Bachelorarbeit am Institut für Informatik

Arbeitsgruppe Intelligente Systeme und Robotik

Object Detection and Recognition with Microsoft Kinect

Marcel Jünemann
Matrikelnummer: 4360370
marcel.juenemann@fu-berlin.de

Erstprüfer: Prof. Dr. Raúl Rojas
Zweitprüfer: Hamid Reza Moballeg
Betreuer: David Latotzky

Berlin, 12. Januar 2012

Abstract

This thesis presents an object recognition algorithm that is designed for the RoboCup@Home. It utilizes 3-dimensional data acquired by a Kinect in order to detect objects that are placed on tables or similar plane surfaces. Fast Point Feature Histograms and hue histograms are combined for the feature extraction from objects, which are then matched with previously trained objects. The algorithm has been implemented by using the recently published Point Cloud Library. The implementation has been evaluated with several objects and has shown a recognition accuracy of 94%.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

12. Januar 2012

Marcel Jünemann

Contents

1 Introduction

The goal of this thesis is the development of an object detection and recognition software for a new RoboCup@Home team, which designs a robot utilizing an electric wheelchair. The RoboCup@Home is an annual competition for autonomous service robot projects and consists of multiple benchmark tests in a home environment setting. Object recognition is primarily required in the "Go Get It!" test, which is explained in the rule book 2011 [8] as follows: In preparation for the test, the team selects 10 objects from 20 predefined everyday household objects. The team is then allowed to train the robot with these 10 objects, so that the robot can recognize them later. During the test, four of these objects are placed in the setting, two on a table or a similar flat surface and two on the floor. The task is to recognize them, grasp them and finally retrieve them to the operator. An additional difficulty is the presence of four objects that the robot did not learn. It must not retrieve these unknown objects and therefore has to be able to distinguish between known and unknown objects. This thesis only covers the detection and recognition part of the task.

Figure 1: RoboCup@Home setting with a robot of the GT@Home team performing the "Go Get It!" test. Image source: [15]

1.1 Outline

Our robot features a Kinect, which is a device published in 2010 as a motion sensing controller for Microsoft's Xbox 360 video game console. The Kinect possesses a depth sensor and therefore provides us with a point cloud of the acquired scene, where each point is annotated with its position in 3-dimensional space. Section 2 presents and discusses an algorithm for the detection of objects. It is based on the assumption that objects are always placed on a plane surface, which is true in the RoboCup@Home context. Thus, it detects tables and other plane surfaces at first, so as to extract the point clouds of objects placed on those surfaces afterwards. Section 3 discusses the feature extraction from objects in order to match detected objects with objects learned in the training stage. Color descriptors as well as surface descriptors are utilized for this task.

An implementation of the detection and recognition is presented in section 4. The implementation uses the recently developed Point Cloud Library (PCL), which simplifies the handling of 3-dimensional point clouds by offering data structures as well as various algorithms that operate with point clouds. PCL is a novel library that was founded in 2010 and officially published in 2011 [21]. Finally, section 5 demonstrates how to use the implementation in practice, and section 6 evaluates the accuracy of the object detection and recognition.

1.2 Related Work

Object recognition is a research topic that is currently intensely investigated in computer vision and related research fields. In context of the RoboCup@Home, the existing teams summarize their approaches in team description papers, for example [10, 17, 23]. The detection of plane surfaces like tables is discussed in [3, 7, 16, 29]. There are various descriptors for the purpose of object recognition, which use color information [6, 26], interest point extraction [1, 12, 25], or 3-dimensional surface structure [9, 19, 20, 22].

2 Object Detection

This section presents an algorithm for the detection of objects. As mentioned in the introduction, we will assume that the objects are placed on a table or a similar planar surface area, e. g. on a shelf or on the floor. Therefore, the largest part of this section covers the detection of these surface areas. Once we estimated a table, the objects on it can be extracted by Euclidean clustering.

2.1 Preprocessing

The 3-dimensional point cloud that we receive from the Kinect contains more than 300,000 points. Although this are fewer points than modern digital cameras acquire, we try to reduce the number of points in the first step because we have to iterate over the entire point cloud multiple times. Since the region that the arm of our robot can reach is limited horizontally, we can ignore the area that is too high above our robot. If the robot arm is not able to grasp objects on the floor, we can also ignore all areas that are too low. Since every point of the input point cloud is denoted with x, y and z coordinates, we filter out all points with too high or too low z-values (see Figure 2 for the coordinate system).

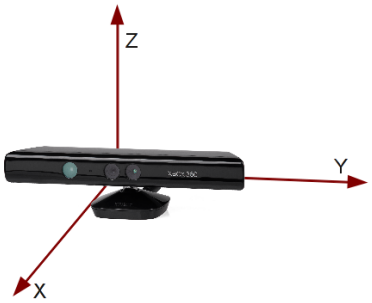


Figure 2: The origin of our coordinate system is the Kinect.

Additionally, to reduce the number of points further, we downsample the point cloud by using a voxelized grid approach, i. e. we divide the point cloud into a grid of voxels which are equivalent to pixels in the 2-dimensional space. In this case, I chose the size of the voxels to be 1cm x 1cm x 1cm. Then, we replace all points in each voxel by a single point, which we place at the centroid of the voxel. Note that all sizes and lengths mentioned in this section, like the voxel size, are only suggestions that I found to work satisfactorily, but will be adjustable in my implementation. The bigger the voxel size, the better is the quality of the object detection, but the longer is the run-time of the detection algorithm.

2.2 Table Detection

In order to detect tables in the downsampled point cloud, I will present three slightly different approaches. The task is not only to detect the points of the cloud that belong to the table, but also to estimate a plane model that fits these points.

2.2.1 Random Sample Consensus

The Random Sample Consensus (RANSAC) algorithm is a general algorithm for fitting models to data that contains many outliers. The algorithm was proposed by Fischler and Bolles [5] in 1981 and is very often used to detect tables (or planes in general) in 3-dimensional point cloud data [18, 16, 29, 3]. In our case the outline of the algorithm is as follows:

1. Select three points from the point cloud randomly. If the three points do not define an unambiguous plane, i. e. they are collinear, select another three points.

2. Estimate the plane model from the three points, i. e. calculate the parameters a, b, c, d for the Hesse normal form

$$ax + by + cz + d = 0$$

3. Count how many points of the cloud support the model. A point is considered an inlier if it's distance to the plane does not exceed a certain threshold (e. g. 1cm).

4. Repeat 1.-3. until the maximum number of iterations (e. g. 1,000) is reached, and return the plane model with the most inliers.

In order to use the RANSAC algorithm to detect more than one table, we remove the inliers of the found table from the point cloud and start the algorithm again.

2.2.2 Surface Normals

All to often, the RANSAC approach considers points, that are not part of any table, as inliers of the plane model. To reduce the number of these false positives, a common improvement is to take the surface normals into account [20, 7].

Estimation of surface normals The surface normal vector $\vec{n}$ at a single point $\vec{p}$ is defined as the vector orthogonal to the plane that tangents the surface at $\vec{p}$. For the estimation we consider only the k nearest neighbors of $\vec{p}$, where k is a constant. If k is too low, the normal estimation is highly influenced by outliers, i. e. by the noisy data acquired through the Kinect. In contrast, if we select k too high, our normals are smoothened and do not represent the details of the surface. A value that provides satisfying results for noisy data is $k = 10$ [20]. We will determine the tangent plane approximately by a least squares approach, thus we will minimize

$$\sum_{i=1}^{k} ((\vec{p_i} - \vec{x}) \cdot \vec{n})^2$$

where $\vec{p_i}$ is a point of the k-neighborhood and $\vec{x}$ is a point of the tangent plane. Of course, the query point $\vec{p}$ itself is a point of that plane, and therefore we could set $\vec{x} = \vec{p}$. However, if we choose $\vec{x}$ to be the centroid of the k-neighborhood

$$\vec{x} = \frac{1}{k} \sum_{i=1}^{k} \vec{p_i}$$

and thus estimate a plane parallel to the tangent plane, we can calculate the $\vec{n}$ with the smallest error directly from the covariance matrix $C \in \mathbb{R}^{3\times3}$

$$C = \frac{1}{k} \sum_{i=1}^{k} (\vec{p_i} - \vec{x}) \cdot (\vec{p_i} - \vec{x})^T$$

Of the three eigenvectors of C, the eigenvector with the smallest eigenvalue approximates $\vec{n}$ [2, 18].

Because this approach could either return an inward-pointing or an outer-pointing normal vector, we flip all normals towards the viewpoint (the Kinect), so we can compare the vectors among each other. Certainly, this is a relatively straightforward algorithm for the problem of normal estimation, and more complex algorithms, which are faster and yield better approximations, exist. [11] compares some of them and performs a benchmark analysis.

k-nearest-neighbors search In the last paragraph we assumed that we know which k points are the nearest to our query point. But the nearest neighbor search is a non-trivial problem, because the computational complexity of the calculation of a full distance matrix is $O(n^2)$, and thus unacceptably slow. The commonly utilized data structure for this task is a k-d tree. Basically, a k-d tree is a binary tree that partitions the nodes in space. In our 3-dimensional case, the first level of the tree partitions all points by their x-value. The second level uses the y-value, the third partitions by z, and the fourth by x again etc. An example of a 3-d tree is visualized in Figure 3. Similar to an usual binary tree, we can build the tree for n points in $O(n \log n)$ [28]. Unfortunately, even for a k-d tree there are no algorithms that perform a k-nearest-neighbors search faster than $O(n)$ for a single point [13]. But if

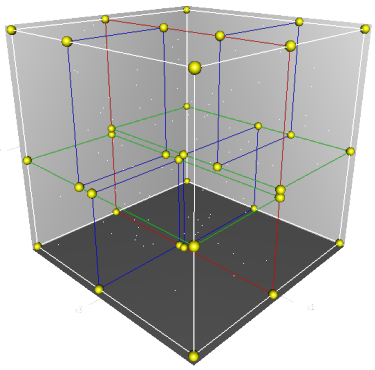


Figure 3: A 3-dimensional k-d tree. The split on the first level of the tree is shown in red. The second level is illustrated with two green rectangles and the third level with four blue rectangles. Image source: [28]

we do not insist on exact results, many algorithms exist that can perform an approximate nearest neighbor search much faster. Since approximations are fine in our case, we use the algorithm proposed by Muja and Lowe in [13]. It combines multiple search algorithms by automatically selecting the best algorithm and it's parameters depending on properties of the given data. My implementation will use the FLANN library, which is an implementation of this sophisticated algorithm and is available as free software.

Integration with RANSAC Now that we know how to calculate the surface normals, how can we use them to improve the table detection? We will integrate the surface normals into the RANSAC algorithm. Previously, we considered a point $\vec{p}$ as an inlier of a plane model E if

$$d(E, \vec{p}) < \delta$$

where d estimates the Euclidean distance between E and $\vec{p}$ and δ is a certain threshold. Considering the surface normals, we change this condition to

$$\alpha \cdot \angle(\vec{n_p}, \vec{n_E}) + (1 - \alpha) \cdot d(E, \vec{p}) < \delta'$$

where $\angle(\vec{n_p}, \vec{n_E})$ denotes the angle between the estimated normal vector at $\vec{p}$ and the normal vector of the plane model. The authors of [20] used $\alpha = 0.1$ and $\delta' = 0.02$ in their implementation.

2.2.3 Orthogonal Normal Filtering

Unfortunately, the RANSAC algorithm is slow and therefore the major bottleneck of the whole detection process. But we can detect tables much faster if we know the angle between the Kinect and the floor. This is the case for our robot, and in general for most of all existing robots. The angle is only hard to determine if the Kinect is continuously shaken, e. g. on a humanoid robot. Once we know the angle between the Kinect and the floor, we know the approximate normal vector of all tables, because tables are in general parallel to the floor. Thus, if the Kinect is straight in relation to the floor, all points of flat surfaces have a normal vector parallel to the z-axis (remember Figure 2 for the orientation of the axis). This is also true for a non-zero pitch angle of the Kinect if we rotate the whole point cloud around the y-axis at first. Therefore, in order to detect tables much faster, we filter out all points whose normal vector is not approximately parallel to the z-axis.

As you can see in Figure 4, not all points of the filtered point cloud are necessarily part of a table or similar surface that we want to detect, but we can now utilize a clustering algorithm in order extract the tables from the point cloud. We consider points to be in the same cluster if their distance is below a certain threshold, e.g. 5cm (Euclidean clustering). This way,

Figure 4: A crowded scene with a table in the foreground. The Figure shows only points whose surface normal is orthogonal to the floor.

we can detect flat surfaces reliably if we drop all clusters that do not meet additional a priori conditions, for example a minimum width of 10cm. After the clustering, we estimate a plane model for each cluster by executing the RANSAC algorithm on the cluster, but this time the algorithm is much faster because it does not operate on the whole point cloud.

2.2.4 Convex Table Hull

Now that we detected all tables and estimated a point cloud as well as a supporting plane model for each, we calculate a convex hull in the next step. The reason for this is shown in Figure 5: The Kinect can not acquire regions of a table that are hidden by an object. Therefore, we get a more appropriate representation of the table by using its convex hull.

2.3 Object Cluster Extraction

For the extraction of the objects on a table you should always have in mind that the Kinect does not acquire a full 3-dimensional representation of the objects. Instead it acquires only a ”2.5-dimensional” representation which contains the points that can be seen from the Kinect’s point of view (as seen in Figure 5). Once we estimated the convex hull of a table, we can extract all points above this convex hull, using the plane mode, in order to get the point cloud of the objects on that table. A problem might be that we also extract points that are part of the table surface, because the data is noisy, our table detection not perfect, and because we defined a range, in which

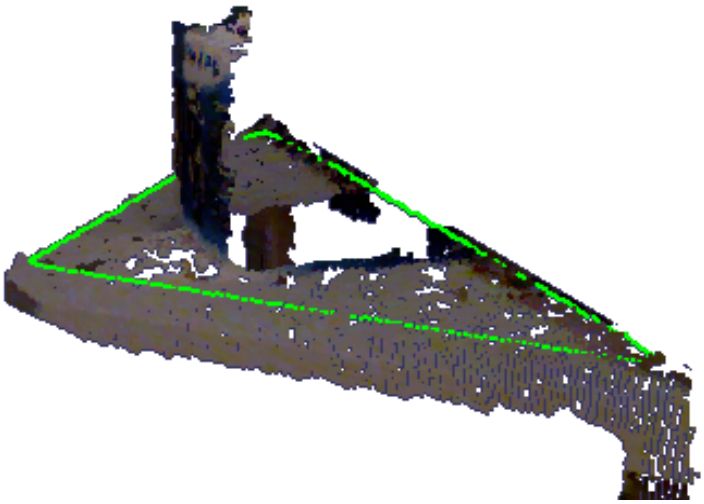


Figure 5: Side view of a triangular table with a box placed on it. The region behind the object is not acquired by the Kinect. Nevertheless, the convex hull drawn in green represents the table correctly.

a point is considered an inlier of the plane model. As a solution, we only extract points with a certain distance to the table, e.g. only points that are at least 5mm above the table. The 5mm cut off from every object's bottom is a clear disadvantage, but our robot can only operate with objects that are at least a few centimeters high anyway, and thus this should not be a serious problem for the detection as well as the recognition.

Since there can be more than one object on each table, we have to cluster the previously extracted point cloud into the single objects. In the context of RoboCup@Home we may assume that there are several centimeters space between all objects, and therefore perform the clustering with an Euclidean approach. For example, points that are at least 5cm away from each other are considered to be of different objects.

2.4 Projected Object Clustering

We can achieve a small speed-up if we perform the clustering in 2-dimensional space. For this purpose, we project the object point cloud to the estimated plane model of the table, and extract the object clusters in 2-dimensional space. Then we calculate the convex hull of each cluster and extract all points above this convex hull to get each object's final point cloud. Besides the speed-up, our clustering is now able to detect objects that consist of horizontally separated components, like the bottle shown in Figure 6. But an even bigger advantage is that we can easily use the original point cloud instead of the downsampled point cloud in the last extraction step. This

way, we get a fully detailed version of all objects, which is essential for the following step, the object recognition. Finally, we can ignore those object clusters that are too small to be an object or too small to be manipulated by our robot, e.g. clusters that are smaller than 5cm in height.

Figure 6: The Kinect was unable to recognize the transparent surface of this bottle. The cup and the front label were recognized, as well as parts of the label on the back side which could be seen through the transparent surface of the bottle.

3 Object Recognition

Object recognition is a non-trivial task. It is not possible to compare objects point by point in order to match them, because even every acquisition of the same object is slightly different. Our algorithm has to recognize objects from different distances, from different points of view, and under different lightning conditions. Thus, object recognition is done by utilizing descriptors which extract certain characteristic properties of the object. This section will present descriptors that focus on color properties of the object, as well as surface descriptors and such descriptors that use special interest points of the object. All of them have in common that they extract a feature vector, i. e. they project the object into $\mathbb{R}^n$. These vectors can then be used to match detected objects with previously trained objects.

3.1 Interest Point Detection

Object recognition systems operating on 2-dimensional images usually extract special interest points from the image. The most widely used algorithm for this is the Speeded Up Robust Feature (SURF) algorithm proposed in [1]. It extracts interest points from the image by analyzing the color intensity distribution, and estimates various features in the region around these points. The interest points can then be matched with interest points from trained images.

Figure 7: The detected interest points of the left image are matched with the detected interest points of the right image. Unfortunately, the left image is a bottle and the right image is a mug with an image of the same bottle on it. Interest point detectors that use only 2-dimensional information are not able to detect this difference, which is important for a robot in a human living environment. Image source: [18]

However, a disadvantage of 2-dimensional object recognition is demonstrated in Figure 7. Since we can use the data of the Kinect's sensor, we will rather use a 3-dimensional descriptor. There are also interest point extractors that operate on 3-dimensional data, like the recently proposed Normal Aligned Radial Feature (NARF) [25]. Nevertheless, interest point descriptors are

designed to extract features of whole scenes, but we already extracted the isolated point clouds of the detected objects. Therefore, we will rather use a descriptor that operates on the isolated point cloud, as presented in the following section.

3.2 3-Dimensional Surface Descriptors

3.2.1 Point Feature Histograms

Point Feature Histograms (PFH) were proposed by Rusu et al. [22] in 2008. PFH is a surface descriptor for a single point of a point cloud and is the base for more complex descriptors which I will present in the following sections. The estimation of PFH requires the preceding calculation of the surface normals described in section 2.2.2. The key idea is to create a feature that represents the difference of a point's normal in relation to the normals of adjacent points, thus describing the geometrical properties of the point.

In the first step of the estimation we search for the k nearest neighbors of the query point p. For each pair $(p_s,\ p_t)$ in the resulting set of $k+1$ points we then calculate three angles α, ϕ and θ which represent the relation between the point's corresponding normals n_s and n_t. We define a frame originated in p_s in the following way:

$$
\begin{aligned}
u &= n_s \\
v &= u \times \frac{p_t - p_s}{\|p_t - p_s\|_2} \\
w &= u \times v
\end{aligned}
$$

where $\|p_t - p_s\|_2$ is the distance between p_s and p_t. α, ϕ and θ are now defined as shown in Figure 8.

$$
\begin{aligned}
\alpha &= v \cdot n_t \\
\phi &= u \cdot \frac{p_t - p_s}{\|p_t - p_s\|_2} \\
\theta &= \arctan(w \cdot n_t, u \cdot n_t)
\end{aligned}
$$

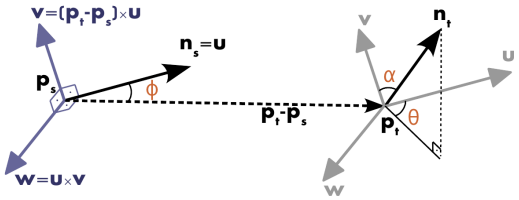


Figure 8: We define a frame uvw in p_s. Our features α, ϕ and θ describe differences between the normals n_s and n_t. Image: [18]

For each of the three angles we get $\binom{k+1}{2}$ values. By dividing the value ranges into q equally sized bins, we create a 3-dimensional histogram with a total of q^3 bins. Therefore, every bin increment stands for a pair of points having certain values for all of the three angles. Of course, a higher number of bins results in a more detailed description of the point's surface. To give an example, the PFH implementation in the Point Cloud Library sets $q = 5$ by default.

In order to use the PFH descriptor as a feature of an entire point cloud, we simply sum up the histograms of all points in the cloud to a single histogram. To preserve invariance to scaling of the object, we normalize the histogram by dividing each value by the number of points in the cloud. Since the run-time of the calculation of a histogram for a single point is $O(k^2)$, the calculation for an entire point cloud consumes $O(n \cdot k^2)$ which is too slow for real time usage on current computers [19].

3.2.2 Fast Point Feature Histograms

A Fast Point Feature Histogram (FPFH) is a surface descriptor which is based on PFH and has been introduced in [19]. The major improvement of the FPFH is that it can be calculated in $O(n \cdot k)$ for a point cloud with n points. This enables a feature extraction in real time while having a discriminative power nearly as good as PFH's.

Instead of calculating the differences between each pair of points in the k-neighborhood of a point p, we calculate them only between p and every other point in the k-neighborhood. This reduces the number of calculations for a single point from $\binom{k+1}{2}$ to k. This is called Simplified Point Feature Histogram (SPFH). Of course, the PFH is a much better descriptor for the surface of a point, because it also takes the relations between neighboring points into account. To improve the SPFH without increasing the computational complexity, we weight the SPFH of the query point p_q with the SPFH values of the k neighboring points in the following way:

$$\mathrm{FPFH}(p_q) = \mathrm{SPFH}(p_q) + \frac{1}{k} \sum_{i=1}^{k} \frac{1}{w_k} \cdot \mathrm{SPFH}(p_k)$$

where w_k represents the distance between the query point p_q and the neighboring point p_k. $\mathrm{FPFH}(p_q)$ is our final Fast Point Feature Histogram for the point p_q. The calculation is clarified in Figure 9. Through the weighting we even include those points into the calculation which are not in the k-neighborhood of the query point, but in the neighborhood of a neighboring point of p_q.

Furthermore, this is not the only improvement of FPFH compared to PFH that was introduced in [19]. A problem of the 3-dimensional histograms is that a large number of bins of the histogram is zero if a specific bin of a single feature is zero. For example, let us assume that q is 5 and there are no values of α in the value range 72° to 90°. One of the five bins (20%) of the α-histogram is now zero. Since the 3-dimensional histogram combines all features, our resulting histogram with 125 bins has at least 25

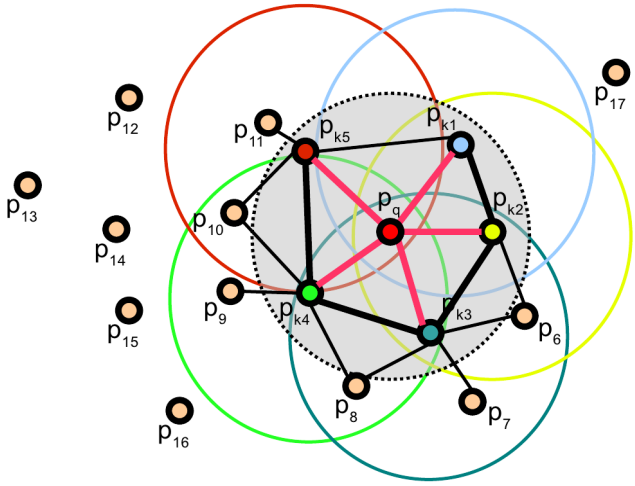


Figure 9: Scheme of the neighborhood of an example point p_q. The FPFH of p_q is influenced by its 5 neighboring points and indirectly by their neighbors. Image source: [18]

(also 20%) bins that are zero. Thus, we give up expensive space in the final histogram for an information that could have been saved much easier. To avoid this, FPFH passes on the combination of the three feature histograms and instead just concatenates them. As compensation to the information loss by not combining the features, we can divide the feature histograms into more bins, because the final histogram now contains $3q$ instead of q^3 bins.

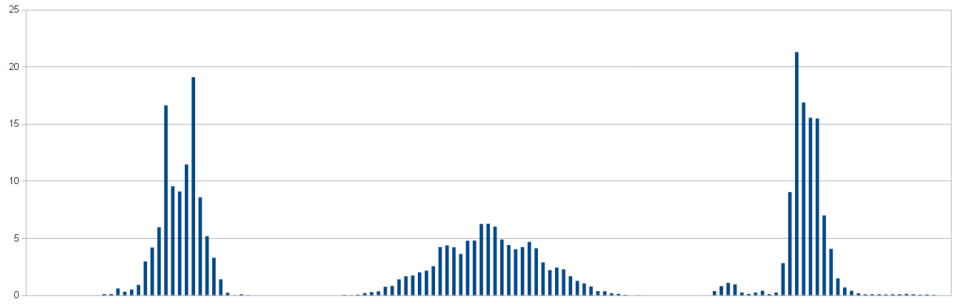


Figure 10: Example Fast Point Feature Histogram of a real object with $3 \cdot 45$ bins

3.2.3 Viewpoint Feature Histograms

An addition to the FPFH was made in [20] in 2010. The authors wanted their robot not only to recognize the object, but also the exact pose of the object. Therefore, the proposed Viewpoint Feature Histogram (VFH) concatenates the Fast Point Feature Histogram with a viewpoint histogram. The viewpoint histogram contains the distribution of the angles between

all surface normals and the vector from the viewpoint to the centroid of the whole object. The centroid is used in order to remain scale invariance. The authors used the VFH for an object recognition system and their tests with over 60 relatively similar objects have shown a recognition and pose estimation accuracy of over 98%.

3.3 Color Descriptors

In order to be able to distinguish objects that only differ in their color and not in their surface structure, we will also utilize a color descriptor.

3.3.1 RGB Histograms

The color information we receive by the Kinect is a common RGB value. We get three integers R, G and B in the range from 0 to 255, denoting the red, green and blue component of the color. A simple approach towards adding a color descriptor to our object recognition is to fill the RGB values of all points into a histogram. Similar to the situation described for the Fast Point Feature Histogram (section 3.2.2), we can either create one 3-dimensional histogram combining all color components, or create three 1-dimensional histograms. Either way, RGB histograms are only a representation of the absolute colors and are variant to basic photometric transformations, such as light intensity changes, which can be expressed as a multiplication of all components with a scalar. Light intensity changes are very common, since they not only occur when the intensity of the light source is changed, but also when our object is shaded. Therefore, we are looking for a color descriptor τ that satisfies the following condition:

$$\tau(\begin{pmatrix} R \\ G \\ B \end{pmatrix}) = \tau(\begin{pmatrix} \alpha & 0 & 0 \\ 0 & \alpha & 0 \\ 0 & 0 & \alpha \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix})$$

3.3.2 Transformed Color Histograms

Transformed color histograms are invariant to light intensity changes because the color components are normalized before comparison. For a given set of RGB values (namely, a point cloud) we compute the mean μ as well as the standard deviation σ for each of the three color channels. The normalization proceeds as follows:

$$\tau(\begin{pmatrix} R \\ G \\ B \end{pmatrix}) = \begin{pmatrix} \frac{R-\mu_R}{\sigma_R} \\ \frac{G-\mu_G}{\sigma_G} \\ \frac{B-\mu_B}{\sigma_B} \end{pmatrix}$$

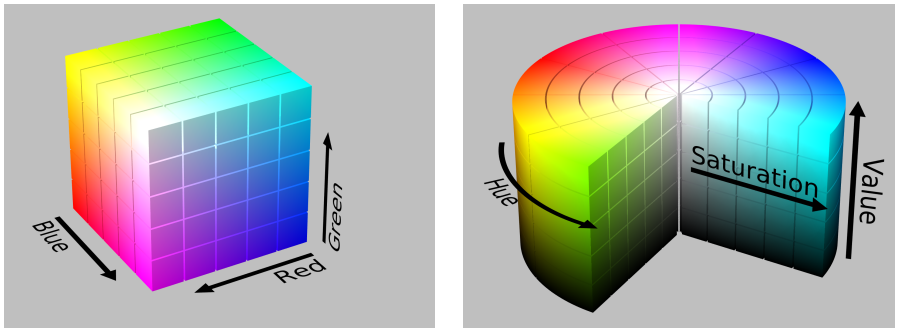
The mean of every channel after the transformation is 0, the standard deviation is 1. Since transformed color values can get exorbitant high if the standard deviation is small, we have to specify a value range which we want to save in our histogram. Examples are $[-3.0; 3.0]$ for saving most of the values, or $[-1.0; 1.0]$ for saving much information per histogram bin, while disregarding $\frac{1}{3}$ of all points. Through the normalization the histogram is not only invariant to light intensity changes, but also to light intensity shifts, which occur due to scattering of light or interreflections [26] and can be specified by an offset that is added to every color component:

$$\tau(\begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} \beta \\ \beta \\ \beta \end{pmatrix}) = \begin{pmatrix} \frac{(R+\beta)-(\mu_R+\beta)}{\sigma_R} \\ \frac{(G+\beta)-(\mu_G+\beta)}{\sigma_G} \\ \frac{(B+\beta)-(\mu_B+\beta)}{\sigma_B} \end{pmatrix} = \begin{pmatrix} \frac{R-\mu_R}{\sigma_R} \\ \frac{G-\mu_G}{\sigma_G} \\ \frac{B-\mu_B}{\sigma_B} \end{pmatrix} = \tau(\begin{pmatrix} R \\ G \\ B \end{pmatrix})$$

Transformed color histograms are also invariant to light color changes. Thus, the descriptor produces the same description for an object under violet light and under red light. These changes can be modeled as multiplying each channel with a different scalar.

$$\tau(\begin{pmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & \gamma \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}) = \begin{pmatrix} \frac{(\alpha \cdot R)-(\alpha \cdot \mu_R)}{\alpha \cdot \sigma_R} \\ \frac{(\beta \cdot G)-(\beta \cdot \mu_G)}{\beta \cdot \sigma_G} \\ \frac{(\gamma \cdot B)-(\gamma \cdot \mu_B)}{\gamma \cdot \sigma_B} \end{pmatrix} = \begin{pmatrix} \frac{R-\mu_R}{\sigma_R} \\ \frac{G-\mu_G}{\sigma_G} \\ \frac{B-\mu_B}{\sigma_B} \end{pmatrix} = \tau(\begin{pmatrix} R \\ G \\ B \end{pmatrix})$$

3.3.3 Hue Histograms



(a) RGB color space (b) HSV color space

Figure 11: (a) shows the RGB color space which is illustrated as a 3-dimensional cube. The HSV color space (b) is a transformation of the RGB space and is illustrated as a cylinder. The diagonal from white to black in the RGB cube is identical to the value axis (V) in the center of the HSV cylinder. The angle around this diagonal corresponds to the hue (H), although, by definition of the HSV space, it is not completely equal. Image source: [27]

The HSV color space A different approach is to transform the RGB value into the HSV color space, which is shown in Figure 11. We are mainly interested in the hue, because it approximates "the degree to which a stimulus can be described as similar to or different from stimuli that are described as red, green, blue, and yellow" [4]. The transformation from RGB to HSV is defined by the following formulas:

$$
\begin{aligned}
V &= \max(R, G, B) \\
C &= V - \min(R, G, B) \\
H &= \begin{cases} \text{undefined} & \text{if } C = 0, \\ 60° \cdot \frac{G-B}{C} \bmod 360°, & \text{if } V = R \\ 60° \cdot \frac{B-R}{C} + 120°, & \text{if } V = G \\ 60° \cdot \frac{R-G}{C} + 240°, & \text{if } V = B \end{cases} \\
S &= \begin{cases} 0, & \text{if } C = 0 \\ \frac{C}{V}, & \text{otherwise} \end{cases}
\end{aligned}
$$

Saturation weighting When we fill the hue values of our point cloud into a histogram, we get a color descriptor that is robust to light intensity changes, as well as to light intensity shifts. The major drawback is that we do not know where to fill in black, white and gray. Since the definition leaves the hue for these colors undefined, we could simply skip these colors and avoid to integrate them into the histogram. But this does not solve

the entire problem: The colors ($H = 120°, S = 1.0, V = 1.0$) and ($H = 120°, S = 0.01, V = 1.0$) would both be classified as green, although we would perceive the first as green and the second as white. The calculated hue is always unreliable for low saturations. To solve this problem, we weight every increment of the histogram with the saturation, as suggested in [26]. Our improved color descriptor is still invariant to light intensity changes, but we have to give up the invariance to light intensity shifts:

$$\begin{aligned} S(\begin{pmatrix} \alpha & 0 & 0 \\ 0 & \alpha & 0 \\ 0 & 0 & \alpha \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}) &= \frac{\max(\alpha R, \alpha G, \alpha B) - \min(\alpha R, \alpha G, \alpha B)}{\max(\alpha R, \alpha G, \alpha B)} \\ &= \frac{\alpha \cdot \max(R, G, B) - \alpha \cdot \min(R, G, B)}{\alpha \cdot \max(R, G, B)} \\ &= \frac{\max(R, G, B) - \min(R, G, B)}{\max(R, G, B)} = S(\begin{pmatrix} R \\ G \\ B \end{pmatrix}) \end{aligned}$$

$$\begin{aligned} S(\begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} \alpha \\ \alpha \\ \alpha \end{pmatrix}) &= \frac{\max(R + \alpha, G + \alpha, B + \alpha) - \min(R + \alpha, G + \alpha, B + \alpha)}{\max(R + \alpha, G + \alpha, B + \alpha)} \\ &\neq \frac{\max(R, G, B) - \min(R, G, B)}{\max(R, G, B)} = S(\begin{pmatrix} R \\ G \\ B \end{pmatrix}) \end{aligned}$$

Because the classification still relies on the hue only, a light intensity shift only scales the histogram. An offset, which is for example produced by a light color change, would be much more drastic in matters of comparison of two hue histograms.

Value threshold The last problem, which I will address, are low color values (V). Similar to a low saturation, which results in white, a low value results in black: ($H = 240°, S = 1.0, V = 1.0$) and ($H = 240°, S = 1.0, V = 0.01$) are both classified as blue, but the second color would be perceived as black by humans and could easily be an imprecision of the camera. We can not weight the histogram with the value this time, because this would give up the invariance to light intensity changes. Instead, I added a value threshold to the color descriptor. We only consider a point if the value of the point's color is above this threshold. One could argue that this also gives up the invariance to intensity changes, but since this holds only for intensity changes that result in a very dark scene anyway, the advantages of a threshold outweigh the disadvantages.

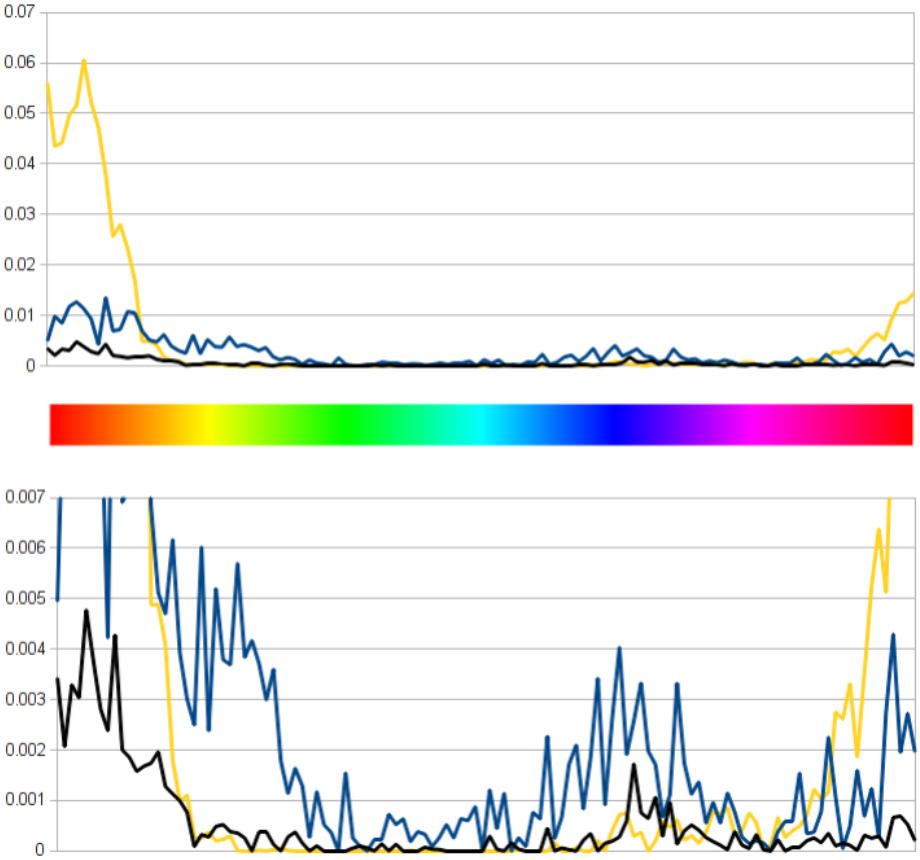


Figure 13: The figure shows three histograms with 120 bins each. The x-axis denotes the hue range, the y-axis denotes the number of occurrences of the respective hue weighted with the saturation. The bottom figure shows the same histograms, zoomed in with factor 10, to be able to distinguish the histograms in areas with low values.

Example To give an example, I created hue histograms for the front view of the three objects shown in Figure 12. I chose objects with very different colors to create three distinct histograms. The results are shown in Figure 13. The yellow line, which has a huge peak in the red area, represents object (a). The histogram of the green object with a blue label (b) is drawn with a blue line. Though it also has it's peak in the red range, the values in the green and blue range are higher than the values of the other two objects. Object (c) is drawn black

Figure 12: Three objects with different colors: (a) red and yellow, (b) green and blue, (c) black

and has in general small values compared to the other objects, since the saturation of black is low. It might surprise that the red region dominates for all objects, although only one of them is partly red. The explanation for this is not obvious, but simple: The used light source, a standard light bulb, does not produce pure white light, but instead light that has a higher red component compared to the green and blue component. Therefore, the saturation of points that are not red can not be as high as the saturation of red points. Also, white and black points whose values exceed our threshold are classified as red, even though with a low saturation. This situation is similar to the Fast Point Feature Histogram, where small angles occur more often than large angles. But it is not necessarily a problem if our algorithm for histogram matching is designed to compensate these imbalances. I will discuss the histogram matching algorithm in section 3.5.

3.4 Object Descriptor Selection

As mentioned earlier, I will combine a 3-dimensional surface descriptor with a color descriptor. I will not use an interest point extractor at all, although the winner teams of the RoboCup@Home 2009 [24], 2010 [17] and 2011 [10] used SURF or a similar descriptor for their object recognition. But the RoboCup@Home does not focus on object recognition, and therefore this is no evidence that SURF is the best known algorithm for object recognition. Since we already extracted the isolated point cloud of an object, an interest point detection seems inappropriate.

As a surface descriptor I will use Fast Point Feature Histograms. I will not use Viewpoint Feature Histograms since the pose information is only useful if the robot has to grasp an object at a certain point of the object, e.g. the handle of a mug. If the robot features only a clumsy arm and grasps all objects by simply approaching them from two sides, the exact pose information is irrelevant.

For the color part I will use hue histograms. While transformed color histograms are, in opposition to hue histograms, invariant to light color changes, their discriminative power is much smaller. They are even unable to detect the differences between an entirely blue and an entirely green object. Therefore, hue histograms are better suited for our object recognition purpose.

Since the hue histogram and the FPFH are both histograms, we can combine them to a single feature histogram by concatenation. The weighting between color and surface descriptor can be regulated via the number of bins and will be discussed in the section that presents my implementation.

3.5 Feature Matching

Now that we decided how to extract a feature histogram from a point cloud of an object, we need an algorithm for matching two histograms, in order to be able to compare a detected object with the objects we learned in the training phase. Since we can write each histogram as a feature vector $(a_1, a_2, ..., a_m)$, the first approach would be to calculate the difference between two feature vectors $\vec{a}$, $\vec{b}$ with the Euclidean metric:

$$d(\vec{a}, \vec{b}) = \|\vec{a} - \vec{b}\| = \sqrt{\sum_{i=1}^{m} (a_i - b_i)^2}$$

But we have seen in Figure 10 and Figure 13 that some bins of the histogram use to have much higher values than other bins, e.g. the red component is more present in the hue histogram than the blue component. Therefore, the Euclidean distance is mostly determined by these few bins. In order to avoid this, we employ the Chi-square metric, which normalizes each bin. In our case, when all values are non-negative, the Chi-square distance is estimated as follows:

$$d_{\text{Chi}}(\vec{a}, \vec{b}) = \sqrt{\sum_{i=1}^{m} \frac{(a_i - b_i)^2}{a_i + b_i}}$$

[18] compares various other distance metrics in context of the Fast Point Feature Histogram. However, the experimental setup is not presented thoroughly and the results are counter-intuitive, because the Manhattan metric is found to have the best accuracy.

The matching of m-dimensional feature vectors can be performed with a m-dimensional k-d tree. Therefore, the feature vectors of all objects learned in the training phase are inserted into such a tree. In order to match a detected object with the trained objects, we perform a k-nearest-neighbor search on the tree with the object's feature vector. Regarding the implementation, the FLANN library mentioned earlier also supports the Chi-square distance metric.

4 Implementation

I implemented the presented algorithms for object detection and recognition as a part of the autonomous wheelchair project. This project is based on the Open Robot Control Software (OROCOS), which organizes all parts of the software in modules. An OROCOS module is basically a C++ class that utilizes an input and output port concept for a smarter management of the data flow between modules. My implemented software includes four OROCOS modules as well as two additional C++ classes. Figure 14 provides an overview of all of them as well as their relationships between each other. This section gives a description of their interfaces, properties, duties and usage of libraries.

4.1 Utilized Libraries

Point Cloud Library (PCL) This library was already mentioned in section 1.1. PCL is extensively used in my implementation since it offers many implementations of algorithms I talked of in the previous sections and without which the software could not have been developed in context of a Bachelor thesis.

Fast Library for Approximate Nearest Neighbors (FLANN) As stated in section 2.2.2, I use this library to estimate the k-nearest-neighbors of a point. FLANN is also utilized and therefore required by PCL.

boost The boost library is a well-known C++ library that provides platform independent implementations of various data structures and algorithms. I will use the library for file system access and shared pointers, which simplify the pointer management. Since PCL also uses shared pointers, boost is also a requirement for PCL.

Mobile Robot Programming Toolkit (MRPT) This library is used by our robot project for various tasks and will especially provide access to the Kinect device.

4.2 RangeScanToPointCloud Module

- **Input port: RangeScanIn**
  Expects Kinect data of type `CObservation3DRangeScanPtr` as provided by `modules.io.mrpthardware.MrptKinect`.
- **Output port: PointCloudOut**
  Provides a 3-dimensional PCL `PointCloud` of the given Kinect data with color information (`PointXYZRGBA`).

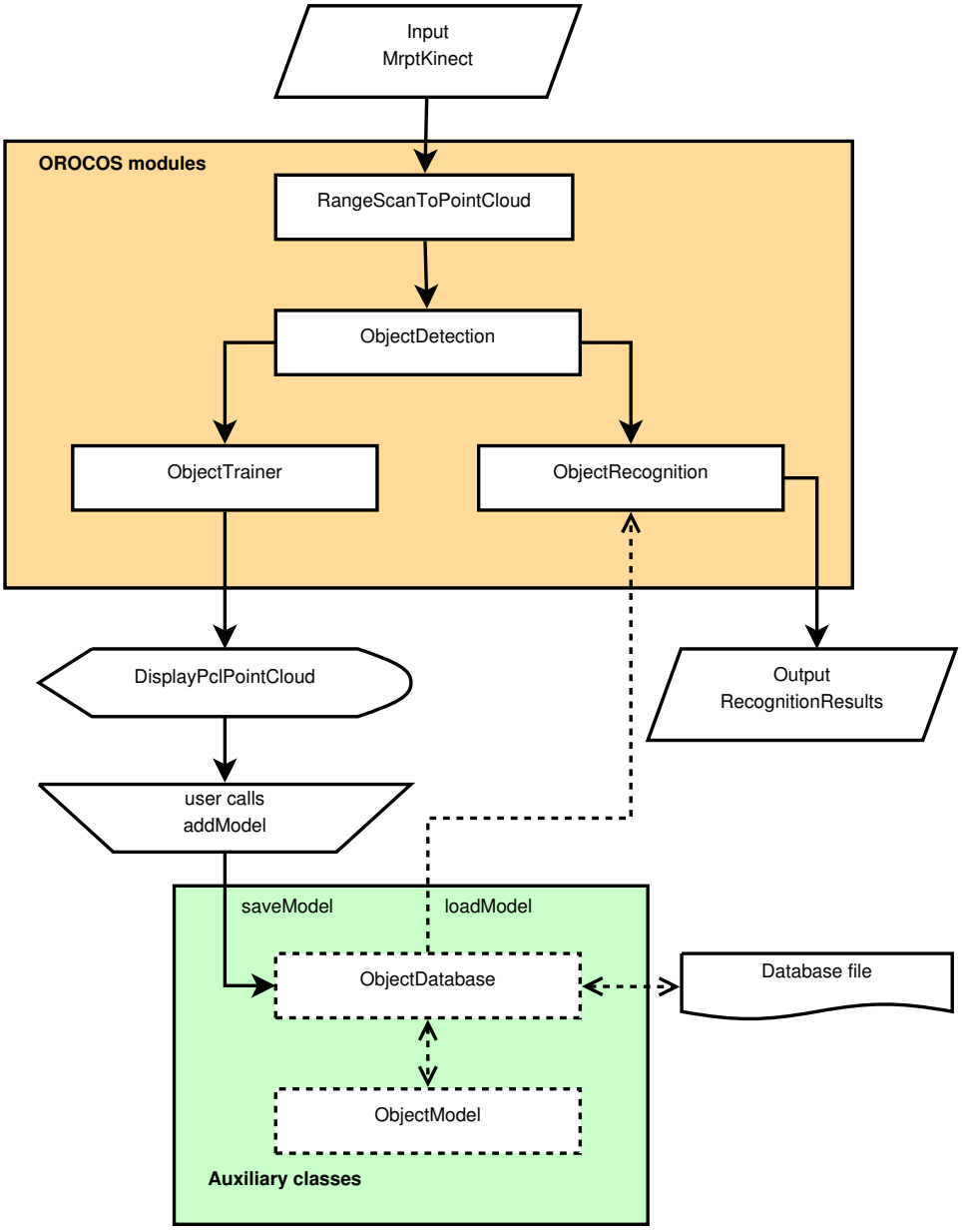


Figure 14: The flowchart shows the relationships between all OROCOS modules involved in the detection and recognition process. The four modules in the orange area and the two auxiliary classes are implemented by me, in contrast to the MrptKinect and DisplayPclPointCloud modules which were already part of the project. In the training phase, the ObjectTrainer module displays the currently detected object to the user. When the detected and displayed object is the one that the user wants to train, he calls the addModel method from the command line interface. This causes the ObjectTrainer module to extract the feature histogram of the object and to save it to an object database. During testing phase, the ObjectRecognition module compares all detected objects with the objects loaded from the object database, and saves the results to a RecognitionResults structure.

The task of this module is a conversion of a depth and rgb image to a 3-dimensional point cloud. MRPT already offers a conversion of the range scan to a MRPT point cloud, called `CColouredPointsMap`. The MRPT point cloud is then copied to a PCL `PointCloud` point by point. MRPT also offers a direct conversion to PCL via `project3DPointsFromDepthImageInto`, but this method is only available if MRPT has been compiled with PCL support, which we can not assume at the moment.

4.3 ObjectDetection Module

- **Input port: PointCloudIn**
  Expects a point cloud of the scene in which the tables and objects should be detected.

- **Output port: DetectedObjectsOut**
  Provides a vector of point clouds where each cloud contains the points of a detected object.

- **Output port: PointCloudOut (optional)**
  Provides the input point cloud plus a green convex hull drawn around detected tables and a red cuboid around detected objects. This output port needs to be enabled through the `EnablePointCloudOut` property, otherwise the module will not use this port in order to save computation time.

The ObjectDetection module implements the procedures for table and object detection that were described in section 2. Extensive use of the algorithms implemented in PCL is made throughout the module. The following list enumerates all steps of the detection process:

1. Remove points that are too low or too high to be a table, i. e. out of the range of the robot arm. This range should be defined by the user through the `HeightFilterMin` and `HeightFilterMax` properties.

2. If `PointCloudOut` is enabled, copy the input cloud to a visualization cloud.

3. Downsample the point cloud with the `VoxelGrid` class.

4. Estimate the normals for the downsampled cloud, using `NormalEstimation` and the ten nearest neighbors.

5. Find all points that have an angle between their normal vector and the z-axis that is smaller than the angle defined in the `TableDetectionMaxAngle` property. Mark these points as table candidate points.

6. Extract table clusters from the table candidate points by Euclidean clustering. Use PCL's `EuclideanClusterExtraction` and a `KdTree` for this step.

7. For each table cluster perform the following steps:

   (a) Estimate a plane model with the RANSAC algorithm that fits the table cluster. Stop the algorithm as soon as 99% of the points are considered lying in the plane, with a user specified threshold, or a fixed number of iterations is reached. Utilizes the `SACSegmentation` class.

   (b) In preparation for the next step, use `ProjectInliers` to project all found inliers to the plane model in order to get a point cloud that fits that model perfectly.

   (c) Calculate the 2-dimensional convex hull of the table with the `ConvexHull` class.

   (d) Discard the table if it does not have a minimum width, minimum length, or minimum number of points as defined by the user. The width does not refer to the real width of the table, but to the width of the projection to the y-axis instead. This calculation is much faster while providing results that are accurate enough. Analog, the length refers to the width of the projection to the x-axis.

   (e) If `PointCloudOut` is enabled, draw the convex hull of the table to the visualization cloud.

   (f) Object detection: Apply the `ExtractPolygonalPrismData` algorithm to get all points above the convex hull of the table.

   (g) Use the `ProjectInliers` class again to project these points to the plane model of the table.

   (h) Find object clusters in the (2-dimensional) point cloud of projected points with `EuclideanClusterExtraction`.

   (i) For each object cluster perform the following steps:

      i. Calculate the 2-dimensional convex hull of the object cluster.

      ii. Extract all points above this convex hull in order to get the final point cloud of the object. This time we use the original input cloud read from the input port instead of the downsampled version.

      iii. Discard the object if it's height is smaller than a threshold. The height is calculated by projection to the z-axis.

      iv. Discard the object if the distance between the object and the table exceeds another threshold. These objects are most likely human hands during the object training.

v. If `PointCloudOut` is enabled, draw a cuboid around the object. Note that all lines of the cuboid are parallel to the x-, y- or z-axis to speed up computation and implementation.

8. Write a vector of all detected objects to the output port, as well as the visualization if requested.

Properties Most of the constants and thresholds in the detection process can be modified by the user via the following 15 properties, which can be changed at runtime. Note that the module expects all lengths to be in meter, although they are listed in centimeter here.

- **DownsamplingResolution** Default value: 1cm
  Minimum distance between any two points after downsampling. This value has a huge impact on the performance of the detection, therefore it is important to select this value according to the needs regarding recognition quality and consumed computation time. The value is passed on to the `setLeafSize` method of PCL's `VoxelGrid` class.
- **HeightFilterMin** Default value: -60cm
  Minimum z-position of tables relative to the Kinect. If the value is zero, only tables above the Kinect are detected.
- **HeightFilterMax** Default value: 0cm
  Maximum z-position of tables relative to the Kinect. If the value is zero, only tables below the Kinect are detected.
- **EnablePointCloudOut** Default value: false
  Whether to write the visualization point cloud to the `PointCloudOut` output port.
- **TableDetectionMinWidth** Default value: 20cm
  Minimum width and length of a table.
- **TableDetectionMinPoints** Default value: 10
  Minimum number of points of a detected table.
- **TableDetectionMaxAngle** Default value: 10°
  Maximum angle of a table point's normal vector relative to the z-axis in degree.
- **TableDetectionTolerance** Default value: 5cm
  Minimum distance between two points that are not considered to be in the same table cloud.
- **TableDetectionThreshold** Default value: 1cm
  Maximum distance for a point to be considered an inlier of a plane model during the RANSAC algorithm for table detection.

- **TableDetectionMaxIterations** Default value: 100
  Maximum number of iterations for the RANSAC algorithm to terminate.
- **ObjectMinPoints** Default value: 10
  Minimum number of points of a detected object point cloud.
- **ObjectMinHeight** Default value: 6cm
  Minimum height of an object.
- **ObjectClusteringTolerance** Default value: 5cm
  Minimum distance between two points that are not considered to be in the same object cluster.
- **ObjectDistanceToTableMin** Default value: 1cm
  Minimum distance from an object to the table. This value should not be too small, otherwise some points of the table surface might be taken as objects.
- **ObjectDistanceToTableMax** Default value: 1m
  Maximum distance from the table to each point of the object.

4.4 ObjectModel Class

The ObjectModel class takes the point cloud of an object, as acquired by the Kinect and segmented by `ObjectDetection`, and extracts the object's feature histogram. Furthermore, it saves additional data like the object's name. Because every object needs to be trained from multiple perspectives, we will have various object models for the same object.

Feature extraction As described in section 3.4, we calculate a histogram that consists partly of a Fast Point Feature Histogram (FPFH) and partly of a hue histogram. The FPFH could be extracted with PCL's `FPFHEstimation`, but this class produces a histogram with only 33 bins, which might not be enough for our purpose. Instead, we use the `VFHEstimation` which procedures a histogram with 308 bins. Since we decided not to use the viewpoint information for the moment, we extract only the 135 bins that represent the FPFH. I selected 120 as the number of bins of the hue histogram because 120 is a denominator of 360 and is in about the same range as 135. However, no evaluation has been done to find the optimal number of bins because there is no obvious metric to estimate that number. The concatenated histogram has 255 bins and is saved in a vector of 255 floats (the feature vector).

Additional data In order to reassign an object model to an object, we save the name of the object as a string. The user can select this name freely, although it will usually be a short description of the object, e.g. `mug` or `bottle`. In future applications the user might want to save additional information about the object. Therefore, we save another string `metainfo` which can be filled by the user. For example, it might be decided that we also need to recognize the pose of the object. In this case the rotation could be saved to the `metainfo` string during training and utilized after recognition.

4.5 ObjectDatabase Class

The ObjectDatabase class offers methods for saving and loading object databases, which are nothing else than sets of object models. It is not sufficient to have only one large object database, because we do not want the program to recognize objects that we trained only for earlier experiments. I chose a file format that allows investigation and modification by a human-being quickly, rather than a binary representation that gives a small speed up.

Directory structure All databases are saved to a directory that is called `objectrecognition` and can be found in the `resources` folder of the project. In order to locate this folder automatically, the working directory has to be either the root of the project (usually `autoauto`) or any subdirectory of it. Alternatively, the `ObjectTrainer` and `ObjectRecognition` modules offer a property called `AutoAutoPath` which takes the path to the root directory. Every database has it's own folder in the `objectrecognition` directory. The name of the folder is the name of the database, thus you can rename a database by renaming the folder.

File format The object models are saved in simple text files with the extension `.mdl`. Every `mdl`-file can contain an arbitrary number of object models, although the object trainer saves an object model to the file of the respective object, i. e. models for an object with the name "red mug" are saved to `red mug.mdl`. However, this is not mandatory and you can copy all models of the database to a single file, as long as the extension is `.mdl` and the structure of each model in the file is the following:

- The first line starts with a # and informs about the date and time the object model was acquired.
- The second line contains the object name.
- The third line contains the `metainfo` string.
- The following 255 lines contain floating point numbers, each representing one bin of the feature histogram.

4.6 ObjectTrainer Module

- **Input port: DetectedObjects**
  Expects the point clouds of all detected objects as provided by the `ObjectDetection` module.
- **Output port: PointCloudOut**
  Offers the point cloud of a single object.

During the training stage, we want to learn all objects one by one. Therefore, we usually provide a clear scene where only the one object we want to learn is present. However, the `ObjectTrainer` module simplifies this by selecting the object that is nearest to the Kinect, thus ignoring any object in the background. Before processing the actual training algorithm, the point cloud of the selected object is shown to the user via the `DisplayPclPointCloud` module. When the user is satisfied by the shown point cloud, i.e. the right object has been detected correctly, he manually starts the training by calling the `addModel` method from the command line interface.

addModel method The `addModel` method requires three arguments: The database to which the model should be saved to, the name of the object which is being trained, and a metainfo string which can be empty. For example:

```
addModel("experiment-2011-12-24", "red mug", "front view")
```

The name and metainfo strings are filled in an instance of the `ObjectModel` class that also extracts the feature histogram from the point cloud. The object model is then saved to the specified database with the `ObjectDatabase` class.

4.7 ObjectRecognition Module

- **Input port: DetectedObjects**
  Expects the point clouds of all detected objects as provided by the `ObjectDetection` module.
- **Output port: RecognitionResults**
  Offers the results of the recognition algorithm in the format that is specified in section 5.3.

The `ObjectRecognition` module takes the detected objects from the `ObjectDetection` module and matches each object with the objects from the object database. The first step of the recognition process is to load the objects from the object database which is specified in the `Database` property. In the next step we build an index as discussed in section 3.5 by utilizing

the `Index` data structure from the FLANN library. The library also provides the distance metric `ChiSquareDistance`. Of course, we do not want to repeat the loading and index creation for every new point cloud, and therefore cache the database and index as long as the user does not change the `Database` property. In order to save computation time, we could also build the index after we finished the training instead of building it at every start of the program, but since the index creation takes only fractions of a second, even for hundreds of object models, I passed on the additional manual operation that would be required after training.
Next, we extract the feature histogram using the `ObjectModel` class for each detected object. Then we perform a k-nearest-neighbours search on the index, where k is defined by the user through the property `HypothesisCount`. We get k object models, which are possible matches, together with the corresponding distances to the analyzed object. These are filled in a `RecognitionResults` structure and written to the output port.

Properties

- **HypothesisCount** Default value: 5
  Number of recognition hypotheses to write to the output port for each detected object, i. e. the number of object models that fit the detected object best are listed in the recognition results. Note that the number of listed object models is not necessarily equal to the number of listed objects, i. e. you could get five different object models of the same object.

- **Database** Default value: ”default”
  The name of the database that should be used for the comparison of objects. At the start of the program, the objects are loaded from the database in the ”default” directory.

- **AutoAutoPath** Default value: ”.”
  The path to the project root or any subdirectory of the project root. This property is used to locate the `resources` directory.

Methods The module offers the methods `printRecognitionResults` and `performExperiment` which can be called from the command line interface to test the recognition functionality. The usage of these methods is described in the section about testing the object recognition (5.2).

5 Using the Implementation

While the previous section has described how the implemented software works, this section will provide a step by step guide on how to use the modules in context of the OROCOS project.

5.1 Training Objects

In order to train an object, create a simple scene with only one table and put the object on it. There should not be any other object in the foreground. Place the Kinect or the whole robot in a spot nearby and start `objectTrainer.xml` with the `xmltest` program from a terminal. For example:

```
$ cd autoauto/build/tests/xmltests/
$ ./xmltest -L 5 xml/vision/objectrecognition/objectTrainer.xml
```

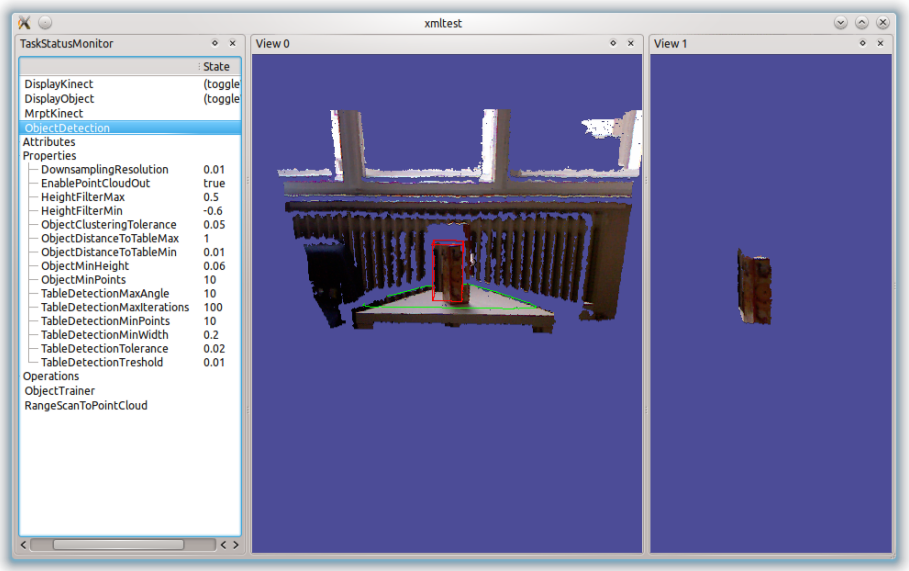


Figure 15: Screenshot of the output produced by `objectTrainer.xml`. (a) The `TaskStatusMonitor` on the left side of the window can be used to modify the properties of the `ObjectDetection` module. (b) Changes are applied to `View 0` immediately, which displays the scene captured by the Kinect. Detected tables and objects are marked and the mouse can be used to explore the 3-dimensional point cloud. (c) `View 1` displays the detected object that is nearest to the Kinect as it will be used for the training.

The created window should look like Figure 15. If the table or object is not detected, you probably have to change the position of the Kinect. You might also modify the properties of the `ObjectDetection` module, which are listed in section 4.3. As soon as the desired object is detected and

displayed correctly, you can train it by calling the `addModel` method from the command line, e. g. :

```
addModel("experiment-2011-12-24", "red mug", "rotation:  0")
```

Since each object has to be trained from each side, rotate the object, wait until it is correctly detected, and call `addModel` again. The rotation should not be greater than 30°, which would result in 12 models of the object after training from each side. Of course, the more models you create, the better the recognition results will be from various viewpoints. All object models are written to the `resources` directory, where you can edit the object databases and models, as discussed in section 4.5.

5.2 Testing the Object Recognition

To test the `ObjectRecognition` module, run `testRecognition.xml`:

```
$ cd autoauto/build/tests/xmltests/
$ ./xmltest -L 5 xml/vision/objectrecognition/testRecognition.xml
```

Similar to `objectTrainer.xml` the acquired scene is displayed with a green polygon drawn around detected tables and a red box around detected objects. Now you know which objects are detected, but if you want to know how these objects are recognized, you have to call the `printRecognitionResults` method from the command line. The results are then written to the standard output as follows:

```
*** detected 2 objects ***
object #1 located at (0.960104m, -0.0157443m, -0.0149633m)
  #46 red mug (rotation: 30) 23.1993
  #52 red mug (rotation: 60) 36.6365
  #41 red mug (rotation: 0) 44.7708
  #47 red mug (rotation: 210) 51.6287
  #36 red mug (rotation: 240) 77.4286
object #2 located at (0.976374m, 0.215263m, -0.151597m)
  #108 bottle () 148.682
  #146 speakerbox (some metainfo) 158.589
  #109 bottle () 162.628
  #85 red mug (rotation: 30) 164.791
  #82 red mug (rotation: 0) 166.643
```

The position of each object (`located at...`) is determined by one random point and is only printed for your information, so you can reassign the objects to the scene. Each hypothesis is printed in the following format:

```
  #ID objectname (metainfo) distance
```

where `ID` is the internal ID of the object model in the database and `distance` is the calculated Chi-square distance between the object model and the detected object. In the example above, the object `red mug` was assigned with viewpoint information during the training stage, thus we can also recognize the rotation of the detected object.

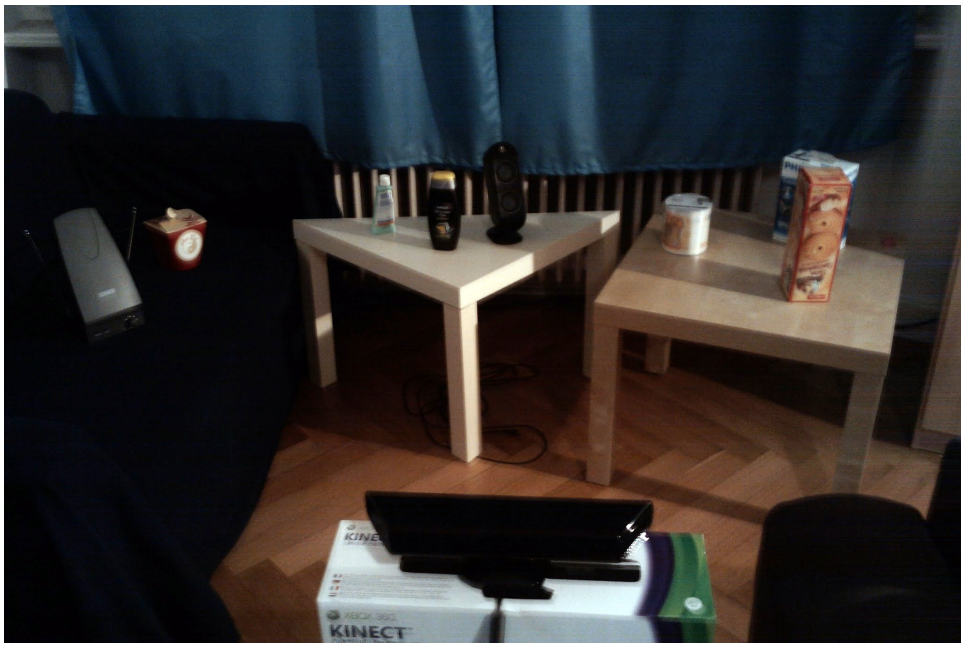
5.3 Using the Recognition Results

The output port of the `ObjectRecognition` module is of type `RecognitionResults`. The data type consists of a vector of all object models from the object database with their names and meta information, as well as all detected objects with their point clouds and their recognition hypotheses including the calculated distances. If you want to use these results in your OROCOS module, you have to include `RecognitionResults.h`, where the data type is defined in the following way:

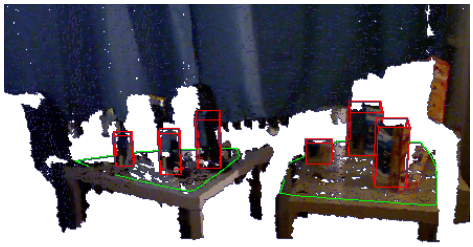
```
struct ObjectModelInfo {
    std::string name;
    std::string metainfo;
};
struct Hypothesis {
    int model; // index of RecognitionResults.models
    float distance;
};
struct DetectedObject {
    pcl::PointCloud<pcl::PointXYZRGBA>::Ptr pointCloud;
    std::vector<Hypothesis> hypothesis;
};
struct RecognitionResults {
    boost::shared_ptr<std::vector<ObjectModelInfo> >
        models;
    boost::shared_ptr<std::vector<DetectedObject> >
        detectedObjects;
};
```

6 Evaluation

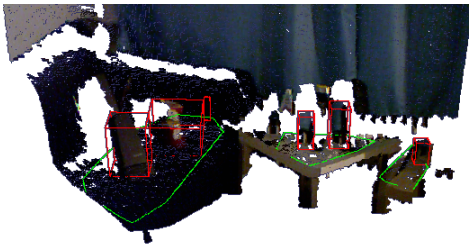
6.1 Object Detection



(a) A table and object detection test setup



(b) Detection results for the right part



(c) Detection results for the left part

Figure 16: Detection results for a scene with 8 objects on 3 different surfaces. Tables are bordered green and objects are bordered red.

In order to test the object detection module, I set up a test scene as shown in Figure 16(a), which consisted of three plane surfaces, two tables and a couch, and placed several objects on them. The detection results are split in two parts 16(b) and (c) because the field viewing angle of the Kinect was too small for this setup and distance. While all visible tables and objects are detected correctly in 16(b), there are three detection errors in 16(c):

1. The box drawn around the second object on the couch is too large. The only possible reason for this is that some of the (blue) points

of the couch are considered to be part of the object. Recalling the algorithm for the extraction of objects on plane surfaces, we know that these points are too high above the estimated plane model. Thus, the error probably occurred due to the surface of the couch, which is not precisely plane, and therefore the estimated plane model was not high enough.

2. A small part of the arm rest is detected as a third object on the couch. Since we have no complete 3-dimensional point cloud of the scene, but a "2.5-dimensional" cloud with only a front view of each object available, it is hard to distinguish between the arm rest and an object. Our algorithm is not capable of identifying these false positive without additional a priori information.

3. The leftmost object on the white table is not detected. As you can see from the convex hull (green), this is because the table is not detected entirely. Especially in the undetected region of the table, there are many "holes" in the point cloud of the table's surface. The Kinect did not perceive this region correctly.

However, this has only been an analysis of a single frame, and you should monitor the detection results in real-time to get an impression of the detection's accuracy. Even when we do not change the scene at all, the results vary from frame to frame. This is due to the highly noisy data that we receive from the Kinect. The three mentioned errors in Figure 16(c) are all fluctuating. This emphasizes that the actions of our robot should not be based on the detection results of a single frame. Indeed, if the robot is able to evaluate the results of multiply detection attempts, we can achieve very accurate results. Nevertheless, even the object detection of a single frame seems to be properly enough for a typical RoboCup@Home scenario.

6.2 Object Recognition

6.2.1 Test Setup

Figure 17: Test set of 10 objects

In the scheme of the RoboCup@Home "Go Get It!" test, I trained the program with the 10 objects shown in Figure 17, which clearly differ in color and structure. I created a simple setting for the training with only one table and one object on it (Figure 18). After an acquisition of an object model, I rotated the object by approximately 22.5°, thus creating 16 models per object. I used the same scene for the testing stage.

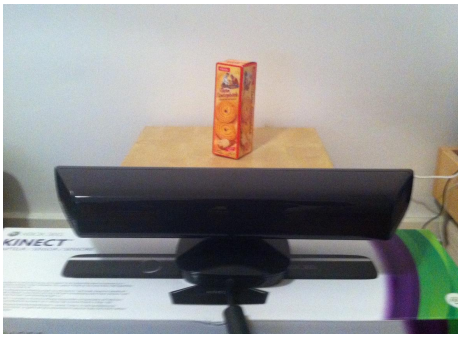


Figure 18: Setup for the test

6.2.2 Test Results

I tested the recognition of each object from 10 different, pseudo-randomly selected angles, and the results are represented in Table 1. 8 of the 10 objects were recognized correctly from each angle. The other two objects were also predominantly recognized correctly, thus resulting in an overall accuracy of 94% out of 100 tests. All mistaken identities occurred when either object #3 or object #8 was tested from a side view, where the point clouds of the objects are very slim. Therefore, the accuracy appears to be low when the analyzed point cloud has relatively few points. I will give a possible explanation for this later.

Object	correct		wrong	
	count	avg. distance	count	avg. distance
#1	10	20.3	0	—
#2	10	25.3	0	—
#3	8	29.3	2	50.4
#4	10	21.6	0	—
#5	10	20.4	0	—
#6	10	10.4	0	—
#7	10	26.5	0	—
#8	6	43.6	4	51.0
#9	10	19.2	0	—
#10	10	18.9	0	—

Table 1: Test results

However, it might even be possible to distinguish correct and wrong matches by the Chi-square distance. The average distance of incorrectly identified objects was about 50, while the average distance of correctly recognized objects was below 30. In this experiment we could have detected all recognition errors through the definition of a threshold at 40, as long as we would not care about the marginal results for object #8. But we should operate with fixed thresholds carefully, because the Chi-square distance might increase with the Euclidean distance between the object and the Kinect. We stated earlier that our features are invariant to scale, but in practice a change in scale also means a change of the viewing angle, because the Kinect has a fixed high above the ground. To achieve a true distance-invariance, the object training has to be done from different distances. In either case, the introduction of a recognition threshold requires further research.

6.2.3 Advanced Test Set

Inspired by the sound results of the recognition test, I repeated the same experiment with a seemingly harder test set, whose objects are more similar to each other. I used the 8 mugs pictured in Figure 19 and received the results shown in Table 2. The overall accuracy was even slightly better this time with a 95% ratio. Only #7 and #8 are occasionally confused due to the fact that their color distributions and surface structures are very similar. In this case it is unlikely that wrong results can be discriminated by the Chi-square distance, because the average values are about the same. Furthermore, the hue histogram seems to serve its purpose, since the mugs #5 and #6 are completely alike except for their colors.

Figure 19: Advanced test set of 8 mugs

Object	correct		wrong	
	count	avg. distance	count	avg. distance
#1	10	35.9	0	—
#2	10	23.9	0	—
#3	10	21.4	0	—
#4	10	20.4	0	—
#5	10	25.7	0	—
#6	10	26.8	0	—
#7	7	21.2	3	25.7
#8	9	27.0	1	29.4

Table 2: Test results for the advanced test set

6.2.4 False Positives

A very important part of the ”Go Get It!” test is the presence of objects that are unknown, because it is a serious problem if a robot recognizes an unknown object as a known object. Therefore, I tested the recognition with five different, untrained objects, each from six different angles. The average Chi-square distance to the best matching object model was 97.7, while the previously presented experiments resulted in an average distance of 23.8 for correctly identified objects. This strongly suggests the introduction of a threshold for the detection of false positives as discussed in section 6.2.2. Still, if such a threshold is indeed needed, the threshold estimation should be preceded by further research about the distance distribution. This requires multiple experiments with various distances, which can be done with a robot much easier, and therefore are not part of this thesis.

6.2.5 Analysis

Although 180 is still a relatively small number of tests, my implementation seems to have an recognition accuracy of over 90% for a set of distinctive objects. Yet, the accuracy ratio of 98.5%, which was claimed in [20] for an even harder test set, could not be reached. The following paragraphs provide possible explanations for this.

The authors of [20] used an electric turn table in order to acquire approximately 900 object models per object automatically. In contrast to the 16 manually acquired object models of my experiment, this results in more exact object model matches. Additionally, the depth sensor of the Kinect, which utilizes an infrared laser, produces even noisier data than the stereo camera used in their experiment.

Furthermore, the output of the Kinect is a RGB-image as well as a depth-image, both produced by different sensors. In order to merge these images into a single 3-dimensional point cloud, a calibration has to be done, i. e. the exact distances between the two sensors as well as other properties must be known. Figure 20 illustrates that the integrated Kinect calibration of the MRPT library does not yield sufficient results. Although the surface of the mug is detected correctly, many points are associated with the color of the background. Since this is a serious problem for the used color descriptor, it might explain the errors of recognition which occurred in the first test, when the point clouds were relatively small and thus led to a large percentage of points with the wrong color.

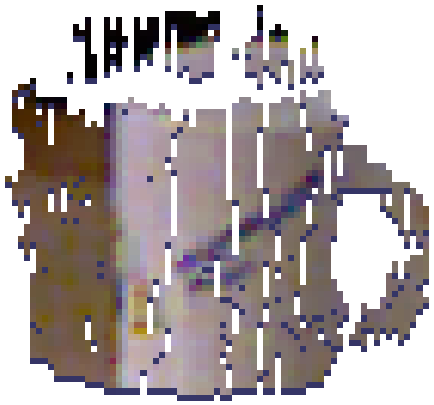


Figure 20: Point cloud of a mug

7 Conclusion and Future Work

In this thesis, I presented a system that detects objects in a 3-dimensional point cloud and matches these objects with previously trained object models. The next step towards a solution to the "Go Get It!" test should be to provide my recognition system with a point cloud that is built of multiple acquisition frames of the Kinect instead of just one. Thus, all acquired point clouds of a certain timespan should be merged into a single point cloud of the environment. The required technique for this is called simultaneous localization and mapping (SLAM). Sufficient real-time algorithms for this task already exist, KinectFusion [14] for example, and are even implemented in the Point Cloud Library. The table and object detection in such a merged point cloud would be more reliably and could be done by the presented software without code changes. Furthermore, the noise in the point cloud could be reduced with the statistical outlier removal algorithm that is already available in the Point Cloud Library.

Since my recognition module does only produce recognition hypotheses, but does not state whether an object is really one of the known objects or not, the recognition results have to be analyzed in order to estimate the action that the robot should perform. An approach for the RoboCup@Home would be to get simply the four objects that have the smallest distance to their respective object model. However, if the number of known objects in the setting is unknown, it has to be done further research in order to estimate a Chi-square distance threshold that can distinguish whether a detected object is a trained object or not.

The presented system can be used in real-time applications, since the analysis of a single point cloud takes approximately 500ms. This duration can be influenced by the various properties which the user can easily adjust. In conclusion, the evaluation has shown that the system's performance is accurate enough for the recognition tasks of the RoboCup@Home.

References

[1] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool, *Surf: Speeded up robust features*, In ECCV, 2006, pp. 404–417.

[2] Jens Berkmann and Terry Caelli, *Computation of surface geometry and segmentation using covariance techniques*, IEEE Transactions on Pattern Analysis and Machine Intelligence **16** (1994), no. 11, 1114–1116.

[3] H. Cantzler, R. B. Fisher, and M. Devy, *Quality enhancement of reconstructed 3d models using coplanarity and constraints*, 2002.

[4] Mark Fairchild, *Color appearance models: CIECAM02 and beyond (tutorial slides)*, IS&T/SID 12th Color Imaging Conference, 2004.

[5] Martin A. Fischler and Robert C. Bolles, *Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography*, Commun. ACM **24** (1981), 381–395.

[6] T. Gevers and A. Smeulders, *Color based object recognition*, Image Analysis and Processing (Alberto Del Bimbo, ed.), Lecture Notes in Computer Science, vol. 1310, Springer Berlin / Heidelberg, 1997, pp. 319–326.

[7] Dirk Holz, Stefan Holzer, Radu Bogdan Rusu, and Sven Behnke, *Real-time plane segmentation using rgb-d cameras*, RoboCup Symposium, 2011.

[8] @Home Technical Committee, *RoboCup@Home Rules and Regulations*, RoboCup@Home Official Website, 2011.

[9] Andrew E. Johnson and Martial Hebert, *Using spin images for efficient object recognition in cluttered 3d scenes*, IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE **21** (1999), no. 5, 433–449.

[10] Kathrin Gräve Dirk Holz Michael Schreiber Sven Behnke Jörg Stückler, David Dröschel, *NimbRo@Home 2011 Team Description*, Proceedings CD RoboCup 2011 (Istanbul, Turkey), 2011.

[11] Klaas Klasing, Daniel Althoff, Dirk Wollherr, and Martin Buss, *Comparison of surface normal estimation methods for range sensing applications.*, ICRA'09, 2009, pp. 3206–3211.

[12] David G. Lowe, *Object recognition from local scale-invariant features*, Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2 (Washington, DC, USA), ICCV '99, IEEE Computer Society, 1999, pp. 1150–.

[13] Marius Muja and David G. Lowe, *Fast approximate nearest neighbors with automatic algorithm configuration*, International Conference on Computer Vision Theory and Application VISSAPP'09), INSTICC Press, 2009, pp. 331–340.

[14] Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohli, Jamie Shotton, Steve Hodges, and Andrew W. Fitzgibbon, *Kinectfusion: Real-time dense surface mapping and tracking.*, ISMAR, IEEE, 2011, pp. 127–136.

[15] Carlos Nieto-Granda, `http://www.cc.gatech.edu/~cnieto6/research.html`, 2012-01-09.

[16] Andreas Nüchter and Joachim Hertzberg, *Towards semantic maps for mobile robots*, Robot. Auton. Syst. **56** (2008), 915–926.

[17] Omori T. Iwahashi N. Sugiura K. Nagai T. Watanabe N. Mizutani A. Nakamura T. Attamimi M. Okada, H., *Team eR@sers 2010 in the @Home League Team Description Paper*, Proceedings CD RoboCup 2010 (Singapore), 2010.

[18] Radu Bogdan Rusu, *Semantic 3d object maps for everyday manipulation in human living environments*, Ph.D. thesis, Computer Science department, Technische Universitaet Muenchen, Germany, October 2009.

[19] Radu Bogdan Rusu, Nico Blodow, and Michael Beetz, *Fast point feature histograms (FPFH) for 3d registration*, The IEEE International Conference on Robotics and Automation (ICRA) (Kobe, Japan), 05/2009.

[20] Radu Bogdan Rusu, Gary Bradski, Romain Thibaux, and John Hsu, *Fast 3d recognition and pose using the viewpoint feature histogram*, Proceedings of the 23rd IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (Taipei, Taiwan), 10/2010.

[21] Radu Bogdan Rusu and Steve Cousins, *3d is here: Point cloud library (pcl)*, International Conference on Robotics and Automation (Shanghai, China), 2011.

[22] Radu Bogdan Rusu, Zoltan Csaba Marton, Nico Blodow, and Michael Beetz, *Persistent point feature histograms for 3d point clouds*, Proceedings of the 10th International Conference on Intelligent Autonomous Systems (IAS-10) (Baden-Baden, Germany), 2008.

[23] Stefan Schiffer and Gerhard Lakemeyer, *AllemaniACs@Home 2011 Team Description*, Proceedings CD RoboCup 2011 (Istanbul, Turkey), 2011.

[24] Stefan Schiffer, Tim Niemüller, Masrur Doostdar, and Gerhard Lakemeyer, *AllemaniACs@Home 2009 Team Description*, Proceedings CD RoboCup 2009 (Graz, Austria), 2009.

[25] Bastian Steder, Radu Bogdan Rusu, Kurt Konolige, and Wolfram Burgard, *Point feature extraction on 3d range scans taking into account object boundaries*, International Conference on Robotics and Automation, 2011.

[26] K. E. A. van de Sande, T. Gevers, and C. G. M. Snoek, *Evaluating color descriptors for object and scene recognition*, IEEE Transactions on Pattern Analysis and Machine Intelligence **32** (2010), no. 9, 1582–1596.

[27] Wikipedia, *HSL and HSV*, `http://en.wikipedia.org/wiki/HSL_and_HSV`, 2011-12-30.

[28] ———, *k-d tree*, `http://en.wikipedia.org/wiki/Kdtree`, 2012-01-03.

[29] Michael Ying Yang and Wolfgang Förstner, *Plane detection in point cloud data*, Proceedings of the 2nd International Conference on Machine Control Guidance Bonn (2010), no. 1, 95–104.