6DOF Object Recognition and Positioning for Robotics Using Next Best View Heuristics

Alexander Ganslandt
elt12aga@student.lu.se

Andreas Svensson
elt12asv@student.lu.se

January 17, 2018

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Mathias Haage, mathias.haage@cs.lth.se

Examiner: Elin Anna Topp, elin_anna.topp@cs.lth.se

Abstract

The accuracy and portability of depth cameras have increased by a lot in recent years, which allows for advanced 3D scanning of the environment for robotic applications. In this thesis we have developed a system that uses a depth camera mounted on a robot arm to identify and localize arbitrary objects, and give hints on how to move the camera to get better localization results. The system works by generating virtual views of objects to identify them in a point cloud generated by the depth camera. This data is then used to estimate a pose of the object, and generate a hint on where to move the camera next. After a new point cloud is taken, it is merged with the previous cloud which allows the system to iteratively get more confident in the identification and pose of the object.

Keywords: Robotics, point clouds, object identification, pose estimation, localization, point cloud merging, hint system, next best view

Acknowledgements

We would like to thank:

- Assistant Professor Mathias Haage as supervisor and mentor.
- Research Engineer Pontus Andersson for help with mounting the depth camera to the YuMi robot.
- The Department of Automatic Control at LTH for access to the robotics lab, a YuMi robot and an Intel RealSense SR300 depth camera.
- Associate Professor Elin Anna Topp as examiner.

Contents

Chapter 1
Introduction

In this section we provide some background information, explain the problem we want to solve and give a quick overview of our proposed solution. We also write about related works, and our contribution to this field.

1.1 Vision Systems for Robotics

Industrial robots today are very efficient at picking up objects in a predefined, structured environment. This is however not sufficient for the increase in human-robot collaboration and interactions where robots need to be able to adapt to changes in unstructured environments. To make this possible, robots need some kind of vision system that allows them to see the world and find objects to interact with. This vision system needs to be both accurate and give as much information as possible about the environment.

One way to give robots vision is to use two-dimensional images with depth information, also known as 2.5D images, RGB-D images or point clouds. These depth images were mostly captured before by using triangulation with two RGB cameras. Nowadays however, it is possible to pick up a single low-cost RGB-D camera that measures both the depth and the color for each pixel, which allows for quick generation of detailed point clouds describing the environment. These point clouds can then be used to find and localize objects by analyzing the environment.

1.2 Problem Description

Giving robots vision using depth images is not more complicated than mounting a depth camera on the robot, preferably on a moving part that allows the robot to gather more information by looking at the environment from different angles. All data that the robot captures then needs to be processed in a smart way to improve the robot's knowledge

of its environment. This is done by using various methods for object identification and localization, which have been heavily researched and developed in the last couple of years. There has however been very little work done in combining these methods to get a system capable of identifying and localizing objects with the addition of improving the results by taking multiple images from different camera viewpoints in an efficient manner. We want to investigate if a system like this can be built, and if the results from this system are good enough to be used in robotic applications. The system should be able to capture multiple point clouds to iteratively improve the identification and localization results for each new cloud. It should also suggest where to move the camera in order to gain as much information as possible and more quickly and accurately identify and localize objects when there exists an ambiguity between similar looking objects. Is a system like this possible to build? Are the results good enough to be usable for robotic applications? Can the system handle arbitrary objects and have an easy method of adding new objects that can be identified and localized?

1.3 Related Work

This section will go through some of the work that has been done related to the object identification, pose estimation, point cloud merging and hint system. Some of this work has been directly used in this thesis and some are covering different methods that could potentially be used to improve the results of this thesis.

R. B. Rusu et al. have created the Fast Point Feature Histograms (FPFH) [1], which are robust multi-dimensional local features used for pose estimation in this thesis. They also propose a sample consensus algorithm used for pose estimation that is partially used in this thesis. This is closely related to the excellent resources and documentation available at the website for the Point Cloud Library [2], which we have used extensively in our work. Another approach for pose estimation is by D. Aiger et al. using 4-points congruent sets [3]. This approach does not rely on normals and features, and is thus more applicable for objects consisting of primitive shapes.

H. Ali and N. Figueroa have implemented the FPFH features to estimate poses of planar metallic objects [4]. This work is very similar to ours in regards to the pose estimation, however they also rely on ICP to fine tune the first rough pose estimation they get using a sample consensus algorithm with FPFH features. In our work we have avoided ICP since the time requirement would be too big when registering multiple views from multiple objects. They are also using only one object, and not multiple arbitrary objects as we are in our work.

Dirk Holz et al. have created a mobile robot bin picking solution [5], which is somewhat similar to our work but using a slightly different approach. Their work uses primitive shape models, instead of feature based models, expressed as graphs to find objects and do grasp planning. They merge multiple point clouds taken of the same bin to acquire new data, however their method for merging is not based on robot position data and instead uses registration. Their method for determining the next best view is somewhat similar to ours where they look at information gain and traveling costs for moving to a new view.

The object identification for this thesis is heavily inspired by the work of W. Wohlkinger and M. Vincze and their publication in [6]. They faced the same problem of object identifi-

cation of RGB-D images using CAD models. Their solution was to render synthetic range images offline of the CAD models. These synthetic range views are then used online to recognize objects using a global feature called the Ensemble of Shape Functions (ESF). In [7], W. Wohlkinger and M. Vincze continue their work in [6] by extending the classification to include a large number of objects. This would have been interesting for our thesis if we decided to expand our model base. They use a method called locally sensitive hashing in order to quickly access the identified model when dealing with a model database of as many as 2500 models with 200,000 synthetically rendered views.

The idea of our hint system emerged originally from the inspiration of object identification in [6] and [7]. The ability to render synthetic views of our CAD models gave us the idea of creating a view-graph for each model where each view was given an absolute value regarding the quality of each view. The concept of our hint system is mostly referred to as the Next Best View (NBV) in the literature. The objective of finding the next best view in a scene has been researched as early as in 1988 by S. A. Hutchinson, R. L. Cromwell, and A. C. Kak in [8]. They faced the problem of selecting the next best sensing viewpoint in order to disambiguate between multiple model hypotheses. They approached the problem in the same way as we do by offline simulating different viewpoints of a CAD object from a tessellated sphere and recording the visible sensor features from each viewpoint. By doing this they reduced the problem of finding the next best view into a simple graph-search problem which is also how we solved it. To find the next best viewpoint they searched through the aspect graph to predict new sensor readings and checked if these were enough to eliminate the ambiguity between the model hypotheses. There are two main differences between our work and [8]. The first one being the way we distinguish and handle views that are similar for different models. We propose a method to predict these similarities offline and assign an absolute value for each viewpoint describing the ability to distinguish between two similar models and model hypothesis. The second main difference is that we also suggest three other measures for the quality of a viewpoint, the visibility, quality of surface normals and quality of local features. A view which maximizes the distinguishability of two objects might not be optimal regarding other qualities of the view. The possibility of generating an online aspect graph was introduced by C. McGreavy, L. Kunze and N. Hawes in [9]. In their application they used a mobile robot to recognize and estimate 6DOF poses for objects. They argue that the implementation of online aspect graphs enables them to update the graph during runtime and thus compensate for obstruction and occlusion in real-time. They use a two stage approach for generating their aspect graph. The first stage uses an environmental analysis which eliminates potential new viewpoints that are occluded by the surrounding environment. The second stage uses a model analysis which computes the visible surface area for each view. The next best view is then selected as the view that maximizes the visibility of the object. However, their choice of next best view only maximizes the visibility of the object and does not take into account other qualities such as local features and surface normals. They also do not use previous data (e.g. merging of point clouds from previous views) for their next view. We believe however that the usage of previous point clouds from previous views will increase the model identification and pose estimation. Another example of building aspect graphs online is the work by C. Maniatis, M. Saval-Calvo, R. Tyleček and R. B. Fisher in [10]. In their work they try to track visible areas of a workspace by minimizing occlusion. They achieve this by capturing a complete 3D scene by using multiple depth sensors around a

work area and updating the 3D scene for each point cloud. In order to track visible areas they updated their 3D scene point cloud for each sample. They define their objective as finding the dynamic next best view (DNBV) with the main goal of overcoming dynamic occlusions. Similar to our approach they define new camera viewpoints by creating a discrete set of camera positions on a tessellated sphere. For each camera viewpoint they assign a cost-function which incorporates the visibility, occlusion, distance traveled, and robot joint movement. Their next view point is then simply the viewpoint with smallest cost-function value. Although the work in [10] does not focus on object recognition and pose estimation, it is interesting to see how they overcame difficult problems such as object occlusions and robot joint movement limitations.

1.4 Contributions

Our main contribution is a system with a pipeline that incorporates object identification, 6DOF pose estimation, point cloud merging and next best viewpoint selection. By using a modified version of the PCL render-views-tessellated-sphere-function, we contribute with an object identification subsystem which has the ability to generate all essential data offline that can later be used by the system. The object identification subsystem uses the proposed method in [6], and we contribute with some evaluations of other proposed identification methods for 2.5D data in [11], [12] and [13]. Our proposed method of defining the quality of a viewpoint by using normal-, feature- and distinguish-utility values is to our extent unique. Our method for finding the next viewpoint by using graph search is not necessarily unique, however, we propose a method for continuously collecting better data by moving the sensor along a generated path which leads to the most optimal viewpoint. We also contribute with a method for merging multiple clouds using robot data that is much faster and more accurate than regular merging using feature-based registration.

Our system can be used in robotics applications that require object identification and localization of arbitrary objects. It is designed to be easy to use and requires very little setup outside of installing the Point Cloud Library.

The development of the system has been split to allow us to work as independently as possible on the different parts of the system, however there has also been some collaboration on certain parts. The work has been regularly discussed and planned to make sure that the different parts of the system fit together, and that each one knows what the other is doing. There has also been a lot of work done outside of the parts mentioned in this report, as we have had to develop many smaller tools to help us test or debug certain aspects of the system.

Alexander has mainly been working with the pose estimation, the scene merging including the camera-to-hand calibration, and the main program that executes all different parts of the system. Andreas has mainly been working on the hint system and the object identification which includes the generation of offline data as well as accessing and storing the data online.

1.5 Outline of Report

The report starts with theoretical background in Section 2, laying the foundation of the theory used in this thesis. Then follows a description of the system structure in Section 3 where the details concerning the different subsystems are explained as well as our general approach. After the system structure description we present the results from the different subsystems in Section 4 as well as the results evaluating the complete interconnected system. The results and overall performance of the system and subsystems are then discussed in Section 5 where we also bring up different improvements that could have been done in order to achieve better results and performance. Finally in Section 6 we summarize our work and reflect upon achieving our main goals and objectives.

Chapter 2
Theoretical Background

This section will cover some theoretical background that is needed for understanding later parts of this report. In Section 2.1 some definitions for point clouds are explained, in Section 2.4 the definition for a pose used in this is explained, in Section 2.5 some of the technology behind depth cameras is explained.

2.1 Point Clouds

In this section, some basic definitions for point clouds are explained. These definitions are used extensively in the rest of this report.

2.1.1 Definition

A point cloud is a set of points in 3D. A point is usually represented in Euclidean space by its Cartesian coordinates (x, y, z), and sometimes also its RGB color. Usually a point cloud is captured using some kind of sensor that has the ability to measure the distance to a number of points in its field of view. The different sensors have different parameters that may give rise to differently looking point clouds, but most of them have the structure explained above.

In this report a point cloud is sometimes called a "cloud" for simplicity. Also, in this report a "scene" is defined as a point cloud that contains one or many objects that should be identified and localized by the system.

2.1.2 Downsampling

Point clouds can have varying point density based on what sensor captured the cloud, and they can also have different densities in different areas of the cloud. This is often

due to certain surfaces having better or worse properties for certain sensors. One common method used to make sure a point cloud has the same density everywhere is to downsample the cloud. Downsampling also saves a lot of memory and speeds up computations, with the disadvantage that some accuracy may be lost. This is commonly done using a voxel grid filtering approach. A voxel is simply a 3-dimensional box with a certain side-length (known as leaf size), and a voxel grid filter divides the point cloud into a grid of voxels. All points inside a voxel are replaced by a new point at the mean position of all the points in the voxel. The new density of the point cloud is thus defined by the leaf size, where a larger leaf size gives a lower density, and a smaller leaf size gives a higher density.

2.1.3 Normal Estimation

Finding the normal of each point is crucial to be able to compute features that will be explained later. There are many ways of estimating the normal of a point, and in this work we have used a simple but effective approach. Estimating the normal of a point is equivalent to estimating the normal of a plane tangent to the point. This plane can be found using least-squares plane fitting for all points in a certain radius around the point of interest. This approach gives one of two possible signs for the normal, where only one of them would be the correct one. There is no real mathematical way of finding the correct sign for the normal, but since a 2.5D depth image is taken from a certain viewpoint, all normals must be facing towards the viewpoint to be valid. In other words, to determine if a normal $\boldsymbol{n}$ for a point p is pointing towards the viewpoint origin v_p, we apply the following formula which computes the angle between two vectors in $\mathbb{R}^3$:

$$\cos(\theta) = \frac{\boldsymbol{n} \cdot \boldsymbol{v_p}}{\|\boldsymbol{n}\| \, \|\boldsymbol{v_p}\|} \tag{2.1}$$

$\boldsymbol{n}$ is the normal vector for the point p and $\boldsymbol{v_p} = p - v_p$ is the vector pointing from the viewpoint origin v_p to the point p. If this value is less than zero i.e. $\theta > \frac{\pi}{2}$ then we have a correct pointing normal that points towards the viewpoint origin. In fact, using our constraint that $\cos(\theta) < 0$ we can simplify Equation 2.1 to:

$$\boldsymbol{n} \cdot \boldsymbol{v_p} < 0 \tag{2.2}$$

to determine if a normal is correctly aligned, since the denominator of Equation 2.1 is always strictly positive ($\|\boldsymbol{n}\| \, \|\boldsymbol{v_p}\| \geq 0$).

2.2 Features

Points in a point cloud are only defined by their (x, y, z) coordinates, and in some cases their color. This does not say much about the properties of a point or its underlying surface. To compare points there needs to be some way to describe how one point is different from another in regards to the surface that the point is located on. A feature aims to describe the difference between a point and the other points around it, which can be seen as describing a point's properties or underlying surface. Features usually consists of a

couple of values computed between one point and another, where the values are usually distances or angles between the normals of the points. It is important that the features are well defined and independent of the viewpoint of the cloud, this means that if the cloud is rotated or translated, the features should remain the same.

The main reason to use features is to find points in two clouds that are similar, so called point correspondences, which will be further explained in the next sections. However, every registration method that is based on features will struggle when point clouds contain mostly primitive shapes such as planes, cylinders, spheres etc. The reason for this is that points on these shapes typically have the same features. As an example there is no way to differentiate a point close to an edge of a plane and a point in the middle of the same plane. There is also no way to differentiate points on different planes, as they all have the same features since their underlying surfaces look the same. This creates a big problem when finding point correspondences, as a point on a plane might be similar to many other points that also lie on planes. This is something to keep in mind when considering the use of features.

Features can be divided into two groups, local and global features, which are further explained in the upcoming two sections.

2.2.1 Local Features

Local features are good at describing the local geometry of the object. A local feature is usually computed by taking the nearest neighbors of the feature point and comparing certain distances and angles between the points. The nearest neighbors are mainly determined in two ways. The first way is by simply taking the N nearest neighbors of the feature point determined by the Euclidean distance in R^3. The second way is by creating a sphere with predefined radius and the center located at the feature point. The neighbors are then the points located inside the sphere. Local features are robust to noise and occlusion.

Three local features called Point Feature Histogram (PFH), Fast Point Feature Histogram (FPFH) and Signature of Histograms of Orientations (SHOT) will be described in the sections below.

Point Feature Histogram (PFH) Point Feature Histograms is a popular and well used local feature proposed by [14]. The feature is computed by comparing the normals in the proximity of the feature point. This is done by first creating point pairs in the k-neighborhood of the feature point, see Figure 2.1.

For each source and target point pair (p_s, p_t) in this neighborhood a coordinate frame is constructed on the source point p_s. The source point is always the point with the smallest angle between the point normal and the line connecting the two points. The coordinate frame can be seen in Figure 2.2 and the axes of the coordinate frame are determined as:

$$\begin{aligned} u &= n_s \\ v &= (p_t - p_s) \times u \\ w &= u \times v \end{aligned} \tag{2.3}$$

With the ***uvw*** frame, the four values α, ϕ, θ and d (see Figure 2.2) can be computed using the normals n_s and n_t of the two points:

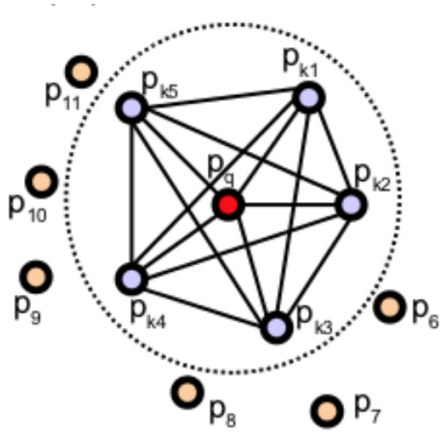


Figure 2.1: Point pairs for the PFH feature. Image from http://pointclouds.org

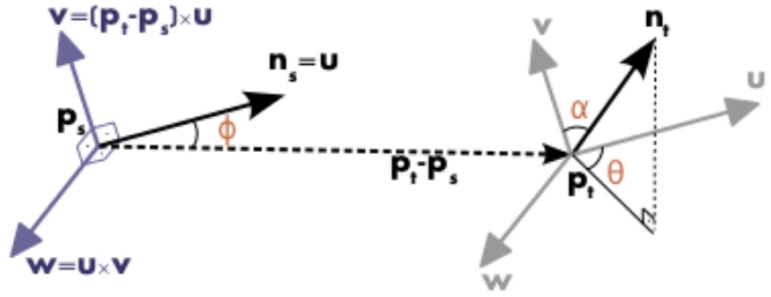


Figure 2.2: The ***uvw*** frame and the computed values α, ϕ, θ and d. Image from http://pointclouds.org

$$
\begin{aligned}
\alpha &= w \cdot n_t \\
d &= \|(p_t - p_s)\|_2 \\
\phi &= u \cdot \frac{p_t - p_s}{d} \\
\theta &= \tan^{-1}(w \cdot n_t, u \cdot n_t)
\end{aligned}
\tag{2.4}
$$

These values are then binned into a final histogram. The quality of the estimated feature is highly determined by the quality of the estimated surface normals.

Fast Point Feature Histogram (FPFH) The computational complexity for PFH is according to [1] $O(nk^2)$ where n is the number of keypoints. That is why [1] developed the Fast Point Feature Histogram (FPFH) which has the improved computational complexity to $O(nk)$. Most of the descriptiveness of the feature is also said to be the same for PFH and FPFH.

The computation of the FPFH is done by first calculating the simplified point feature histogram (SPFH) for all keypoints and their neighbors. The final calculation of the FPFH is then done by:

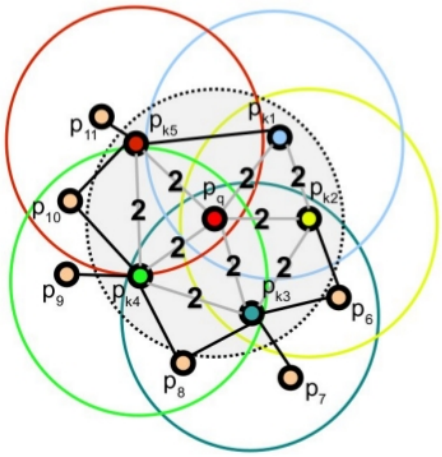


Figure 2.3: A visual representation of the neighbors and their influence for the computation of the FPFH. The neighbors of the feature point are determined by the search radius r, however some point pairs outside of r will still contribute to the final value (though at most $2r$) [1].

$$FPFH(p) = SPFH(p) + \frac{1}{k}\sum_{i=1}^{k}\frac{1}{w_k}\cdot SPFH(p_k) \tag{2.5}$$

where w_k is the distance between the keypoint p and a neighbor point p_k in $\mathbb{R}^3$. The visual representation of the computation and neighbor influence can be seen in Figure 2.3, which shows that some point pairs might be counted twice (indicated by 2).

Signature of Histograms of Orientations (SHOT) The Signature of Histograms of Orientations (SHOT) feature is a local feature introduced by [15] and is a combination of signatures and histograms. It is a very robust and descriptive local feature as well as computationally effective. It uses a unique and repeatable local reference frame to compute the descriptor. First a set of local histograms is computed for the geometry created by the sphere in Figure 2.4. The sphere is divided along the radial, azimuth and elevation axes with 8 azimuth divisions, 2 elevation divisions and 2 radial divisions. This results in 32 local histograms which are then used to compute the final descriptor. For each local region that makes up a local histogram, the angle between the feature point and the normals of the points within the region is calculated and binned into a histogram. The authors of [15] also recommends that the final descriptor is normalized to sum up to 1 for robustness to variations of the point density.

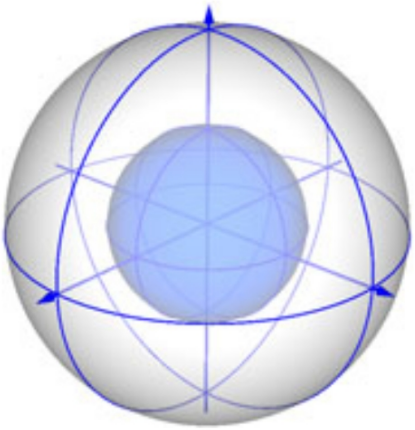


Figure 2.4: A visual representation of the local region used to compute the SHOT descriptor. The sphere centered at the feature point is divided 8 times along the azimuth axis, 2 times along the elevation axis and 2 times along the radial axis. However, for simplicity only 4 azimuth divisions are shown here [15].

2.2.2 Global Features

Global features are good at describing the whole geometry of an object. Instead of estimating features at specific keypoints as in the local case, one feature for the whole object is computed. This implies that the object is pre-segmented from the scene. Global features are less robust to noise and occlusion and are heavily dependent on good object segmentation. Global features are however good at describing objects with low geometrical variation such as household objects and man-made objects.

Viewpoint Feature Histogram VFH The viewpoint feature histogram (VFH) is a global feature proposed by [12]. The feature is used to explain the object geometry with respect to its viewpoint using surface normals. The viewpoint feature histogram consist of two components, a viewpoint component and a surface shape component. The viewpoint component is computed by measuring the angle between the view direction and each normal in the cluster, which is then binned into a histogram. The view direction is simply the normalized vector between the sensor and the center of mass of the cluster (see Figure 2.5). The second component is the surface shape component which is a modified version of the local feature FPFH (Section 2.2.1). Instead of computing local features at certain keypoints, the FPFH is computed at the center of mass with a radius large enough to encapsulate the whole cluster. The normal of the center of mass is set to the viewpoint direction. The resulting histograms from the two components (1 for the viewpoint and 3 for the extended FPFH) are concatenated into one final histogram for the feature.

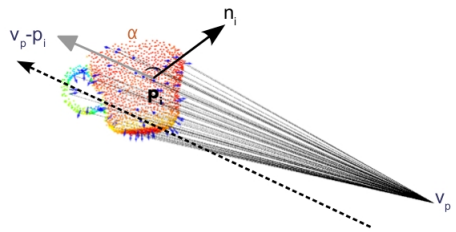


Figure 2.5: The Viewpoint Feature Histogram uses the viewpoint of the scene by measuring the angle between each surface normal and the viewpoint direction. The viewpoint direction is the vector pointing from the sensor origin to the center of mass of the cluster [12].

Clustered Viewpoint Feature Histogram CVFH The VFH global feature is not very robust to sensor noise and performance is reduced even for partial occlusions. To address this performance issue the authors of [13] came up with the Clustered Viewpoint Feature Histogram (CVFH). This global feature processes the segmented point cloud by first removing points with high curvature normals, such as sharp edges and noise, in order to later create smooth and stable regions using a smooth region growing algorithm. This creates new stable clusters which are then used instead to compute the global feature (see Figure 2.6). For each new cluster a VFH global feature is computed. The authors of [13] also describe the use of a camera roll histogram in order to recover the 6DOF pose of the object.

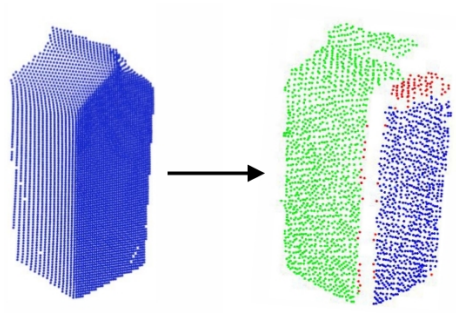


Figure 2.6: The smooth region growing algorithm is used to create new stable clusters (represented by different colors). The left image is the original point cloud representing a milk carton and the right image is the point cloud containing two new stable cluster regions (blue and green points, the red points do not belong to any cluster) [13].

Global Signature of Histograms of Orientations (GSHOT) The Global Signature of Histograms of Orientations (GSHOT) is just an extended version of the local SHOT feature to work as a global feature. This is done by computing one SHOT feature at the objects center of mass with a radius large enough to include the hole object.

Ensemble of Shape Functions (ESF) Ensemble of Shape Functions (ESF) introduced by [6] is especially designed for 3D CAD object identification where synthetic views of the CAD models are generated (more about synthetic views in Section 3.4.1). The authors claim that the ESF feature is robust and suited for real-time applications and does not require any preprocessing or normal estimation.

The ESF feature is computed by first dividing the (pre-segmented) point cloud into a voxel grid (see Figure 2.7).

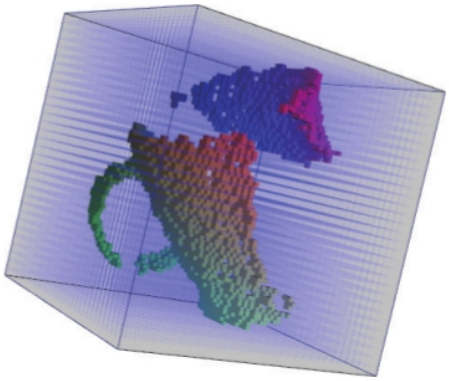


Figure 2.7: A 64x64x64 voxel grid created from a point cloud captured by a depth sensor [6].

For each point, three random points are chosen from which 3 point pairs are created. These point pairs are then used to calculate three shape functions D2, D3 and A3.

D2 This function is used to compute the distances between each point pair. The line connecting each point pair (see Figure 2.8) is then categorized as either IN, OUT or MIXED. IN corresponds to the line being completely inside the surface, OUT corresponds to the end points belonging to different surfaces and MIXED is a combination of both. All categories IN, OUT, and MIXED are assigned a histogram and the point pairs are binned into those accordingly. The D2 function also has one extra histogram measuring the ratio of the lines in MIXED being on the surface. In total the D2 shape function accounts for four distinct histograms.

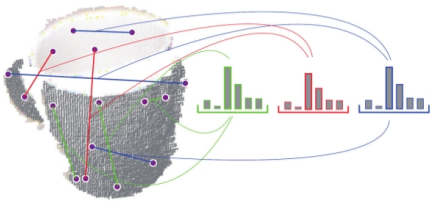


Figure 2.8: Point pairs and their lines. The green lines are inside the surface, red lines have end points belonging to different surfaces and blue lines are are mixture of both [6].

D3 This function calculates the area formed by three points as in Figure 2.9. The same methodology is applied here as in D2, the areas are divided into IN, OUT, and MIXED and binned accordingly.

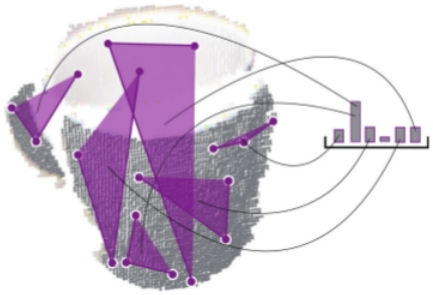


Figure 2.9: Areas formed by the point pairs [6].

A3 The last shape function computes the angle between the lines of the point pairs as can be seen in Figure 2.10. The opposing line then determines the classification (IN, OUT or MIXED) creating three additional distinct histograms.

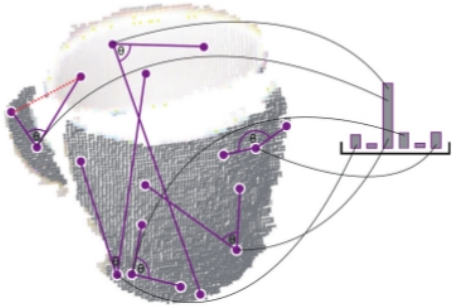


Figure 2.10: Angles formed by the lines from the point pairs [6].

In total there will be 10 histograms describing the underlying surface of the object.

As mentioned above the ESF produces its feature value by randomly choosing points in the voxel grid. This implies that the feature value will differ when recomputing the feature value for the same cluster and thus making the ESF feature nondeterministic. According to the authors in [6], on average the ESF feature is able to find the correct view with an accuracy of 90% within the ten nearest neighbors.

2.2.3 Feature Matching

Feature matching is used in order to find point correspondences between point clouds. A point correspondence is a point that is similar for both point clouds. This similarity is usually represented by geometry for point clouds which means that if two points from different point clouds share common geometry then the points are said to be similar or correspondences. Feature matching can be used for object identification using global features (explained in Section 3.5) and object pose estimation using local features (explained in Section 2.3).

Performing feature matching is the same for global and local features. In Figure 2.11 we see an example from the paper in [6] where a pre-rendered point cloud of a mug and a banana is matched to a real point cloud of a mug. In this case the histograms are matched and a score regarding the match is computed. We see clearly that the histograms of the pre-rendered mug and real mug are a better match than the pre-rendered banana and the real mug.

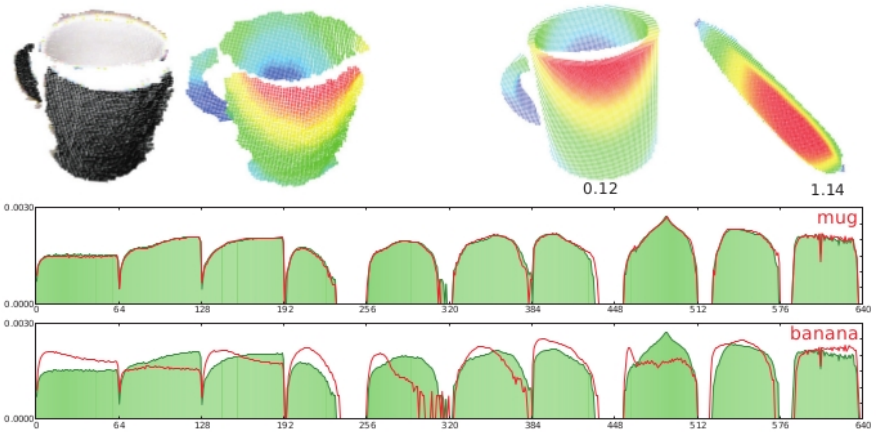


Figure 2.11: Matching of a point cloud captured by a real depth sensor and two pre-rendered point clouds of a mug and a banana. The green histograms correspond to the real object and the red curve corresponds to the pre-rendered objects [6].

The score of the match is computed by taking the norm of the difference between the histograms of two features. What kind of norm depends on what features you use. For instance, the matching of the ESF global feature is best computed by the L1-norm (Equation 2.6) as suggested by [6] whereas the matching of SHOT and FPFH features might be better computed by the L2-norm (Equation 2.7):

$$\text{L1-norm:} \quad m = \|\boldsymbol{h_1} - \boldsymbol{h_2}\|_{L_1} = \sum_{i=1}^{N} |h_1(i) - h_2(i)| \tag{2.6}$$

$$\text{L2-norm:} \quad m = \|\boldsymbol{h_1} - \boldsymbol{h_2}\|_{L_2} = \sum_{i=1}^{N} \left(h_1(i) - h_2(i)\right)^2 \tag{2.7}$$

where m is the matching score, $\boldsymbol{h_1}$ and $\boldsymbol{h_2}$ are the feature histograms (or descriptors), h is the value at position i of the histogram and N the size of the histogram.

2.3 Registration

Registration is the method used when aligning different point clouds with each other. The goal is to find a transformation of one point cloud such that it fits as well as possible with another point cloud. Naturally, a part of the first point cloud needs to be visible in the second point cloud, such that it can be seen how the point clouds fit together. The overlap between the point clouds is used as a measurement of how well the point clouds are aligned.

Many of the common methods for registration are based on point correspondences. Point correspondences use features which, explained above, aim to describe the underlying surface of a point and can be used to compare how similar different points are. In the case of registration of two point clouds, usually local features are used. The idea is to find a point in the first point cloud that has the same features as a point in the second point cloud, this is called a point correspondence. These two points can then be assumed to be the same point expressed in both clouds. By finding a number of these correspondences a transformation from one cloud to the other can be computed, such that all point correspondences align.

However, when a point cloud is captured by a sensor there is always noise present. This noise can lead to inaccurate normal and feature computations. This can in turn lead to false point correspondences between clouds, which will lead to bad transformations. It is therefore important that the registration method is fast and robust even in the presence of noise, which usually entails some form of early correspondence rejection, and an evaluation of the transformation to determine if it is accurate enough.

The most common implementation of this method is using a RANSAC loop. RANSAC stands for RANdom SAmple Consensus and is an iterative method that is well suited for data with outliers due to noise. It has many uses in different fields, but has been particularly useful in computer vision. As the name suggests, the method is based on using the least possible number of random samples from a dataset to fit some model parameters. The method then checks how many inliers the model gets, where an inlier is defined as an element in the dataset that is consistent with the model. This is done for a number of iterations, and the model with the most inliers is chosen in the end.

The way RANSAC is used for registration is the following. At least three point correspondences are necessary when finding a transformation from one cloud to another, as three point correspondences completely define all 6 degrees of freedom needed (position and rotation). Three points are thus chosen in one of the clouds, and three points with similar features are chosen in the second cloud. A transformation is then computed and one of the clouds is transformed into the other. The method then computes the number of inliers, which is defined as the number of points in one cloud that are close to points in the other cloud, based on some distance threshold. To allow for comparison between clouds of different sizes, the "inlier fraction" is usually used instead of counting the number of inliers. The inlier fraction is defined as the number of inliers in one cloud divided by the number of points in that cloud, and can be seen as a fraction of how well the clouds overlap. An inlier fraction of 1 would then mean that both clouds completely overlap, and an inlier fraction of 0 means that the clouds do not overlap at all. After this is done, the loop restarts by choosing three new points. This is done for a number of iterations and the transformation with the highest number of inliers (or highest inlier fraction) is chosen as the final transformation.

Due to noise and points with similar features, many of the point correspondences are not correspondences at all and will thus result in a bad transformation. It is expensive to compute the transformation between the two corresponding point sets and it is thus important to find and reject wrong correspondences as early as possible. This can be done using various correspondence rejection methods (also known as prerejective methods), and the one used in this thesis is based on geometric constraints [16]. The method simply compares the distances between all points in one point set, with the same distances between all points in the second point set. If the distances are too different (defined by a threshold),

all points are rejected and the RANSAC loop is restarted by picking new point correspondences. This works based on the fact that the same points in two clouds will always have the same distances between them regardless of how the rest of the clouds look. If this is not the case, the points are not correspondences and can be rejected. In this way no transformation has to be computed which decreases the execution time by a lot.

2.4 Pose of an Object

The pose of an object is defined as the position and rotation of the object in a scene. The position in Cartesian space is given by the (x, y, z) coordinates of the object's center of mass. The rotation can be represented in many ways, in this thesis we have used both Euler angles (ZYX convention) and quaternions. There are thus 6 degrees of freedom for each object that needs to be found to completely identify the pose of the object. The position and rotation of an object can be represented as a transformation matrix, which is defined as the following

$$\boldsymbol{T} = \begin{pmatrix} \boldsymbol{R} & \boldsymbol{t} \\ 0 & 1 \end{pmatrix}$$

Where $\boldsymbol{T}$ is the transformation matrix, $\boldsymbol{R}$ is the rotation matrix and $\boldsymbol{t}$ is the translation vector. When this transformation matrix is left-multiplied with a point in homogeneous coordinates, it will both rotate and translate the point according to $\boldsymbol{R}$ and $\boldsymbol{t}$ respectively. This can be used to efficiently transform point clouds, where the transformation matrix is left-multiplied with each point in the cloud. In other words, the pose of an object in a scene is the transformation matrix applied to the object such that it fits perfectly into the scene.

Worth noting is that the pose of an object in a scene is defined in the coordinate system of the scene, which is the same as the coordinate system for the camera that captured the scene. To be able to use the pose of an object to, for example, pick up the object using a robot, the pose needs to be transformed from the camera coordinate system to the robot coordinate system. Transformations between camera and robot coordinate systems are explained further in Section 3.6.2.

2.5 Depth Sensors

This section explains what a 2.5D image is and how they are captured by depth sensors such as the depth camera used in this thesis.

2.5.1 2.5D Images

A camera with a depth sensor, like the Intel Realsense SR300 camera this thesis uses, is able to capture images with depth information. Each pixel in the 2D image is assigned a depth value which enables the camera to capture so called 2.5D images. The reason they are not called 3D images is because only the visible part, not the complete part, of the 3D environment is captured.

2.5.2 Capturing 2.5D Images

One way to acquire 2.5D or range images is by using triangulation with a stereo camera setup. At least two cameras are required in order to use triangulation where a point in 3D is projected into the 2D camera images. By finding point correspondences in the images one can setup a set of equations and solve for the corresponding 3D point.

Structured light can also be used to capture range images. A depth sensor using structured light is composed of two cameras and a projector (light source). The light source projects patterns onto the environment (usually in the infrared spectrum) which become distorted by the surrounding objects. This distortion is then captured by the cameras which uses this information along with triangulation to solve for the depth image of the scene.

Time-of-flight cameras are also a popular alternative of capturing range images. They work by measuring the delay of a light pulse emitting from a light source. Using the speed of light, this delay can be used to calculate the distance traveled by the light.

2.5.3 Intel SR300 Depth Camera

The Intel RealSense SR300 is a consumer grade depth camera mainly designed for human-computer interaction. It contains a regular RGB camera, an infrared camera and an infrared projector, and thus uses structured light to measure depth. It has an operating range of about 0.3 to 2 meters and is very small, measuring in at 110 mm in width, 12.6 mm in height and 4.1 mm at its thickest point. It is sold for around $100 as of this writing, making it very affordable compared to similar products.

Chapter 3
System Structure

This section will explain the different parts of our system in detail. Section 3.1 gives an overview of the system, and explains how the different parts fit together to form the pipeline. Section 3.2 explains the point cloud capturer. Section 3.3 explains how the raw point clouds are segmented to find interesting clusters. Section 3.4 explains how the synthetic views are rendered and how the data is generated offline. Section 3.5 explains how the object identification works. Section 3.6 explains how multiple scenes are merged together using data from the robot. Section 3.7 explains how the pose estimation works. Finally, section 3.8 explains how our implemented hint system works. The models referred to in this section can be found in Figure 4.1 in Section 4.

3.1 Overview

The entire system consists of a couple of separate subsystems that each have their own tasks and that can be modified as long as the inputs and outputs remain the same. Figure 3.1 shows a high level overview of the system. Circles symbolize point clouds, rectangles symbolize subsystems and triangles symbolize output data. All subsystems in this figure are explained in detail in the following sections, but will be summarized here to give the reader an understanding of each subsystem's role in the system.

The first subsystem in the pipeline is the point cloud capturer. This subsystem is responsible for capturing point clouds (also known as scenes) using the Intel RealSense SR300 depth sensor, and save the clouds in PCD format [17]. The point cloud (named single cloud in the figure) is then processed by the segmentation subsystem, that segments the cloud into clusters. The clusters are then used in the object identification subsystem, that identifies the most probable objects that each cluster represents, this data is the "ID data" triangle in the figure. The single cloud is also merged with any previous clouds (if they exist) using the scene merging subsystem, which is visualized by the arrow to itself. The merged cloud is then segmented and used in the pose estimation subsystem together

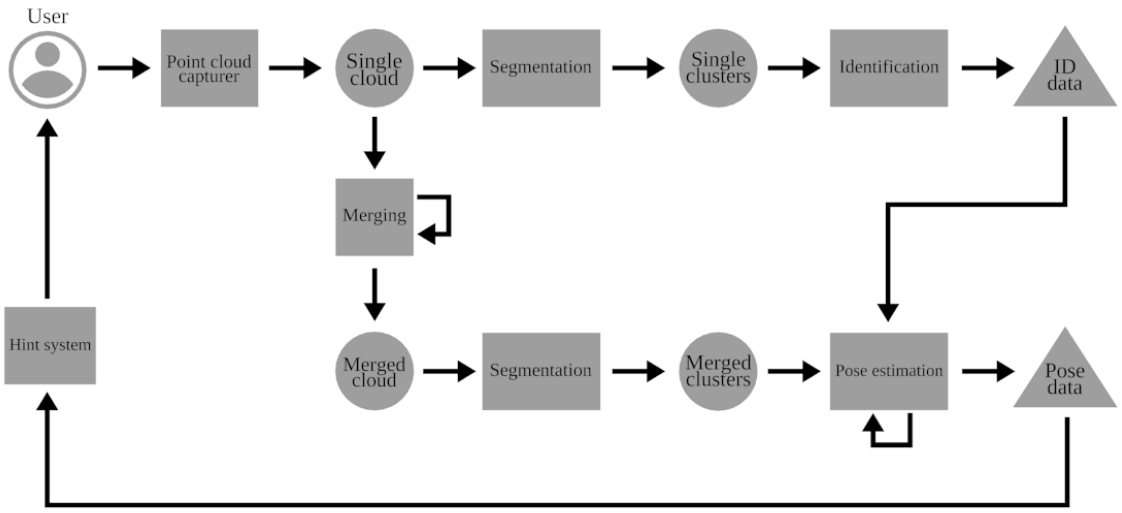


Figure 3.1: A high level overview of the system. Note that the linking of data for the pose estimation has been omitted to not clutter the figure.

with the results from object identification and any pose estimation results from previous clouds (visualized by the arrow to itself). This subsystem estimates the poses for the objects found in each cluster, which is the "Pose data" triangle in the figure. Finally, the pose data is sent to the hint system that generates a hint for where the camera should be placed to capture a new point cloud that will give as much information as possible about the object. This hint is shown to the user, that then decides where to capture the next point cloud.

3.2 Point Cloud Capturer

The point cloud capturer uses the Intel RealSense SDK for Windows [18], which has since been discontinued in favor of Intel RealSense SDK 2.0 also known as librealsense [19]. The subsystem is written in C# and consists of a simple window with two live camera feeds, one for color information and one for depth information. There is a single button that captures a point cloud and saves it as a PCD file. A screenshot of the program can be seen in Figure 3.2.

3.3 Segmentation

A point cloud of a scene usually contains many points that are not particularly useful for object recognition or pose estimation. For example, if the goal is to find a mug in a scene which contains the mug and some other objects on a table, the only necessary points are those that belong to the mug. All other points can be removed since they do not directly contribute to finding the mug. These unnecessary points both slow down execution of algorithms and lead to poorer results.

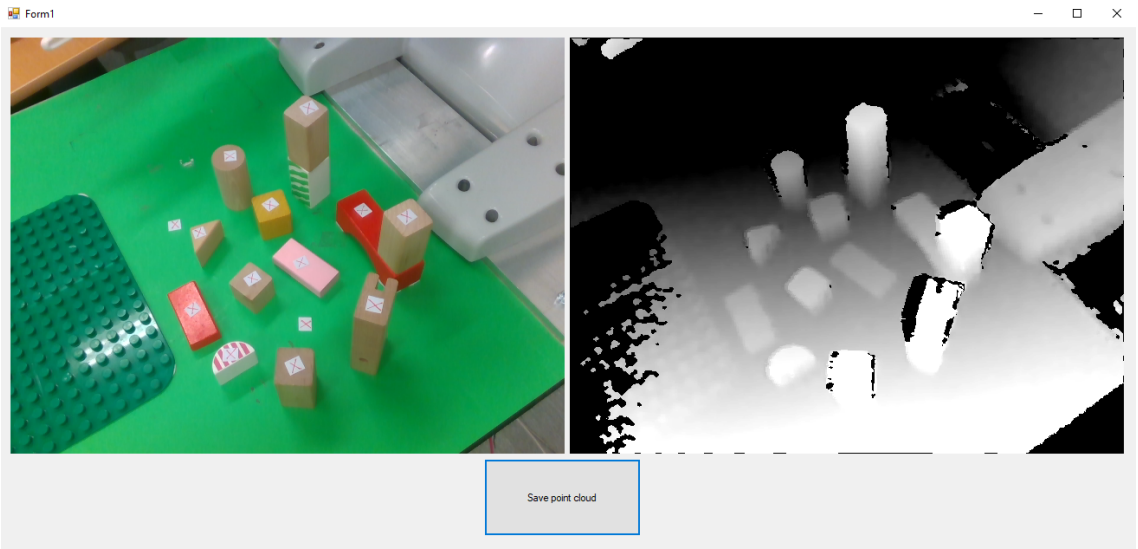


Figure 3.2: A screenshot of the point cloud capturer program. The left part of the window shows a live feed from the RGB camera, and the right part of the window shows a live feed from the depth camera, with closer points being more white than far away points.

3.3.1 Removal of Unnecessary Points

There are many different ways of removing unnecessary points from a point cloud depending on the application. For example, if there is guaranteed to be a large flat surface (like the top of a table) in every point cloud, one can find and remove the largest plane in every cloud (using a RANSAC loop). However, this would cause problems if a point cloud does not contain a table, since the program would still remove the largest plane. This could then remove a plane in the object of interest. The program will thus be limited to only point clouds containing objects on a flat surface and will not work well for any arbitrary point cloud.

A more accurate way of removing unnecessary data is by using background removal. This requires the user to capture a point cloud of the background of a scene which can then be subtracted from a new scene containing objects. Since it might be hard to get the entire background in one point cloud, one can capture multiple clouds of the background and merge them together using the method explained in Section 3.6. The subtraction works by removing points in the new scene that are close (defined by some radius) to points in the background scene. In this way interesting objects can quickly be segmented from the background. However, due to inaccuracy and jitter in the depth sensors, this subtraction is far from perfect. Despite this we found this method to work well for our purposes, but it does require the user to be able to capture the background before the objects of interest are placed into the scene. This might be a limitation for some applications.

After either the plane or the background has been removed, the clouds are also downsampled. As explained in Section 2.1.2 this normalizes the point cloud density and saves a lot of memory while speeding up computations on the cloud.

In our work we use background subtraction if the user has created a background cloud,

otherwise the system falls back to using plane segmentation. However, even when using background subtraction we still find the largest plane in the scene which is used later in the hint system, explained in Section 3.8.

3.3.2 Clustering

The point clouds usually consist of clusters of dense points after unnecessary points have been removed from the point cloud (either using plane removal, background removal or some other method). These clusters may be partially connected to each other by areas of points with less density than the clusters. The clusters are found and segmented from the cloud using a Euclidean cluster extraction method. This is a method very similar to a flood fill and the algorithm works as follows [20].

1. Create an empty list of clusters C, and an empty queue of points Q
2. For every point $\boldsymbol{p}_i \in P$, where P is the input point cloud, do the following:
   (a) Add $\boldsymbol{p}_i$ to the queue Q
   (b) For every point $\boldsymbol{p}_i \in Q$ do the following:
      i. Find the set of neighbors P_k^i of $\boldsymbol{p}_i$ in a sphere with radius r
      ii. For every neighbor $\boldsymbol{p}_i^k \in P_i^k$, check if the point has been processed, if not add it to Q
   (c) When all points in Q have been processed, add the points in Q to the list of clusters C and set Q to an empty list
3. The algorithm is finished when all points $\boldsymbol{p}_i \in P$ have been processed

A limit on how many points should be required for a cluster to be valid is also used, this will remove small clusters that are most likely not part of an object. After running this algorithm the cloud will be split up in clusters represented as 'subclouds', that can then be used by the other parts of the system.

3.4 Offline Training and Data Generation

In order to speed up online recognition and pose estimation we generate all necessary data for the models offline. We start of by generating synthetic views of the CAD models. For each view we then estimate global and local features. The global features will be used for online model recognition and to detect similar views between models and the local features will be used for online 6DOF pose estimation. We also compute utility-values for each view describing the overall quality of the view, which will be used later in the hint system.

3.4.1 Synthetic View Rendering

The model base for the system consists of CAD models which are complete 3D objects. In order to identify and estimate poses for the 2.5D cluster point clouds we need to create a set of 2.5D views of our CAD models. However, capturing all 2.5D views of a model with a real depth sensor can be cumbersome and time consuming. It is also error prone since some views might be missed. That is why we decided to use a virtual rendering program that generates synthetic 2.5D view point clouds of the CAD models. This is done by using PCL and a function that render views from a tessellated sphere. It enables us to capture all views of an object using just the CAD model as input. The function puts a virtual camera uniformly on either the faces (triangles) or vertices on a tessellated sphere with the CAD model in the center. The sphere is generated by a regular icosahedron which can be seen in Figure 3.3. The tessellation level determines how many times the triangles are divided in the original icosahedron. A tessellation level of 0 means that the regular icosahedron is used with 12 vertices and 20 faces. A tessellation level equal to 1 means that all triangles are divided one time (see Figure 3.4) thus creating 42 vertices and 80 faces. A tessellation level of 2 means that the triangles are divided 2 times creating 162 vertices and 320 faces and so on.

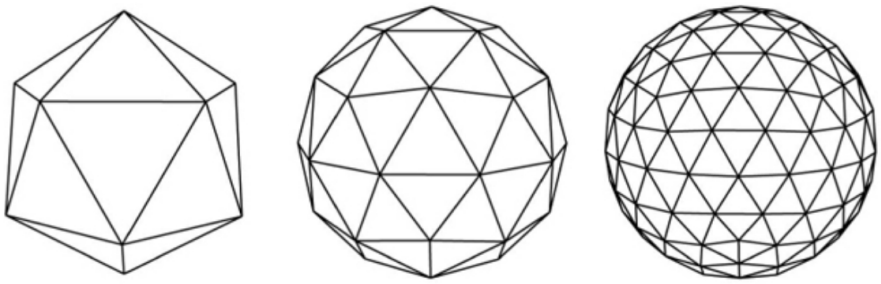


Figure 3.3: This figure shows the regular icosahedron (left) along with tessellation level 1 (middle) and 2 (right) [21].

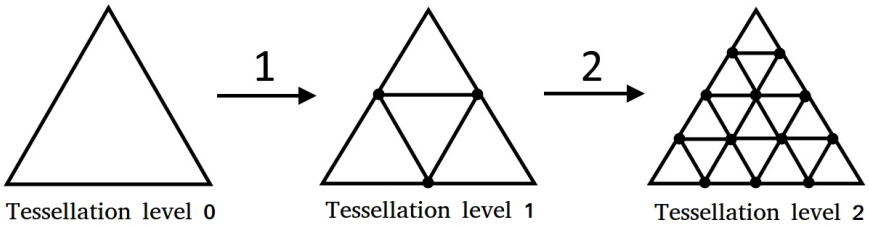


Figure 3.4: Dividing a triangle (face) one time when using a tessellated icosahedron means inserting an extra vertice on the link between two existing vertices (middle). Each new vertice is then connected according to the image creating four new triangles.

By choosing to put the virtual camera on either the faces or the vertices and applying a tessellation level of 0 and 1 we can choose whether to render 12, 20, 42, 80, 162 or 320 views. In Figure 3.5 we see some of the synthetic views rendered from a CAD model of a

wireless xbox controller (see Figure 4.1 in Section 4 for a description of the wireless xbox controller model).

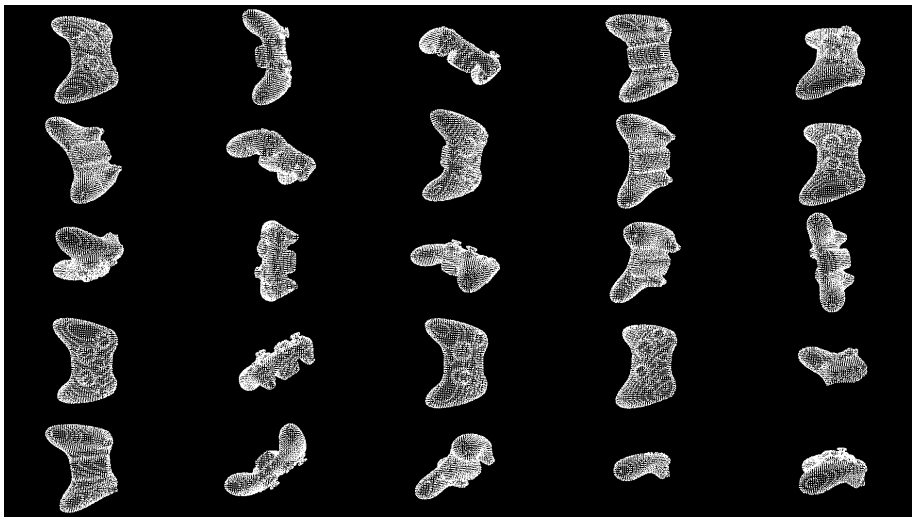


Figure 3.5: Rendered synthetic range images of a CAD model.

Using the information from the tessellated sphere we can construct a graph where each node is a camera position and rotation, and every link is a connection to the nearest neighbors (nearest views). We call this a view-graph and it is illustrated in Figure 3.8 where a tessellated sphere with tessellation level 1 is used.

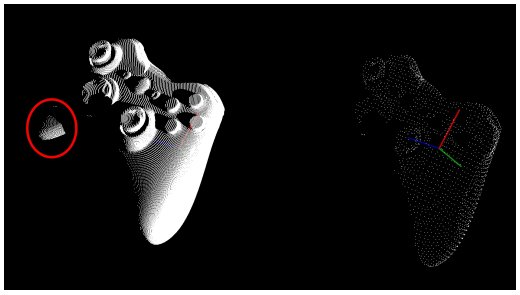
We decided to render 80 views of each model by using a tessellation level equal to 1 and putting the virtual camera on the faces of the sphere. This proved to capture a sufficient amount of data for each model.

3.4.2 Processing Rendered Views

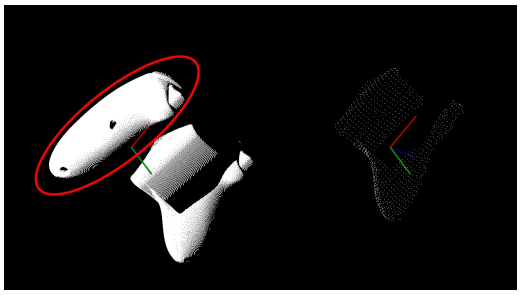
The rendered 2.5D view point clouds has to be processed to mimic the real 2.5D views from the camera sensor. First the view point clouds are downsampled using the same voxelsize as the real 2.5D point clouds from the depth sensor. Just like the real sensor the virtually rendered views might have disconnected clusters of the object as can be seen in Figure 3.6. These are processed by only keeping the largest cluster in the view thus removing smaller clusters. The clusters are extracted by the use of Euclidean cluster extraction as explained in Section 3.3.2. The final processed versions and the original point clouds can be seen in Figure 3.7.

After the processing of the rendered 2.5D view point clouds the normals for each view are estimated. In order to have correct pointing normals (normals should be pointing in the direction of the sensor origin) the viewpoint for the normal estimation in PCL has to be placed at the virtual camera position for each view. We used the setKSearch(int k) method in PCL with $k = 10$. This provided good normals without dealing with NaN-values which can occur when using the setRadiusSearch(double *radius*) method in PCL.

As a final step all views are merged in order to create a complete point cloud model of the rendered CAD model. This is easily done by just adding all view point clouds into one large point cloud. This is possible since all view point clouds have the same pose which is

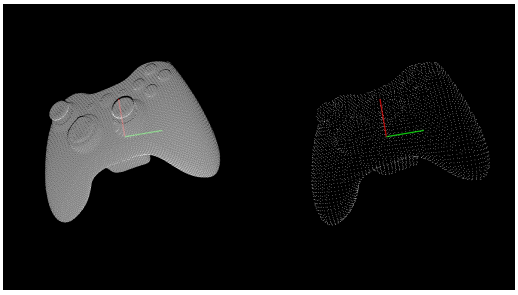


(a) Original rendered point cloud (left) and processed point cloud (right). Notice the smaller cluster in the left cloud that has been removed in the processed cloud.

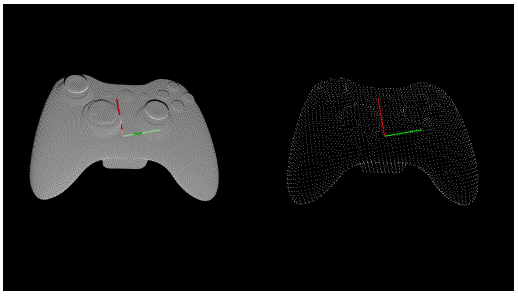


(b) Original rendered point cloud (left) and processed point cloud (right). The size of the cluster matters for each rendered view. Only the largest cluster is kept.

Figure 3.6: When processing the rendered 2.5D views only the largest clusters are kept. Smaller clusters (encircled by red) are discarded in the final processed point cloud.



(a) Original rendered point cloud (left) and processed point cloud (right) of the wireless xbox controller model.



(b) Original rendered point cloud (left) and processed point cloud (right) of the wireless xbox controller model.

Figure 3.7: This figure illustrates the rendered view point clouds before (left) and after (right) processing.

the pose of the original CAD model. The final merged point cloud of the complete model is then downsampled using the same voxelgrid size as before. The fact that all view point clouds have the same transformation will be used later when merging views and finding better views in the hint system. The same transformation for a view point cloud can be used on other view point clouds of the same model as well. The complete model for the wireless xbox controller model and its view-graph can be seen in Figure 3.8.

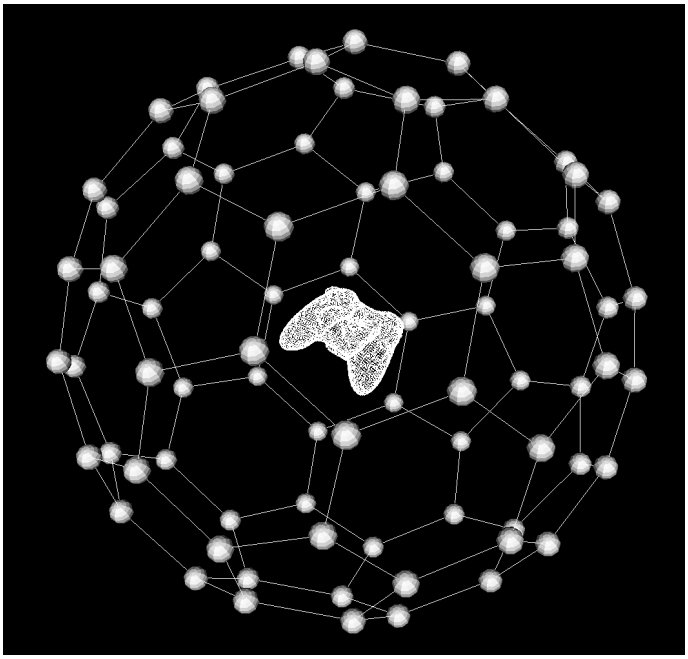


Figure 3.8: View-graph and complete model. Each white sphere corresponds to a virtual camera position. Each camera position is represented by a node in the view-graph and each node has a connection to its nearest neighbor.

3.4.3 Estimating Global and Local Features

As global features we decided to use ESF features. These features are very easy to estimate in PCL without estimating any normals or keeping track of view-direction. We compared the results of matching global SHOT, CVFH and ESF features (see Section 2.2.2). When matching global SHOT features the results were mostly good. However, the global SHOT features seemed to be very sensitive to missing parts and minor occlusion. The CVFH features did not perform very well either. The clusters obtained from the smooth-region-growing algorithm were not consistent between the rendered views and the real views from the sensor.

As local features we decided to use FPFH features. These are widely used features for local registration and are very fast to compute. This is important since the local features for parts of the scene need to be computed online.

3.4.4 Detecting Similar Models

One of the big problems of object recognition is when there are similarly looking views among the models. Say we have two simple models, cube A and cube B with equal dimensions (see Figure 3.10). A is a solid cube and B is a cube with a circular hole carved into one of its sides. It is not hard to realize that for some views it is not possible to distinguish between these models and in order to separate these models we have to look at the side containing the circular hole.

By matching each of the rendered views with the existing models and their views we

can detect which views that are similar and label them. If we identify this view during the online recognition phase we can use our view-graph to search for views that separate the models better.

The matching of objects is done by taking two models and their global features. Each model has N views and thus N global ESF features. Let us say the models are called A and B. We start by matching model A with model B which will be denoted later on as $A \rightarrow B$. For the first view V_1^A in model A we take the global feature for that view and match it with all the views $V_k^B,\ k = 1, \ldots, N$ and their global features in model B. An illustration of this can be seen in Figure 3.9. This will give us 80 matching scores: $m_k^{V_1^A \rightarrow V_k^B},\ k = 1, \ldots, N$ for the first view V_1^A in model A. We then select the lowest matching score and assign this value to the first view V_1^A in model A:

$$u_1^{A \rightarrow B} = \min\left(m_1^{V_1^A \rightarrow V_1^B},\ m_2^{V_1^A \rightarrow V_2^B},\ \ldots,\ m_N^{V_1^A \rightarrow V_N^B}\right) \tag{3.1}$$

If we get a low score for $u_1^{A \rightarrow B}$, then there exists some view V_k^B in model B which is very similar to the first view V_1^A in model A. We do this for all views in model A which will give us a set of values $u_k^{A \rightarrow B},\ k = 1, \ldots, N$ for each view V_k^A in model A, all describing the worst case or the best matching view in model B. This is then also repeated for model B by matching model B with model A, denoted by $B \rightarrow A$.

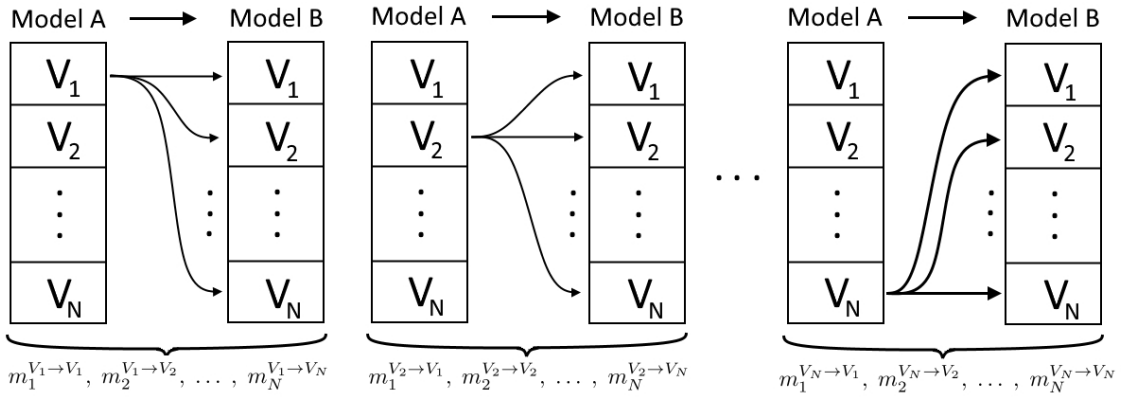


Figure 3.9: This figure illustrates the procedure of finding similar views for two models A and B. Each view in A is matched with all views in B to find the best matching view. Each view V in A will get a set of matching scores m for which the minimum matching score is selected.

It is important to note that matching model A with B is not the same as matching model B with A. We illustrate this by an example where we have two models A and B. Model A is a solid cube and model B is solid cube except for one side which contains a circular hole (see Figure 3.10). We then place a depth camera around both objects with viewpoint directions perpendicular to all sides of the objects. This results in six views for both objects, each view capturing one side of the cube. As we can see in Figure 3.10, the matching $A \rightarrow B$ will yield a low matching score for all views in model A. This is expected since all views $V_k^A,\ k = 1, \ldots, 6$ for model A are identical to the views $V_k^B,\ k = 2, \ldots, 6$ for model B. However, when we match $B \rightarrow A$ we will have a low matching score for the

views $V_k^B,\ k = 2, \ldots, 6$ and a high score for the first view V_1^B containing the circular hole. There is not any similar view in A for V_1^B in B so the lowest matching score $u_1^{A \to B}$ will be high and thus $u_1^{A \to B} \neq u_1^{B \to A}$.

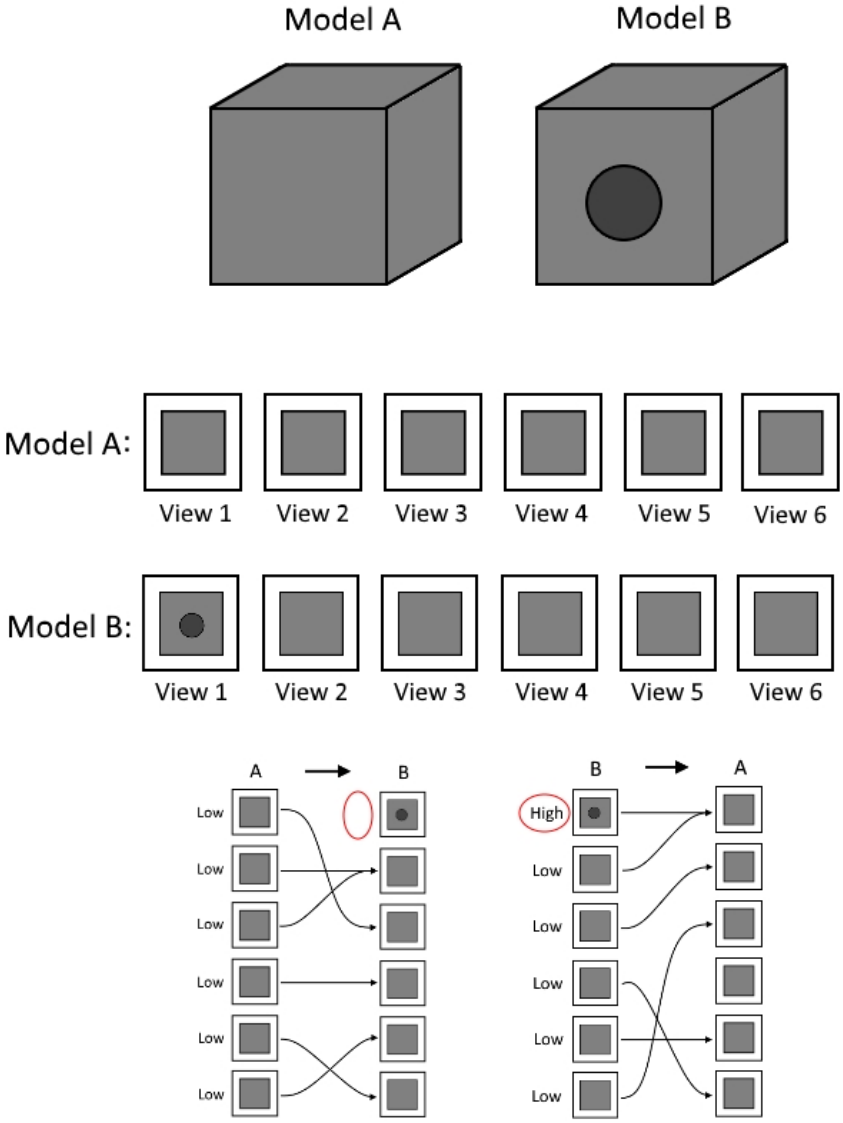


Figure 3.10: An example illustrating the matching order of two models. For this figure we have two models A and B which are cubes of identical dimensions. Model B has a circular hole on one of its sides. A camera is then placed around both objects capturing a 2.5D view of each side of the objects, giving us a total of six views of each model. These views are then matched in the same manner as in Figure 3.9 and the lowest score is selected for each view. The matching score is located to the left of each view, the arrows indicate the best matching view.

When adding a new model to the system we count the number of rendered views with low distinguishability and a warning message is displayed if the ratio of views with low distinguishability is higher than 50%. This enables the system to warn the operator if he

or she decides to add a very similar model to the existing model base (which will make the identification process very hard for these models).

3.4.5 Utilities

For each rendered view a number of utilities are computed to determine the quality of the view. This quality measure will later be used in the hint system to search for better views. There are many quality measures for a given view but we decided to focus on four values; visibility, normals, features and distinguishability. As mentioned in Section 1.4 we came up with all utility values, except the view-utility, on our own.

View-Utility

The view-utility for a rendered view of a point cloud is a measure of how much of the object that was visible for the given view. This is typically referred to as the entropy in related works regarding the next best view. A high view-utility means that a large part of the object is visible. The view-utility is computed by taking the number of points in each point cloud for all the rendered views. All these values are then normalized between 1 and 0 by applying the following formula:

$$u_i(x) = \frac{x - \min}{\max - \min} \tag{3.2}$$

where $u_i(x)$ is the view-utility for view $i = 0, \ldots, 79$, x is the number of points in the point cloud for view i and max, min are the maximum and minimum number of points for all views respectively.

Normal-Utility

The normal-utility is a measure of the quality of the normals for a rendered view. Many depth sensors such as the Intel RealSense SR300 camera suffer from noise when the surface normals of the object points are close to perpendicular with respect to the viewpoint direction. In order to incorporate this into our system we decided to count the number of "bad normals" for each view. A bad normal is defined by first measuring the angle θ between the normal and the viewpoint direction (see Equation 2.1 in Section 2.1.3). If the angle θ is less than 110 degrees the normal is considered a bad normal. The final utility value is computed by:

$$u_i(x) = \frac{x - \min}{\max - \min} \tag{3.3}$$

where $u_i(x)$ is the normal-utility for view $i = 0, \ldots, 79$, x is the number of bad normals in the point cloud for view i and max, min are the maximum and minimum number of bad normals for all views respectively. This also results in normalizing the normal-utility between 1 and 0.

Feature-Utility

The feature-utility is a measure of how unique the local features of visible points are for a certain view. Having more points with unique or distinct local features in a view leads to faster and more accurate pose estimation due to more easily finding point correspondences.

The feature-utility is computed by first finding distinct features in a view. This is done by computing the average feature for the current view, and selecting the points with features that differ a lot from the average feature, based on a threshold. These points are called distinct points. A score is then computed where a point with a close to average feature is worth 1 point, but a point with a distinct feature is worth 10 points. This score is then compared to the same score for different views of the same object, and all scores are normalized between 0 and 1. This means that the view with the least distinct features will have a feature-utility of 0, while the view with the most distinct features will have a feature-utility of 1.

Distinguish-Utility

The distinguish-utility is a measure of how similar two objects are. When detecting similar views, each view is given a distinguish-utility which determines if the view is distinguishable from the other model and its views. If a view is very similar to another view for another model then the matching of these views will be high and thus given a low distinguish-utility and vice versa.

The distinguish-utility is computed by matching the global ESF features as explained in Section 3.4.4. A good match between two views means that the matching score is low (typically less than 0.2) and a bad match yields a higher matching score. A good match for two views results in a bad distinguish-utility because these views will be hard to distinguish from. From Section 3.4.4 we get that the distinguish-utility for model A when matching with model B is computed by:

$$u_i^{A \to B} = \min \left(m_1^{V_i^A \to V_1^B},\ m_2^{V_i^A \to V_2^B},\ \ldots,\ m_{79}^{V_i^A \to V_{79}^B} \right) \tag{3.4}$$

where $u_i^{A \to B}$ is the distinguish-utility for view $i = 0, \ldots, 79$ when matching model A with model B. A distinguish-utility close to 1 has good distinguishability and a distinguish-utility close to 0 has bad distinguishability. Notice in Equation 3.4 that the distinguish-utility is not normalized. The normalization is carried out later online when running the hint system, as will be explained in Section 3.8.

3.5 Object Identification

For a system with many models it is time consuming to perform a 6DOF pose estimation on all object and their views in order to determine which model is best. To solve this problem we implemented an object recognition pipeline that gives the most probable models.

There are many ways of performing 3D object recognition. Implicit Shape Model [11] is a method that relies on local features. Each model is trained using the local features and given a code-book with code-words containing activation strategies. Each code-word

votes for the center of mass of the model and is then used online to determine which model is present using a voting scheme. For the implementation of Implicit Shape Model in PCL we found that it is suitable for complete 3D object identification and less suitable for identification of objects captured with a RGB-D camera which only produces 2.5D images.

We decided to use global features matching, which is also very popular ([6], [7], [22], [23]). As described in Section 2.2.2, a global feature is computed for the whole point cloud and encodes the whole geometry of the object. Capturing all different views offline of a model enables us to create a list of global features for each model, each feature corresponding to a synthetic view. We then estimate a global feature for each unknown point cloud cluster and match it to the precomputed lists of global features giving us a set of the best matching models and their best matching views. Each matching view for each model is given a matching score which depicts the quality of the match. The score is computed by taking the L1-norm of the difference between two ESF feature histograms (as explained in Section 2.2.3) and a low score indicates a good match. Remember from Section 2.2.2 that for ESF features, the correct view is found with an accuracy of 90% within the 10 best matching views for a model. This means that if we have a point cloud cluster of an object for a specific view and search for the best matching view in our list of ESF features, we will with 90% certainty find the correct view within the 10 best matching views for that model. As a result, for the identification of an unknown point cloud cluster, we save the 10 best matching views for the 10 best matching models.

The search for matching global features is done using ordinary linear search which has a computational complexity of $O(n)$. Since we are only dealing with a few models (at most 50), resulting in at most $50 \cdot 80 = 4000$ global features to search through, this is not a major problem. However, dealing with a database consisting of up to 100,000 models would have had some performance issues when using linear search. The authors in [7] overcame this problem by using locality sensitive hashing to fast access the appropriate models. They claim that the time complexity of the search using this technique reduces from $O(n)$ to $O(1)$.

It is worth mentioning that the identification method described above using global descriptor matching is only suitable for 2.5D images captured by a single depth camera. Using the complete 3D scene or merged point clouds will not yield good results.

3.6 Scene Merging

This section will explain the methods used for merging two scenes using data from the robot. The camera is mounted on the robot's hand using a special tool as can be seen in Figure 3.11. This allows for using the position and rotation of the robot hand in the robot coordinate system to compute the relation between two scenes captured from different views, and use this data to merge the scenes. The scenes are considered successfully merged when one scene is transformed into the other scene such that the common points in both scenes overlap perfectly.

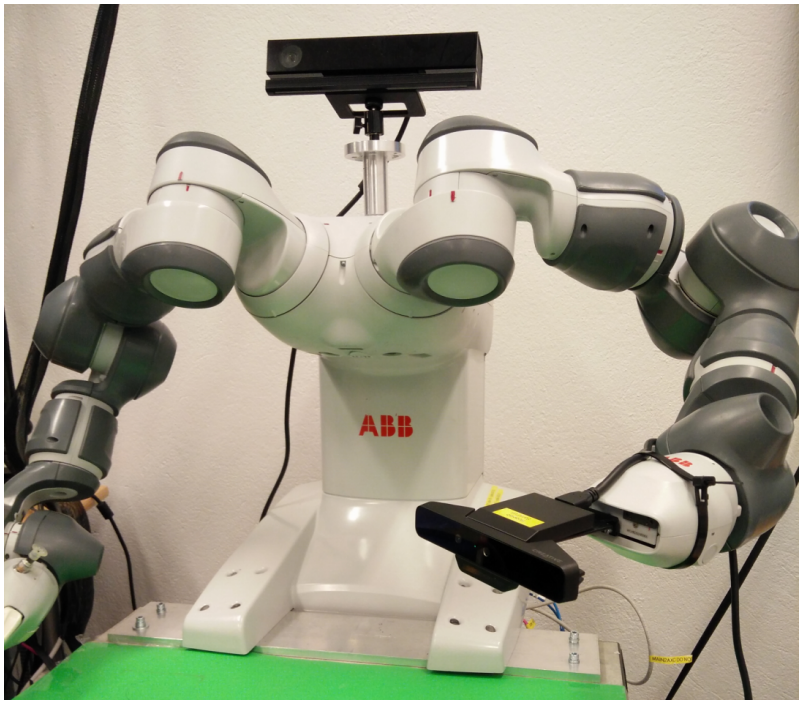


Figure 3.11: The camera mounted on the hand of a YuMi robot.

3.6.1 Fetching Robot Data

The ABB Robot Web Services API [24] is used to fetch data from the robot in a simple way. This is a platform that can interact with the robot controller using the HTTP protocol which allows for communication with the robot in a flexible way. The position and rotation of the robot hand can be read using this platform in a few lines of code, without requiring any special libraries or languages. To access the data on this platform we use Curl [25]. Curl is a library for transferring data using various protocols, and in this case using HTTP. This is used together with the Robot Web Services to get the hand position and rotation data from the robot. The data is fetched as HTML code which is then processed to find certain tags that give the position (x, y, z) and rotation (as a quaternion) of the hand. This data is then used to create the transformation matrix for the hand (see Section 2.4).

3.6.2 Robot-Camera Calibration

To avoid needing to hard code the position and rotation of the camera in relation to the robot hand, we instead use a point-picking method to compute the transformation between the camera and the robot hand, from now on known as the camera-to-hand transformation. This requires the user to capture at least three point clouds from different angles and pick at least three corresponding points in all clouds. This is then used together with the robot data to compute the camera-to-hand transformation. This only needs to be done once as long as the camera does not change position or rotation in relation to the hand. The relation between the robot coordinate systems and the camera coordinate systems can be seen in Figure 3.12. This figure shows the setup used when finding the camera-to-hand transformation using three point clouds captured from different camera positions and rotations.

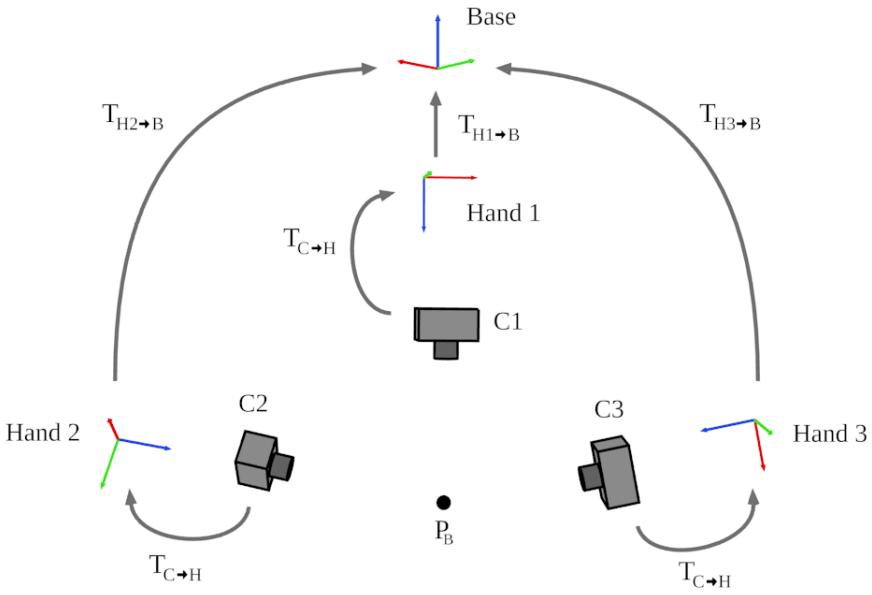


Figure 3.12: The robot camera system.

The robot's base coordinate system is denoted "Base", and the hand coordinate systems are denoted "Hand 1", "Hand 2" and "Hand 3" for the different positions and rotations of the hand. The transformation from hand x to the base coordinate system is given by $T_{Hx\to B}$, and this transformation is computed by the robot controller. The transformation from the camera to either hand is given by $T_{C\to H}$, and this is the unknown transformation that should be found using the points picked by the user. Note that this transformation is the same for all camera orientations, since the camera is assumed to never change orientation in relation to the hand. The point denoted P_B in Figure 3.12 is a known point in the base coordinate system. This point is assumed to be visible in all camera orientations and is called P_{C1}, P_{C2} and P_{C3} for camera 1, 2 and 3 respectively.

Defining the system like this makes it possible to derive an expression for $T_{C\to H}$. This is done by first finding expressions for P_{C1}, P_{C2} and P_{C3} in the following way.

$$P_{C1} = T_{H\to C}T_{B\to H1}P_B = T^{-1}_{C\to H}T^{-1}_{H1\to B}P_B \tag{3.5}$$
$$P_{C2} = T_{H\to C}T_{B\to H2}P_B = T^{-1}_{C\to H}T^{-1}_{H2\to B}P_B \tag{3.6}$$
$$P_{C3} = T_{H\to C}T_{B\to H3}P_B = T^{-1}_{C\to H}T^{-1}_{H3\to B}P_B \tag{3.7}$$

By solving Equation 3.6 for P_B and inserting it into Equation 3.5 the following relation between P_{C1} and P_{C2} is found.

$$P_{C1} = T^{-1}_{C\to H}T^{-1}_{H1\to B}T_{H2\to B}T_{C\to H}P_{C2} \tag{3.8}$$

In the same way, by solving Equation 3.7 for P_B and inserting it into Equation 3.5 the following relation between P_{C1} and P_{C3} is found.

$$P_{C1} = T^{-1}_{C\to H}T^{-1}_{H1\to B}T_{H3\to B}T_{C\to H}P_{C3} \tag{3.9}$$

The only unknown variable in these equations is $T_{C\to H}$ which can be solved by expressing Equation 3.8 and 3.9 as a non-linear least squares minimization problem in the following way.

$$\min_{T_{C\to H}} \sum_i \left(P^i_{C1} - T^{-1}_{C\to H} T^{-1}_{H1\to B} T_{H2\to B} T_{C\to H} P^i_{C2}\right)^2 \tag{3.10}$$

$$\min_{T_{C\to H}} \sum_i \left(P^i_{C1} - T^{-1}_{C\to H} T^{-1}_{H1\to B} T_{H3\to B} T_{C\to H} P^i_{C3}\right)^2 \tag{3.11}$$

Where P^i_{C1} is point i in camera 1, P^i_{C2} is the corresponding point in camera 2 and P^i_{C3} is the corresponding point in camera 3. Note that P_B is not present in these equations, thus it is only necessary to know points in the camera coordinate systems and not in the robot base coordinate system. This makes calibration much easier for the user.

Equation 3.10 and 3.11 can be joined to form one equation in the following way.

$$\min_{T_{C\to H}} \sum_i \left(\begin{bmatrix} P^i_{C1} - T^{-1}_{C\to H} T^{-1}_{H1\to B} T_{H2\to B} T_{C\to H} P^i_{C2} \\ P^i_{C1} - T^{-1}_{C\to H} T^{-1}_{H1\to B} T_{H3\to B} T_{C\to H} P^i_{C3} \end{bmatrix}\right)^2 \tag{3.12}$$

The two equations are simply put in a matrix which is then minimized as before.

When $T_{C\to H}$ has been computed, it can be used in the following way to merge any two point clouds captured with the camera as long as the camera is not moved in relation to the robot hand.

$$P_{CX} = T^{-1}_{C\to H} T^{-1}_{HX\to B} T_{HY\to B} T_{C\to H} P_{CY} \tag{3.13}$$

P_{CX} would then be all points in one cloud with corresponding robot hand transformation $T_{HX\to B}$, and P_{CY} all points in the other cloud with corresponding robot hand transformation $T_{HY\to B}$. Since all transformations are known, P_{CY} can be directly transformed into the coordinate system of P_{CX}, and thus any two clouds can be merged very quickly after the calibration has been done.

This minimization was implemented using a non-linear least squares minimization method called "lsqnonlin" in MATLAB. The minimization tries to find the camera-to-hand transformation by minimizing Equation 3.12 with respect to $T_{C\to H}$, which is defined using 7 unknown parameters. These unknown parameters are the four parameters making up a quaternion for the rotation, and three parameters for the translation in each axis.

3.6.3 Advantages Compared to Registration

Merging two point clouds can also be done using registration as explained in Section 2.3. This method is very similar to the method explained above, but instead of a user manually picking points they are automatically found by using features. The method then tries to find the transformation between two point clouds as in Equation 3.13, however it will find the direct transformation and not the four different robot transformations since it does not use any robot data. This means that the transformation will only be valid for these two clouds, and when new clouds should be merged it needs to find new corresponding points and redo the computations. This is why the points need to be found automatically, otherwise a user would have to pick points for every cloud they want to merge which defeats the point of the system.

Finding corresponding points automatically requires that the scenes have points with good features that allow for accurate registration, which might be problematic if the scene does not contain objects with distinct shapes. The scenes also contain many points which will slow down and decrease the accuracy of the registration algorithm. To get a higher accuracy the ICP algorithm can be used after registration, but this takes time as well. These problems are non-existent when using the robot data for merging, however this requires the camera to be mounted on a robot which is not practical in many applications.

3.7 Pose Estimation

This section will explain the various steps taken in the pose estimation subsystem. The pose estimation depends on older pose estimation results, which makes it a bit complicated to explain in a straight forward manner. For this reason, this section starts off with a high-level overview that roughly explains the goal and method of the pose estimation before going into the details.

3.7.1 Overview

The goal of the pose estimation is to use identification data for each cluster in a scene to find the best poses of the identified objects in that cluster, where a pose is defined as both position and rotation of an object. The basic idea of the pose estimation is to use the methods explained in Section 2.3 to register an object view to the cluster. The identification data contains identified objects and their most probable views for each cluster. However, the identification only works for single clouds, and to get increasingly better pose results for each new cloud, the identification data needs to be available for merged clouds as well. This is because the pose estimation relies on inlier fractions to determine how good a pose is, and thus it will get more accurate when more clouds are merged and more points are available in the merged cloud. For this reason there needs to be a way to link the object identification data from the single cloud to the merged cloud. To get even better results the pose estimation uses historical data from the pose estimation on the previous cloud, which allows it to get more accurate for every new cloud. This means that there needs to be a way of linking pose estimation data from the previous cloud as well. The linking of identification and pose data is explained in Section 3.7.2.

The final result of the pose estimation is the following. Every cluster will have a list of objects, and every object in this list contains at least one object view. This means that the pose estimation has managed to fit this object view into the cluster with a sufficiently high inlier fraction. Each object view will also have the pose (transformation) for that object view in the cluster, as well as the inlier fraction for that transformation. The pose estimation also applies the final transformation on the complete object cloud and computes the inlier fraction, this can be seen as an accuracy measurement of how well the complete object fits into the cluster. The more points are on the object in the scene, the higher this accuracy will be (assuming that the pose is correct). In this way there can be many different objects with many different views and transformations for the same cluster, and they can be sorted from best to worse using the inlier fraction.

3.7.2 Linking Data to Merged Cloud

Linking clusters from different clouds is necessary to allow the latest merged cloud to use current identification data and previous pose data. The segmentation subsystem will not segment all clouds into the same clusters since the segmentation is different depending on the position of the points in a cloud. This means that the latest single cloud might have more or less clusters compared to the latest merged cloud. The goal of the "linker" is to link identification data (from the latest single cloud) to the latest merged cloud, such that the clusters in the merged cloud are identified according to the latest identification data. This is done by linking the clusters of the single cloud to the clusters of the merged cloud, to see which single cloud clusters correspond to which merged cloud clusters. When this is done the identification data for a single cloud cluster can then be used for the corresponding merged cloud cluster.

The way the clusters are linked is fairly simple. All clusters in the single cloud are first transformed using the robot data such that they are in the same coordinate system as the merged cloud. Then the following algorithm is executed

1. Select a cluster C_s in the single cloud
2. For each cluster C_m in the merged cloud
    (a) If any point in C_s is identical to any point in C_m, the clusters are linked

This method takes advantage of the fact that every merged cloud contains every previous single cloud. By transforming the clouds to the same coordinate system, the points will perfectly line up and can easily be compared.

In the same way the linker is used to link the previous merged cloud to the current merged cloud, such that pose data for the previous merged cloud can be used in the current merged cloud. This is done in the same way as for the single cloud. After the linking is done, the latest merged cloud now has both identification data and pose estimation data for most clusters, except for those clusters that were not found in any previous cloud. This data is used in the view merging.

3.7.3 View Merging

As explained in the overview, the pose estimation uses merged views from the latest identification data and previous pose data to estimate the best poses for clusters in the latest merged cloud. This is done by merging the object views from the previous pose data with the object views from the latest identification data, and use these merged views when estimating the new poses. The reason for using object views instead of the entire object model is to get better feature correspondences in the registering phase. The features on the full object model can be vastly different compared to a cluster in the scene since occluded points in the cluster will affect its features. These points are not occluded in the object model, which will give rise to different features. By using identified views for the pose estimation we can avoid this problem and get better and fast pose estimation.

The view merging function is fairly simple. It loops through the previous pose data and finds all objects in the pose data that are also identified in the latest single cloud. If an object has a good pose in the previous merged cloud, and the same object is identified in

the latest single cloud (although with different views), then there is a high chance that it is the correct object. Some of the best views from the previous pose data is then merged with some of the best views from the latest identification data for the same object, to form new merged views which are added to a list. Due to the way the views are created (see Section 3.4.1) this is simply done by adding the two view point clouds together. When this has been done for all poses in the previous pose data, some of the best unused views from the latest identification data is added to the list as well. Keep in mind that these are not merged views, they are simply taken straight from the identification data, and their purpose is to avoid that the pose estimation gets "stuck" on only using previous data. If the previous data is not good, the pose estimation needs to get new views such that it can continue, and this can be seen as a sort of "exploration" of possible views for the cluster. The list with all merged views is then used in the registration which gives new poses where these views fit in.

3.7.4 Registration

One problem with finding the pose of object views without distinct shapes is that they fit well into many different clusters. For example, an object view consisting of mostly a plane can fit well into any cluster containing a similar plane, even if the rest of the cluster looks vastly different. To solve this one can register the cluster to the object views instead of the object views to the cluster. This means finding a transformation such that the cluster is transformed onto an object view with a sufficiently high inlier fraction for the cluster. If the inlier fraction is too low (defined by a hard threshold), the cluster does not fit well with the object view and that object view can be discarded. Note that the inlier fraction for the *object view* can be rather high in this case, if the object view fits well in the cluster. This first step of the pipeline can be seen in the left part of Figure 3.13. In this figure C is a cluster and $V1, V2$ and $V3$ are object views. A transformation for the cluster to each object view is found, which is visualized as $T_{C->VX}$ in the figure, where X is either 1, 2 or 3. The green check mark signals that the cluster fit well into the object view, while the red cross signals that the cluster did *not* fit well into the object view. By registering the cluster to the object view many clusters can be immediately and quickly discarded due to low inlier fraction. The remaining clusters will then go on to the next stage of the registering process.

The object views that the cluster fit well into are certain to be somewhat similar in shape to the cluster. These object views now need to be registered to the cluster to find the object views that fits the best. This is done by simply registering each object view to the cluster and computing the object view inlier fraction for each one. The transformation of the object views can then be sorted on inlier fraction to find the best poses. This can be seen on the right side of Figure 3.13. In this figure the object view $V2$ has an inlier fraction that is too low, and it is discarded. $V1$ has an inlier fraction above the threshold and is thus chosen as the best object view and its transformation $T_{V1->C}$ is chosen as the best pose for this cluster. This is done for each cluster found in the original scene, where each cluster will have different object views depending on the results from the object identification.

In practice, the pose estimation is done by finding the 10 best (with respect to inlier fraction) transformations for the object view in the cluster. These 10 transformations are then grouped in the transformation space, meaning that similar transformations are

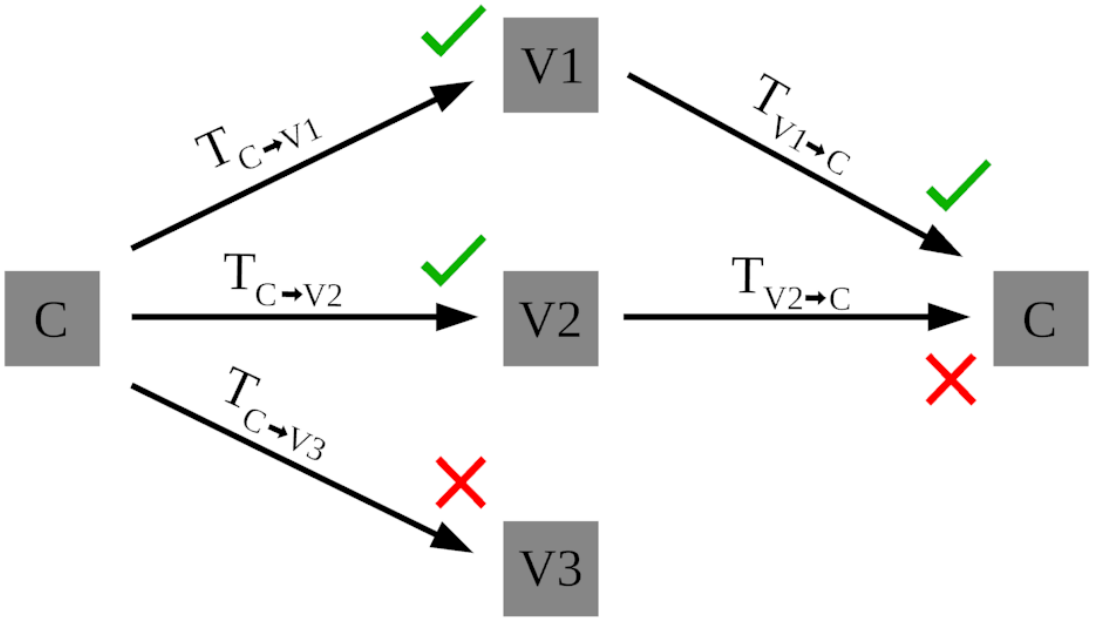


Figure 3.13: The pose estimation pipeline for one cluster. The cluster is first registered to each object view to determine if the cluster fits well into the object view. The object views that passed this first phase are registered to the cluster to find the final transformation.

grouped together and form a transformation cluster (not to be confused with point cloud clusters). Transformations are defined as similar if the distance and rotations between them are smaller than certain thresholds. The average transformation and the average inlier fraction is then computed for each transformation cluster, and the final transformation is chosen as the one with the highest average inlier fraction.

The registration is implemented using a RANSAC loop with a prerejection step that is based on geometric constraints between point sets, as was explained in Section 2.3.

3.8 Hint System

The hint system is the final stage of the pipeline. The task of the hint system is to reject bad poses, correct viewpoints from the pose estimation and decide where the camera should be moved in order to collect better data.

3.8.1 Determining Valid Nodes

For each pose result the identified view and pose is loaded into the hint system along with the associated view-graph and complete point cloud for the model (the definition and appearance of the view-graph is explained in Section 3.4.1). The view-graph and complete point cloud for the model are then transformed using the estimated pose to fit into the scene.

We can then later on select new camera positions by picking nodes in the transformed view-graph. For practical reasons however, we would only like to chose nodes that are above the work table (moving the camera to a position underneath the table is neither possible nor preferable). We would thus only like to chose between the nodes that are positioned above the work table.

The equation of the estimated plane in $\mathbb{R}^3$ from the segmentation in Section 3.3 is given by:

$$ax + by + cz + d = 0 \tag{3.14}$$

where (a, b, c, d) are the plane coefficients. The normal $\boldsymbol{n}$ of the plane is given by: $\boldsymbol{n} = (a, b, c)$ and the distance from an arbitrary point $p = (x_0, y_0, z_0)$ to the plane is given by:

$$D = \frac{ax_0 + by_0 + cz_0 + d}{\sqrt{a^2 + b^2 + c^2}} \tag{3.15}$$

where D is the distance from the point p to the plane. If $D > 0$ then the point is located on the same side as the direction of the normal vector and equivalently if $D < 0$ then the point p is on the opposite side. We can always assume that the camera is located above the work table and facing downwards. By always having the plane normal aligned towards the sensor viewpoint (the position of the camera) we can define that points that are above the plane, i.e. on the same side as the direction of the normal vector, are in fact also above the table. Using this information we can select nodes in the graph that are positioned above the plane and thus also above the table by imposing:

$$D = \frac{ax_0 + by_0 + cz_0 + d}{\sqrt{a^2 + b^2 + c^2}} > 0 \tag{3.16}$$

Since the denominator is always strictly positive in Equation 3.16, we can simplify it into:

$$D = ax_0 + by_0 + cz_0 + d > 0 \tag{3.17}$$

We will denote nodes that are above the table ($D \geq 0$) as *valid nodes* and nodes that are below the table ($D < 0$) as *invalid nodes*. For practical reasons, we also only validate nodes that are above the table by a certain threshold: $D \geq d_{thr}$, where d_{thr} was set to 10 cm.

3.8.2 Pose Rejection and Viewpoint Correction

While an estimated pose might have a large amount of inliers, it is not guarantied that the pose is valid in terms of the real world. In the real world the robot has a camera mounted on the robot arm and a work table where all objects are placed. The pose estimation however ignores all these facts and solely tries to find a match with as many inliers as possible. As

a result, the best pose and viewpoint for that pose that comes out of the pose estimation might be a pose or viewpoint that is located underneath the table. That is why the hint system first performs a validation test in order to reject bad poses. After the validation test, the identified viewpoint is corrected to align with the real camera and its viewpoint.

Pose Rejection

First we perform a validation test that rejects bad poses where the complete model is located inside or partly underneath the table. It is safe to assume that the objects are not embedded into the table. The way this is done is by checking if there are any points in the complete model that are below the estimated plane from the segmentation in Section 3.3. For each point in the complete model we use Equation 3.17 to see if any point is below the plane. Of course some poses might have one or two points that are theoretically below the plane depending on how well the plane was estimated during the segmentation. In order not to reject these poses a threshold is also introduced here in which only poses that have any points below a certain threshold are rejected. This threshold was experimentally set to 1 cm which means that a pose that has a complete model for which there is any point below the plane by 1 cm is rejected.

Identified Viewpoint Correction

The poses from the pose estimation are estimated using the identified synthetic view point clouds V as explained in Section 3.4.1. The identification of these views are not perfect which is why the pose estimation uses the 8 best matching views from the object identification. However, it is not guarantied that the best matching pose has the correct viewpoint (the correct viewpoint being the viewpoint of the real camera). That is why we use our view-graph to search for the node which has the best viewpoint alignment with the real camera. We do this by measuring the angle between the z-axis of each valid node in the view-graph and the real camera, using Equation 2.1 in Section 2.1.3. We then select the node with the smallest angle and thus the smallest z-axis difference (the z-axis is always the view-direction of the camera). Once we have found the correct viewpoint and corresponding point cloud from this view, we recompute the inlier fraction to determine if this still remains a good pose. If the pose is good then the inlier fraction should not differ that much from the original viewpoint of the pose (in fact, the inlier fraction might be even better given the correct viewpoint). If however the inlier fraction for the corrected viewpoint is below a certain threshold we can reject the pose in the same way that was done in Section 3.7.4.

Once the validation test and camera correction have been carried out we end up with two sets of pose results, valid and invalid. If there are no valid pose results for a cluster then the hint system notifies this to the operator. For scenes with valid pose results, the model with the best pose is selected according to the inlier fraction. All other models with valid pose results are considered similar models and a strategy to distinguish between these models will be explained later using the combined distinguish-utility.

3.8.3 Definition of a Good View

In order to search for a better view we need to define what makes a view better than another view. We decided to let this be determined by an absolute value where a high value yields a better view compared to a view with low value. This value is a weighted sum of four utilities; view-utility, normal-utility, feature-utility and distinguish-utility which where all explained in Section 3.4.5.

The distinguish-utility for a given view is determined by how similar the identified model is with respect to the other similar models and their views. Let us say we have a scene and a cluster within that scene that we have successfully identified as model A. This model also has an estimated pose with the highest number of inliers in the scene and thus the best matching model. Let us also say that the models B and C have valid pose estimations but with lower amounts of inliers than model A for their poses. The inlier fractions for model B and C are though sufficiently high enough not to draw the conclusion that we identified model A. In order to distinguish between these models we have to move towards a view that maximizes the distinguish-utility for these models. As explained in Section 3.4.5, the distinguish-utility is determined by the matching score of two models. So in our example we have a set of distinguish-utilities for model A and B and their rendered views, and model A and C and their rendered views. So in order to distinguish between model A and B we would like to move towards a view that maximizes the distinguish-utility for model A and B. Likewise for model A and C we have to move towards a view that maximizes the distinguish-utility for model A and C. This raises the question of how to move towards a view that maximizes the distinguish-utility between all similar models, in this case model A, B and C. We solved this by combining all distinguish-utilities for the identified model A and the similar models by:

$$u_i^{\text{Combined}} = \max(u_i^{A \to B}, u_i^{A \to C}, \ldots) \tag{3.18}$$

where u_i^{Combined} is the combined distinguish-utility for model A given the similar models B, C,... for view i. After combining the distinguish-utilities we normalize them as we have done for all other utility values (see Section 3.4.5).

The equation for the summed utility value f_i for a view i can be seen below in Equation 3.19.

$$f_i(\boldsymbol{u}) = \boldsymbol{w}^T \cdot \boldsymbol{u} = w_1 \cdot u_1 + w_2 \cdot u_2 + w_3 \cdot u_3 + w_4 \cdot u_4 \tag{3.19}$$
$$\boldsymbol{w} = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{pmatrix}, \quad \boldsymbol{u} = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix}$$

where $\boldsymbol{w}$ is a weight vector and $\boldsymbol{u}$ is the utility value vector containing the four different utilities for the given view i described above (u_1 = view-utility, u_2 = normal-utility, u_3 = feature-utility and u_4 = combined distinguish-utility). All utilities are normalized between 1 and 0 and by applying the constraint $|\boldsymbol{w}| = w_1 + w_2 + w_3 + w_4 = 1$ we also get that $f(\boldsymbol{u}) \in [1, 0]$.

In case we do not find any similar models, e.g. if only model A is present in our valid pose results, we only add the other three utilities to the weighted sum:

$$f_i(\boldsymbol{u}) = \boldsymbol{w}^T \cdot \boldsymbol{u} = w_1' \cdot u_1 + w_2' \cdot u_2 + w_3' \cdot u_3 \tag{3.20}$$
$$\boldsymbol{w} = \begin{pmatrix} w_1' \\ w_2' \\ w_3' \end{pmatrix}, \quad \boldsymbol{u} = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix}$$

where u_1 = view-utility, u_2 = normal-utility, u_3 = feature-utility. In order to still fulfill the constraint $f(\boldsymbol{u}) \in [1, 0]$ we have to modify the weights w_1, w_2 and w_3. This was done by adding an equal amount of w_4 to all other weights:

$$w_1' = w_1 + \frac{w_4}{3} \tag{3.21}$$
$$w_2' = w_2 + \frac{w_4}{3} \tag{3.22}$$
$$w_3' = w_3 + \frac{w_4}{3} \tag{3.23}$$

The choosing of weights was not a trivial task as will be explained later in Section 5.5. However, we ultimately decided to use the combination $\boldsymbol{w}^T = \begin{pmatrix} \frac{2}{9} & \frac{2}{9} & \frac{2}{9} & \frac{1}{3} \end{pmatrix}$ which puts more weight on the distinguish-utility while still having equal weights for the view, normal and feature-utilities.

3.8.4 The Search for an Optimal View

Once the summed utility $f_i(\boldsymbol{x})$ has been computed for each view i in the graph, the search for an optimal view with better utility begins. As explained in Section 3.4.1, each model has a graph where each rendered view has a node (virtual camera position) and links that are connected to the nearest neighbors of the node (see Figure 3.8).

The view with the best utility is sought for using the valid nodes in the graph. This is done by simply going through all the valid nodes and picking the node that has the highest summed utility $f(\boldsymbol{x})$.

3.8.5 Path to Optimal View

Finding the most optimal view given the current view is a relatively simple problem. However, we do not want our system to necessarily jump directly to the most optimal view if the view is located far away from the current view. The goal of our system is to continuously retrieve data and thus obtain a better pose estimation over time. So instead of making a "big jump" to the most optimal view, we would like to collect as many good views as we can along the way. To implement this we decided to use the Dijkstra's search algorithm which explores all paths starting at the current node and ending at the destination node (the node with the best summed utility). To choose which path is the most optimal one we introduced a cost function that penalizes big jumps and prioritizes views with high utility along the way. We also made the graph containing the valid nodes completely connected

meaning that between each pair of valid nodes there exists a link, implementing the possibility of going from any valid node within the graph instead of just moving along the existing links in the graph. So in theory it would be possible to make a big jump from the starting node (currently identified node) to the best node with the highest summed utility. To prevent this though we decided to implement a cost function that prevents the path from making big leaps in the graph and still prioritizing views with high utilities. The cost function for a given path is given in Equation 3.24.

$$J(f_i(\boldsymbol{x}), d) = g(f_i(\boldsymbol{x})) + h(d) \tag{3.24}$$

where $f_i(x)$ is the summed utility for a given view i, $g(f_i(\boldsymbol{x}))$ the cost function for the utility of the next view and $h(d)$ is the cost function for traveling the Euclidean distance d to the next view.

The way we decided to implement the functions g and h was purely experimental by trying out different functions. We wanted to have something that really penalized big jumps and still had a linear behavior for small to medium jumps (skipping one or maybe two bad view-nodes). An exponential function fulfills both of these criteria. We also wanted a function for g that prioritized a good view-node over a bad view-node even though the path of the bad view-node is shorter than the path of the good view-node. Instead of searching for the shortest path in distance we would search for longer paths that have better view-nodes. This would make it more expensive to jump to a closer view-node with low utility than exploring a longer path that has better view-nodes with higher utilities.

3.8.6 Moving the Camera to a Better View

The hint system suggests a trajectory or path that leads the camera to an optimal view while still collecting as much good data as possible along the path as possible. However, it is worth noting that the path is only a suggestion and the operator does not need to traverse the path in order to get to the optimal view. Although, if the operator wants to move according to the path then the next position for the robot hand needs to be computed. As explained earlier, the view-graph is transformed to fit the scene according to the transformation given by the best valid pose estimation. The position and rotation for the next viewpoint for the camera is thus given directly in the view-graph. The problem is that the transformation of the next viewpoint is for the camera base coordinates so we have to apply a change of basis in order to get the transformation for the robot base coordinates. This change of basis is found by taking the current position of the robot hand and applying the transformation from the robot hand to the camera. In other words:

$$C = T^R_{H\to B} T^R_{C\to H} \tag{3.25}$$

where C is a basis transformation from the camera base coordinates to robot base coordinates, $T^R_{H\to B}$ is the current position of the robot hand in robot base coordinates and $T^R_{C\to H}$ is the camera-to-hand transformation estimated through the calibration in Section 3.6.2. So if we now apply this to the next position we get:

$$T^R_{C_{\text{next}}} = CT^C_{C_{\text{next}}} \tag{3.26}$$

where $T^R_{C_{\text{next}}}$ is the next position for the camera in robot base coordinates, $T^C_{C_{\text{next}}}$ is the next position for the camera in camera base coordinates and C is the change of basis transformation. So now we have the next position for the camera in robot base coordinates and to get get this position for the robot hand we apply the inverse of the camera-to-hand transformation:

$$T^R_{H_{\text{next}}} = T^R_{C_{\text{next}}}(T^R_{C\to H})^{-1} \tag{3.27}$$

3.8.7 Using Past Data

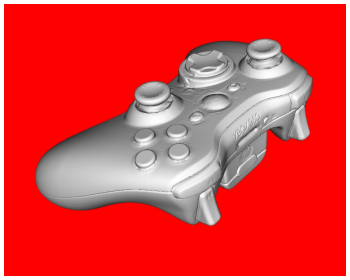
It is important that the hint system keeps track of previous camera positions in order to not suggest the camera to move to an already visited camera view. That is why the hint system saves the current camera position for all scenes. If the hint system detects that there exists data from previous views it loads all previous camera positions. For all pose estimation results it then tries to find all views that aligns with the current and previous camera positions using the same method as explained in Section 3.8.2. All nodes that share the same or similar viewpoint as the previous camera viewpoints are marked as invalid nodes. Doing this will prevent the graph-search from visiting and suggesting these nodes again.

Chapter 4
Results

This section shows and explains the results of each subsystem and of the entire system as a whole. Section 4.1 shows the resulting raw and unprocessed point cloud from the SR300 camera, obtained from the point cloud capturer. Section 4.2 shows the segmentation of the point cloud from the point cloud capturer which results in a number of unknown clusters. Section 4.3 shows the results from the identification of an unknown point cloud cluster. Section 4.4 shows the results from the calibration program used to find the camera-to-hand transformation as well as the resulting merged point clouds. In Section 4.5 we show the results from the pose estimation on each identified cluster. Section 4.6 then shows how the hint system computes a new and better view. Finally, in Section 4.7 we show the results from the complete system with all subsystems interconnected. The models referred to in this section can be found in Figure 4.1.

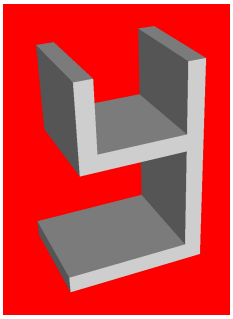
(a) **Playstation 3 controller**. Model accessed from GrabCAD[26]

(b) **Playstation 4 controller**. Model accessed from GrabCAD [26]

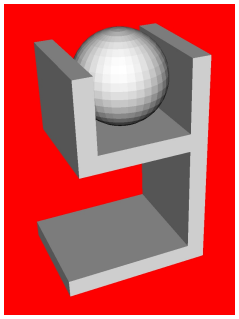


(c) **Wireless xbox controller**. This CAD model is scanned using a depth sensor with high detail. This is the wireless version which contains a small battery pack underneath the controller. Model accessed from Thingiverse [27].

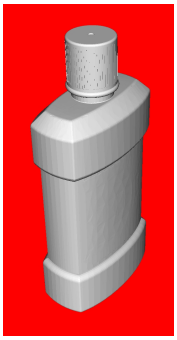
(d) **Wired xbox controller**. This is the wired version of the controller which doesn't contain a battery pack underneath. Apart from that, this model is identical to the wireless version. Model accessed from GrabCAD [26].



(e) **Wood object**. This is a small wooden structure which contains a little holding place for a small object on the top.



(f) **Wood object with ball**. This is a wooden structure with a small ball located in the holding place. Apart from the small ball, this model is identical to the "Wood object".



(g) **Listerine bottle**. This is a CAD model of a Listerine mouthwash bottle. Model accessed from GrabCAD [26].

Figure 4.1: A list of all the CAD models used for this thesis. The models were accessed from GrabCAD Community [26] and Thingiverse [27] (except for the wood structure models).

4.1 Point Cloud Capturer

As explained in Section 3.2, the point cloud capturer simply captures point clouds using the Intel RealSense SR300 depth sensor. A typical point cloud can be seen in Figure 4.2. The left part of the figure shows the scene visualized as a regular 2D image, and the right part shows the same scene visualized as a point cloud. This shows the shadows that occur in point clouds, which are caused by the fact that the sensor is unable to capture points behind obscuring surfaces.

(a) A scene visualized using a regular 2D image. (b) The same scene visualized as a point cloud.

Figure 4.2: A typical point cloud of a scene.

4.2 Segmentation

The results for the segmentation using plane removal can be seen in Figure 4.3. Figure 4.3(a) shows the original point cloud straight from the point cloud capturer. The objects are lying on a green table which is mostly flat. Figure 4.3(b) shows the point cloud with the largest plane removed. It can be seen that due to imperfections in the depth sensor, some parts of the plane are not removed, mainly at the edges of the plane. Figure 4.3(c) shows the downsampled point cloud. This is downsampled using a leaf size of 3 millimeters. Figure 4.3(d) shows the clustered point cloud. It can be seen that some small clusters of points visible in the previous figure are now gone. This is because the system requires a certain number of points in each cluster, otherwise the cluster is not valid. Figure 4.3(e) is the same figure as before, but with all clusters in a different color to more easily distinguish them. It can be seen that the clustering is not perfect, and some objects are joined into the same cluster.

A similar result is achieved when using segmentation based on background subtraction. Figure 4.4 shows the same scene as for the plane segmentation, but segmented using background subtraction. Usually background subtraction results in a better segmentation since small variations in the background are taken into account as well. The effect of imperfections in the sensor is also decreased when merging many views of a background,

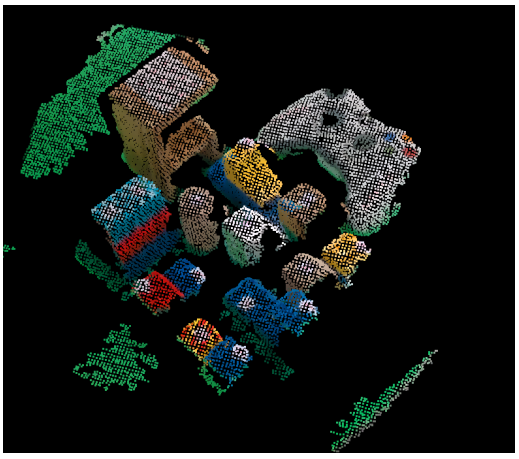
which will make the segmentation results better. This can be seen when comparing Figure 4.3(e) with Figure 4.4(b), in the former figure the Lego bricks and the cylinder to the left of the scene are joined in one cluster (blue points), but in the latter the two objects are different clusters (yellow and cyan points). The explanation for this is that the background segmentation has managed to more accurately remove part of the background between these objects, which allows them to be separated.

The adjustable parameters used for the segmentation are the following. Downsample leaf size is 3 millimeters. The maximum distance from a point to a plane to count as an inlier is 3 millimeters. The number of iterations used when finding a plane using RANSAC is 200. The maximum distance between points to count as an inlier when using background segmentation is 6 millimeters. The radius used when estimating normals is 4 millimeters. The maximum distance between two points to belong in the same cluster is 1.1 times the leaf size, or 3.3 millimeters. The minimum number of points in a cluster is 200, and the maximum is basically infinite.

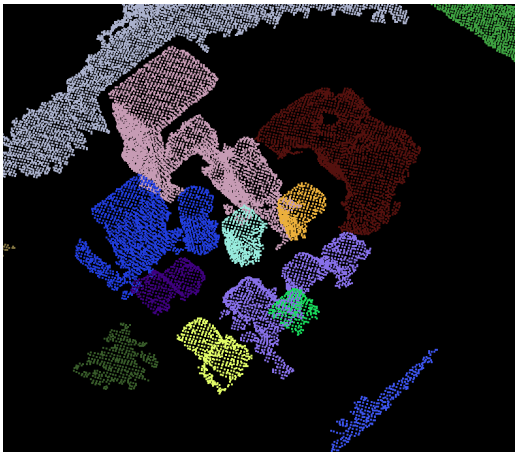
(a) The original point cloud.

(b) The same cloud with the largest plane removed.

(c) The downsampled cloud.

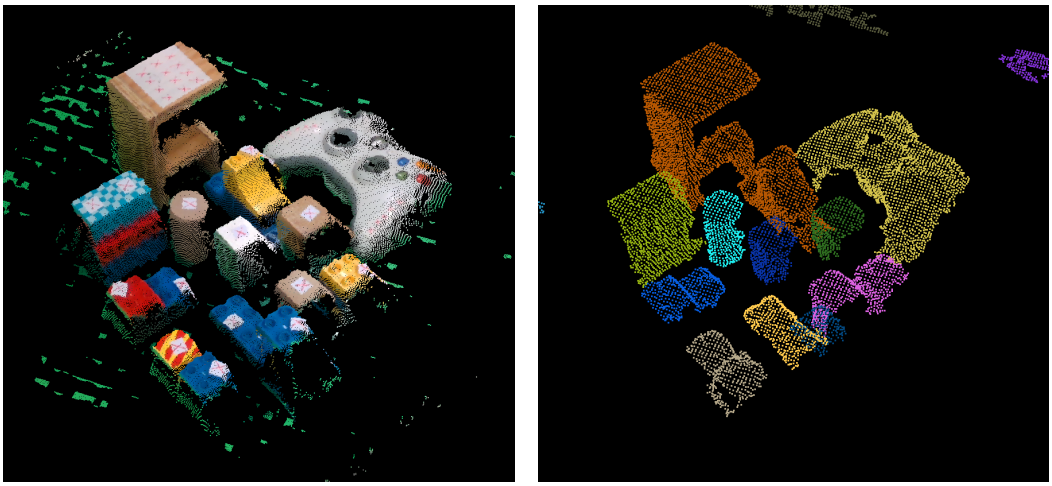


(d) The clustered cloud.



(e) Same as (d), but with each cluster visualized in a different color.

Figure 4.3: Segmentation results when using plane removal.



(a) The same scene as in 4.3(a), but with the background subtracted.

(b) The clustered cloud.

Figure 4.4: Segmentation results when using background subtraction.

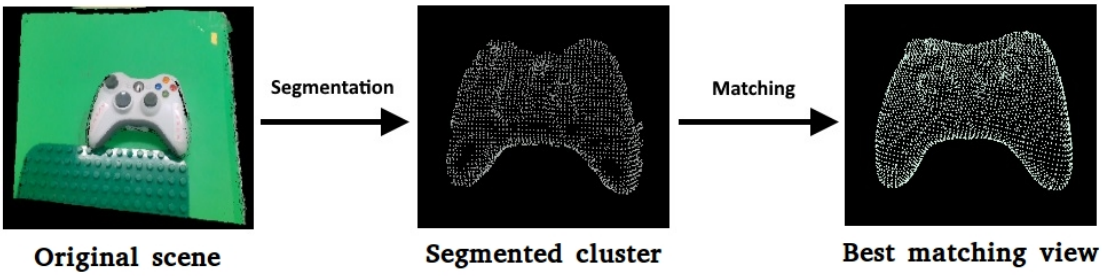
4.3 Object Identification

In this section we show the identification results from one scene containing the wireless xbox controller model (see Figure 4.5). In the scene we have performed segmentation on the original scene point cloud and obtained the cluster representing the object. A visual representation of the identification results is shown in Figure 4.6, accompanied by Table 4.1 showing all the most significant data from the identification result.

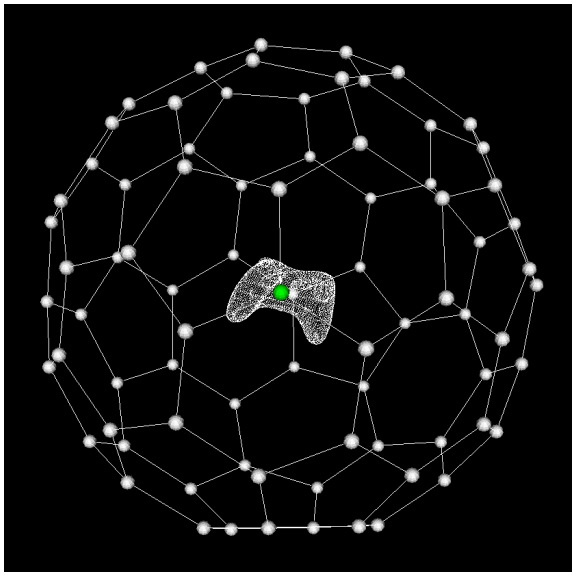
Figure 4.5: Original point cloud of the scene captured by the camera containing a wireless xbox controller.

The identification of the cluster containing the wireless xbox controller model is shown in Figure 4.6 where the best identified view as well as the view-graph are displayed. The matching score of the global ESF features which determines the quality of the match, is

shown in Table 4.1. We display the 10 best matching views for two models. The reason for this, as mentioned in Section 2.2.2, is because within the ten best matching views for the ESF feature we will with 90% certainty find the correctly identified view.



(a) Visual representation of the identification process. The original scene point cloud is segmented into clusters. Each cluster is then matched with the trained model base to find the best matching synthetic view-point-cloud.



(b) The view-graph for the identified model. The green sphere is the position of the virtual camera that rendered the best matching view in (a)

Figure 4.6: A visual representation of the identification results from the scene in Figure 4.5.

Wired xbox controller	
View	Score
76	0.075
72	0.087
14	0.090
24	0.095
9	0.098
11	0.104
15	0.104
8	0.123
27	0.134
79	0.139

Wireless xbox controller	
View	Score
12	0.087
8	0.088
15	0.092
11	0.102
14	0.105
9	0.110
52	0.119
13	0.133
4	0.135
48	0.142

Table 4.1: The matching score from the identification of the wireless xbox controller. The first column shows the synthetic view index and the second column shows the matching score. The score is computed by taking the L1-norm (see Equation 2.6) for the difference between the cluster histogram and the synthetic view histogram. A low score approximately between $0.05 - 0.2$ indicates a good match. As we can see, both the wireless and wired xbox controller models have similar matching score which is expected since we cannot differentiate between these models from the current view (see Figure 4.1). Only the matching scores from two models were shown here for simplification.

4.4 Scene Merging

The calibration program with point picking used to find the unknown camera-to-hand transformation is shown in Figure 4.7. In this figure there are three different clouds taken from different angles, and the user has picked a couple of corresponding points in all clouds. The points can be seen as red spheres in each cloud. When this is done the data is sent to a compiled MATLAB script that uses a non-linear least squares minimization to find the camera-to-hand transformation as explained in Section 3.6.2. When the transformation has been found the clouds can be merged, which is shown in Figure 4.8. Figure 4.8(a) shows all clouds in the same viewer, without any attempt at merging them. Figure 4.8(b) shows all clouds after using robot data and the camera-to-hand transformation to merge them. It can be seen that there are less shadows in this merged scene since the points come from three different angles. With a good calibration (well chosen points and enough difference in camera angles for the clouds) the merging is very good with an offset of 2-3 mm at most. The non-linear least squares minimization function used is the "lsqnonlin" function in MATLAB.

The adjustable parameters that were changed from default for this function are the following. FunctionTolerance and OptimalityTolerance was set to $1.0 \cdot 10^{-10}$ and MaxFunctionEvaluations was set to 2000. The starting point for the transformation was very close to the actual transformation.

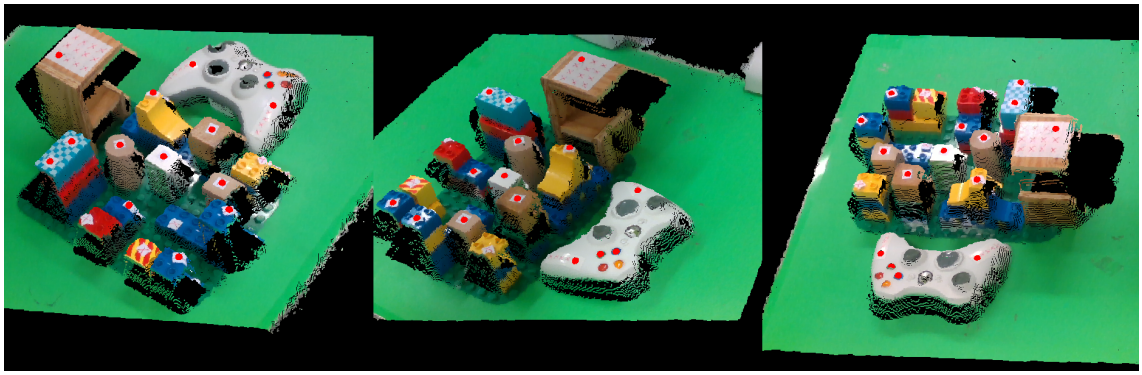
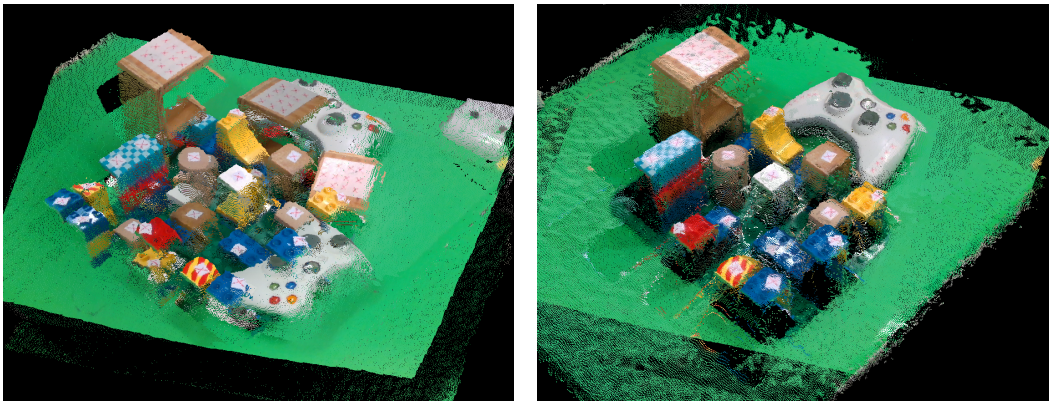


Figure 4.7: The calibration program with point picking.



(a) The point clouds shown in the same viewer.

(b) The point clouds after merging using robot data and camera-to-hand transformation.

Figure 4.8: The merged clouds.

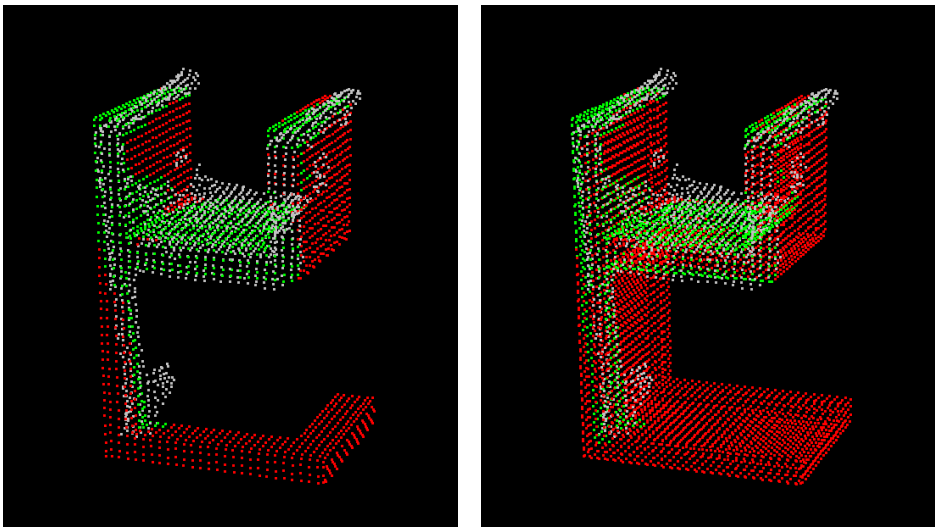
4.5 Pose Estimation

Figure 4.9 shows a good pose for an object view of the wood object model. In this figure the pose seems to fit well, but it is hard for the system to measure that since the inlier count is still very low due to a lack of points in the cluster. This can be seen by the many red points in Figure 4.9(b).

Figure 4.10 shows a bad pose for an object view of the wood object model in the same cluster as in Figure 4.9. In this figure the pose does not fit well, but again it is hard for the system to measure it since it still has quite a few inliers. However, we can see that this pose has fewer inliers than the pose in Figure 4.9(a).

Figure 4.11 shows two point clouds of a scene taken from different angles, as well as the identified views for the wood object model in both scenes, and the final merged view of these two views. This merged view would be used in the pose estimation of the merged scene consisting of the two scenes in the figure.

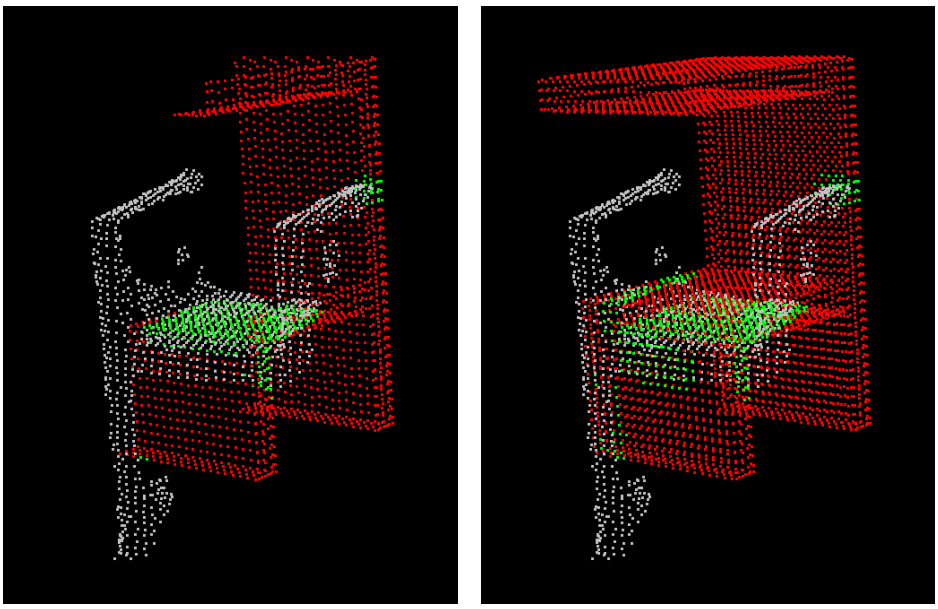
The adjustable parameters we have used for the view merging is the following. For each object found in the previous pose estimation results, 2 of its best views are merged with 8 of its best views from the current object identification results. We have a maximum of 40 merged views to limit the time the pose estimation takes.



(a) A good pose found for one view of the wood object model.

(b) The same pose as in (a) applied to the full CAD model of the wood object.

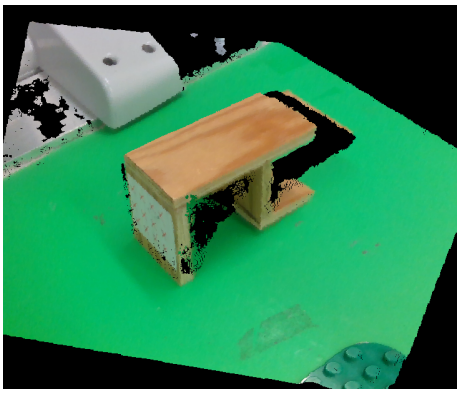
Figure 4.9: A good pose estimation of the wood object model. Gray points are the cluster, green points are object inliers and red points are object outliers.



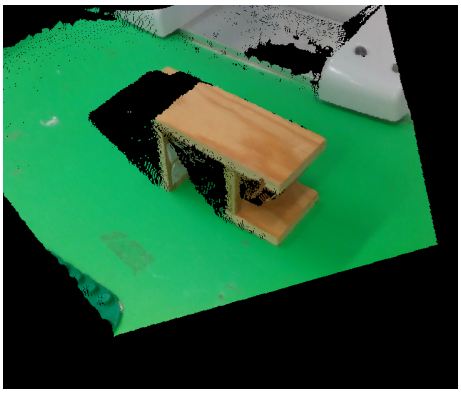
(a) A bad pose found for one view of the wood object model.

(b) The same pose as in (a) applied to the full CAD model of the wood object.

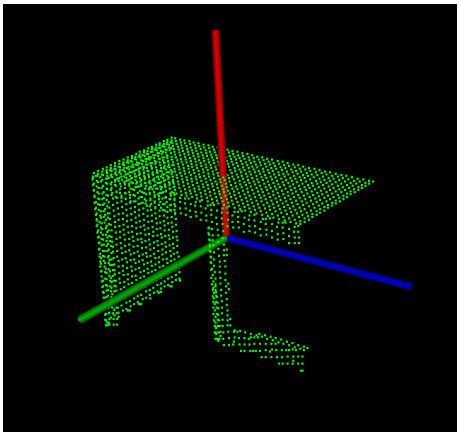
Figure 4.10: A bad pose estimation of the wood object model. Gray points are the cluster, green points are object inliers and red points are object outliers.



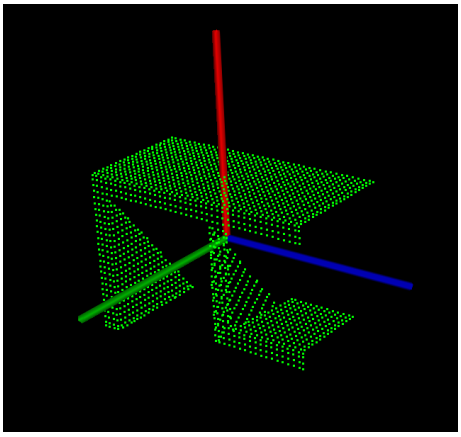
(a) A scene where the underside of the wood object model is visible.



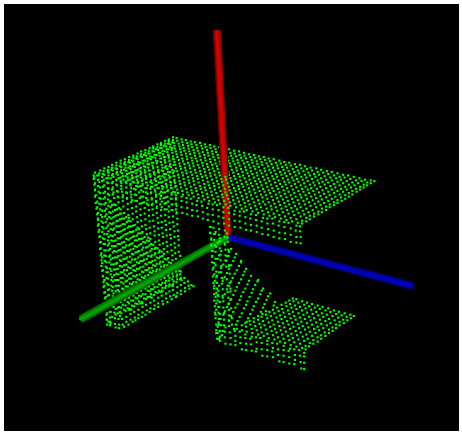
(b) The same scene as (a) taken from another camera angle.



(c) An identified view of the wood object model in scene (a).



(d) An identified view of the wood object model in scene (b).



(e) The resulting merged view after merging the views in (c) and (d).

Figure 4.11: Two camera angles for the same scene, their identified views and the final merged view to be used for pose estimation.

Figure 4.12 shows the results of the cluster-to-object registration in the pose estimation subsystem. The gray cloud is the cluster, which is the lower base of the YuMi robot. The green cloud is the object view, which is a controller. A pose for the object view could probably be found such that the inlier fraction is high by jamming the controller into the robot base. However, the system first registers the cluster to the controller, and since it will get a very low inlier fraction for that registration this object view is immediately discarded. Without this form of rejection, the controller would get matched into the cluster and have a fairly high inlier fraction.

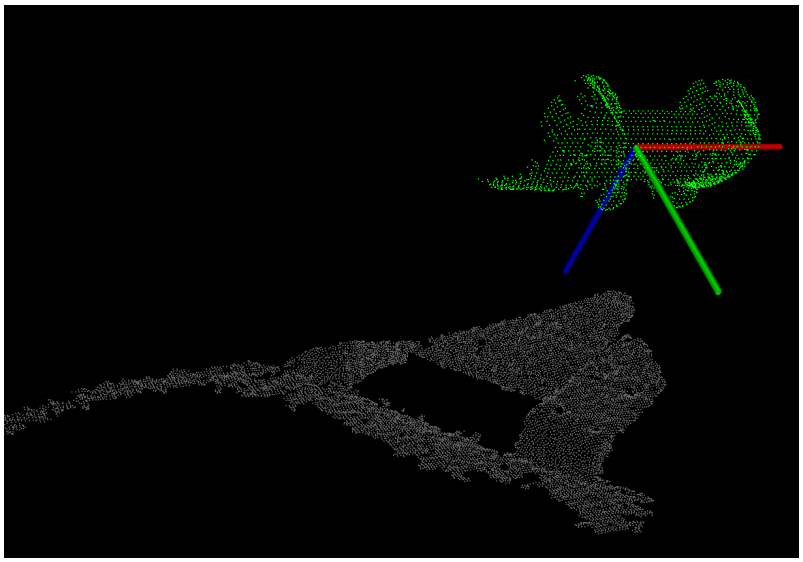


Figure 4.12: An object view of a controller (green cloud) that was rejected by the pose estimation because the cluster (gray cloud) could not fit into it with high enough inlier fraction.

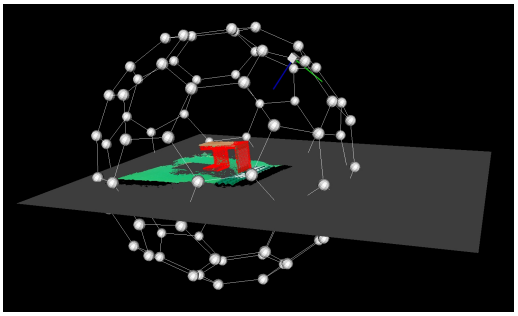
The adjustable parameters we have used in the pose estimation are the following. Downsample leaf size is 3 millimeters. The radius used when estimating local features is 15 millimeters. Number of iterations for each pose estimation is 10,000. The maximum distance between points two count as inliers is 1.5 times the leaf size, or 4.5 millimeters. The similarity threshold between edge lengths of the underlying geometrical correspondence rejector in the sample consensus method is 0.8. The minimum inlier fraction needed for a cluster when being registered to an object view is 0.5. The minimum inlier fraction needed for an object view when being registered to a cluster 0.1.

4.6 Hint System

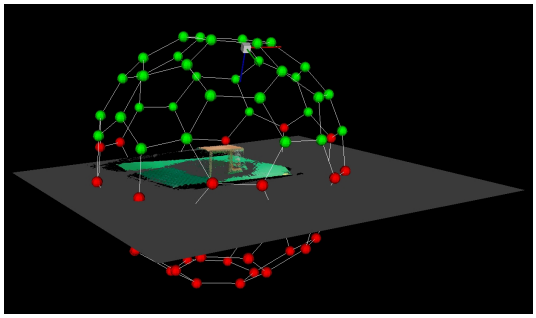
In this section we first show how the pose rejection and viewpoint correction works for the hint system as well as sorting out the valid and invalid nodes in the view-graph. We then try to visually explain the impact of the different utilities as well as the weighted sum of utilities in the view-graph.

4.6.1 Determining Valid Nodes

From Section 3.8.1 we concluded that valid nodes that are positioned 10 cm above the working table in front of the robot. This was achieved by selecting the nodes that were 10 cm above the estimated plane from the segmentation algorithm. In Figure 4.13(a) we see a scene with the estimated plane and transformed view-graph for a pose. The red point cloud represents the complete model which is transformed into the scene with the estimated pose. In Figure 4.13(b) we see the valid nodes marked by green spheres and invalid nodes marked by red spheres. Determining the valid nodes will assure that the hint system later only searches through the set of valid nodes to find a new position.



(a) In this figure we see the estimated plane as well as the fitted view-graph for a given pose.



(b) Here all valid nodes are marked by a green sphere and the invalid nodes are marked by red spheres. A valid node is a node that is located 10 cm abode the estimated plane.

Figure 4.13: These figures display the transformed view-graph and valid nodes for a scene. The view-graph is centered around the identified object by applying the same transformation as the estimated pose from the pose estimation algorithm. The real camera position is marked by a coordinate system and a white box.

4.6.2 Pose Rejection and Viewpoint Correction

In Section 3.8.2 we explained that some of the poses and their identified viewpoint from the pose estimation algorithm are incorrect with respect to the real world. There was a validation test followed by a viewpoint correction performed on the pose estimation data to determine whether a pose was valid or invalid. The validation test examines if the complete model for a pose had any part underneath the working table. In Figure 4.14 we see an example of a pose that is marked as invalid due to the complete model being partly underneath the table.

Pose estimations that pass the validation test are then given new viewpoints that corresponds to the real camera viewpoint. If a pose uses previous data and has multiple merged scenes then all identified views are aligned according to the current and previous camera locations. In Figure 4.15 we see three examples of viewpoint correction. In the first Figure 4.15(a) we see an example of a single viewpoint correction. The identified view from the pose estimation is for convenience here marked by a black sphere and as we can see,

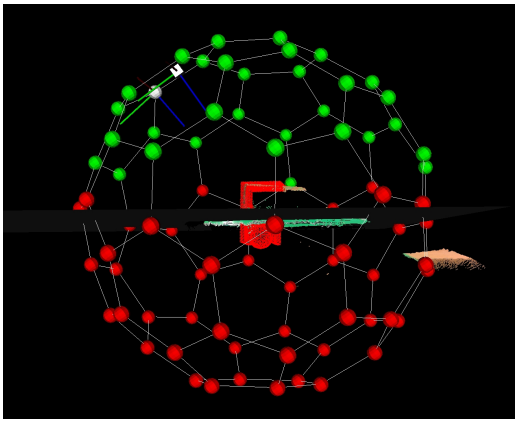
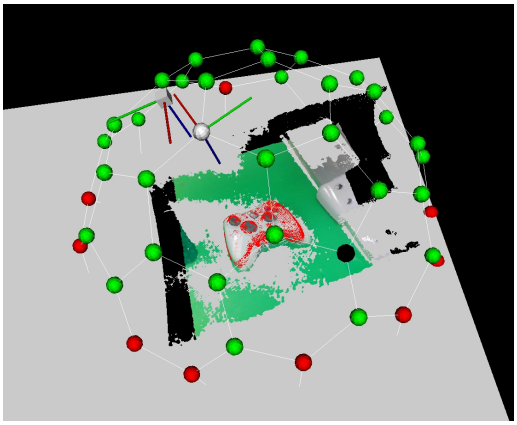
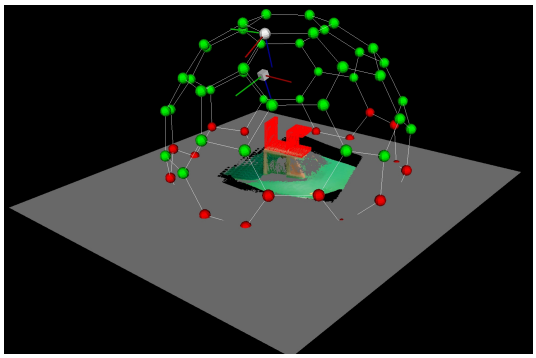


Figure 4.14: Validation test for an estimated pose. The complete model for this pose is partly located underneath the table which means that this pose is invalid

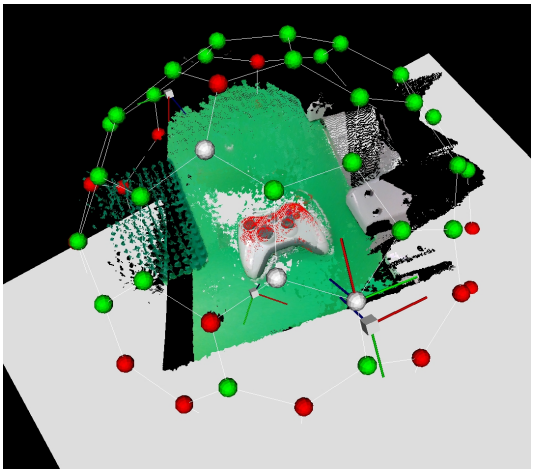
it is incorrect with respect to the real camera viewpoint (the white box). The estimated pose however is very good so we should not discard this pose but rather fix the viewpoint to align with the real camera. In the second Figure 4.15(b) we see a similar viewpoint correction for the wood object model. The identified viewpoint from the estimated pose is located underneath the table and so we search for the view that aligns the best with the real camera instead. The problem here is that for the new viewpoint that aligned with the real camera, the inlier fraction is very low (in fact it is almost zero). A pose with an inlier fraction this low would not pass as a valid pose during the pose estimation so we mark this pose with corrected viewpoint as invalid. Finally in the third Figure 4.15(c) we see multiple viewpoint corrections for both current and previous camera positions. All corrected, identified views are marked by white spheres and the current and previous camera viewpoints and positions are marked by white boxes (previous camera positions are marked by smaller white boxes).



(a) A viewpoint correction for an estimated pose. The identified view from the pose estimations is marked by a black sphere and the corrected viewpoint is marked by a white sphere. The corrected viewpoint is aligned with the real camera viewpoint.



(b) Another viewpoint correction for an estimated pose of the wood object model. The identified viewpoint from the pose estimation is located below the plane. The point cloud corresponding to the corrected viewpoint (white sphere) is however invalid due to low amount of inliers in the cluster.



(c) Viewpoint corrections for multiple viewpoints. This occurs when the hint system detects previous merged data. All viewpoints are aligned with the current camera position (white box) and previous camera positions (small white boxes).

Figure 4.15: These images displays the camera viewpoint correction performed by the hint system. A viewpoint or multiple viewpoints from the pose estimation is corrected to align with the real camera viewpoints. Corrected viewpoints are marked by white spheres.

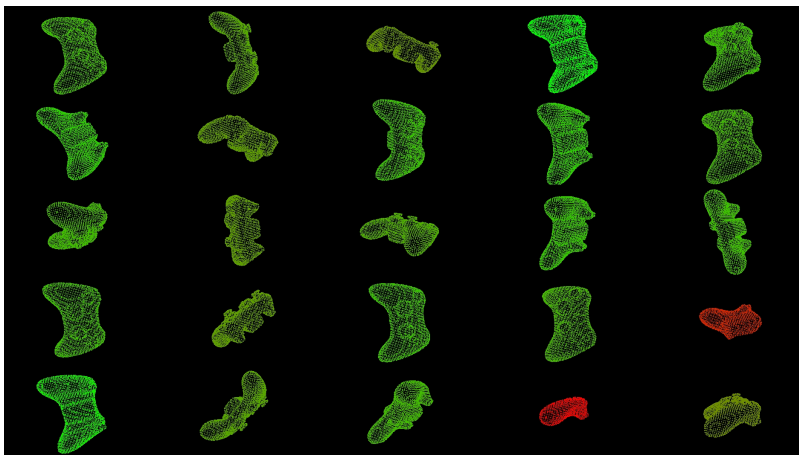
4.6.3 Utilities

In this section we show visually how the utilities work by using the rendered views. Each utility will be displayed first and at the end we show what the weighted sum of utilities look like. The explanation of all utilities can be found in Section 3.4.5. All examples will contain the utilities of the wood object model and the wireless xbox controller model.

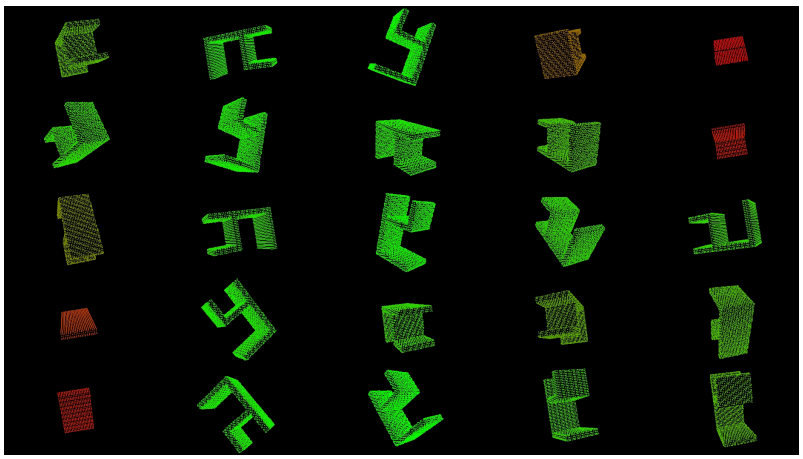
A sample of 25 views is shown with a color scheme for which a green point cloud is equivalent with high utility value and a red point cloud is equivalent with low utility value.

View-Utilities

The view-utility is a measurement of how much of the object that is visible for a view. In Figure 4.16 we see the view-utilities for both objects. It is easy to see that a high view-utility means that we see a bigger portion of the object for that view. The low utility values are mostly due to the Euclidean cluster extraction which was explained in Section 3.3.2 and 3.4.2 where we only keep the largest cluster if the point cloud for a rendered view was disconnected. The view-utility will thus help the system pick views that display a larger, connected part of the object.



(a) The view-utility for 25 rendered views of the wireless xbox controller model.

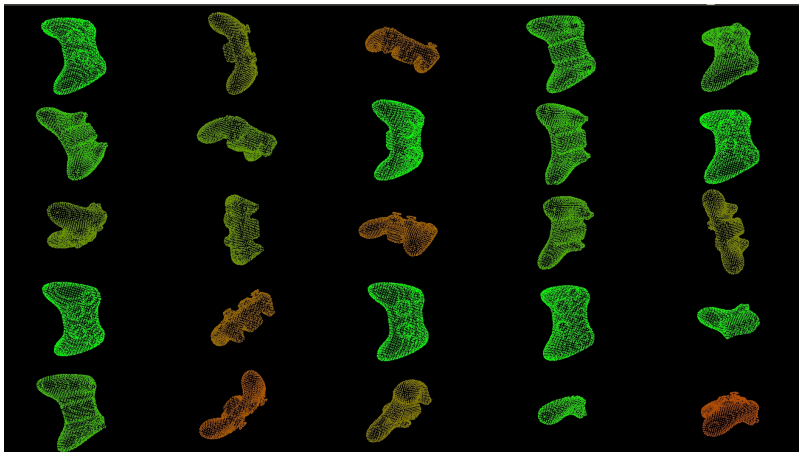


(b) The view-utility for 25 rendered views of the wood object model.

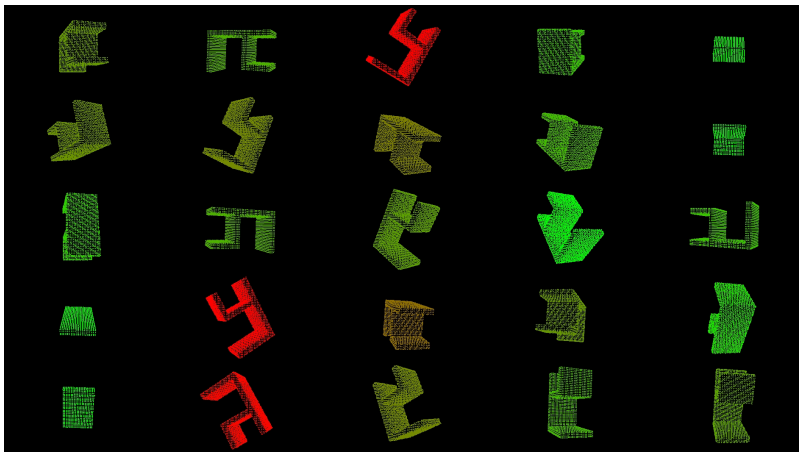
Figure 4.16: The view-utilities for the wireless xbox controller and the wood object.

Normal-Utilities

The normal-utility is a measurement of the quality of the surface normals for a view. A bad surface normal points perpendicular or close to perpendicular to the viewpoint direction of the camera which will most likely lead to measurement noise and outliers for that surface using the depth sensor. In Figure 4.17 we see the normal-utilities for the wireless xbox controller model and the wood object model. The normal-utilities tend to favor views where the normals of flat surfaces are not perpendicular to the camera viewpoint which is exactly what we want.



(a) The normal-utility for 25 rendered views of the wireless xbox controller model.



(b) The normal-utility for 25 rendered views of the wood object model.

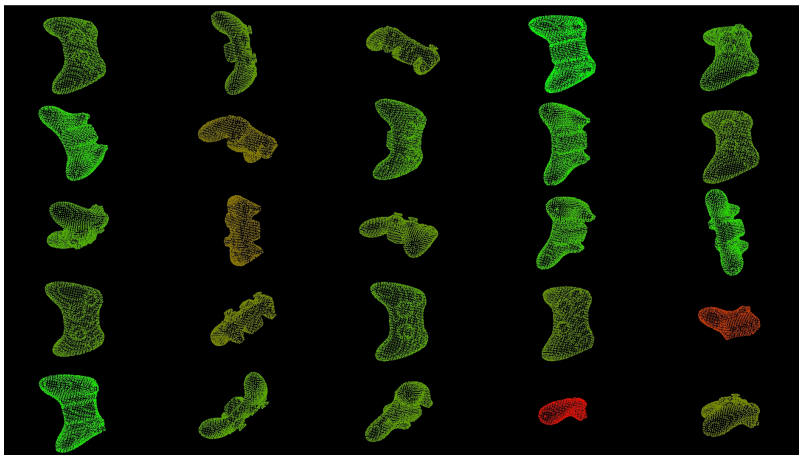
Figure 4.17: The normal-utilities for the wireless xbox controller and the wood object.

Feature-Utilities

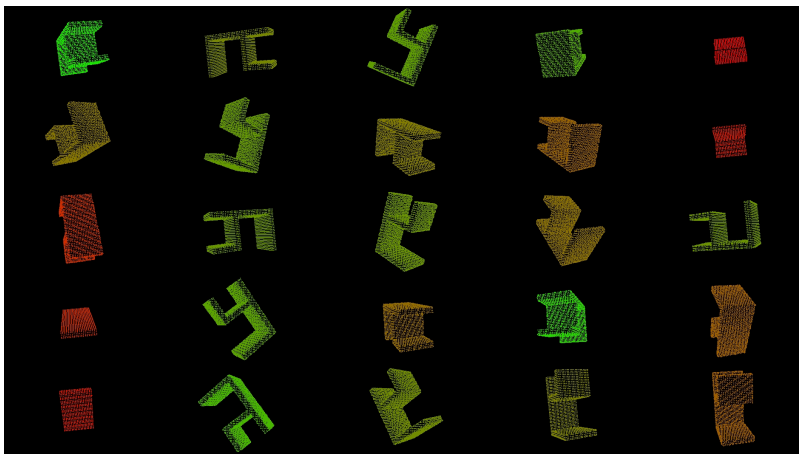
The feature-utilities are a measurement of the quality of the local features for a view. Figure 4.18(a) shows the feature-utility for 25 different views of the wireless xbox controller

model. Due to the wireless xbox controller having distinct features on both the front and back, there is not a big difference between these views. However, generally the views showing the underside of the controller have a higher score due to very distinct features around the edges of the battery pack. The side views have a very low score since there is no interesting geometry there and most points have identical features.

Figure 4.18(b) shows the feature-utilities for 25 different views of the wood object model. For this object it is much more apparent what sort of views lead to higher feature-utility. It can be seen that views which show more edges and corners have a higher feature-utility. This is because most of this object consists of planes where points will have the same features, but the edges and corners will have vastly different features and thus they will be more distinct.



(a) The feature-utility for 25 rendered views of the wireless xbox controller model.



(b) The feature-utility for 25 rendered views of the wood object model.

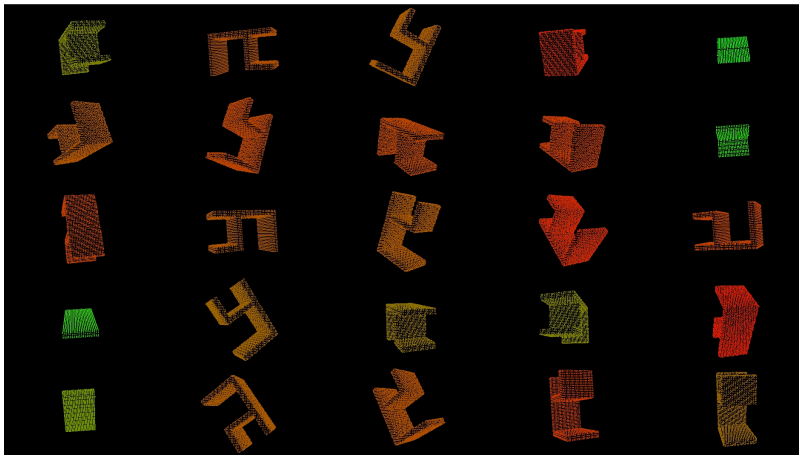
Figure 4.18: The feature-utilities for the wireless xbox controller and the wood object.

Distinguish-Utilities

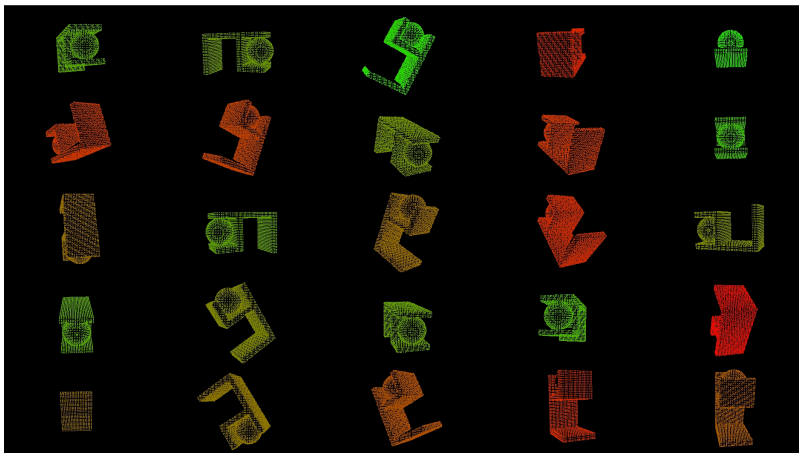
The distinguish-utility is a measurement of how similar two objects are and how much we can distinguish between them for different views. A low utility value means that the view is very similar to another view for another object. This time we decided to show the utility values for four models. The first two models are "Wood object" and "Wood object with ball" which are two very similar objects of a wooden structure. "Wood object with ball" is the exact same object as "wood object" but with a small ball located in a small holding place. The last two objects are the wireless and wired xbox controller models which are also very similar to each other. The wireless xbox controller model is a controller which is wireless and thus contains a battery pack underneath the controller. The wired xbox controller model is, as the name suggests, a wired controller and does not have a battery pack underneath the controller. Remember from section 3.4.4 that the distinguish-utilities are computed by matching two models and extracting the best matching score for each view. Remember also that the matching of model A with B is not the same as the matching of model B with A.

In Figure 4.19 and 4.20 we see the distinguish-utilities for all models and their most similar model. For Figure 4.19 we see that the utility value is high for views where we see the holding place and ball which are, not surprisingly, the views where we can distinguish between these models. The views with low utility values are views that are identical with respect to the other model and thus non-distinguishable. We see the same principle for Figure 4.20, i.e. views with high utility are views that show the bottom of the controller where the battery pack is (or is not) located. Views on top of the controller (where all buttons are located) are identical and thus very hard to distinguish between and so these views are given a very low utility value.

Notice also in both Figure 4.19 and 4.20 how the sequence of matching the models yields different results. In e.g. Figure 4.19(a) and 4.19(b) we see that the view in the first row and third column have different color and thus also different utility value. In Figure 4.19(a) we matched the "Wood object" model with the "Wood object with ball" model and in Figure 4.19(a) we did the opposite i.e. matched "Wood object with ball" with "Wood object".

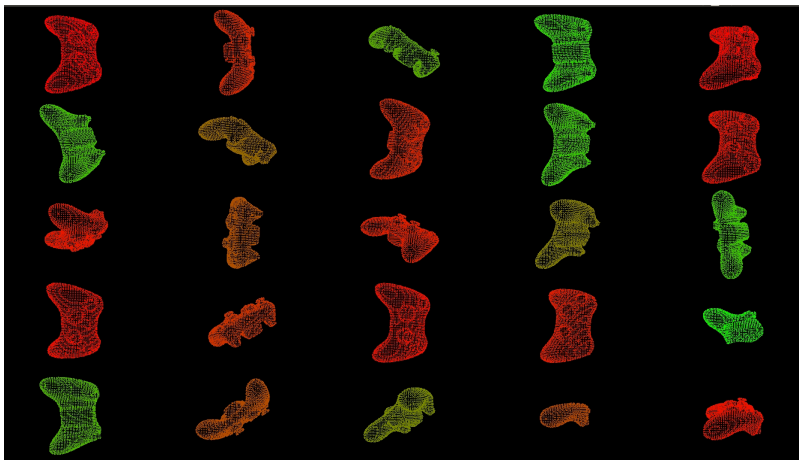


(a) The distinguish-utility for 25 rendered views of the wood object model.

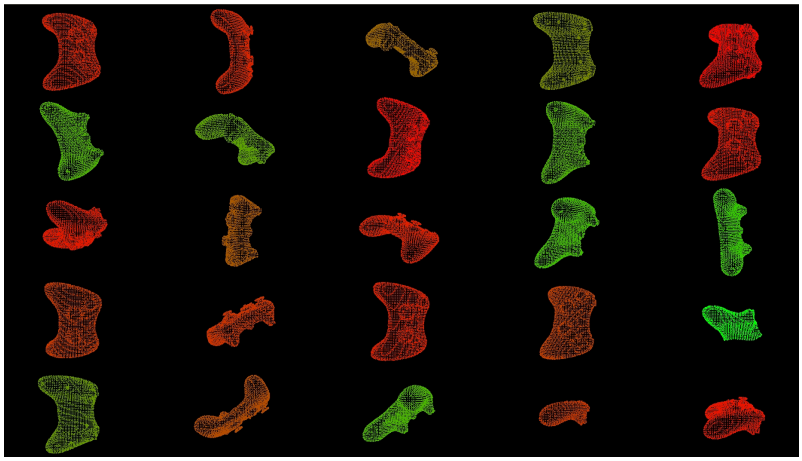


(b) The distinguish-utility for 25 rendered views of the wood object with ball model.

Figure 4.19: The distinguish-utilities for the wooden structure with and without ball.



(a) The distinguish-utility for 25 rendered views of the wireless xbox controller model.



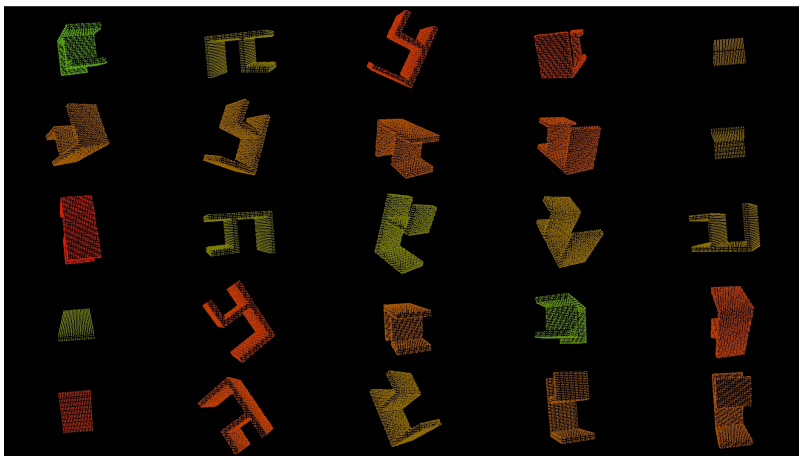
(b) The distinguish-utility for 25 rendered views of the wired xbox controller model.

Figure 4.20: The distinguish-utilities for the wired and wireless xbox controller.

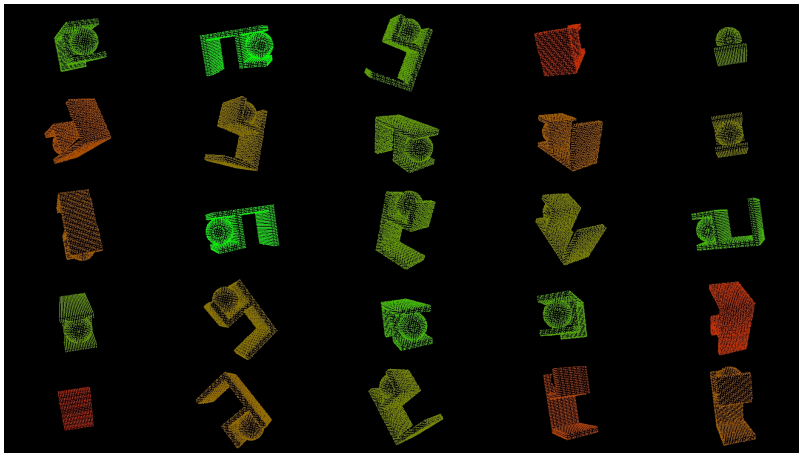
Summed Utilities

The summed utilities are the result of combining all utilities according to Equation 3.19. The weights used here are: $\boldsymbol{w}^T = \begin{pmatrix} \frac{2}{9} & \frac{2}{9} & \frac{2}{9} & \frac{1}{3} \end{pmatrix}$. We show the results in the same way as for the distinguish-utilities i.e. for the model pairs "Wood object" vs "Wood object with ball" and "Wireless xbox controller" vs "Wired xbox controller". These results are what the hint system uses in order to find a new and better view.

The summed utilities are shown in Figures 4.21 and 4.22. Note that using different weights will have different impact on the summed utilities.

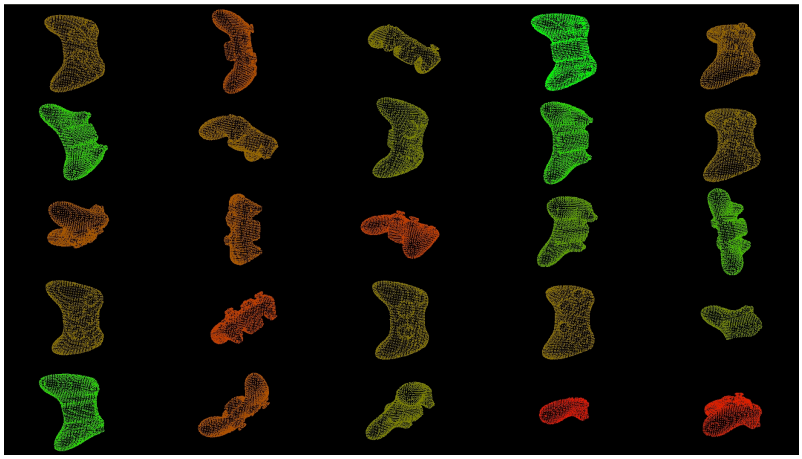


(a) The summed utility for 25 rendered views of the wood object model.

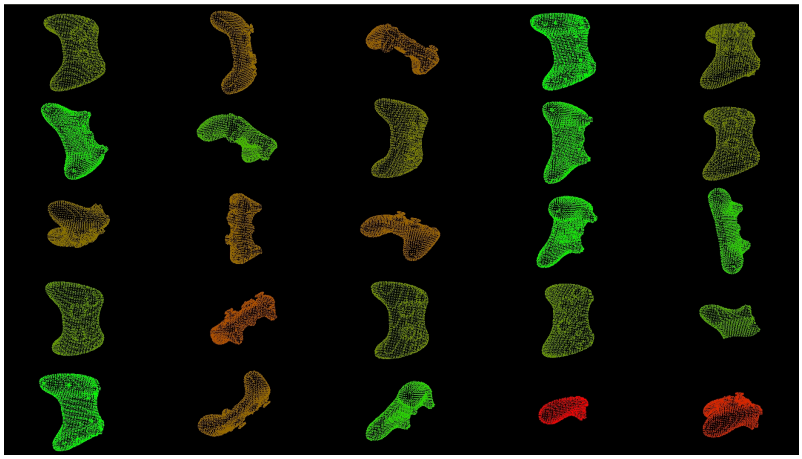


(b) The summed utility for 25 rendered views of the wood object with ball model.

Figure 4.21: The summed utilities for the wooden structures with and without ball.



(a) The summed utility for 25 rendered views of the wireless xbox controller model.



(b) The summed utility for 25 rendered views of the wired xbox controller model.

Figure 4.22: The summed utilities for the wired and wireless xbox controller.

4.7 System Function Evaluation

In this section we show the complete functionality of the system. The task of the hint system is to suggest a better viewpoint given the currently identified viewpoint. It searches through the valid nodes and finds the most optimal one using the weighted sum of utility values. It also suggests a trajectory or path that takes the camera from the current view to the most optimal view. This path is supposed to penalize big jumps in the graph and prioritize good views along the way. In all the figures displayed in this section we have the following color scheme. The nodes marked as white spheres are the identified views and the white boxes are the current and previous camera position (the large box indicates the current camera position and the small boxes indicate previous camera positions). All valid nodes are colored according to their utility value where a high utility value is marked by a green sphere and a low utility value is marked by a red sphere. Invalid nodes are blacked out. The most optimal view is marked by a blue sphere and the path leading towards the optimal view is colored with blue lines. To visualize the estimated pose we transform the complete model (red point cloud) with the estimated pose to fit the scene. We also show the accuracy for each model in the scene which is a measure of how many inliers there are for the complete model.

We ran two experiments, one for the wireless xbox controller model and one for the wood object model. The experiments start in a viewpoint where no or small distinction can be made between the models. We then moved the camera along the suggested path by the hint system in order to gather more and better data. This was done three times capturing three views for each experiment. The idea is that the pose estimation and identification of each model are supposed to get better the more data we collect.

4.7.1 Initial View

In Figure 4.23 and 4.24 we see how the hint system works for the initial view of a scene. We see that the matching pose of the wireless xbox controller model in Figure 4.23 is bad simply due to not having enough data to make a good pose estimation. The hint system suggests that we move upwards with the camera to capture a new and better view. Note that the hint system only follows the matching model and not the actual cluster in the scene. In this case moving upwards will probably capture more of the backside of the controller and thus correct the pose. We also see that the best matching similar model was the playstation 3 controller model. The most probable reason why the wired xbox controller model did not show up as a similar model was because it did not have any valid pose results.

In Figure 4.24 we see the initial view for the wood object model. As we can see the initial pose estimation is far from perfect which is due to not having enough data in order to make a good pose estimation. We thus need to move the camera to a new position to collect more data. The same phenomenon is seen here as in Figure 4.23 in which the view-graph is centered and transformed according too the fitted model and not the cluster. However, moving the camera to the next suggested view will also here lead to more data which will improve the pose estimation.

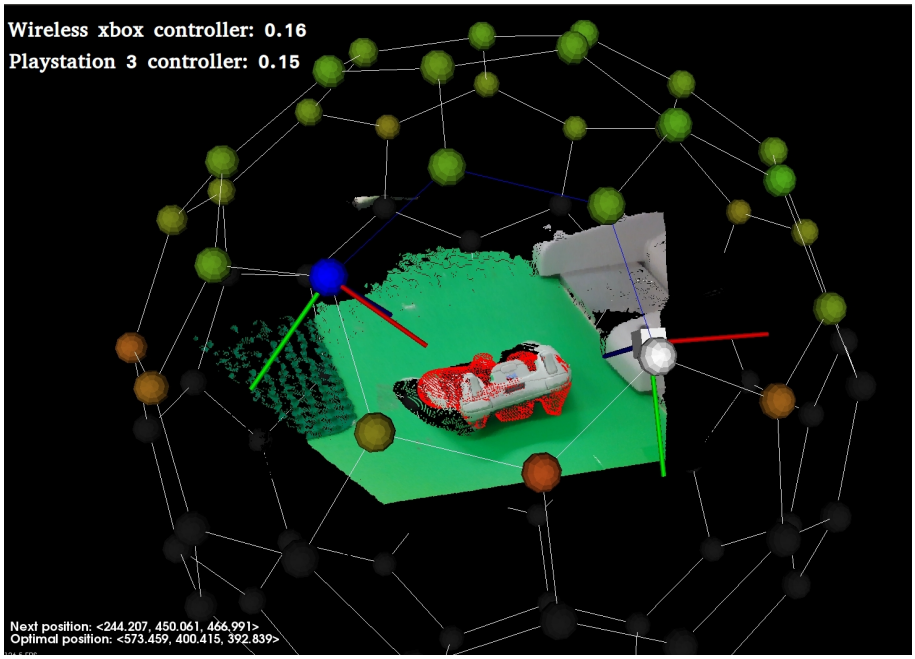


Figure 4.23: Initial view of the wireless xbox controller model. The matching pose of the object is incorrect. However, the hint system is unaware of this fact and thinks that the controller is facing upwards when in fact the controller is facing downwards. Nonetheless, The hint system suggest that we move up to get a better view. This will probably lead to a better view of the controller and thus correct the pose.

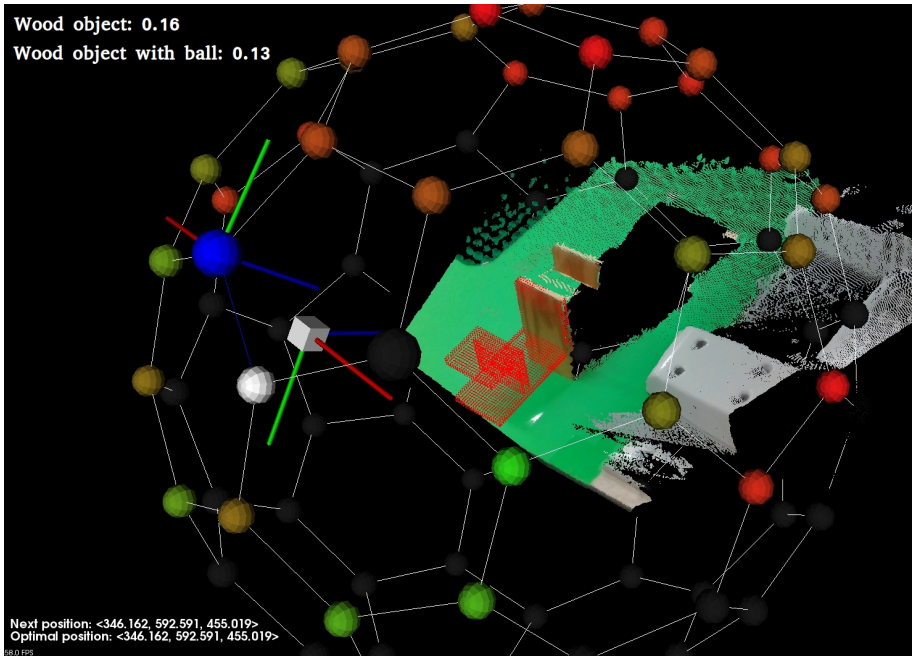


Figure 4.24: Initial view of the wood object model. As we can see the match from the pose estimation is poor and we need to collect more data to make a better estimate.

4.7.2 Second View

In Figure 4.25 we see the result of moving the camera according to the hint system for the initial view. Luckily, we now have a better view of our object and so the pose estimation for our model is aligned with the cluster. However, the hint system still did not detect the wired xbox controller model as a similar model which means that the pose estimation still has not found any valid pose for this model. The hint system did not detect any similar models so it tries to maximize the other utility values in order to get a better view (this was explained in Section 3.8.3).

In Figure 4.26 we see the second and merged view for the wood object model. Since we received more data for the second view we are now able to make a better pose estimate for the model. We have two similar models for this scene, "Wood object with ball" and "Listerine bottle". In order to maximize the distinguish-utility for these models the hint system suggest that we move the camera above the object in order to get a better view of the holding place (which does not contain a ball for our current model). It might have been better to ignore or put less weight on distinguishing between the "Wood object" and the "Listerine bottle" since the accuracy is very low for this model.

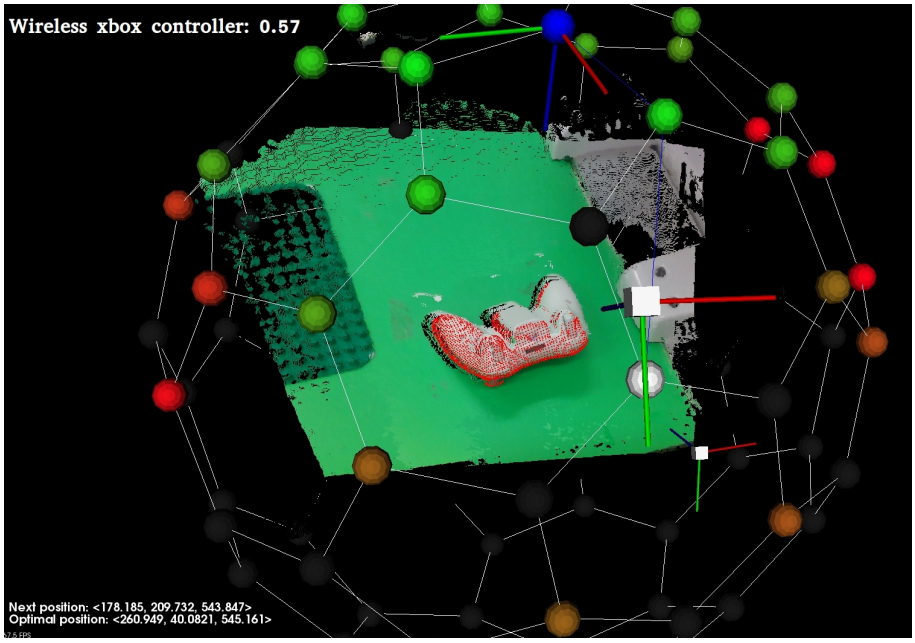


Figure 4.25: Second view of the wireless xbox controller model. We now have a better view of the controller and thus we also get a much better pose estimation of the controller.

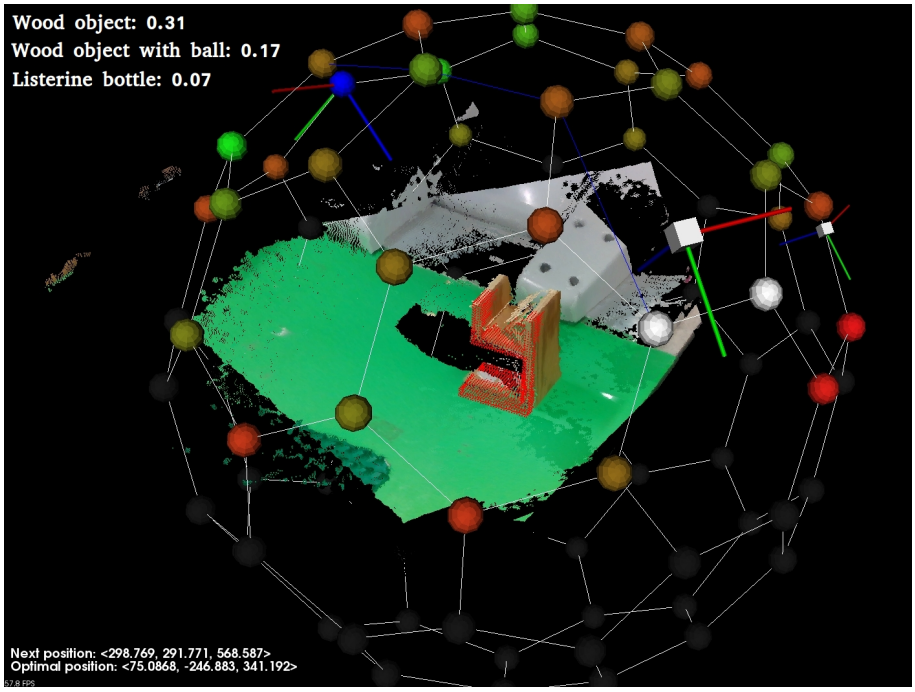


Figure 4.26: Second view of the wood object model. We definitely got a better pose estimate this time compared to the initial view. However, the pose is still not sufficient and we still have two similar models to distinguish from.

4.7.3 Third View

In the third and final view we should have more data in order to make a better pose estimation and better distinguish between similar models than from the initial view. In Figure 4.27 we see the third view for the wireless xbox controller model. The accuracy for the wireless box controller model is much higher for this view than the other views. Also note that in this view the hint system successfully managed to detect the wired xbox controller model as a similar model. This is because for this view the pose estimation for the wired xbox controller model was sufficiently good and thus passed as a valid pose during the pose estimation. The reason why we probably will not get a better difference in accuracy for these models is because of the way we measure the accuracy. The accuracy is a number describing the fraction of inliers for the complete model. Both the wireless xbox controller and the wired xbox controller are very similar except for the battery pack on the back meaning that the only difference in accuracy is completely due to the battery pack (which is only a small part of the complete controller).

In Figure 4.28 we see the third and final merged view for the wood object model. This time we can probably make the assumption that we with high probability identified the "Wood object" model due to the low accuracy for the "Wood object with ball" model. Notice however that the pose still is not good enough and more data probably needs to be received in order to make a better pose estimate. This could be achieved by putting more weight on the feature-utility and thus collecting more views with better features.

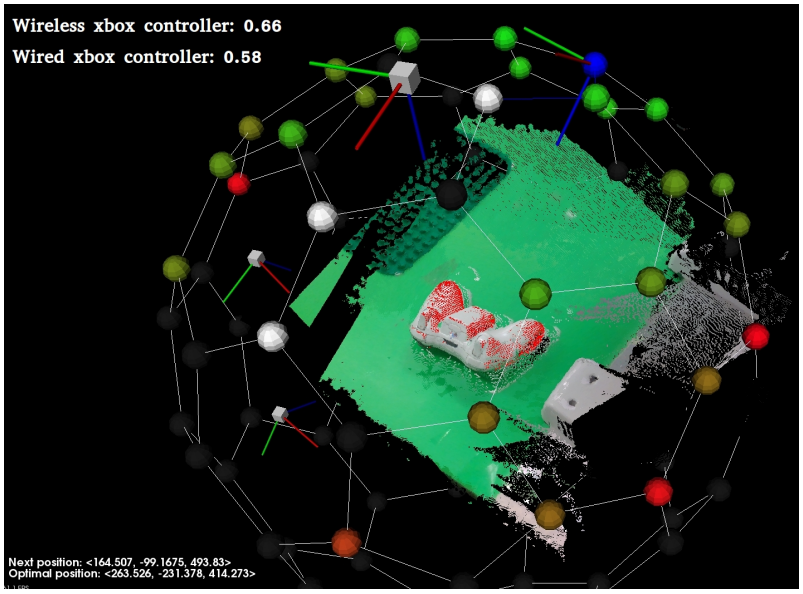


Figure 4.27: Third and final view of the wireless xbox controller model. We now have decent accuracy for the pose estimation. Notice also that the wired xbox controller model now has a valid pose from the pose estimation and thus appears as a similar model. The difference in accuracy for both models suggests that we are more likely to be looking at the wireless xbox controller as opposed to the wired xbox controller.

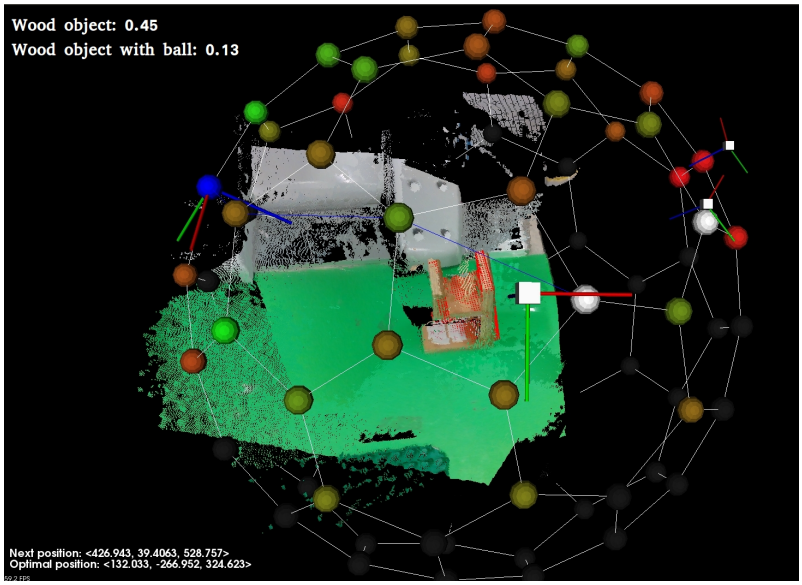


Figure 4.28: Third and final view of the wood object model. Due to the low accuracy of the "Wood object with ball" we can probably assume that we identified the "Wood object". As the red point cloud suggests the pose still needs to be improved which could be achieved by finding views that maximizes the feature-utility.

For both experiments we see that with more data the system is able to estimate better poses and achieve higher confidence for the object identification. The hint system always tries to find a better view which means that if we reach the optimal view, the hint system will search for the second most optimal view and so on until there are no valid nodes left. The advantage of our hint system, as opposed to just moving the camera randomly to collecting more data, is that we are guaranteed to move the camera in an optimal manner and collecting better views.

Chapter 5
Discussion

In this section we will go through the limitations of the system and how the system could be improved.

5.1 Point Cloud Capturer

The point cloud capturer works well but is inconvenient to use since it requires Windows while the rest of the system runs on Linux. This could be solved by using "librealsense" (which is now officially supported by Intel and is also known as "Intel RealSense SDK 2.0") instead of the currently used "Intel RealSense SDK for Windows". It also takes around 1 second to capture a point cloud, which is too long if it were to be used for some real-time application. We believe this could be improved when switching to librealsense and further improved by writing the program in C++ instead of C#.

5.2 Scene Merging

We spent a lot of time working on the scene merging before we got it to work sufficiently well. Most of the problems are related to the calibration where the unknown camera-to-hand transformation is found. The point picking method we use is not ideal to get an accurate calibration of the system. Even a small error when picking corresponding points can produce a large offset when multiple clouds are merged. This small error might occur because the user picks a wrong point, or because the point cloud from the camera does not have the required precision for the point that the user picks. The latter is especially noticeable when the camera is positioned too close to an object, then the points on that object will not be positioned correctly in relation to the points around the object. This is simply a limitation in the depth camera, but it is not easily noticed when calibrating the system and the user might think that something else is wrong.

Even a good calibration might have a small offset (around 1 mm) when merging certain clouds due to imprecisions when picking points. An interesting idea to fix this might be to run an automatic registration method after two clouds have been merged, which could then fix this small offset. ICP is a well researched and widely used registration method that works well for aligning two clouds that are already very close. However, it takes a long time if there are many points in the cloud, and it would have to be done every time a new cloud should be merged. It might however be possible to use the results from the ICP algorithm to fine tune the camera-to-hand transformation, which would allow the calibration to be better for every new cloud that is captured.

A better way of doing calibration would be to use already existing calibration techniques for RGB cameras. These usually consist of taking multiple images of a checkerboard pattern from many different angles, and using that to compute the extrinsic parameters of the cameras which would give the position and rotation of all camera angles. This could then be used in a similar way as the calibration method in Section 3.6.2 to find the camera-to-hand transformation. This is possible since the depth sensor has a built-in RGB camera. We did some tests on this but had a few problems with the accuracy of the extrinsic parameters. Due to lack of time and since we already had a sufficiently good calibration method we did not continue developing this method.

5.3 Object Identification

The object identification is able to successfully identify which views corresponding to which objects that are most likely present in the scene. One major problem however is the way the synthetic views are rendered offline. In the real world the point clouds from the depth sensor are cluttered with sensor noise and outliers. The synthetic rendered views however do not share these traits and thus capture perfect range images of the objects. This had some limitations when it came to the recognition of certain views with lots of sensor noise such as when the surface normals are pointing perpendicular to the viewing direction of the depth sensor. This was however compensated for later in the hint system by using the normal-utility (Section 3.4.5). The usage of the normal-utility thus limited the system from suggesting views that contained so called "bad normals".

We also noticed that views with occlusion resulted in lower performance for the object identification. This was probably a limitation of the ESF global feature which was used to match the model views with the scene. The problem of successfully identifying objects in views with occlusion is a well researched area. The biggest problem probably arises from the usage of global features for the object identification. Global features are well known for not being well suited for dealing with point cloud occlusion. A better solution could have been to use implicit shape model as suggested in [11] where they use local features for the identification. Local features such as the SHOT feature are much better at handling with occlusion. The reason why we did not use implicit shape model was explained in Section 3.5. We did not get any good results for the implementation in PCL but maybe another implementation would have worked better.

Another approach for the object identification would have been to also make use of the RGB data from the camera. Many methods exist within computer vision which recognize objects using 2D images. Neural networks for instance are very popular and successful

methods which identify objects with 2D images. A combination of these methods along with our existing pipeline would have been very interesting to implement.

5.4 Pose Estimation

The biggest limitation for the pose estimation is speed. It is set to run for a fixed number of iterations even if it finds the perfect pose on the first iteration of the algorithm. For large objects and clusters (containing many points) each iteration can take quite some time and most of that time might be wasted if a perfect pose has already been found. We have registered times of 1-2 minutes to find a pose of an object for some clusters, even on computers with very good hardware. The speed could be improved in many ways by using smarter algorithms that adapt the number of iterations necessary based on the results of previous iterations. It would also be interesting to try a method that estimates the pose for all object views at the same time, instead of completely finishing the pose estimation for one object view before starting on the next. By running one iteration on one object view, and then switching over to run one iteration on the next object view etc., the system could quickly see which object views give better results and filter them out quickly. This would avoid wasting time on estimating poses for object views which will not have better poses than other object views.

Another part of the pose estimation that could be improved is the view merging. Currently the system only merges a hard coded number of views. It might happen that all these views are wrong and the system is unable to find any good pose using these views. It would then be good if the system continued to merge new views which would allow it to continue searching for a good pose until it has tried all possible view combinations, or until a timeout specified by the user has passed. This could also be used in the opposite way, meaning that the system would stop after it has found a very good pose and it is unlikely that it would find a better one. We believe that smart algorithms like this could improve both the results and the execution time by a lot.

5.5 Hint System

Our suggested hint system implementation achieves decent results but there are many things that could have been done differently. The utilities in our current implementation of the hint system are static and do not change when running the system. Let us say that we have collected data for a view V_a in our system. The node n_a corresponding to view V_a in our view-graph has the nearest neighboring nodes n_b, n_c and n_d. Since n_b, n_c and n_d are neighboring nodes to n_a we get that visiting these nodes, given that we have already visited n_a, will result in a low amount of new data. Most of the object for n_b, n_c and n_d have already been seen for n_a. However, our existing hint system does not take this into account and will suggest neighboring nodes to already visited nodes even though visiting these nodes will not lead to any significant amount of new data. A better way would thus have been to keep track of shared data between nodes and continuously update the utilities for each node given the current state. The easiest way to implement this would probably be to use dynamic, state driven utility weights $\boldsymbol{w}$. These weights would change when running

the system and thus change the utility for some nodes for a given state.

The hint system was not either able to detect whether more data should be collected in order to better distinguish between objects or if more data should be collected in order to make a more accurate 6DOF pose estimation. In some cases we would want to put more weight on the distinguish-utility in order to separate and correctly identify similar models and in some cases we would like to collect better features in order to make a better 6DOF pose estimation. As mentioned before our current implementation uses static utility weights and thus to solve this we would need to have dynamic weights that change over time. If the system detects that there are similar models and that we cannot distinguish between them then the hint system would have to put more weight on the distinguish-utilities instead. Likewise, if the system detects that there are no similar models and that the current pose estimation does not have a sufficient amount of inliers in order to make a good 6DOF pose estimation, the hint system would have to put more weight on the feature-utilities in order to prioritize views with better local features.

Some improvement could also have been done concerning the utility values. The introduction of more utility values could definitely have made a difference for our hint system. One utility in particular is something that describes the point density and connectivity of a point cloud. Small and narrow objects were likely to be removed completely due to outliers and sensor noise. Imagine an object consisting of two boxes connected by a small and narrow rod. Due to sensor noise characteristics it is very likely that the rod will be removed during scene preprocessing (removing outliers and extracting clusters) which will result in two separate clusters instead of one for the complete object. By measuring the point density for the rendered synthetic views of our object we could detect views where this would appear and compensate for it by giving these views a lower utility value.

One big feature for our hint system is the ability to detect similar objects by matching models offline and detecting similar views. As can be seen in the results we did achieve good results for most of our objects that were very similar. However, this method was far from perfect as can be seen in Figure 4.19 in the last row and first column. Here we see that the views for both models are identical and so the distinguish-utility for that view should have been very low. Instead we see that this value is slightly higher than moderate which suggest that this view for both models has moderate distinguish ability. Outliers like this were present in some model view-graphs and could have some negative impact on the overall online performance of the hint system. We believe that the ESF features for flat surfaces such as planes are inconsistent which would explain why these views are seen as "different" when matching the ESF features.

Our current implementation of the hint system (and system in general) only uses data from the depth sensor and thus ignores the RGB-data. If we look at the work in [8], we see that they chose the next best sensor viewpoint by minimizing the ambiguity between different predicted model hypothesis. The key difference between their implementation and our is that they include the possibility of choosing between different sensor viewpoints instead of just one. In some cases when distinguishing between two objects it could be as simple as just comparing their colors using the RGB data. In other cases, the next best camera viewpoint for distinguishing between similar models might be closer and more optimal when using the RGB data, as opposed to using the depth data.

Compensation for environmental occlusion caused by other objects could have some major effects as well for our hint system. In [9] and [10] they define the next best view

by minimizing or excluding occlusions by creating an octree voxel map representation of the scene point cloud. The first and probably easiest way to implement compensation for occlusion into our hint system would be to use the approach in [9] where a viewpoint is eliminated for any kind of occlusion. This would enable us to detect nodes (or viewpoints) in our view-graph that are occluded by other objects in the scene and a simple way to compensate for this would be to mark these nodes as invalid. The second and more dynamic approach would be to use the method in [10]. They define a vector that keeps track of the current number of projected occluded voxels for each camera viewpoint. They then define a hard threshold that eliminates a viewpoint if the number of occluded voxels exceeds that threshold. Introducing a new dynamic utility value that keeps track of the number of occluded voxels could thus have been a solution for our hint system. For more informations about these methods we refer to the work in [9] and [10].

In our thesis we implement our system using the ABB YuMi robot which has two independently working robot arms. A problem arises when both arms are actuated, which is a main feature of the YuMi robot. If the camera is mounted on one arm, then a natural behavior of the hint system would be to avoid colliding or obstructing the other robot arm. In some cases the second robot arm might also be occluding an object from different viewpoints. Introducing robot joint movement limitation and collision detection would thus have had a great impact on the performance of our hint system. A similar problem is found and solved in [10] where they incorporate the robot joint movement as a cost in their cost-function. They introduce this cost in order to limit large changes in the robot joint space configuration. Though their problem is not exactly the same as ours, their approach of incorporating the robot joint movement in the cost function could also be applied in our implementation. Again, this cost would be introduced as a utility value in our hint system which would decrease the possibility of moving the robot arm and camera to a viewpoint which is non-optimal with respect to the other robot arm.

5.6 Overall

As mentioned above, the modification that we believe would improve almost all aspects of this system is to use the color data in the point clouds. In our work we have ignored the color data in the algorithms, and only used it for visualization purposes. The color data can be used in feature descriptors which could improve the results from the identification, pose estimation and hint system. This would however require color data on the CAD models as well, which increases the work needed to add new objects to the system.

The system works well for the objects that we have tested in our work, but more testing is necessary to see how well it works for different objects. In general, objects with more geometrically interesting areas that give more distinct features should be easier to both identify and localize. It is hard to know how the system would perform for objects that are mainly primitive shapes, like cubes, cylinders, spheres etc. since these objects do not contain many distinct features. For these objects it might be beneficial to use some other pose estimation method that is more focused on primitive shapes, as explained in the related works section. This would definitely be an interesting continuation of our work, to add more objects of different geometry and see how well the system performs. This could also be used to see how many clouds are needed on average before a good pose is found

for different objects. In our tests we need on average 2 to 3 clouds before we get a good identification and pose estimation of an object. However, if there are many similar objects, more clouds may be needed to get good results since certain parts of these objects have to be captured in the clouds before the system can know which object it is looking at.

We have tested the system with some small objects as well, that were somewhere around 2x2x1 centimeters in size, but we got bad results. The main problem is that the depth camera smooths the edges of objects, which is especially noticeable for such small objects. This causes the cluster of the object to be very different compared to the rendered views of the object, which makes it hard to find a good pose. The results might be better with other depth cameras that are more intended for high-detail environment scans.

The system we have developed is mainly usable for robotic applications that have the possibility of mounting a depth camera to a robot arm. Ideally the depth camera would be integrated in the arm such that the robot still has the ability of using other tools with the same arm. The system could be extended to allow the robot to follow the hints automatically and not rely on a human guiding it, which would further improve the usability of the system.

It was difficult to compare our complete system with other related works. This was mainly due to the fact that there did not exist a "general" or "state-of-the-art" system which uses our design. As a result, there was not any data for which we could evaluate our system and compare it to other related work. When examining the different subsystems such as the object identification and pose estimation there exist many sets of data for which one can evaluate these subsystems. However, this thesis did not propose any new and unique methods for these subsystems, the evaluation for these methods are instead verified in [6] and [1].

Chapter 6
Conclusion

In this master's thesis we developed a system for CAD object identification and 6DOF pose estimation in robotics. The identification was performed using global descriptor matching with synthetically rendered views of CAD models. For each identified view we then estimated a 6DOF pose for the object in the scene by matching local features using a robust RANSAC feature matching algorithm. The pose estimation and identification was then further improved by our hint system which suggested where the robot should move the camera in order to get a better view. Not only did we get a better view but we also kept track of our previous camera positions and data. We merged the previous point clouds with our new point cloud and thus gained more information describing the 3D scene for each new view. Thanks to our hint system and merging of point clouds we steadily improved the object identification and pose estimation as opposed to just moving the camera randomly in order to get more data.

There is much research in the area of finding a better viewpoint such as [8], [10] and [9]. However, our solution used an offline training algorithm to generate utility values describing the absolute quality of a view. The quality mainly depended on four values: visibility, surface normals quality, local features quality and distinguishability between similar models. Our method of detecting similar models and distinguishing between them using global ESF matching was also different and achieved good results. Another significant difference was the way we handled merging of previous point clouds. By using the previous point clouds we iteratively gained more information and was able to identify and estimate more accurate poses over time.

We draw the conclusion that it is possible to build a system with object identification and localization using a depth camera. By moving the camera to better views as suggested by the hint system and merging all new point clouds the results were good enough to be usable for robotic applications. Adding new objects was very easy, and the user feedback for the results were clear and easy to understand. However, more objects with varying geometry needed to be tested to make sure that the system was robust even for many different objects, especially objects consisting of primitive shapes like cubes and cylinders.

Bibliography

[1] R. Bogdan Rusu, N. Blodow, M. Beetz. *Fast Point Feature Histograms (FPFH) for 3D Registration.* International Conference on Robotics and Automation. Kobe: IEEE, 2009.

[2] *Point Cloud Library (PCL).* `http://www.pointclouds.org/` (accessed 2017-12-08).

[3] D. Aiger, N. J. Mitra, D. Cohen-Or. *4-Points Congruent Sets for Robust Pairwise Surface Registration.* ACM Trans. Graph 27, 2008.

[4] H. Ali, N. Figueroa. *Segmentation and Pose Estimation of Planar Metallic Objects*. Ninth Conference on Computer and Robot Vision, Germany, 2012.

[5] D. Holz, M. Nieuwenhuisen, D. Droeschel, J. Stückler, A. Berner, J. Li, R. Klein, S. Behnke. *Active Recognition and Manipulation for Mobile Robot Bin Picking*. In Gearing up and accelerating cross-fertilization between academic and industrial robotics research in Europe, Springer 2014.

[6] W. Wohlkinger, M. Vincze. *Ensemble of Shape Functions for 3D Object Classification.* International Conference on Robotics and Biomimetics. Phuket: IEEE, 2011.

[7] W. Wohlkinger, M. Vincze. *Shape Distributions on Voxel Surfaces for 3D Object Classification From Depth Images.* International Conference on Signal and Image Processing Applications. Kuala Lumpur: IEEE, 2011.

[8] S. A. Hutchinson, R. L. Cromwell, A. C. Kak. *Planning Sensing Strategies in a Robot York Cell with Multi-Sensor Capabilities.* IEEE Transactions on Robotics and Automation, 1988.

[9] C. McGreavy, L. Kunze, N. Hawes. *Next Best View Planning for Object Recognition in Mobile Robotics.* Proceedings of the 34th Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG), 2017.

[10] C. Maniatis, M. Saval-Calvo, R. Tyleček, R. B. Fisher. *Best Viewpoint Tracking for Camera Mounted on robot Arm with Dynamic Obstacles*. Cornell University Library, arXiv.org, Ithaca, New York, 2017.

[11] J. Knopp, M. Prasad, G. Willems, R. Timofte, L. Van Gool. *Hough Transforms and 3D SURF for robust three dimensional classification.* European Conference on Computer Vision. Crete: ECCV, 2010.

[12] R. Bogdan Rusu, G. Bradski, R. Thibaux, J. Hsu. *Fast 3D Recognition and Pose Using the Viewpoint Feature Histogram*. International Conference on Intelligent Robots and Systems. Taipei: IEEE/RSJ, 2010.

[13] A. Aldoma, M. Vincze, N. Blodow, D. Gossow, S. Gedikli, R. Bogdan Rusu, G. Bradski. *CAD-Model Recognition and 6DOF Pose Estimation Using 3D Cues*. International Conference on Computer Vision Workshops. Barcelona: IEEE, 2010.

[14] R. Bogdan Rusu, Z. Csaba Marton, N. Blodow and M. Beetz. *Persistent Point Feature Histograms for 3D Point Clouds.* Conference on Intelligent Autonomous Systems. Baden-Baden, 2008.

[15] F. Tombari, S. Salti, L. Di Stefano. *Unique Signatures of Histograms for Local Surface Description.* European conference on computer vision. Crete: ECCV, 2010.

[16] A. G. Buch, D. Kraft, J.-K. Kämäräinen, H. G. Petersen and N. Krüger. *Pose Estimation using Local Structure-Specific Shape and Appearance Context*. International Conference on Robotics and Automation (ICRA), 2013.

[17] *PCL documentation on PCD file format*. `http://pointclouds.org/documentation/tutorials/pcd_file_format.php` (accessed 2017-11-08).

[18] *Intel RealSense SDK for Windows*. `https://software.intel.com/en-us/realsense-sdk-windows-eol` (accessed 2017-11-08).

[19] *Intel RealSense SDK 2.0*. `https://github.com/IntelRealSense/librealsense` (accessed 2017-11-08).

[20] *PCL Documentation Euclidean Cluster Extraction*. `http://pointclouds.org/documentation/tutorials/cluster_extraction.php` (accessed 2017-10-10).

[21] *Spherical temperature averaging using Icosahedral grids*. `http://clivebest.com/blog/?p=7820` (accessed 2017-12-08).

[22] W. Wohlkinger, A. Aldoma, R. Bogdan Rusu, M. Vincze. *3DNet: Large-Scale Object Class Recognition from CAD Models.* International Conference on Robotics and Automation. Saint Paul: IEEE, 2012.

[23] A. P. Rhodes, J. A. Christian, T. Evans. *A Concise Guide to Feature Histograms with Applications to LIDAR-Based Spacecraft Relative Navigation.* American Astronautical Society, 2012. 64 (4), pp. 414–445.

[24] *ABB Robot Web Services*. http://developercenter.robotstudio.com/webservice (accessed 2017-11-08).

[25] *Curl*. https://curl.haxx.se/ (accessed 2017-11-08).

[26] *GrabCAD Community*. https://grabcad.com/ (accessed 2017-12-13).

[27] *Thingiverse*. https://www.thingiverse.com/ (accessed 2017-12-13).