	261	300	19,188	
	129	262	300	18,887	
	129	262	300	18,787	
71,5	126	264	300	18,818	
	127	265	300	18,728	
	126,8	262,8	300	71,1184	
	85,675676	99,695586	99,333333	99,466294	94,901532
	125	251	302	50,825	
	131	251	300	50,878	
	128	251	301	50,907	
51	125	249	301	50,783	
	130	250	301	50,795	
	127,8	250,4	301	50,8376	
	86,351351	95,572519	99,003322	99,681569	93,642398
	140	248	300	20,397	
	140	245	300	20,265	
	146	247	300	20,219	
21,1	141	248	300	20,219	
	144	248	300	20,375	
	142,2	247,2	300	20,295	
	96,081081	94,351145	99,333333	96,184834	96,58852
	148	245	299	4,36	
	152	245	299	4,591	
	150	247	299	4,486	
4,5	149	244	300	5,041	
	149	244	299	4,994	
	149,6	245	299,2	4,6944	
	98,930481	93,51145	99,59893	95,858896	97,346954

The following diagram demonstrates the influence of box rotation relatively to RGB-D camera on the accuracy of measured dimensions of obstacle:



c) Third static experiment

	1 Box 262x148-298					2 Box 405x314-240				
	length	width	height	angle	type	length	width	height	angle	type
x distance	143	263	297	38,838	box	221	412	243	33,51	box
[mm]	145	262	298	39,1	box	246	411	243	33,345	box
900 - 1200	144	264	298	38,754	box	268	409	243	33,389	box
	145	263	298	38,85	box	275	405	241	33,458	box
	146	261	297	38,778	box	284	406	242	33,497	box
arith. mean	144,6	262,6	297,6	38,864		258,8	408,6	242,4	33,44	
accuracy [%]	97,703	99,772	99,866	99,651		82,42	99,119	99,01	99,287	
x distance	139	252	300	39,123	box	220	392	245	34	box
[mm]	135	256	300	39,112	box	237	395	245	34,1	box
1200-1500	137	254	299	39,25	box	246	392	245	34,15	box
	138	256	299	39,15	box	242	396	244	33,975	box
	137	253	298	39,21	box	249	392	245	34,275	box
arith. mean	137,2	254,2	299,2	39,169		238,8	393,4	244,8	34,1	
accuracy [%]	92,703	97,023	99,599	99,569		76,051	97,136	98,039	98,768	
x distance	123	245	302	40,826	box	216	392	243	33,67	box
[mm]	127	250	301	40,2	box	274	388	243	33,975	box
1500-1900	126	255	302	39,95	box	221	391	243	33,967	box
	128	251	300	39,85	box	235	395	246	34,075	box
	127	249	301	40,1	box	211	389	245	34,052	box
arith. mean	126,2	250	301,2	40,185		231,4	391	244	33,948	
accuracy [%]	85,27	95,42	98,938	97,051		73,694	96,543	98,361	99,211	
x distance	113	241	300	41,382	box	249	392	238	34,256	box
[mm]	123	243	301	41,23	box	211	392	238	34,375	box
1900-2200	124	244	303	40,975	box	210	398	235	34,075	box
	125	246	301	41,078	box	209	390	237	34,024	box
	118	244	300	41,127	box	221	395	237	34,255	box
arith. mean	120,6	243,6	301	41,158		220	393,4	237	34,197	
accuracy [%]	81,486	92,977	99,003	94,756		70,064	97,136	98,75	98,488	

The previous table shows the results of experiment for two obstacles of box shape. Dimensions, angle of rotation and the type of obstacle determined by algorithm were measured at 4 different distances of obstacle from the RGB-D camera.

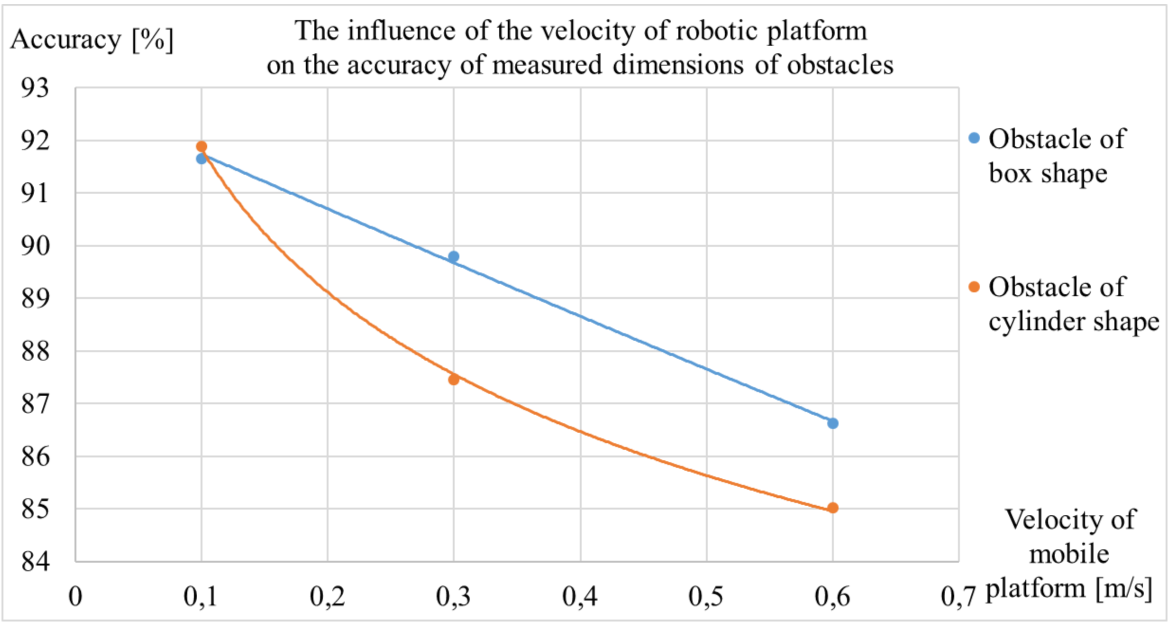
The following table shows the results of experiment for two obstacles of box shape and for one of cylinder shape:

	3 Box 569x364-668					4 Box 155x92-103					1 Cylinder 33-185		
	length	width	height	angle	type	length	width	height	angle	type	radius	height	type
x distance	322	553	670	42,822	box	89	158	99	43,8	box	34	178	cylin.
[mm]	327	568	669	42,967	box	91	154	99	43,675	box	29	179	cylin.
900 - 1200	326	565	670	42,774	box	89	157	99	43,525	box	27	154	cylin.
	334	564	669	42,289	box	90	154	100	43,391	box	32	175	cylin.
	335	569	668	42,621	box	92	157	101	43,45	box	31	174	cylin.
arith. mean	328,8	563,8	669,2	42,695		90,2	156	99,6	43,568		30,6	172	
accuracy [%]	90,33	99,086	99,821	98,807		98,043	99,359	96,699	99,545		92,727	92,973	
x distance	305	561	670	42,711	box	81	153	102	42,97	box	38	145	cylin.
[mm]	300	561	669	42,875	box	84	151	102	42,985	box	34	173	cylin.
1200-1500	304	557	670	42,721	box	81	154	102	42,875	box	32	165	cylin.
	311	559	669	42,672	box	85	155	99	43,01	box	35	176	cylin.
	321	565	670	42,389	box	84	154	101	42,995	box	28	172	cylin.
arith. mean	308,2	560,6	669,6	42,674		83	153,4	101,2	42,967		33,4	166,2	
accuracy [%]	84,67	98,524	99,761	98,759		90,217	98,968	98,252	99,071		98,802	89,838	
x distance	267	557	666	42	box	82	149	96	42,725	box	36	142	cylin.
[mm]	274	553	665	40,2	box	78	153	96	42,758	box	34	163	cylin.
1500-1900	278	552	662	39,95	box	77	155	96	42,975	box	33	175	cylin.
	312	620	648	0	undef	83	148	101	42,674	box	28	148	cylin.
	288	554	668	40,1	box	84	151	99	42,95	box	30	178	cylin.
arith. mean	283,8	567,2	661,8	32,45		80,8	151,2	97,6	42,816		32,2	161,2	
accuracy [%]	77,967	99,684	99,072	75,098		87,826	97,548	94,757	98,724		97,576	87,135	
x distance	329	706	646	0	undef	80	176	96	42,625	box	34	132	cylin.
[mm]	368	622	648	0	undef	91	161	96	42,497	box	34	157	cylin.
1900-2200	318	698	647	0	undef	81	153	96	42,698	box	27	161	cylin.
	315	661	645	0	undef	78	163	97	42,865	box	28	142	cylin.
	321	701	649	0	undef	77	164	99	42,96	box	31	134	cylin.
arith. mean	330,2	677,6	647	0		81,4	163,4	96,8	42,729		30,8	145,2	
accuracy [%]	90,714	83,973	96,856	0		88,478	94,859	93,981	98,522		93,333	78,486	

The following table shows results of experiment for last three obstacles of cylinder shape:

	2 Cylinder 27-251			3 Cylinder 59-52			4 Cylinder 113-298		
	radius	height	type	radius	height	type	radius	height	type
x distance	25	243	cylin.	59	46	cylin.	108	291	cylin.
[mm]	25	243	cylin.	58	41	cylin.	106	291	cylin.
900 - 1200	26	243	cylin.	58	42	cylin.	108	291	cylin.
	27	245	cylin.	59	45	cylin.	110	294	cylin.
	25	247	cylin.	58	47	cylin.	107	295	cylin.
arith. mean	25,6	244,2		58,4	44,2		107,8	292,4	
accuracy [%]	94,815	97,291		98,983	85		95,398	98,121	
x distance	22	246	cylin.	58	46	cylin.	103	298	cylin.
[mm]	23	246	cylin.	57	47	cylin.	99	293	cylin.
1200-1500	26	246	cylin.	58	46	cylin.	102	299	cylin.
	22	246	cylin.	56	41	cylin.	105	292	cylin.
	27	243	cylin.	59	48	cylin.	107	297	cylin.
arith. mean	24	245,4		57,6	45,6		103,2	295,8	
accuracy [%]	88,889	97,769		97,627	87,692		91,327	99,262	
x distance	27	247	cylin.	56	44	cylin.	103	297	cylin.
[mm]	26	238	cylin.	56	42	cylin.	101	293	cylin.
1500-1900	27	239	cylin.	57	42	cylin.	106	298	cylin.
	19	241	cylin.	55	46	cylin.	104	295	cylin.
	21	221	cylin.	58	45	cylin.	99	292	cylin.
arith. mean	24	237,2		56,4	43,8		102,6	295	
accuracy [%]	88,889	94,502		95,593	84,231		90,796	98,993	
x distance	24	215	cylin.	54	40	cylin.	95	295	cylin.
[mm]	26	200	cylin.	56	38	cylin.	107	291	cylin.
1900-2200	25	227	cylin.	60	39	cylin.	103	293	cylin.
	20	235	cylin.	52	42	cylin.	102	294	cylin.
	23	205	cylin.	54	45	cylin.	101	291	cylin.
arith. mean	23,6	216,4		55,2	40,8		101,6	292,8	
accuracy [%]	87,407	86,215		93,559	78,462		89,912	98,255	

d) Dynamic experiments



The previous diagram demonstrated the influence of the velocity of robotic platform on the accuracy of measured dimensions of obstacles.

Results of measured dimensions of 4 obstacles of box shape at 3 different velocities of mobile platform are shown in the following table:

	1 Box dimensions [mm]			2 Box dimensions [mm]			3 Box dimensions [mm]			4 Box dimensions [mm]			Total
	148	262	298	92	155	103	38	51	177	49	288	138	accuracy
	125	234	302	101	155	91	29	35	174	50	279	130	
	134	232	301	100	159	94	31	37	174	48	287	128	
v = 0,1	129	243	302	99	157	91	29	34	176	45	277	132	
	129	239	300	98	152	89	32	39	173	47	284	131	
	133	244	304	98	161	96	33	37	175	49	289	130	
arith mean	130	238,4	301,8	99,2	156,8	92,2	30,8	36,4	174,4	47,8	283,2	130,2	
accuracy [%]	87,83784	90,99237	98,74089	92,74194	98,85204	89,51456	81,05263	71,37255	98,53107	97,55102	98,33333	94,34783	91,65567
	137	253	301	101	159	89	29	32	171	47	285	133	
	123	234	302	102	157	92	31	37	140	48	293	130	
v = 0,3	121	219	302	97	164	89	29	33	170	51	282	129	
	132	226	300	99	152	95	28	36	173	43	278	129	
	118	224	304	101	161	91	27	41	165	48	277	124	
arith mean	126,2	231,2	301,8	100	158,6	91,2	28,8	35,8	163,8	47,4	283	129	
accuracy [%]	85,27027	88,24427	98,74089	92	97,73014	88,54369	75,78947	70,19608	92,54237	96,73469	98,26389	93,47826	89,7945
	118	266	302	93	167	92	25	35	154	30	235	126	
	123	223	302	103	159	87	30	37	147	45	289	127	
v = 0,6	115	219	305	101	169	88	29	35	174	49	269	131	
	123	212	297	99	158	93	28	37	170	41	273	127	
	109	231	304	107	164	93	24	34	159	39	257	122	
arith mean	117,6	230,2	302	100,6	163,4	90,6	27,2	35,6	160,8	40,8	264,6	126,6	
accuracy [%]	79,45946	87,8626	98,6755	91,45129	94,85924	87,96117	71,57895	69,80392	90,84746	83,26531	91,875	91,73913	86,61492

Results of measured dimensions of 3 obstacles of cylinder shape at 3 different velocities of mobile platform are shown in the following table:

	1 Cylinder dimensions [mm]		2 Cylinder dimensions [mm]		3 Cylinder dimensions [mm]		Total
	59	52	25	251	31	193	accuracy
	58	44	24	255	30	174	
	59	46	14	178	33	153	
v = 0,1	60	46	26	232	35	182	
	59	47	25	251	37	162	
	59	49	25	165	32	151	
arith mean	59	46,4	22,8	216,2	33,4	164,4	
accuracy [%]	100	89,23077	91,2	86,13546	92,81437	85,18135	90,76032
	57	47	24	191	37	174	
	58	46	18	167	29	121	
v = 0,3	60	39	25	229	31	175	
	57	47	21	251	35	181	
	59	42	17	213	36	131	
arith mean	58,2	44,2	21	210,2	33,6	156,4	
accuracy [%]	98,64407	85	84	83,74502	92,2619	81,03627	87,44788
	58	46	18	231	28	85	
	56	47	19	154	27	175	
v = 0,6	54	37	26	223	33	181	
	55	39	19	221	32	173	
	57	45	20	201	21	141	
arith mean	56	42,8	20,4	206	28,2	151	
accuracy [%]	94,91525	82,30769	81,6	82,07171	90,96774	78,23834	85,01679

e) Specification of PC on which the system has been tested is provided in a table below:

Lenovo Edge 540	
CPU	Intel Core i5-4200M processor (2 cores, 2.50GHz, 3MB cache)
RAM	4GB PC3-12800 DDR3L
Graphics	Intel HD Graphics 4600
HDD	500 GB SATA

f) Performance experiments

The following table demonstrates measured processing time needed to process cluster of an obstacle and an input point cloud that contains this obstacle:

Obstacle	Dimensions [mm] /	Processing time [ms]						
	points in input cloud [-]	1	2	3	4	5	arith. mean	total
Box	347x263-422	202	209	200	198	195	200,8	545,4
Cloud	6595	347	317	362	350	347	344,6	
Box	405x314-240	180	186	183	176	195	184	508,4
Cloud	4969	352	314	324	316	316	324,4	
Box	262x148-298	38	31	30	28	34	32,2	365,4
Cloud	2248	309	354	354	351	298	333,2	
Box	155x92-103	9	13	13	16	10	12,2	322,8
Cloud	991	304	337	313	300	299	310,6	
Box	569x364-668	192	179	186	184	178	183,8	588,4
Cloud	8254	396	409	408	395	415	404,6	
Cylinder	33-185	15	17	15	15	16	15,6	351,2
Cloud	898	356	326	391	307	298	335,6	
Cylinder	27-251	16	13	16	23	22	18	342,6
Cloud	909	317	305	342	342	317	324,6	
Cylinder	59-52	15	15	22	12	13	15,4	323,8
Cloud	826	309	313	316	305	299	308,4	
Cylinder	113-300	141	155	176	131	144	149,4	467,6
Cloud	2455	319	342	311	294	325	318,2	
Cylinder	152-250	74	65	79	58	90	73,2	378,2
Cloud	1715	316	315	307	295	292	305	
Undefined	320x300-500	257	251	264	249	253	254,8	593,8
Cloud	3320	349	331	359	326	330	339	
Undefined	42x70-135	5	10	3	9	10	7,4	324,8
Cloud	751	316	304	327	342	298	317,4	
Undefined	316x293-165	147	107	148	159	113	134,8	458,2
Cloud	2129	343	320	326	319	309	323,4	
Undefined	400x297-205	100	185	96	108	99	117,6	437,2
Cloud	2381	300	314	333	335	316	319,6	
Undefined	260x255-145	116	89	95	87	112	99,8	432
Cloud	1687	329	298	323	355	356	332,2	