TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

ACIN
AUTOMATION & CONTROL INSTITUTE
INSTITUT FÜR AUTOMATISIERUNGS- & REGELUNGSTECHNIK

Object Recognition for Robotic Grasping Applications

DIPLOMARBEIT

Conducted in partial fulfillment of the requirements for the degree of a

Diplom-Ingenieur (Dipl.-Ing.)

supervised by

Univ.-Prof. Dr. techn. M. Vincze
Dr. techn. J. Prankl

submitted at the

Vienna University of Technology
Faculty of Electrical Engineering and Information Technology
Automation and Control Institute

by
Thomas Muttenthaler
Sonnleithnergasse 9/26
1100 Vienna
Austria

Vienna, in October 2017

Preamble

Firstly, I want to thank my supervisors Markus Vincze and Johann Prankl for sharing their knowledge and professional advice. Furthermore, I want to express my acknowledgment to the PCL open source community for providing useful libraries, which facilitated my work.

Finally, I want to thank my parents and friends for their support along the whole way during the last years.

Abstract

Recent developments in modern robotics show that making robots more autonomous is an important, but difficult task for the robotic industry. Imagine a simple pick and place scenario from our daily life. For example, taking a cup of coffee from the kitchen table and putting it in the sink. This seems like a fairly simple task, at least for a human being. If we want to teach a robot to deal with this situation autonomously, some advanced problems need to be solved.

First of all the robot needs to find the table and recognize the cup of coffee. Furthermore, the robot needs to bring its arm and fingers in the right position to grasp the object properly. The goal of this thesis is to integrate an object recognition system into the framework of an existing robotic arm, which has already smart grasping algorithms implemented. Combining the vision system with these grasping algorithms enables the robotic arm to recognize and grasp objects, like a cup of coffee, autonomously. Till now the user of the robot needed to provide the location and orientation information of objects manually. Typing in this data takes time and makes the robot less autonomous.

In this thesis, we tested and evaluated existing general purpose object recognition algorithms and optimized the results with respect to speed and accuracy for basic table top situations. The whole recognition process is divided into different steps, from modeling real objects to extracting location and pose of recognized objects in a given scene. Finally, we created simple interfaces of the modeling and recognition system to make it easily operable for users who are new to computer vision.

Kurzzusammenfassung

Neueste Entwicklungen in der modernen Robotik zeigen, dass autonom arbeitende Roboter zusehends an Bedeutung gewinnen. Stellen Sie sich eine einfache Tätigkeit aus ihrem Alltag vor. Zum Beispiel sie wollen eine Tasse Kaffee vom Küchentisch in die Abwasch stellen. Das sollte keine große Herausforderung darstellen, zumindest nicht für einen Menschen. Soll diese Tätigkeit aber von einem Roboter erledigt werden, müssen einige Probleme gelöst werden. Zuerst muss der Tisch gefunden werden und auch die Tasse auf dem Tisch erkannt werden. Danach muss der Roboter noch eine passende Greiftrajektorie berechnen, um die Tasse richtig greifen zu können. Das Ziel dieser Arbeit ist es, ein Objekterkennungssystem in ein existierendes Robotersystem zu integrieren. Der verwendete Roboterarm ist schon in der Lage selbstständig Objekte zu greifen, aber derzeit müssen noch die Positionsdaten des zu greifenden Objektes manuell übergeben werden. Das ist natürlich zeitaufwendig und stellt eine Einschränkung dar. In dieser Arbeit habe ich verschiedene Objekterkennungsalgorithmen untersucht und diese auf die Roboterumgebung angepasst. Die ganze Objekterkennung beinhaltet auch die Modellierung von zu erkennenden Objekten. Zu guter Letzt wurde die Benutzeroberfläche des Modellierungs- und Erkennungssystems überarbeitet, um eine einfache Bedienung der Software gewährleisten zu können.

Contents

List of Figures

List of Tables

1 Introduction

Developments during the last years in industrial automation and modern robotics show, that making a robot more autonomous has become an important issue. One reason for this development is that companies try to make their factories more flexible. For example, lets take a look at one robotic arm inside a factory. Think of a simple task like moving a work piece from place A to place B. We need to program the exact trajectory of the robotic arm and the end effector to fulfill this task. If we slightly change the object or the places where we pick or place the object the robot needs to be reprogrammed. This is tremendously time consuming especially if the factory consists of hundreds of robots. One solution to avoid reprogramming all the robot is to give the robot the abilities to recognize objects itself and to grasp such objects autonomously. Consequently, the robots and the factory become more flexible, because they can adapt to chances in the work process efficiently.

For another example of an application area for autonomous robots, lets switch from highly automated factories to a more familiar environment, our homes. Imagine we sit in our living room and we want to get our cup of tee, which we forgot on the kitchen table. Luckily we don't need to get it ourselves, because we have a household service robot. After we gave the order, our robot starts to go to the kitchen, picks up the cup of tea and carries it back to us in the living room. This seems like a fairly simple task, at least for a human being. If we want to teach a robot to deal with this situation autonomously, some advanced problems need to be solved.

To find the kitchen some navigation algorithms are required. Inside the kitchen the robot needs to find the table and the cup on the table, which is done by computer vision algorithms. Finally, with the help of grasping algorithms the robot can bring its arm and end effector in the right position to grasp the cup of tea properly and carry it back to us.

One can see, a whole bandwidth of different engineering fields are required to solve a seemingly easy task. In this thesis we are going to focus on just one key part of the described task chain. We want to use a camera to give a robot "eyesight", which allows it to recognize objects autonomously and fulfill tasks on top of it. A more precise description of the goal of this thesis is given in the next section.

1.1 Problem Statement

The initial setup consists of a robotic arm mounted on a table, similar to the setup shown in Figure 1.1. The arm is capable of picking up objects which are placed on the table next to it.

Figure 1.1: Static robotic arm setup[1]

The robot has smart grasping algorithms implemented already. But to calculate a grasping trajectory in order to pick up an object from the table, the location, the orientation and a shape model of the targeted object need to be provided beforehand. My goal is to integrate a vision system into the robotic arm setup, to make the whole process from object recognition to object grasping fully autonomous. The setup is mainly used for demonstration and testing purposes by engineers with little computer vision background. Consequently, the vision system needs to be intuitively operable.

1.2 Solution Statement

The vision system is going to be realized in two stages. The first one is the object modeling stage which allows scanning real-world objects with a RGBD camera and creating a model database. We are going to introduce a simple GUI based tool, which makes the modeling process fairly easy. Only a minimum of hardware and little computer vision knowledge is necessary to operate the tool

and to create highly accurate 3D models. The same tool calculates a surface model which can be used directly for the grasping system.

The second stage is the object recognition stage. Using the models of the model database and receiving an image of a scene with objects, the recognition system calculates the location and orientation of objects found in the scene. The recognition algorithms are going to be adjusted and optimized for table top situations. Evaluating a variety of objects in different scenes showed the efficiency of the chosen recognition approach. Moreover, the algorithm offers a high flexibility to optimize the results for specific situations, like table top scenes for robotic grasping applications.

The location information is passed on to the object grasping system, which calculates a grasping trajectory in order to pick up an object. To keep the usability of the recognition system simple a visual programming language is going to be used. Moreover, the results are going to be visualized simply, which provides a nice setup for demonstration purposes. An overview of the whole system is given in Figure 1.2.

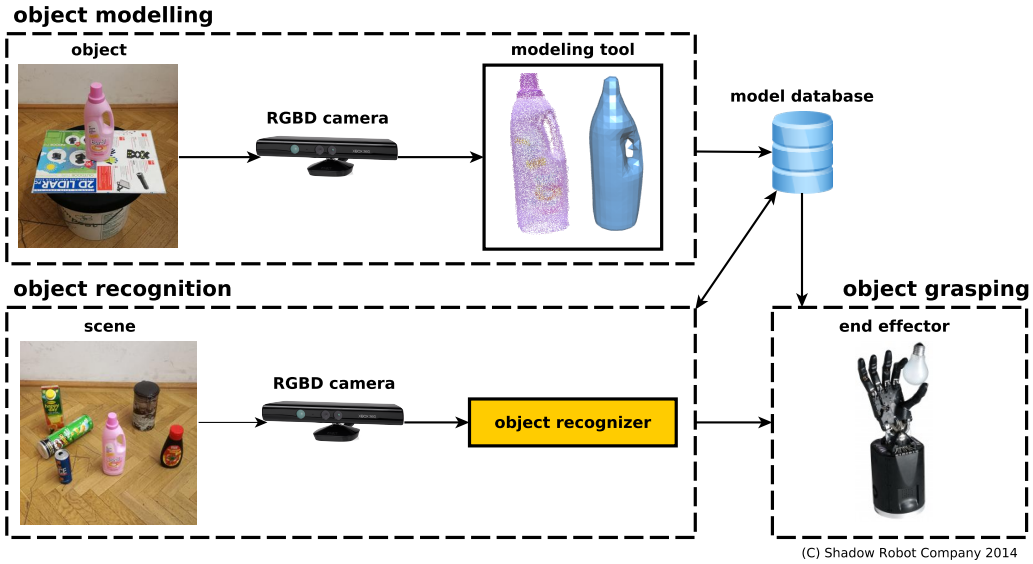


Figure 1.2: System overview

1.3 Guideline through Work

After the brief introduction this work continues with Chapter 2, which gives an overview of the existing object recognition systems. It also covers the object modeling stage, which is needed for the recognizer. The following Chapter 3 a overview of the whole recognition system is given. In Chapter 4 my modeling approach is explained. In Chapter 5 all parts of the recognizer are explained. Finally in Chapter 6 the recognizer from the previous chapter is evaluated.

2 State of the Art

The whole recognition progress is generally dividable into two main parts, the object modeling stage and the actual object recognition stage.

We start with a real-world object which we want to be recognized by the robot. First we need to teach the robot how this object looks like in terms of color and shape by creating a model. Possible approaches are scanning the object with a camera or using a laser scanner. The specifications of the objects can be stored in a model database. After modeling, the actual recognition step can start. The recognizer takes a picture of an arbitrary scene which includes known objects. The algorithm starts to compare the image of the scene with the model database and tries to find all known objects in the scene.

2.1 Object Modeling

This section deals with the process of obtaining a model from the real-world object which can be used for recognition purposes.

2.1.1 RGBD Camera Data based Modeling

A convenient approach to obtain spatial and color information of an object is to use a RGBD camera. These cameras give us all necessary information in a single step. They provide a color picture with depth information, a so called 2.5D image. Usually the images are represented by point clouds. To obtain a full 3D model of an real object basically three steps need to be done. Obtaining the raw data by taking pictures of the object from several perspectives, transforming the images to the same coordinate system and finally generating a model. A framework for the described modeling approach is given by the RTM tool[1] of the V4R research group [2]. We are going to explain this tool in more detail, because it builds the base of the object modeling step of my later work. The setup consists of a turntable, an object which is placed on the turntable and a RGBD camera as shown in Figure 2.1

Turning the turntable by hand allows the camera to take pictures from all sides of the object. Additionally, the algorithm keeps track of the angle of the

[1] http://www.acin.tuwien.ac.at/forschung/v4r/software-tools/rtm

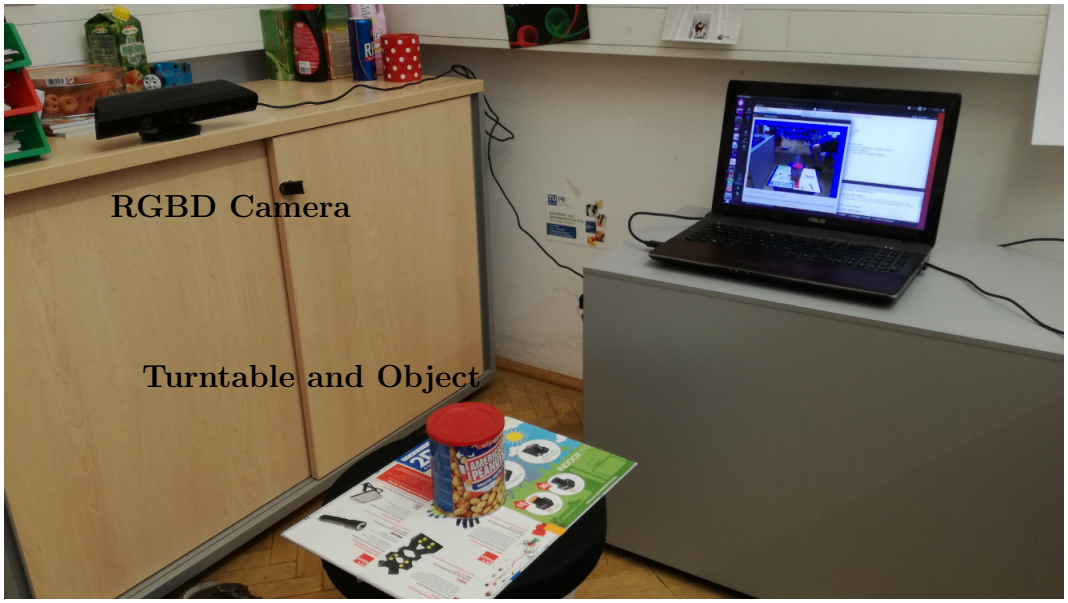


Figure 2.1: V4R object modeling setup.

object, which is needed for the registration of all images. An explanation of the registration process is given later in this section. First of all the algorithm needs to distinguish between points belonging to the object and points belonging to the background. The RTM tool provides two approaches for the segmentation, which are described in [2].

- an interactive segmentation approach, and
- segmentation based on a tracked region of interest (ROI).

The interactive segmentation relies on multi-plane detection and smooth segmentation (Figure 2.2, left). The image is segmented into multiple parts by clustering the planes and remaining areas depending on normal deviation of neighboring image points. Interaction by the user is needed to manually select all segments which belong to the modeled object. For the second segmentation option, we need to select a planar surface before the camera tracking starts. After clicking on the flat surface below the object, a ROI is calculated automatically on top of this surface (Figure 2.2, right, blue grid box surrounding the red mug.). The assumption the object is within the ROI allows the algorithm to segment all points which belong to the object and filter the background.

Now all images which were accumulated need to be registered. During the data acquisition the algorithm keeps track of the angle of the object by a local feature based method. A local feature describes a distinct point which can be

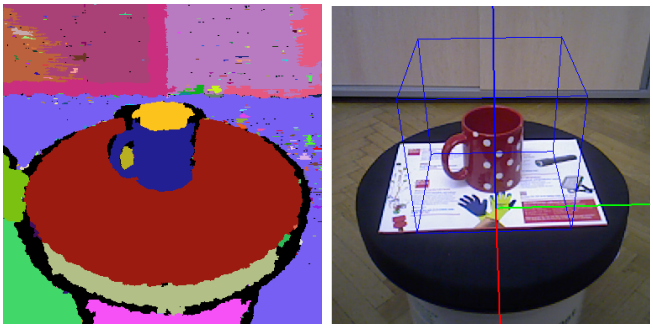


Figure 2.2: Labels of planes and smooth clusters (left) used for automatic adjustment of region of interests (right) and for interactive object segmentation ©[2015] IEEE[2].

easily detected on multiple images of an object from different perspectives. A more detailed description of features is given in the next section. The transforms between two images can be calculated, if we know the location and orientation of(at least three) features, which exist in both images [2]. To improve the tracking quality also the supporting surface is used to keep track of the angle. This is the reason a textured pattern was chosen (Figure 2.2, right).

Due to concentration of several transformations during the registration stage some drifting errors occur. Two methods to improve the registration result are implemented. First of all the framework allows applying **Bundle-Adjustment**, which directly reduces the re-projection error of correspondences used during camera tracking. The second method is a multi-view **Iterative Closest Point** algorithm [3] that globally reduces the registration error between overlapping point clouds by iteratively adapting the transformation between camera poses.

2.1.2 Laser Scan

If the models created by RGBD cameras are not accurate enough, we can use a laser scanner system instead. Laser based systems allow us to obtain precise spatial data. As mentioned before the recognizer does not only need spatial information, but color information as well. Laser scanners can't extract colors, therefore an additional camera is needed. An example setup of a laser scanner plus a camera is shown in Figure 2.3.

As we can see the setup for this modeling system is quite complex. To acquire data from all perspectives of the object the measurement system is moved on a predefined trajectory over the object. Additionally, the object is placed on a computer-controlled turntable [5].

Figure 2.3: Laser Object Scanning Rig: the box contains a computer-controlled turntable. ©[2015] IEEE [4]

2.1.3 In-Hand Scanner

The previous frameworks divided the object modeling into two stages. First, the images are accumulation and then a post processing stage combines all data to a full model. In here [6] a method was proposed which combines these two stages, and calculates a model in real time. This allows the framework to visualize the model throughout the whole modeling process. Since no prost-processing is done artifacts and noise need to be detected and removed during recording.

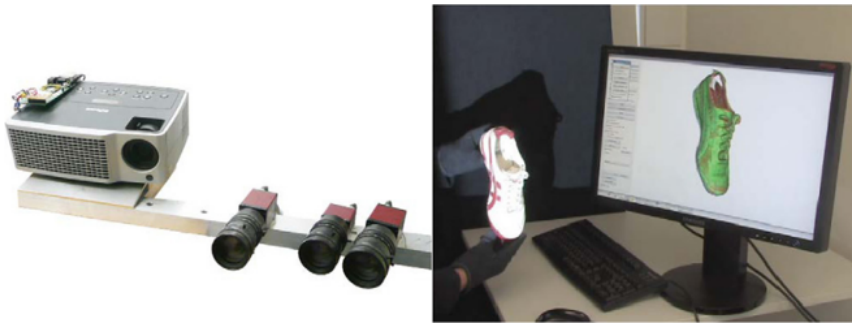


Figure 2.4: In-hand scanner,©[2009] IEEE

2.1.4 Feature Based Modeling

The object modeling approaches mentioned in the previous sections have the goal to create a full 3D point cloud model. Depending on what we need the model for a simpler representation of an object may be sufficient. The framework called MOPED [7] uses a feature based model. The model does not include every single point of the object, but only certain key points alongside with SIFT descriptors, which describe the local area around a key point. To improve the result of the final model bundle adjustment is used on all key points. A more detailed description about this optimization algorithm is given in Section 4.3.

This model can be used directly for object recognition based on SIFT. For a more detailed explanation of SIFT, see Section 2.2.2.

2.2 Object Recognition

After a model database was created by the modeling tool, the actual recognition algorithm can be used.

First we are going to explain the basic working principle of modern object recognition algorithms. Figure 2.6 shows an example of a recognition pipeline. The recognizer uses a RGBD camera to take an image of a scene which includes known objects, as shown in Figure 2.5.

Figure 2.5: Example of a typical table top scene

The input data for the recognizer are the point cloud models from the model database and the captured scene, which is also represented by a point cloud. But the recognizer does not compare point clouds directly, it works on a more abstract level. It finds features of the models of the database and also of the

scene. As mentioned in Section 2.1.1, features describe distinct specification of an object which can be easily detected on multiple images of an object. For example, corners in a picture or some unique texture usually provide good features. More details about different kind of features are given in the next section. From now on the scene and the models are not represented by point clouds anymore, but by lists of key points and features. Similar to the representation mentioned in Section 2.1.4. Consequently, if an object from the model database is present in the captured scene, the same features can be found in the database and in the scene. Feature matching algorithms figure out which features in the scene correspond to features in the model database. Finally, from these correctly matched features the transformation between the recognized objects and the scene coordinate system can be estimated. Finally, this transformation found between features can be applied to the original point clouds of the model database to transform them to the correct location and orientation in the scene. Basically a recognition algorithm returns the identification, location and orientation of recognized objects relatively to the coordinate system of a given scene.

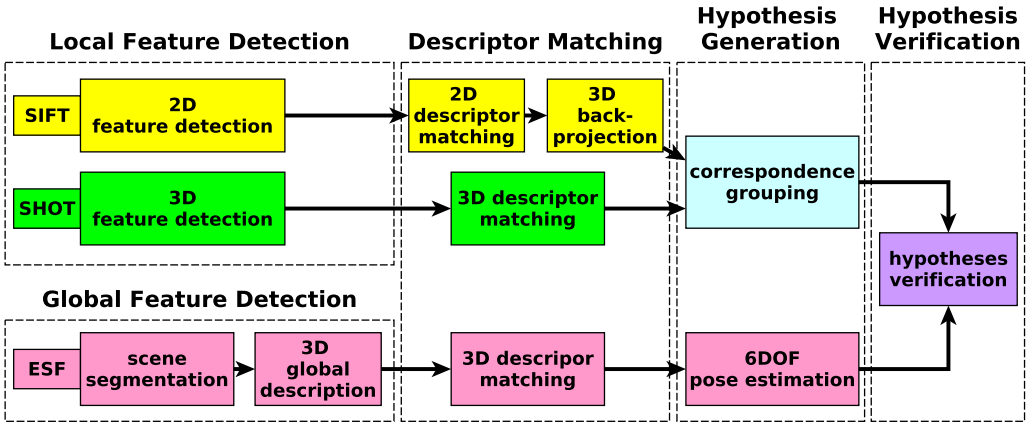


Figure 2.6: Example of a recognition pipeline [8]

2.2.1 Global Feature Detection

As mentioned before one way to identify objects is to find unique specifications in shape or color, so called features. These features are described by descriptors and are stored in the model database alongside the point cloud model of the object. To measure specifications of the whole object, data needs to be segmented. All points belonging to the same object need to be clustered

together as a pre-processing step. For different features, different descriptors are used. First we want to start with the class of global descriptors. They give a description of the whole object. This can be color distribution, or some information of the general shape of the object.

One example of a global shape descriptor is the ESF (Ensemble of Shape Functions) descriptor.

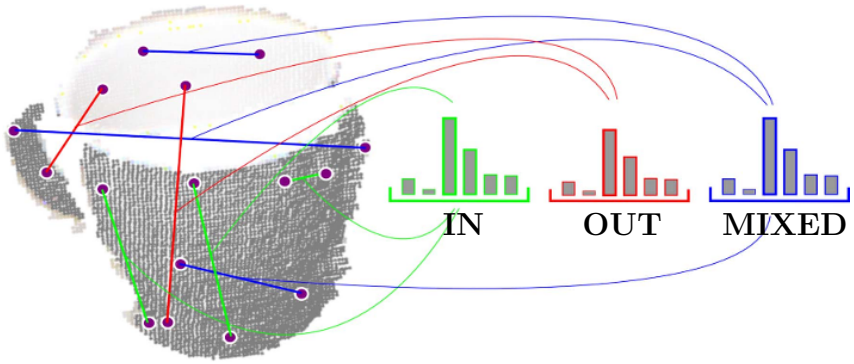


Figure 2.7: ESF descriptor: Histograms over Distances ©[2011] IEEE [9].

This descriptor is based on the shape distribution of objects. The approach is simple, just take two random points of the point cloud of our object model and measure the distance. Repeat this process many times until we can build a significant histogram from the distance data we collected. As shown in Figure 2.7, multiple histograms of different categories can be made.

Green represents the connecting lines lying ON the surface of the object, red represents the lines where only the endpoints are on the surface and the connecting line is OFF surface and the blue colored line distances are classified as MIXED as they are partly ON and OFF the surface [9]. Moreover, additional histograms can be created, for example the ratio between IN and OUT distances. Also, the area between three sampled points can be used to create a histogram. All these obtained histograms together represent an ESF descriptor.

2.2.2 Local Feature Detection

Another type of features are the local features which are represented by local descriptors. In contrast to global descriptors, local descriptors don't describe specifications of the whole object. They only describe small areas around interest points of the object. An interest point or a key point is a discriminable point in an image, for example a corner. Similar to global descriptors local descriptors can represent shape or texture specifications, but restricted to local

areas only. In the following paragraphs we are going to introduce one algorithm which relies on local texture information and one on local shape information.

SIFT: Scale-invariant feature transform

SIFT is a popular algorithm to detect and describe local features of 2D images. [10] The algorithm is patented by the University of British Columbia [11]. Examples of SIFT features are given in Figure 2.8.

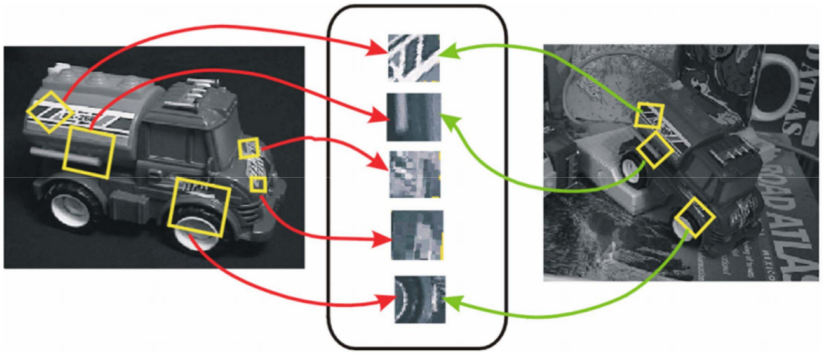


Figure 2.8: Example of SIFT features of same object in different images marked with yellow rectangles. Features are invariant to rotation and image scale. Furthermore, they are fairly robust to illumination changes, change of viewing angel and noise. [12]

To find interest points the difference of Gaussian (DoG)[13] function is used. Extrema of the DoG function applied on multiple scales of the image represent interest points. SIFT features are invariant to scale variation and rotation. Moreover, SIFT is robust in change of illumination, noise and minor changes of the viewing angle. The area around the key point is described by a SIFT descriptor, which is calculated of the local image gradient around the key point [10]. Finally, we want to single out that in contrast to ESF(Section 2.2.1) and SHOT(which we are going to explain next) SIFT uses a gray scale 2D image to find and describe features. This means a projection of the point cloud is used and via back-projecting the 2D key point into the 3D space, the 3D locations of key points are obtained.

SHOT: Unique Signatures of Histograms for Local Surface Description

SHOT descriptors [14] work on 3D point clouds and describe the local shape of objects. It is based on eight histograms which describe the angular difference between the surface normal and a local reference frame (LRF). The LRF

defines the orientation of a corresponding key point. As for the structure of the descriptor, an isotropic spherical grid is used, that divides the sphere along the radial, azimuth and elevation axes, as sketched in Figure 2.9.

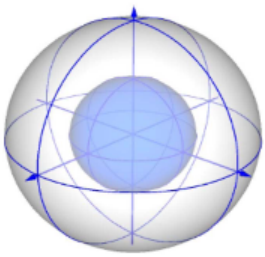


Figure 2.9: SHOT descriptor. The smaller dark blue sphere in the center contains eight sectors. The other eight sectors are located between the surface of the dark blue sphere and the surface of the bigger white sphere. Each of the sixteen sectors is described by its own angle histogram. [14]

2.2.3 Feature Matching

Once the features in the scene are found and the descriptors calculated, they need to be compared with the descriptors of the model database. If the same descriptor appears in the scene and in the model database, they probably belong to the same object. We want to single out, that the descriptors of the objects in the model database only need to be calculated once. Then they are stored alongside the models and can be reused later.

Figure 2.10: Feature matching result.

For the feature matching process different algorithm are available, which differ in speed and quality of the result . The simplest matching algorithm is based on brute force. It just takes a descriptor of the model database and compares it with every descriptor found in the scene one by one. Using some distance measure, the two closest descriptors are defined as match. Of course the simplicity of the algorithm comes with huge calculation costs. Therefore, some more sophisticated methods were invented. For example the SIFT algorithm uses a modification of the k-d tree algorithm, called the Best-bin-first search method [15]. It can find the nearest neighbor (in terms of similarity of two descriptors) with high probability efficiently. An example of a SIFT matching result is shown in Figure 2.10.

2.2.4 Hypotheses Generation

After the matching process of local features, only pairwise connections between models and the scene are known. In Figure 2.10 we see that several features of the object are found in the scene, but we don't know yet, that they belong to the same object. Therefore, corresponding grouping is done to cluster all matched features of an object together. One popular corresponding grouping algorithm is based on a 3D Hough voting scheme described in [16]. Another example is enforcing simple geometric constrains between pairwise correspondences which is based on [17]. From each group of clustered features in the scene a hypothesis can be generated. A hypothesis consists of an identifier according to the model database and a 6DOF position. In the end all the generated hypotheses from local features and hypotheses generated from global features are put together. Not all of them are true, many hypotheses overlap, violate the table plane or can't explain many points from the scene. An example of generated hypotheses is shown in Figure 2.11 center.

2.2.5 Hypotheses Verification

The hypothesis verification finds out, which set of the generated hypotheses explains the scene the best and filters false positives. Basically this is done by optimizing a cost function[18], which looks like this:

$$X^* = \min_{X \in B} \{f_S(X) + \lambda f_M(X) + f_C(X) + f_E(X)\} \ . \tag{2.1}$$

Where X is a set of hypotheses which is part of the solution space B, which contains all generated hypotheses. The scalar value λ is a regularizer aimed at penalizing model outliers, f_S and f_M account, respectively, for geometrical cues defined on scene points and model points. For example, the point cloud

model gets projected into the scene and it overlaps with a relatively high number of points of the scene. This could be an indicator for a valid hypothesis and consequently results in a relatively low value of f_S and f_M. f_C evaluates the color registration between model and scene. Finally, f_E considers physical constraints. For example, it evaluates if a hypothesis is in contact with the table plane, which could be interpreted as an indicator for a valid hypothesis. The solution X^* of the optimization problem 2.1 is an optimum set of hypotheses which explains the scene the best and is also the final output of the recognizer. For a detailed description see [8], [18] and [19] and . An example result of a hypotheses verification is shown in Figure 2.11.

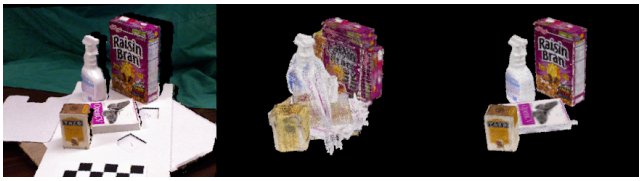


Figure 2.11: Example of hypotheses generation and verification. left: input scene, center: generated hypotheses, right: verified hypotheses ©[2013] IEEE [18]

2.2.6 Object Recognition Frameworks

To combine all the different stages explained in the previous section (feature detection → feature matching → hypotheses generation → hypotheses verification) some frameworks were developed. The framework proposed in [8] allows combining various descriptors, like SIFT, SHOT, ESF and more for hypotheses generation. It is based on 3D point cloud models and automatically calculates all necessary descriptors from the model. Originally, the framework was developed for the STRANDS [2] project, which deals with mobile robotics, but it is also useful for general purpose object recognition. Another framework called MOPED, which was already mentioned in Section 2.1.4, is presented in [7]. It works with 2D cameras and only uses SIFT for object recognition, which limits the usability to textured objects. Moreover, the hypothesis verification stage is less effective because only a feature based model (see Section 2.1.4) is used.

[2] http://strands.acin.tuwien.ac.at

2.2.7 Deep Learning

Just to present a full picture of modern object recognition methods, we shortly mention deep learning as a different approach. Deep learning concerns with artificial neural networks and other machine learning algorithms which use several hidden layers for learning feature representations from large amounts of data. [20]

As I am writing this thesis, especially ANN (Artificial Neural Network) is a very popular method in machine learning and deserves a closer examination.

ANN: Artificial Neural Network

Artificial neural networks are computational models that consist of a number of simple processing units (artificial neurons) that communicate by sending signals to one another over numerous weighted connections. [21] The structure is inspired by the human brain, which also consists of interacting units, called neurons. The artificial neurons of an ANN are arranged in layers and connected so that one layer receives input from the preceding layer of neurons and passes the output on to the subsequent layer. Mathematically a single neuron can be modeled as

$$f(x_j) = f(\alpha_j + \sum_{i=1}^{k} w_{ij} y_i) \tag{2.2}$$

For the real function f usually the sigmoid (logistic or tangent hyperbolic) function is chosen. The input $(y_1 ... y_k)$ is weighted by the weighting factors w_{ij}. A graphical interpretation is shown in Figure 2.12.

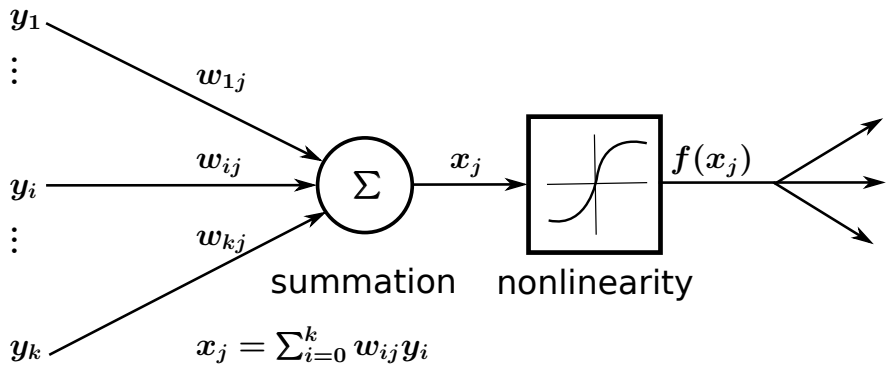


Figure 2.12: Model of a single artificial neuron.[21]

Combining and interconnecting numerous of these artificial neurons forms a whole artificial neuronal network, which can be used to classify data. We start

with a training stage, where known training data is used to adapt the weighting factors w_{ij} to get the right relation between input and output. Afterwards, the trained network can be used to classify unknown data. For example, as input data we can use features(see Section 2.2.1 and Section 2.2.2) and as classification result, we receive the object class the features belong to. This allows identifying objects in a given scene.

In general, deep learning algorithms need a huge amount of training data, compared to feature matching based methods. Moreover, it is very difficult to estimate the exact pose of objects, therefore we did not use deep learning algorithms for this thesis.

2.3 Visual Programming Languages

To provide a simple user interface, we decided to use a block based visual programming language to operate the object recognition system. This section provides an overview of some visual programming languages.

2.3.1 Blockly

Blockly[3] is a library that adds a visual code editor to web and Android apps. The Blockly editor uses interlocking, graphical blocks to represent code concepts like variables, logical expressions, loops, and more. It allows users to apply programming principles without having to worry about syntax. An example Blockly program is given in Figure 2.13.

Blockly can be run on a local webserver and the programming can be done in a normal web browser. The webserver is run as a ROS[4] node which makes the communication between Blockly blocks and the remaining ROS network possible. ROS is a robot operating system which allows different software parts of a robotic system to communicate. A more detailed explanation of ROS is given in Section 3.2. Before the Blockly code is executed, it is going to be translated into one of five programming languages of our choice. Blockly supports Python, JavaScript, PHP, Lua and Dart. Moreover, it is possible to create custom blocks. First we need to create a graphical block in JavaScript and finally add the functional code of the block in one of the supported languages. Examples of the blocks we created, are given in Section 5.2.

How the interface for creating a block, called block factory, looks like is shown in Figure 2.14. If we use custom blocks, we can easily pass information from one

[3]https://developers.google.com/blockly
[4]http://www.ros.org/

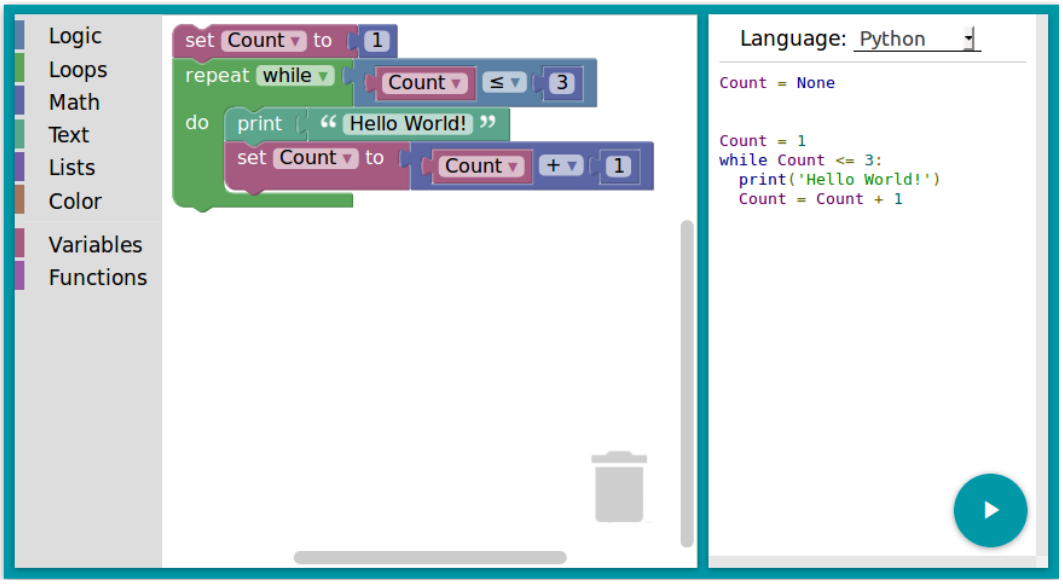


Figure 2.13: Example of a Blockly interface. On the left, the toolbar is shown which contains all available blocks. In the center, some basic blocks are combined which print the string "Hello World" three times. Finally, on the right, the translation from Blockly blocks to Python is given.

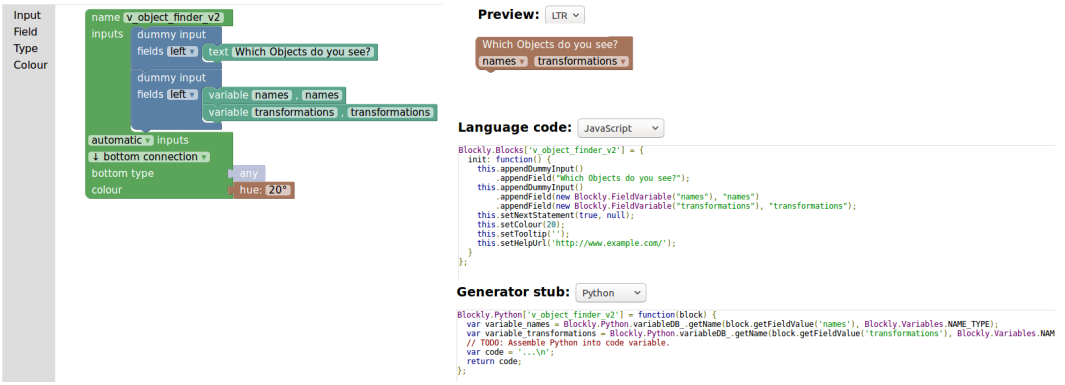


Figure 2.14: Example of the Blockly block factory. left: input, output and color of the block is chosen. Language code: This is the automatically generated code which corresponds to the blocks on the left. Preview: This is how the block is going to look like. Generator stub: In here the functionality of the block is written into the code variable.

block to another block. This gives an simple communication interface between two systems. For example, the recognition system which can be implemented in one block passes recognition results to the grasping system which can be implemented in another block.

The big advantage of Blockly over other visual programming languages is, that it is not a new programming language. It translates Blocks to Python in real time and runs Python code. Everything which can be programmed in Python can be realized in a custom Blockly block too.

2.3.2 Scratch

To create a script in Scratch[5] different blocks are put together in a visual editor, similar to Blockly. It is mainly used for educational purposes, for creating small games and visualizations. The options for creating custom blocks are very limited. It is not possible to use our own functional source code for custom blocks.

2.3.3 Snap

Snap[6] is similar to Scratch. It offers more options for creating custom blocks, but it is still not possible to use our own functional source code for a custom block.

2.3.4 Waterbear

Waterbear[7] offers a promising approach, since it is not a new programming language, similar to Blockly. It translates blocks to common languages like JavaScript. Moreover, it supports custom blocks using custom source code. As I am writing this thesis, only a pre-alpha release is available.

[5] https://scratch.mit.edu/
[6] http://snap.berkeley.edu
[7] https://github.com/dethe/waterbear

3 System Overview

Figure 3.1 shows a schematic drawing of the whole recognition system [1]. First of all the objects get modeled and stored in the model database. Then the recognizer uses the model database to recognize objects from a captured scene. For the whole data acquisition a regular RGBD camera is used. Finally, the recognizer passes location and orientation information to the grasping system. The model database provides the robotic arm with a surface model of the recognized object. This work focuses on modeling, recognition and providing the necessary data for the grasping system. The grasping algorithm and the actual physical grasping process is not part of it.

3.1 Sensor Principe and Data Acquisition

As mentioned before, for modeling and recognition a RGBD camera is used to obtain data to model objects and to capture an input scene for recognizing objects. For all experiments described in Chapter 6 the Microsoft Kinect (Figure 3.2) was used. In general this framework works with several RGBD cameras like Asus Xtion[2] or Astra Pro[3]. Only the intrinsic parameter and the resolution of the camera need to be adjusted. The Microsoft Kinect is an end user camera which was originally developed for the entertainment electronics industry. The camera provides an RGB picture with additional dept information. This allows the camera to detect arm and hand movements and introduces a new way of communication between user and video game console. A huge advantage of the Kinect is the simple data format. It provides an RGB picture and the depth information for each pixel simultaneously. The data can be interpreted as point clouds directly, which makes the Kinect also interesting for computer vision [22]. The received point clouds represent so called 2.5D data. This means only an image from a certain perspective is available. For example if one object occludes another object in the scene, the occluded object

[1] The end effector shown in Figure 3.1 is only one example. The recognition system works with various grasping systems. The picture of the end effector was provided by Shadow Robot Company (http://www.shadowrobot.com)

[2] https://www.asus.com/us/3D-Sensor/Xtion_PRO_LIVE

[3] https://orbbec3d.com/product-astra-pro

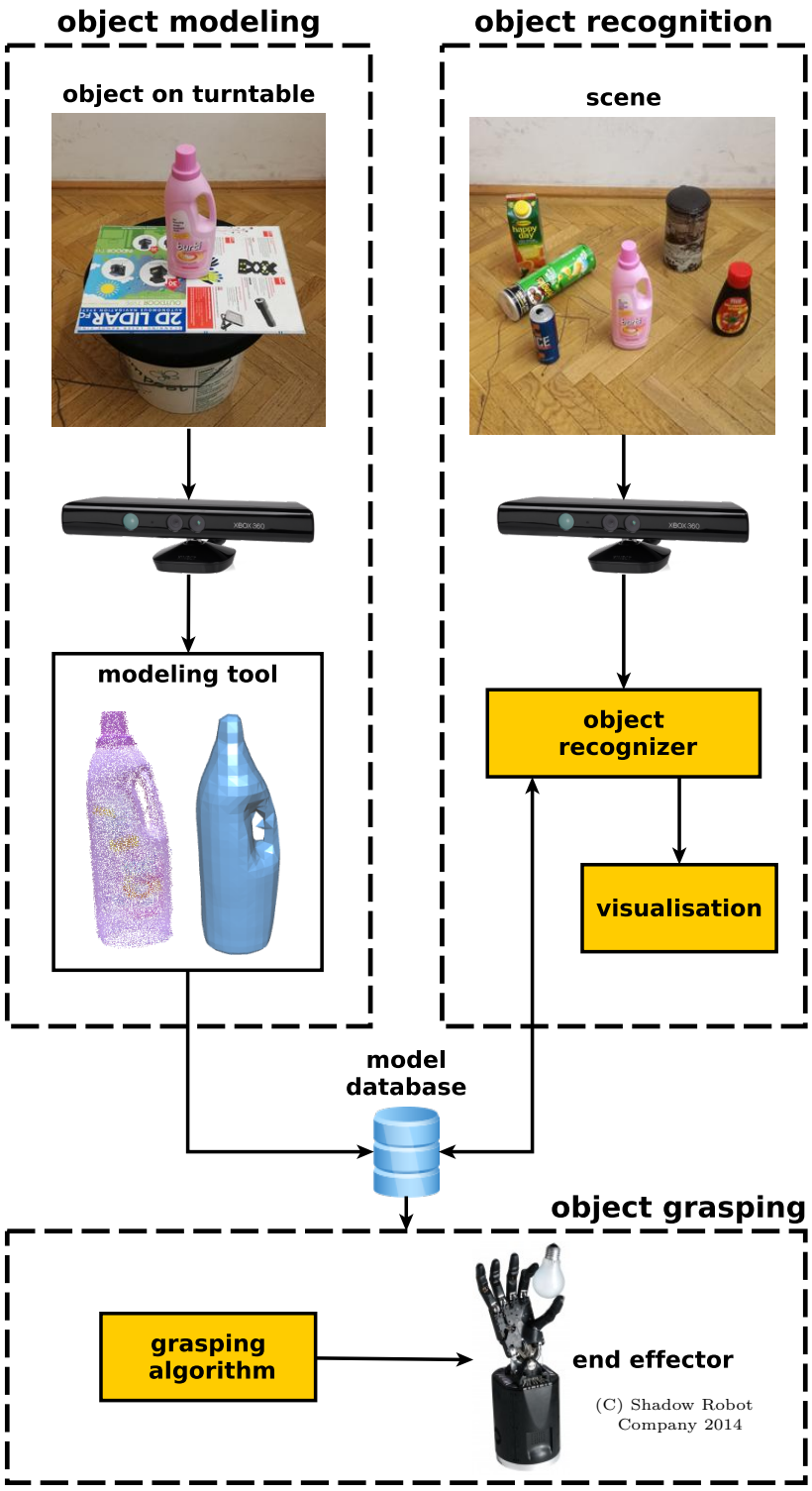


Figure 3.1: System Overview [22]

Figure 3.2: Microsoft Kinect ©[2012] IEEE [22]

is not visible in the point cloud. One advantage of 2.5D data is, that it can be stored in organized point clouds[4]. An organized point clouds resembles an organized image (or matrix) like structure, where the data is split into rows and columns. Every entry in the matrix contains spatial and color information. The advantages of a organized data set is that by knowing the relationship between adjacent points (e.g. pixels), nearest neighbor operations are much more efficient, thus speeding up the computation and lowering the costs of certain algorithms.

Moreover, through high production numbers the price of the Kinect is considerable low.[5]

The camera consists of a RGB sensor, a infrared projector and an infrared sensor. The depth computation is all done by the PrimeSense hardware built into Kinect and is not publicly available. Most likely depth was measured with a method using structured light. The Kinect uses infrared laser light, with a speckle pattern. Due to deformation of this pattern the distance of single speckles can be estimated[23]. Figure 3.3 shows a simplified setup for a depth measurement using structured light.

Using the intrinsic camera parameter as focal length f and the baseline b between the projector and the observing camera, the depth d of the pixel (x,y) can be calculated as

$$d = \frac{b * f}{m(x,y)} \tag{3.1}$$

Where $m(x,y)$ describes the disparity value. The depth range and depth accuracy directly relate to the baseline. Therefore, a longer baseline allows a more robust depth measurement.

[4]http://www.pointclouds.org/documentation/tutorials/basic_structures.php

[5]The company PrimeSense, which produced one key component of the Kinect was bought by Apple in 2013. Since then the Kinect and other cameras which used PrimeSense components are not available anymore.

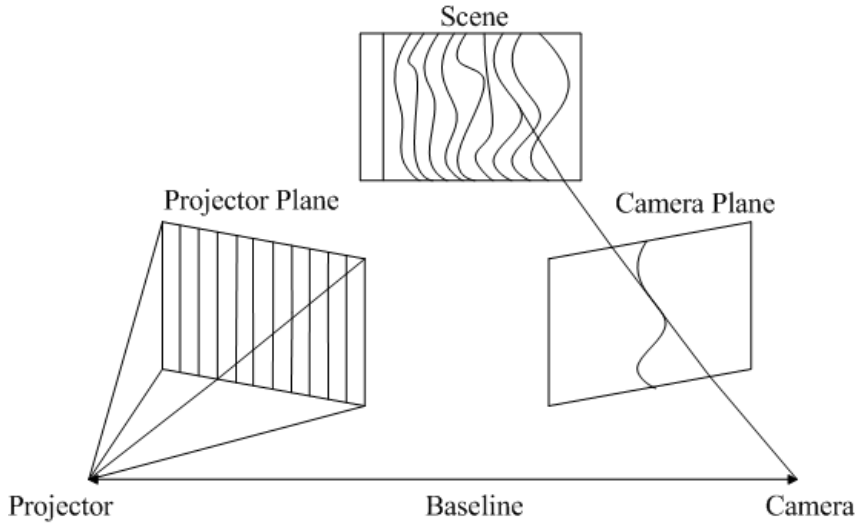


Figure 3.3: Principle of structured light method ©[2017] IEEE[24]

3.2 Implementation Overview

Figure 3.1 shows the implementation structure of the modeling and recognition system. Before I'll explain the details, We give a short introduction into ROS[6] (Robot Operation System). ROS is a framework which allows integrating different components of a robotic system. For example an object recognition system was developed by one research group and an object grasping system by another group. With ROS these two systems can be encapsulated and it provides a simple communication protocol. For example the recognition algorithm publishes the position of a recognized object and the grasping system receives this information. This means the grasping system does not need to know how the whole recognition system works in order to use the recognition results. The yellow boxes in Figure 3.1 represent encapsulated systems called nodes. The arrows represent the communication channels between these nodes.

The whole recognizer is embedded into a ROS framework. The recognizer is split up into different nodes to make the system more flexible. For detailed explanation of the single nodes see Chapter 4 and 5. The recognition process starts with a picture from the Kinect which is connected to the system by a OpenNI Node. The Node receives data from the camera and provides the point clouds to the visualization node and recognition server. The server waits for a goal from the recognition client. As soon as the goal is received the recognition server collects the scene in form of a point cloud from the OpenNI node and

[6]http://www.ros.org

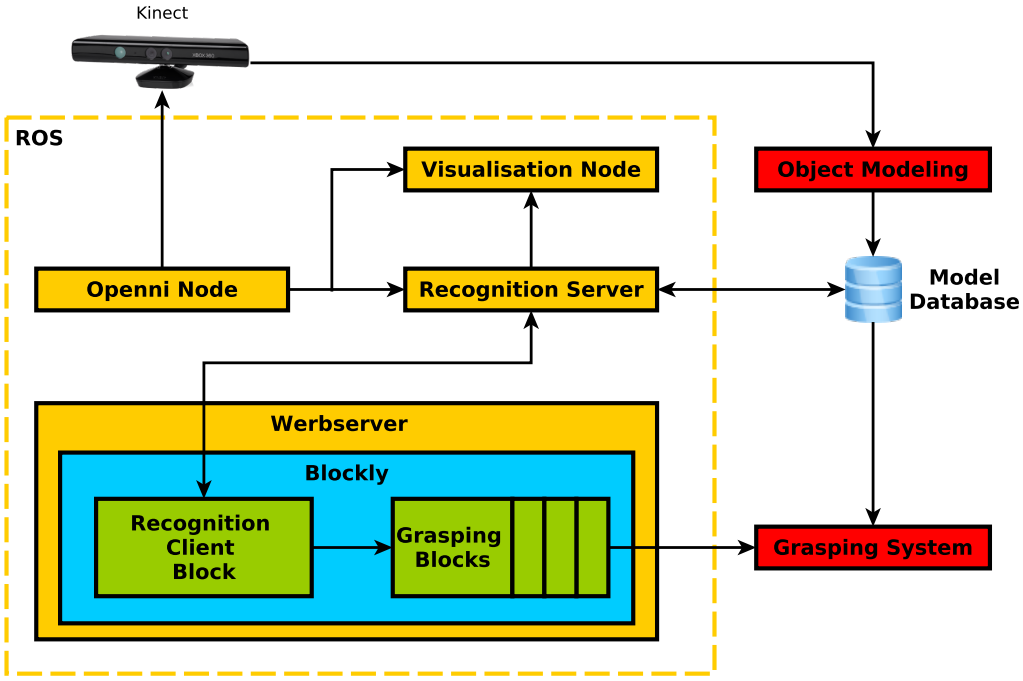


Figure 3.4: Implementation Overview, yellow: ROS nodes, blue: Blockly, green: Blockly blocks, red: other [22]

loads the models from the model database. Now the server recognizes objects in the scene based on the models in the database. After the recognition progress is finished the server sends the location and orientation information of recognized objects back to the recognition client. As we can see, the client is implemented into a Blockly block. An introduction into Blockly is given in Section 2.3.1. Inside the Blockly framework the recognized object information is passed on to the grasping system. Finally the robot can physically grasp the object and solve further tasks.

4 Modeling

The object recognizer and the grasping system need a model of every object which needs to be recognized and grasped. An offline modeling stage is required to create such a model database. We decided to modify and use the object modeling framework introduced in Section 2.1.1 because it provides a GUI based tool to model objects, simply by using a RGBD camera. Contrary to the other frameworks mentioned in Section 2.1, no advanced modeling setup is necessary. The modeling tool was mainly developed for scientific use. As we mentioned before, several options need to be selected to adjust the modeling results. For example the segmentation method and the post processing algorithms need to be chosen in order to start the modeling process. To understand all options, advanced knowledge of computer vision is required. Moreover, it requires some training time in order to operate the tool correctly. Since one goal of this thesis is to make the usage of the whole system as simple as possible, we predefined a default set of options. We tried to find a good trade-off between accuracy of the resulting model, speed of the algorithm and complexity of the graphical interface of the tool.

The physical setup for the modeling process is the same as explained in Section 2.1. The object is placed on a turntable and faces the RGBD camera. Then the turntable is turned by hand to get an image from all sides of the object.

4.1 Segmentation

During the data acquisition several key frames, which are represented by point clouds, are recorded. The camera tracks the angle of the object by a feature based method. If the same features (at least three) are found in two consecutive key frames, the transform can be estimated.

Then, the segmentation algorithm which separates the object from the background needs to be chosen. We compared the two available options, the interactive method and the region of interest (ROI) method, which were mentioned earlier in Section 2.1.

The interactive method models flat parts, larger than a certain threshold as planes and the remaining areas are recursively clustered. To cluster the image

into larger parts, the user is required to select seed points, by clicking on parts which belong to the model. Starting from the selected points, the algorithm clusters neighboring points together depending on the deviation of the surface normals of neighboring image points.

For the region of interest (ROI) approach, before the recording session starts, the supporting surface needs to be modeled to calculate a ROI automatically. The user selects the supporting plane by clicking on a point on the plane. This point is used as a seed point, similar to the interactive method explained before. Again, the neighboring image points are clustered together depending on their surface normals. The clustering stops if the plane constrictions are not fulfilled anymore. Then, a ROI is calculated, which is located directly above the plane.

Every point, which is located inside the ROI is considered to be part of the model. This reduces the segmentation process to a simple filter, which removes all data located outside the ROI. It turned out that the segmentation results are similar and both methods allow creating models which fulfill the requirements of the recognizer. We decided to use the ROI method, where only one single click is required to choose the supporting plane. This makes the tool easier to use, compared to the interactive method, where manually all parts of the object need to be selected.

4.2 Noise Model

After the turntable, with the object on top of it, was turned to acquire images of all sides of the object, the post processing starts. All the key frames acquired before need to be put together to form a full model. The key frames contain redundant data. Let's assume, roughly 20 pictures are taken during data acquisition. This means in average every 18° one point cloud of the object is recorded. Therefore, two consecutive frames contain quite a bit overlap. Combining these redundant data means, we need a quality measure in order to judge which points are finally used to create our model. Therefore, a noise model[2] is introduced which assigns a noise measure to every single point. We observed, that noise increased linearly with distance to the sensor. Since we can choose the distance between model and sensor, this does not affect us significantly. But more important, data quickly deteriorates when the angle between the sensor and the surface gets above 60°. Moreover, data close to edges of objects, therefore close to depth discontinuity, are not reliable. These points tent to jump between foreground and background. In other words, points belonging to the foreground (i.e. objects) appear on the background (i.e. table plane) and the other way around. Using the observations above, a simple noise model can be formulated. Let $P = \{p_i\}$ represent a point cloud, $N = \{n_i\}$

represent the normal information of P and $E = \{e_i\}$, where e_i indicates, if p_i is located at a depth discontinuity or not. The quality measure w_i can be formulated as

$$w_i = (1 - \frac{\theta - \theta_{max}}{90 - \theta_{max}})(1 - \frac{1}{2} exp^{-\frac{d_i^2}{\sigma_L^2}}) \ . \tag{4.1}$$

where θ represents the angle between n_i and the sensor, $\theta_{max} = 60°$, $d_i = ||p_i - p_j||_2$ (p_j being the closest point to depth discontinuity, where $e_j = true$) and $\sigma_L = 0.002$ represents the lateral noise sigma.[2]

Finally, the redundant data can be averaged and combined by using w_i as a weighting function.

4.3 Optimization

To improve the registration results two optimization algorithms are available. First, the optimization algorithm Bundle-Adjustment (BA), which minimizes the re-projection error of feature correspondences used during camera tracking.[25] This optimization problem is commonly formulated as a least-square problem. The goal is to minimize the error between the observed feature location and the projection of the corresponding 3D point on the image plane of the camera. Let x be a vector of parameters and $f(x) = [f_1(x), \ldots, f_k(x)]$ be the vector of reprojection errors for a 3D reconstruction. Then the optimization problem can be formulated as the following non-linear least square problem:[26]

$$x^* = \min_x \sum_{i=1}^{k} ||f_i(x)||^2 \ . \tag{4.2}$$

To solve this problem the Levenberg-Marquardt [27] (LM) algorithm is a popular choice. It solves a series of regularized linear approximations to the original nonlinear problem.

The second optimization algorithm is based on a multi-view Iterative Closest Point algorithm [28], that globally reduces the registration error between overlapping point clouds. The basic idea of ICP is to use a distance metric between two point sets and iteratively minimize the distance. Let $A = \{a_1, a_2, \ldots, a_n\}$ and $B = \{b_1, b_2, \ldots, b_n\}$ be two corresponding sets of points, where a_i and b_i represent single points. Solving the optimization problem

$$T^* = \min_T \sum_{i=1}^{n} ||Ta_i - b_i||^2 \tag{4.3}$$

gives us an optimum transformation T^*, which minimizes the error between A and B. T is composed of a translation vector t and a rotation matrix R. Usually the correspondences between two sets of points are not known, therefore pre-processing of the data is required. For example, the correspondence (a_i,b_i) can be created by finding the nearest neighbors between each point of A and B. Figure 4.1 [1] visualizes the algorithm.

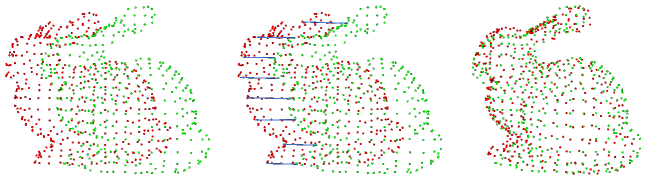


Figure 4.1: Visualization if the ICP algorithm. left: two input point clouds A(green) and B(red). center: distances between two corresponding points of A and B. right: optimum transformation T^* was applied to align A and B.

This algorithm just explains the basic function of ICP, for dealing with multiple point clouds ICP based algorithms as introduced in [28] can be used.

Both algorithms, BA and ICP generally improve the registration results, but need some calculation time. To decide if these post processing steps are necessary we took basic measures of the two objects shown in Figure 4.2 and compared them with the model. The results are listed in Table 4.1.

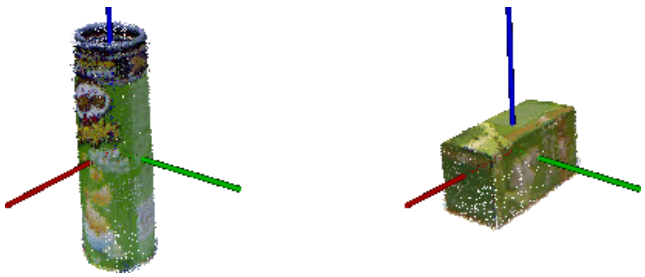


Figure 4.2: Point cloud models of objects: left Pringles, right. tea.

As we can see, the post processing improves the size of the models by several millimeters. Therefore, we decided to use Bundle-Adjustment and Iterative Closest Point as default options in the modeling tool. Of course these measures along the main axes only do not capture the full impact of the optimization algorithms on the models. Since we only wanted to find out, if these post

[1] Figure 4.1 by Dirc Holz, http://www.pointclouds.org/assets/iros2011/registration.pdf, CC BY 3.0

object	measures of object	measures of model without optimization	measures of model with optimization
pringles	Ø74 x 258	Ø88 x 256	Ø73 x 257
tea	151 x 82 x 71	156 x 78 x 77	Ø150 x 80 x 69

Table 4.1: Measurements of objects. Pringles: diameter x length, tea: length x hight x depth

processing steps are necessary, comparing the main measures was sufficient. Since the bottom of the objects can not be seen by the camera, the original modeling tool offers a feature which combines multiple modeling session to receive a fully defined model. As I am writing this thesis, this feature is not working sufficiently. More specific the different sessions are not aligned perfectly and produce an inaccurate model. Therefore, my simplified version does not support the multi-session feature. More details about this feature can be found here [2].

4.4 Surface Reconstruction

After post-processing, the final point cloud model is ready, but additionally, the modeling tool needs to provide a surface model, which is necessary for the grasping algorithm. To receive a dense surface model from the point cloud the Poisson Surface Reconstruction algorithm, proposed in here [29], was used. It finds a globally consistent surface model and finally creates a watertight polygonal mesh.

The idea of Poisson Surface Reconstruction is to start with an oriented point sample, computing a 3D indicator function $\boldsymbol{\chi}$ (defined as 1 at points inside the model, and 0 at points outside), and then obtaining the reconstructed surface by extracting an appropriate isosurface. The oriented point samples can be viewed as samples of the gradient of the model's indicator function (see Figure 4.3). Therefore, the problem of computing the indicator function reduces to finding the scalar function $\boldsymbol{\chi}$ whose gradient best approximates a vector field $\vec{\boldsymbol{V}}$ defined by the samples, i.e.

$$\boldsymbol{\chi}^* = \min_{\chi} ||\nabla \boldsymbol{\chi} - \vec{\boldsymbol{V}}|| \ . \tag{4.4}$$

If we apply the divergence operator, this variational problem transforms into a standard Poisson problem:

$$\Delta\boldsymbol{\chi} \equiv \nabla \cdot \nabla\boldsymbol{\chi} = \nabla \cdot \vec{\boldsymbol{V}} \tag{4.5}$$

This formulation has some advantages, it considers all data at once and gives a global solution. Poisson reconstruction creates very smooth surfaces that robustly approximates noisy data.

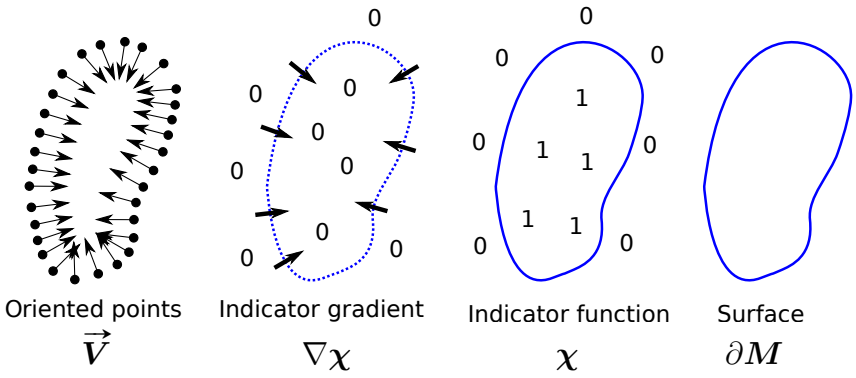


Figure 4.3: Illustration of Poisson reconstruction in 2D [29]

One problem occurs if the algorithm is applied on data with incomplete depth data. For example, if an object was modeled but a part (for example the bottom) is missing. Poisson reconstruction tends to add extensions to the undefined parts of the model (Figure 4.4, left).

Figure 4.4: Poisson surface mesh before and after cropping using the convex hull ©[2015] IEEE [2]

One way to remove these extensions is to initially calculate the convex hull of the point cloud. After applying Poisson reconstruction, all vertex which lay outside the convex hull are projected on the hull. This ensures that the resulting mesh model does not have any parts outside the hull (Figure 4.4[2], right).

[2]Figure 4.4 shows a textured surface reconstruction. In this work we did not use texture, because the mesh model is meant for grasping applications, where colors are not important.

4.5 User Interface

Finally, we want to present the new simplified interface of the modeling tool (Figure 4.5) compared to the original one (Figure 4.6). The following section gives a short overview, how to operate the tool to receive a model. First we have to select the surface under the model then a region of interest (ROI) is created automatically. Everything within this region is considered to be part of the model. After clicking the button "Start Modeling" we need to turn the model for 360° and the tool records point clouds from all directions of the model. Simultaneously, the tool takes track of the orientation of the objects, using features of the supporting surface and the object itself. If one whole rotation is finished, we need to click on "Finish Modeling". Now the post-processing algorithms will create a consistent model using the data recorded before. First, the ROI-segmentation will remove all points, which are outside the ROI (blue grid box in Figure 4.5). Since the tool kept track of the orientation of the object during recording session, transforms between all point clouds and a reference coordinate system exist. This allows to transform all point clouds into this reference coordinate system and create a model already. Since tracking errors may occur during recording session, optimization algorithms may improve the accuracy of the model. Therefore, Bundle-Adjustment (BA) is applied, which directly reduces the re-projection error of correspondences used during camera tracking. The second optimization method is a multi-view Iterative Closest Point (ICP) algorithm that globally reduces the registration error between overlapping point clouds by iteratively adapting the transformation between camera poses. The outcome is a point cloud model, which can be used for recognition purposes directly. Moreover, a surface mesh model is needed, which can be calculated from the point cloud model by applying Poisson Surface Reconstruction. It finds a globally consistent surface model and creates a watertight polygonal mesh. Finally, we can store the point cloud model and the mesh model and use it for further purposes, like object recognition and object grasping applications.

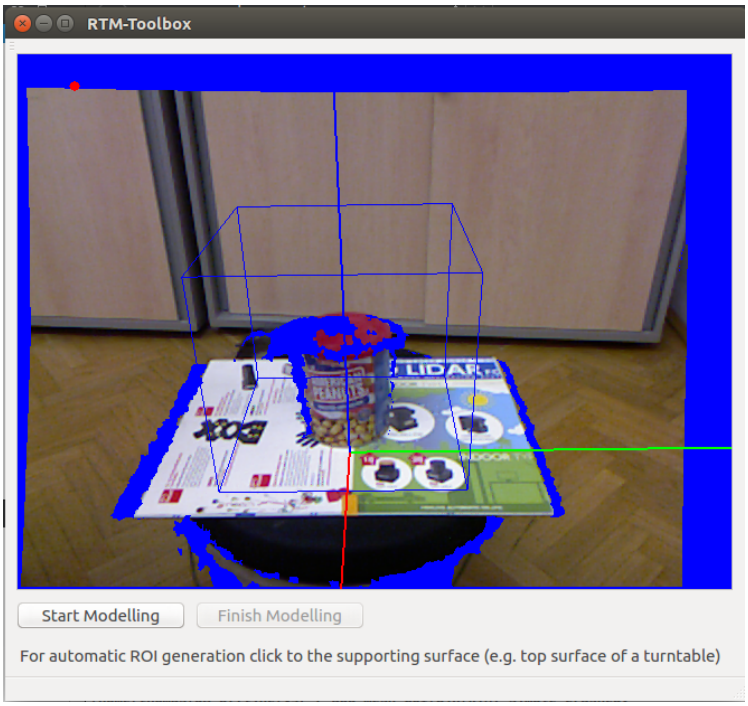


Figure 4.5: Simplified version of the RTM-tool

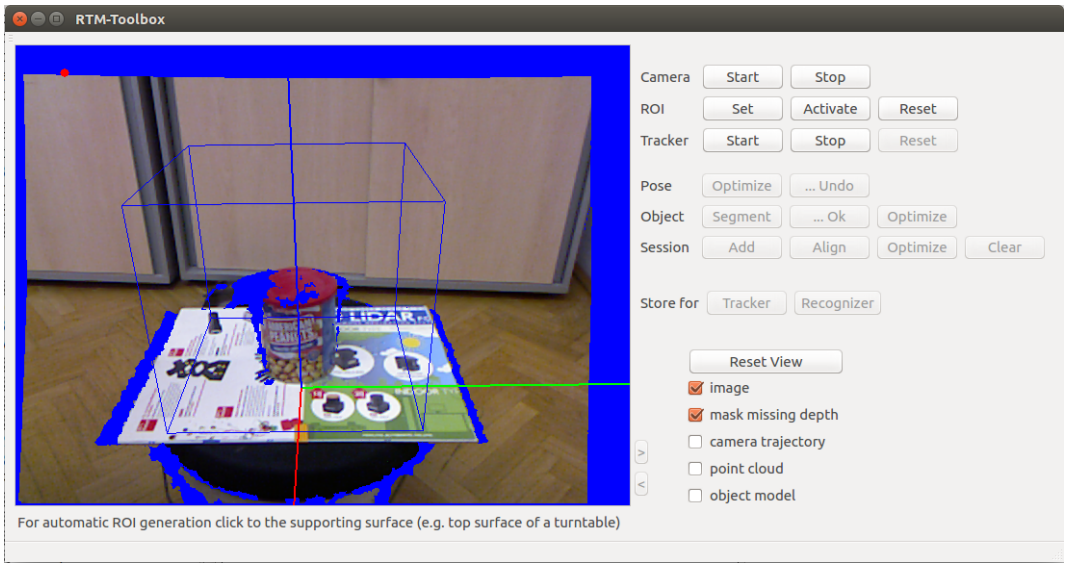


Figure 4.6: Original version of the RTM-tool

5 Recognition System

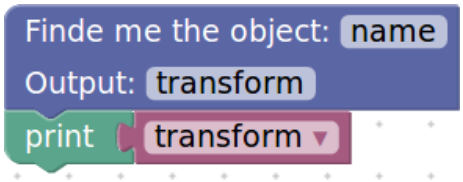
Using the models created with the tool introduced in the previous chapter, the actual object recognition can start. The main part of the recognition system is the recognition server, which contains the algorithms for recognizing objects. For details about the recognition server see Section 5.3. To create a user friendly environment, we used a visual programming interface based on Blockly (see Section 2.3.1). Also, some grasping algorithms of the robotic setup implemented in Blockly. This provides a simple communication interface between the grasping system and the recognition system.

5.1 Client/Server Architecture

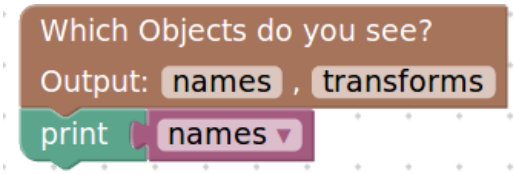
As mentioned in Section 2.3.1, Blockly only support Python, Java Script, PHP, Lua and Dart as source code languages for blocks. Due to performance reasons the recognition algorithms are written in C++. Since C++ is not supported, it is not possible to implement the recognizer directly into a Blockly block. A delicate approach to solve this problem is to create an interface between Blockly and the remaining ROS network, by implementing an client/server architecture. More details are explained in the following section.

5.2 Recognition Client

ROS allows running nodes in Python and in C++ within the same ROS network. For example, it is possible to run a server in C++ and a client in Python. This gives us the necessary interface to communicate between a Blockly block written in Python and another node written in C++. We implemented the client into a Blockly block and the corresponding server is implemented as a C++ node, which contains the recognition algorithms (see Section 5.3). In short, the Blockly block gives an order to start the object recognition task and to return the results. Based on this method, we created two custom blocks in Blockly. One of them does not have any input parameter and returns the name, position and orientation of all recognized objects. The second one needs one input parameter, the name of an object which needs to be recognized. This



(a) A Blockly program which recognizes a specific object in a given scene and prints the output "transform" which includes the position and orientation of the recognized object.



(b) A Blockly program which recognizes all objects in a given scene and prints the names of the recognized objects.

Figure 5.1: Two simple Blockly programs, which use costume blocks to control the recognizer.

block returns just the position and orientation of the object which was given as input parameter and all other objects in the scene are ignored. In Figure 5.1 an easy example is given how these recognition blocks can be used. Of course, in order to recognize objects from a given scene, these objects need to be modeled first and stored in the model database before, as described in Chapter 4.

5.3 Recognition Server

The recognition server is the main node of the recognition system. It has implemented all the actual recognition algorithms which were described in Section 2.2. Additionally, features were implemented to optimize the recognition process with respect to speed. Details about these features are presented in the following sections.

After running the recognition server node, the recognizer is initialized automatically. All user defined parameters, which are stored in configuration files are loaded into the recognizer. For an overview of available parameters, see Appendix B. Then the models from the database are loaded. Is there a new model, which were never used for recognition before, all necessary descriptors for this model are calculated. Depending on which recognition algorithms are in use, descriptors for SIFT, SHOT and/or ESF are calculated and stored in the model database. As mentioned in Section 2.2.3, the descriptors for the models only need to be calculated once. For every following recognition task the descriptors are simply loaded from the database. Now the recognition server is ready for orders from the client. After an order was received, the recognizer starts working. First an image of the Kinect is loaded via the openNI node.

Next, the key points and descriptors of the image are calculated. As described in Section 2.2.3 SIFT uses a modified k-d tree algorithm to match features between the scene and the features stored in the model database. Based on the correctly matched features, hypotheses are generated (Section 2.2.4). Enforcing simple geometric constrains between pairwise correspondences allows grouping matched features together and assign them to a certain model of the database. In other words, all groups of pairwise matched features in a scene represent a model of the model database. To improve the quality of the results a verification stage was introduced in Section 2.2.5. It finds out which set of the generated hypotheses explains the scene the best and filters false positives. Finally, the recognizer sends back the results to the client, which contain the identifications, positions and orientations of the recognized objects represented by the matrix T. It transforms the model coordinate system into the scene coordinate system which gives a distinct location of every recognized model in the scene.

$$T = \begin{pmatrix} r_{00} & r_{01} & r_{02} & t_1 \\ r_{10} & r_{11} & r_{12} & t_2 \\ r_{20} & r_{21} & r_{22} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.1}$$

Where the matrix entries r_{ij} describe the rotation and the entries t_k describe the translation.

In order to visualizes the results, they are also sent to the visualization node, see Section 5.5.

5.3.1 Parameter Mapping

As mentioned in the previous section, the user needs to define parameters before the recognizer can be used. The configuration files contain about 80 parameters, see Appendix B. Some of them are simple to understand. For example the parameters "do_sift" and "do_shot" are activating or deactivating recognition pipelines. If each of these two parameters are set to "0" or "1" the recognizer uses SIFT and/or SHOT for object recognition. But for interpreting most of the other parameters, an advanced knowledge in computer vision is required. Since one goal of my theses is to make the recognizer available for users without such knowledge, we implemented a feature which simplifies the configuration process. The user does not need to define all parameters by himself or herself, he or she can choose between a number of different parameter settings. These predefined settings give the recognizer certain specifications. The following descriptions of available settings are qualitative, for a quantitative evaluation, see Chapter 6.

fast: If the recognizer is initialized with this parameter setting, low calculation effort is needed. On the other hand the results are not as accurate compared to the other settings.

accurate: This parameter setting makes the recognizer slower than the previous one. But the results are more accurate. This is the default setting of the recognizer because it gives a balanced trade-off between accuracy and speed.

high recall: The third option has the best results in terms of recall, but also takes the longest time for the recognition process. A high recall means, many objects were recognized (many true positives) without taking into account false positives, see Chapter 6.

These three parameter sets are just examples, the number of sets could be scaled up arbitrarily.

5.3.2 Initialize Empty Workspace

As we mentioned before, every time a recognition task is started, a new image of the scene needs to be processed. This is necessary to keep the results up to date, because something could have changed in the scene. For example an object could have been removed or maybe a new object could have been added. This is time consuming, because the recognizer starts a whole new recognition process on the new scene. In the following section we propose an algorithm to speed up the recognizer while maintaining the accuracy of the recognition results. We assumed that at least the work space around the objects does not change. Even if some objects are moved, the table top, or the floor beneath the objects does not move. If we know how our empty work space looks like, we can subtract the picture of the empty work space from the picture of the new scene. Afterwards the input picture of the recognizer only consists of relevant data, namely the objects without any background.

An example of a scene where the empty work space was removed is given in Figure 5.2

In order to make the algorithm fast, we assumed, that the camera is mounted on a fixed position. It does not move during the whole time the recognizer is used. With this assumption an efficient algorithm can be written, because these two scenes can be compared pixel by pixel and no cost intensive difference segmentation algorithm [1] needs to be used.

[1] http://docs.pointclouds.org/trunk/classpcl_1_1_segment_differences.html

Figure 5.2: Working principle of empty work space removal. left: point cloud A of empty work space, center: point cloud B of the scene, right: point cloud C of a scene where the empty work space was removed.

To understand the problems which comes with the difference segmentation we give an overview on the implementation.

As we said before, the algorithm compares the two scenes pixel by pixel. This is possible because the two scenes are represented by an organized point cloud. As described in Section 3.1, organized point clouds are represented by a matrix like structure.

$$A = \begin{pmatrix} a_{00} & a_{01} & \dots & a_{0M} \\ a_{10} & a_{11} & \dots & a_{1M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N0} & a_{N1} & \dots & a_{NM} \end{pmatrix}$$

Every entry of A represents a pixel of the point cloud, which contains color and spatial information $a_{ij} = \{x, y, z, rgb\}$. The indices i and j represent the x- and y- axes of the image. The indices N and M represent the resolution of the image in the respective direction.

Therefore, the algorithm just needs to compare the depth information stored in the two matrices A and B to detect spatial differences Δz_{ij},

$$\Delta z_{ij} = z(a_{ij}) - z(b_{ij}) \tag{5.2}$$

where $z(a_{ij})$ and $z(b_{ij})$ represents the depth values of the pixels a_{ij} and b_{ij} of the point clouds A and B. If

$$\Delta z_{ij} > c * z^2(b_{ij}) \tag{5.3}$$

where c is a constant value, the pixels a_{ij} and b_{ij} are considered to be different and stored in a point cloud C. On the right side of 5.3 we chose a threshold which is proportional to $z^2(b_{ij})$ in order to model the noise behavior of the RGBD camera. The point clouds A and B were acquired by a RGBD camera.

The further the scene is away from the camera, the higher the noise level of the respective pixels gets. The output of the algorithm is a point cloud C which only contains the differences between a point cloud A and a point cloud B An example is given in Figure 5.2.

Since only spatial information is used, some errors can occur which are explained in more detail in the following Section 5.3.3.

Using this algorithm speeds up the process, because all irrelevant data is removed from the scene before the recognition process is started. First of all, only descriptors of the remaining data need to be calculated. Moreover, no false positive hypotheses are provoked by misleading key points on the work space. This leads to a faster hypothesis verification too, because fewer hypotheses need to be validated.

One drawback of this method is, that we need to acquire a point cloud of the empty work space. This is done automatically during the initialization of the recognizer. Therefore, the work space needs to be empty during the start up of the recognizer. This could be inconvenient, because we need to remove all the objects from the work space before we can start the recognizer. Moreover, if we forget to empty the work space, the recognizer considers all objects, which are present during initialization as part of the empty work space. This means these objects are not going to be recognized during the following recognizer calls. One way around this problem is offered in the next section.

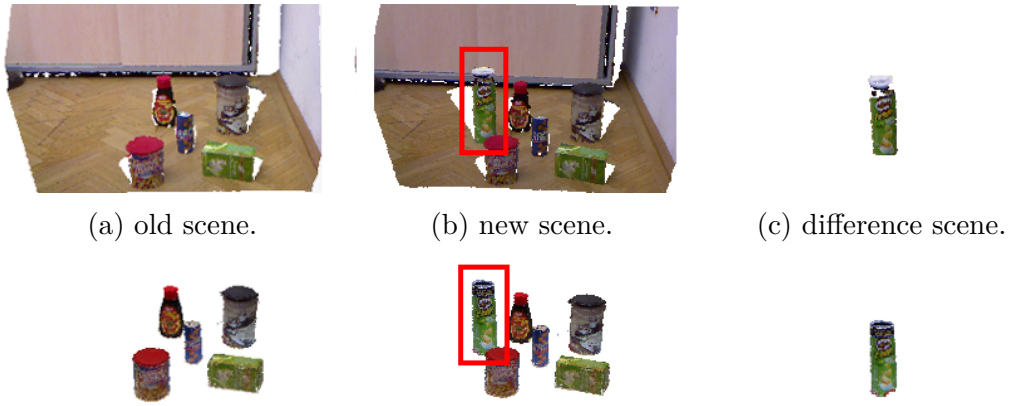
5.3.3 Object Change Detection

The algorithm in the previous section only filters the empty work space of the scene. We still need to run a whole new recognition process on the remaining objects in the scene. For example, if we add objects to the scene, the recognizer needs to re-recognize all the objects, even if most of the objects were recognized in the last recognizer call already. Now the idea is to reuse the information from previous recognition results to reduce calculation effort. Figure 5.3 shows the scenes and the recognition results of two consecutive recognizer calls. As we can see, the only difference between the two scenes is the Pringles can. Nevertheless, for the second recognizer call all objects need to be re-recognized.

We propose a method which re-uses the results from the first scene for the recognition of the second scene.

To explain the working principle of the algorithm, we use the example shown in Figure 5.3.

A represents the point cloud of the *old scene*, B represents the point cloud of the *new scene* and $C = diff(A,B)$ represents the output of the difference segmentation, the *difference scene*.



(a) old scene. (b) new scene. (c) difference scene.

(d) verified hypotheses of the old scene. (e) verified hypotheses of the new scene. (f) generated hypotheses of the difference scene.

Figure 5.3: Example of scenes.

Moreover, the *old scene* A contains the objects $O_A = \{o_1, o_2, ...o_N\}$ like shown in Figure 5.3d, where o_i represents the objects in the scene and $N = |O_A|$ is the total number of objects in A. In our example $N = 5$. The point cloud B, the *new scene*, (Figure 5.3b) contains the objects $O_B = \{o_1, o_2, \dots o_M\}$, where $M = |O_B|$ is the total number of object in B. In our example $M = 6$.

Let's assume we did object recognition on the *old scene* A and recognized all objects O_A, then we alter the scene to the *new scene* B, and we want to do object recognition on B to get O_B. One solution is to run a new object recognition on B, which gives us directly O_B, but takes a lot of calculation effort. To speed up the process we can reuse recognized objects from O_A from the previous object recognition of A. Therefore, we only need to extract objects which obey:

$$O_{A \cap B} = O_A \cap O_B \ , \tag{5.4}$$

and only recognize the new objects, which are element of O_B but not element of O_A. In order to do so, we first apply difference segmentation, similar to the "Initalize Empty Work Space" algorithm, which gives us

$$C = diff(A,B) \ . \tag{5.5}$$

For example in Figure 5.3 only one object was added. Consequently, the algorithm removes the whole scene except one object. In other words, the point cloud C only contains points which belong to the Pringles can, because it describes the difference between the point clouds A and B (see Figure 5.3c). Object recognition on the *difference scene* C gives us $O_C = \{o_1, o_2, ...o_R\}$, where

$R = |O_C|$, the total number of objects in the *difference scene* C, in our example, only the Pringles can is present, therefore $R = 1$. It holds

$$O_C = O_B \setminus O_{A \cap B} = O_B \setminus O_A \ . \tag{5.6}$$

To get all objects O_B we need to apply hypotheses verification of the *new scene* B on

$$O_{A \cup C} = O_A \cup O_C \ . \tag{5.7}$$

For clarification, we should mention, that usually hypotheses verification is the last step of object recognition and is used to verify generated hypotheses. Hypotheses which can't explain enough points of the scene are removed, as explained in Section 2.2.5. But in this algorithm, we use hypotheses verification also to verify, if verified hypotheses O_A of the point cloud A are also valid hypotheses O_B of the point cloud B.

Therefore, hypotheses verification removes all objects $O_D = O_A \setminus O_B$ from the set $O_{A \cup C}$ which results in O_B because

$$O_B = O_{A \cup C} \setminus O_D \ . \tag{5.8}$$

This seems unnecessary complicated, because in our example $O_B = O_A \cup O_C$, but in general, this is not valid. For example, if in the scene B (Figure 5.3b) not only the Pringles can was added, but also the coffee can was removed, then the coffee can is element of O_A but not element of O_B, therefore $O_B \neq O_A \cup O_C$.

The flow chart shown in Figure 5.4 gives an overview of the implementation of the algorithm described above.

General Conditions

Finally, we want to discuss some general conditions of the algorithm to ensure a correct functionality. Similar to the "Initialize Empty Work Space" algorithm explained in Section 5.3.2, the first task of the "Object Change Detection" algorithm is the difference segmentation. As explained before the difference between two scenes is found only by spatial differences, or more precise, by differences in depth data between two point clouds. This can lead to wrong segmentation results if symmetrical or spatial similar objects are involved. For example, if the peanuts can shown in Figure 5.3b is turned by 180° around its symmetry axis and the "Object Change Detection" algorithm is run. The algorithm is going to remove the peanuts can and the difference scene is going to look exactly like shown in Figure 5.3f. After the following hypotheses verification stage, two outcomes are possible. One outcome is that the hypotheses of the peanuts can passes the hypotheses verification, because it is still spatially consistent with the scene. Since the results are used for grasping applications

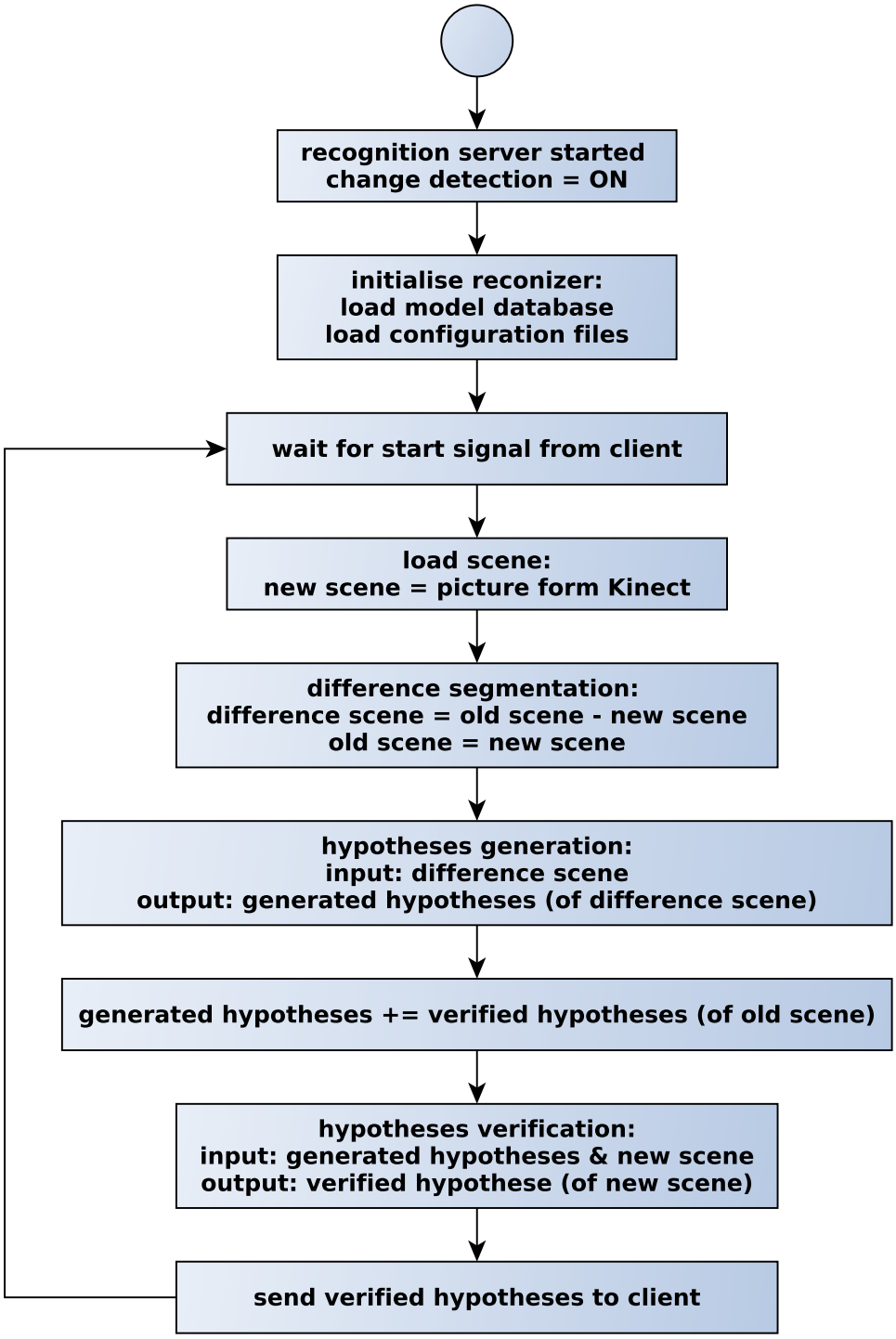


Figure 5.4: Flow chart of "Object Change Detection" algorithm.
The names *old scene*, *new scene*, *difference scene*, *generated hypotheses* and *verified hypotheses* refer to the examples given in Figure 5.3.

spatial consistent changes of an object can be neglected and the results can be considered as true. The other outcome is that the hypotheses of the peanuts can fails the verification, because the color of the scene is not consistent with the color of the hypotheses anymore. This means the peanuts can is not recognized anymore and consequently no position or orientation information is available. The actual outcome depends on the configuration of the recognizer. Another problem occurs, if two spatially similar objects are exchanged. For example, if the coffee can in Figure 5.3b is removed and the peanuts can is moved to the exact same position as the coffee can has been before. Then the "Object Change Detection" algorithm filters the peanuts can from the scene. This happens, because it is spatially very similar to the coffee can. After the hypotheses verification step the hypotheses from the coffee can is going to be removed, because it is not there anymore. Moreover, the peanuts can on the new position will not be recognized, because it was filtered from the scene before. As we are going to show in Chapter 6, these errors cause a slight reduction of the recognition accuracy.

5.4 OpenNI Node

The recognizer uses the point clouds provided by the Kinect or any other connected RGBD camera. We decided to use a separate node which handles the communication between the ROS network and the Kinect, shown in Chapter 3. Since we use a Kinect, we implemented a node based on ROS OpenNI [2], which is an open source project focused on the integration of the PrimeSense sensors with ROS.

The node supports the standard ROS point cloud format which makes the data transfer within the network easy. For example point clouds can be sent to the recognition server and the visualization server simultaneously. The main advantage of a stand-alone camera node is, that it makes the system more flexible. If a different camera for the recognizer needs to be used, just the camera node needs to be adapted to the new camera. The other nodes in the recognition system don't need to be changed. Another advantage is, that the Kinect data is available throughout the whole ROS network. For example if another node is implemented later, it can easily access the Kinect without interfering with other nodes.

[2] http://wiki.ros.org/openni_kinect

5.5 Visualization Node

To make the whole recognition system more appealing for users, an easy to understand visualization of the recognition process is desirable. Moreover, visualized object recognition results are convenient for debugging, because users can judge immediately if objects were recognized correctly. An example of the visualization window is given in Figure 5.5.

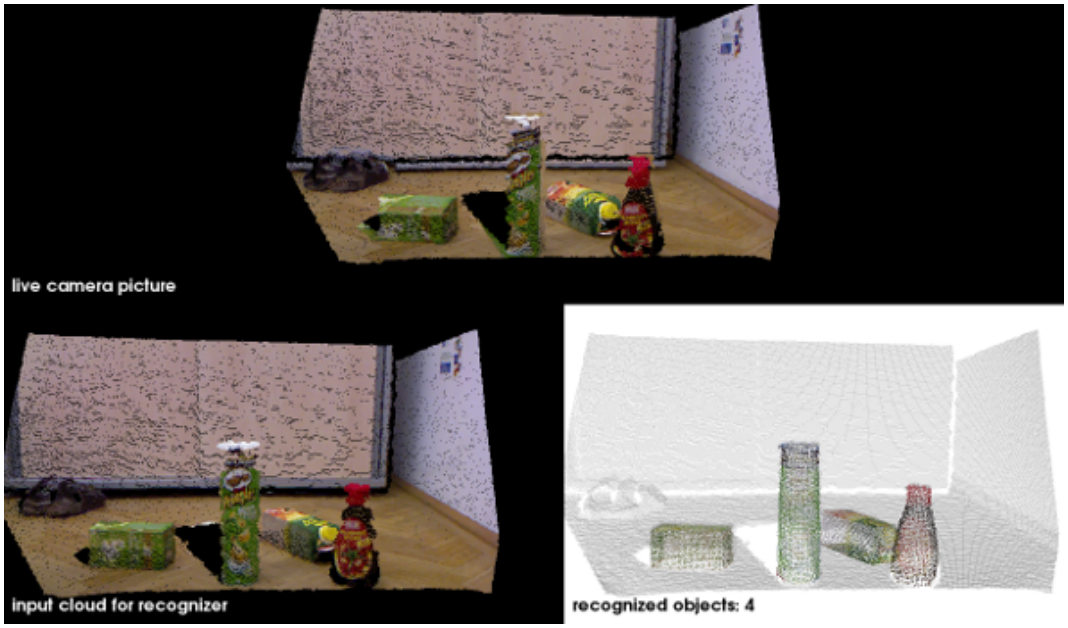


Figure 5.5: Visualization window.

The top picture is a live stream from the Kinect camera directly. The picture on the left-bottom is the input cloud of the recognizer during the last recognizer call. This is especially useful if the algorithms described in Section 5.3.2 or Section 5.3.3 are used, because the user can directly see the working principle of these algorithms. If the "Initialize Empty Work space" algorithm is used the input cloud consists only of the objects without any background as described in Section 5.3.2. If the "Object Change Detection" algorithm is used the input cloud consists only of the new, or moved objects compared to the previous recognizer call. More details of the algorithm are given in Section 5.3.3. And finally the picture in the bottom-right corner are the point clouds of the recognized objects projected into the scene.

As we can see, for the visualization data from the OpenNI node and the results from the recognition server are needed. Therefore, we decided to implement a separate visualization node as shown in Figure 3.4. Another advantage of a

separate node is, that the visualization window is available during the whole recognition process.

6 Evaluation

The algorithms we described in the last chapter are based on the recognizer proposed in this PhD thesis [30] Therefore we decided to use the same evaluation criteria for the recognition results. Moreover, these criteria are commonly used in the computer vision community. One measure to judge the performance of an algorithm is the recall. It gives an objective number on how well all objects in a scene were recovered. It gives a measure if the present objects were explained by an appropriate verified hypotheses. The recall is defined as

$$\text{recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}} \ . \tag{6.1}$$

Another measure is the precession. At this point we are more interested if the algorithm produces a fair amount of false positives alongside the true positives, which was not taken into account by the recall measure. Precession is defined as

$$\text{precession} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}} \ . \tag{6.2}$$

The third measure we used is the F-score. It is convenient, because the whole recognition system is judged by a single number. The F-score is defined as

$$\text{F} = 2 * \frac{\text{precision } * \text{ recall}}{\text{precision} + \text{recall}} \ . \tag{6.3}$$

An object is counted as a true positive if the proposed verified object hypotheses is sufficiently close to the ground truth-object in the scene. In particular the hypothesis needs to align the ground truth object in a way that the centroids are within 5cm. Furthermore, the maximum in-plane and out-of-plane rotation needs to be less than 30°. For a more detailed explanation of the evaluation criteria see [30].

For the evaluation we used an object set consisting of nine objects shown in Figure 6.1. Since the recognition results are used for grasping tasks we were restricted on the size and form of the objects. The requirements were, that the objects should be in similar size and shape as objects used in the ycb data set [5]. Moreover, we used mostly objects with a fair amount of texture to improve

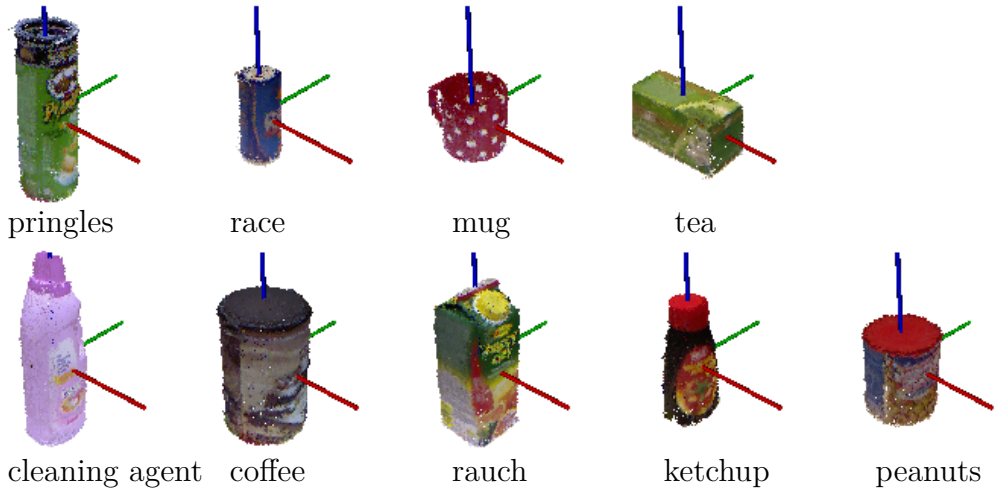


Figure 6.1: Objects stored in the model database.

Figure 6.2: Examples of the constant workspace dataset.

recognition results. All models were created by the modeling tool proposed in Chapter 4.

We created a test set of scenes which share the same work space. Moreover, the location and orientation of the camera was not altered during recording the scenes. This was necessary to properly test the "Initialize Empty Work space" algorithm proposed in Section 5.3.2. Some examples of test scenes are given in Figure 6.2. To test the algorithm "Object Change Detection" proposed in Section 5.3.3 no requirements like a constant work space are necessary for the test set. In order to give an example of the full potential of the algorithm we created a test set where from one scene to another only few changes occur. Two examples of this data set are shown in Figure 5.3. In Table 6.1 statistics of the used data sets are listed.

dataset	objects	sequences	object instances in all sequences
Const. Workspace	9	17	98
Change	7	4	22

Table 6.1: Statistics of the datasets.

6.1 Parameter Mapping

In Section 5.3.1 a system was proposed to simplify the configuration process for the recognizer. Three different parameter settings were created which change the recognition results and the computational time of the recognizer. The statistics of the results of different settings are shown in Table 6.2. Moreover, a ROC of the recognizer is given in Figure 6.3. The three configurations, *fast*, *accurate* and *high recall* only use the algorithm SIFT for object hypotheses generation. It turned out, additional recognition pipelines like SHOT or ESF do not improve the recognition results, see Table 6.2 last two lines. SHOT and ESF decreased the performance of the recognition system and also increased the median calculation times. This dominance of SIFT is mainly due to my choice of objects (see Figure 6.1) we used for evaluation. As mentioned before, we purposely used objects which have textured surfaces in order to increase recognition results.

parameter set	P(TP)	P(FP)	recall	precision	fscore	median time	
fast	0.67	0.03	0.67	0.96	0.79	3.1	s
accurate	0.85	0.07	0.84	0.92	0.88	5.5	s
high recall	0.90	0.14	0.87	0.89	0.86	11.9	s
high recall + SHOT	0.93	0.63	0.92	0.59	0.72	48.9	s
high recall + ESF	0.88	0.18	0.87	0.83	0.85	14.7	s

Table 6.2: Evaluation of *Parameter Mapping*.

For all three configuration we used a basic parameter set and adjusted some parameter to influence the results of the recognizer and the calculation time. An overview of all parameter available in the configuration files is given in Appendix B.

Some examples of different parameters of different parameter sets are given in Table 6.3.

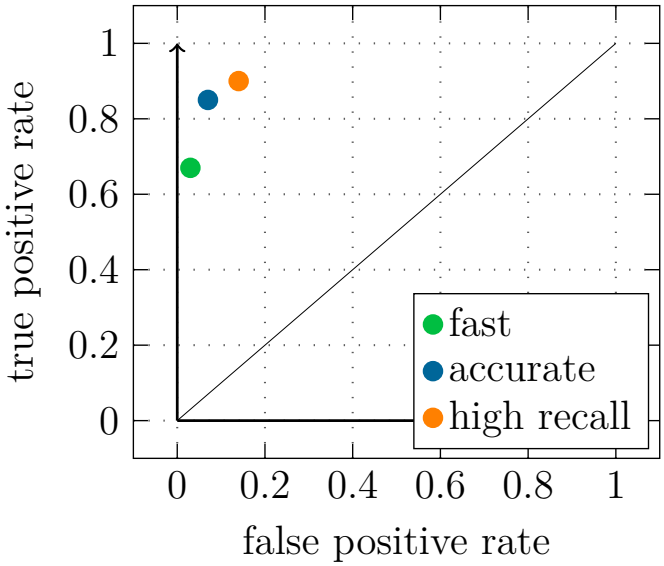


Figure 6.3: ROC

knn:
Is the number of nearest neighbors a feature of the scene is matched to features of the model database. It influences the SIFT hypotheses generation stage. The higher the value of parameter is chosen, the more hypotheses are generated. In general increasing the value of this parameter increases the calculation effort but also improves performance of the recognition results.
distance metric:
Two different distance metrics were used to define the similarity between two SIFT descriptors. For the *fast* setting the Hellinger distance metric was used, which gives a measure on the similarity between two probability distributions. For the *accurate* and *high recall* settings the standard L1 norm was used. Defining the distance between two features is not trivial, therefore it is hardly possible to predict which distance metric is optimal for certain scenes or objects.
min. fitness:
This is a measure between zero and one for the general confidence of a generated object hypotheses. This parameter influences the hypotheses verification stage. If an object hypotheses has a fitness lower than the value of min. fitness, it gets rejected. The higher this number is chosen the more wrong hypotheses get filtered, but also more right positive hypotheses get rejected too.
resolution:
The resolution of models and scene used to verify hypotheses. Before the hypotheses verification starts the point clouds of the scene and the object models get down sampled. In general a smaller resolution slows down the

verification stage, but increases the accuracy of the recognition results.
inlier threshold:
It describes the inlier distance in meters between model and scene points during the verification stage. One measure to judge the confidence of a generated hypotheses is the number of points in a scene are explained by a hypothesis. In general the rejection rate of hypotheses gets higher if we decrease the value of the inlier threshold.
filter boarder points:
With this parameter the boundary point filter is chosen. It filters key points close to the boundary of the point clouds within the scene. In general, activating the filter speeds up the calculation process because less key points are used for recognition. On the other hand sometimes valid key points are filtered which decreases the quality of the results.

parameter set	knn	distance metric	min. fitness	resolution in mm	inliers threshold	filter border points
fast	1	Hellinger	0.3	20	0.03	1
accurate	2	L1 norm	0.5	12	0.02	0
high recall	4	L1 norm	0.3	5	0.01	0

Table 6.3: Examples of different parameters.

6.2 Initialize Empty Workspace

In Section 5.3.2 an algorithm was introduced to speed up the recognition process while maintaining the quality of the recognition results. In Table 6.4 the evaluation results are listed. All three algorithms were initialized with the same parameter set. The evaluation was done on the *Constant Work space* data set. As we can see, the median time could be reduced by roughly 20%. The precission was increased by 6%. Occasionally the background provokes false positive hypotheses. Since the background was removed the false positive rate got lower and leads to a higher precision. Also, the recall got slightly reduced. One explanation is, that the difference segmentation does not work perfectly. Occasionally, not only the background, but also small parts of an object gets filtered too, which makes it more difficult to recognize an object.

Algorithm	recall	precision	fscore	median time	
Original	0.87	0.90	0.88	10.5	s
Initialize Work space	0.82	0.96	0.88	8.5	s
Change Detection	0.78	0.88	0.83	5.6	s

Table 6.4: Evaluation on *Constant Work space* data set.

6.3 Object Change Detection

In Section 5.3.3 an algorithm was proposed to speed up the calculation time of the recognizer. Especially if small changes in the scene between two consecutive recognizer calls occur, we expect a significant improvement of the results with respect to speed. In Table 6.5 the evaluation results are shown. The median calculation time could be reduced by more than a factor of three. The *Change* data set was created to show the full potential of the "Object Change Detection" algorithm. As explained earlier, in this chapter a set of different test scenes were used where only one object was added between two scenes (see Figure 5.3). Moreover, we tested the algorithm on the *Constant Workspace* data set. The results are listed in Table 6.4. Compared to the original algorithm, the median calculation time could be reduced significantly. But on the other hand also recall and precision were reduced slightly. This can be explained by the issues which occur when spatially similar objects are involved, see Section 5.3.3.

Algorithm	recall	precision	fscore	median time	
Original	0.91	0.91	0.91	9.4	s
Change Detection	0.95	0.95	0.95	2.9	s

Table 6.5: Evaluation on *Change* dataset.

6.4 Evaluation of Demonstration Dataset

In this section we are going to evaluate the recognizer for certain model database and set of scenes. These models are going to be used for grasping purposes during demonstrations of the whole pipeline. The demonstration includes object recognition, synchronizing the camera coordinate system and robot coordinate system and finally physically grasping the object. The objects were

placed on a table within the reach of a robotic arm, which autonomously can grasp recognized objects. The tested model database is shown in Figure 6.4. Examples of the demonstration scenes set is given in Figure 6.5.

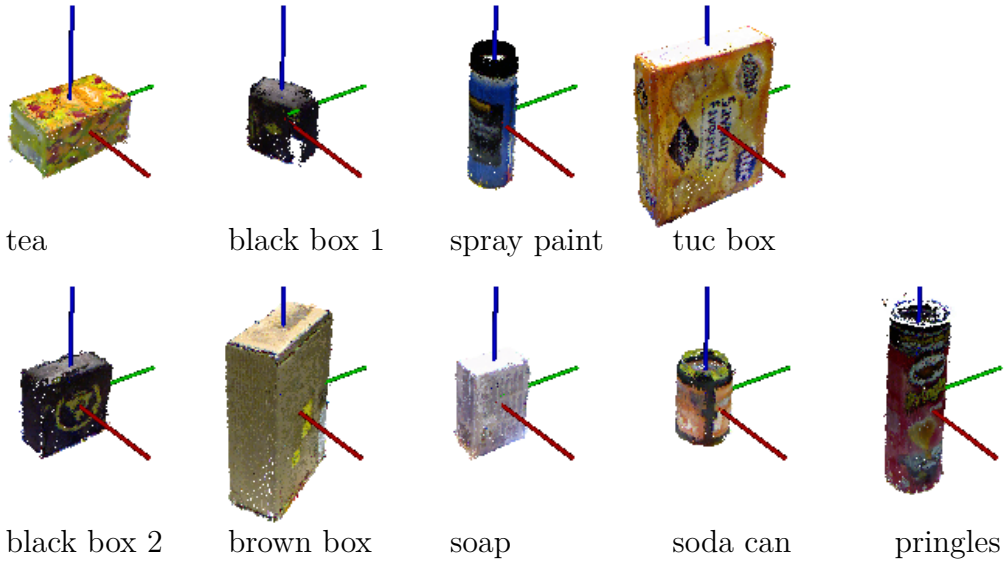


Figure 6.4: Objects tested for the demonstration.

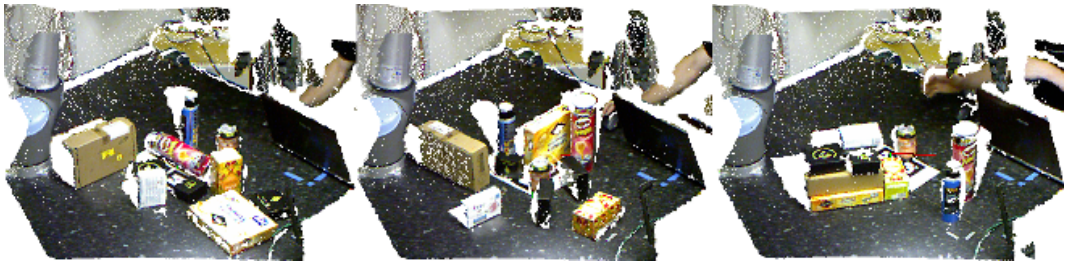


Figure 6.5: Examples of the demonstration dataset. Objects from the demonstration dataset were placed on a table in reach of a robotic arm.

If an object is recognized and the robot tries to grasp it, the position and location information need to be correct. Otherwise, the robotic arm may cause physical damage due to collision with falsely recognized objects or the table plane. Therefore, we chose a parameter set which ensures a low false positive rate and a high precession. An extract of the used parameters is given in Table 6.6. Since calculation time was not restricted for the demo we chose a high value for knn, which increases the number of generated hypotheses. On the other hand, to maintain a low false positive rate we chose a small resolution

parameter set	knn	resolution in mm	remove planes	inlier threshold
demo set	5	5	1	30000

Table 6.6: Extract of the demo parameter set.

for hypotheses verification. The table plane had some texture and therefore added many interest points to the scene. This caused a high false positive rate. To solve this problem, we removed planes from the scene to filter all these unnecessary interest points. We chose a quite high inlier threshold for removing planes to ensure only the dominant table plane is filtered. The results of the evaluation is shown in Table 6.7.

As expected some objects with little texture are difficult to recognize. For example the two small black boxes or the brown box shown in Figure 6.4. Also, the large yellow box was difficult to recognize, even if it appears to have sufficient texture. The surface of the box is quite shiny, which led to bright reflections due to the given illumination situation (Figure 6.5, left). The Pringles and the small soda can were easy to recognize and therefore good candidates for initial demonstrations.

parameter set	recall	precision	fscore	median time	
demo set	0.62	0.93	0.75	13.2	s

Table 6.7: Evaluation of demo set.

7 Outlook

The implemented single-view object recognition approach provides useful results for textured objects, if the parameters are chosen wisely. But even used in optimal condition the possible outcome is limited.

First of all a good parameter set for the recognizer needs to be found. Even if a feature was introduced, which simplifies this process, certain parameter sets need to be predefined. A calibration tool could be implemented, which finds a parameter set for certain test data automatically. Since many parameters are involved, correlations between single parameters and recognition results could be determined to achieve feasible calculation times.

In general for grasping tasks the recognizer knows which object needs to be found beforehand. Therefore, the parameter set and the recognition algorithm could be optimized to find one single object in the scene.

One efficient general purpose approach to improve the results is to up-great to a multi-view recognition system. Multiple RGBD cameras could be positioned around the scene, which allows to get a complete view on targeted objects. Moreover, a camera could be mounted on the robotic arm directly, which could be used to verify the proposed object location during the grasping process.

Finally, one highly restricting factor is the sensor itself. The RGBD cameras available nowadays provide relatively low resolution images with noisy depth data. If in near future better sensors are released, recognition results are going to be more accurate. Especially methods relying on local geometrical features, like SHOT, could benefit significantly.

8 Conclusion

In this thesis a state-of-the-art object modeling and object recognition system was integrated into an existing robotic system. The whole setup consists of a RGBD camera and a robotic arm which can fulfill a "smart grasping" task. My work is the final peace to make the pipeline from object recognition to object grasping completely autonomous. Now, providing the location of objects is not necessary anymore.

Also, the object modeling tool was modified to make it compatible to the grasping system. Moreover, the interface were simplified to ensure an easy usability of the tool. The recognition system was integrated into ROS and Blockly. This block based graphical programming language allows connecting different custom blocks, for example a recognition block and a grasping block. This provides a delicate communication interface between these two systems. The recognition block passes on a list of recognized objects with corresponding location information to the grasping block. Throughout the process the recognition results are visualized, which offers a convenient way for the user to judge the accuracy of the results subjectively. The object recognition system was adjusted in order to fulfill the requirements of the robotic grasping system. Since up to 80 parameters are available to adjust the recognizer, a parameter mapping feature was introduced. The user simply chooses between predefined settings which give the recognizer certain properties, like low calculation effort or high accuracy. The setup, which includes a RGBD camera mounted on a fixed position, gave the opportunity to optimize the recognition algorithm to this situation. The algorithm "Initialize Empty Work space" was proposed. Before the recognition starts, the empty work space of the scene is captured, and filtered afterwards during recognition tasks. Since fewer data need to be processed, the calculation effort decreases. The algorithm "Object Change Detection" takes the idea of filtering the empty work space one step further. Additionally, information of previous recognition tasks are used to reduce calculation effort. Finally, the proposed algorithms were evaluated and tested on the robotic system.

A Manual

The following sections give a short manual, how the proposed modeling and recognition framework can be used. We need a Kinect and for simplifying the modeling process a turntable, as shown in Figure A.1 is recommended. Moreover, we need to install the V4R Toolbox, the sr_vision ROS Package and the Blockly web server on our Ubuntu 16.04 computer.

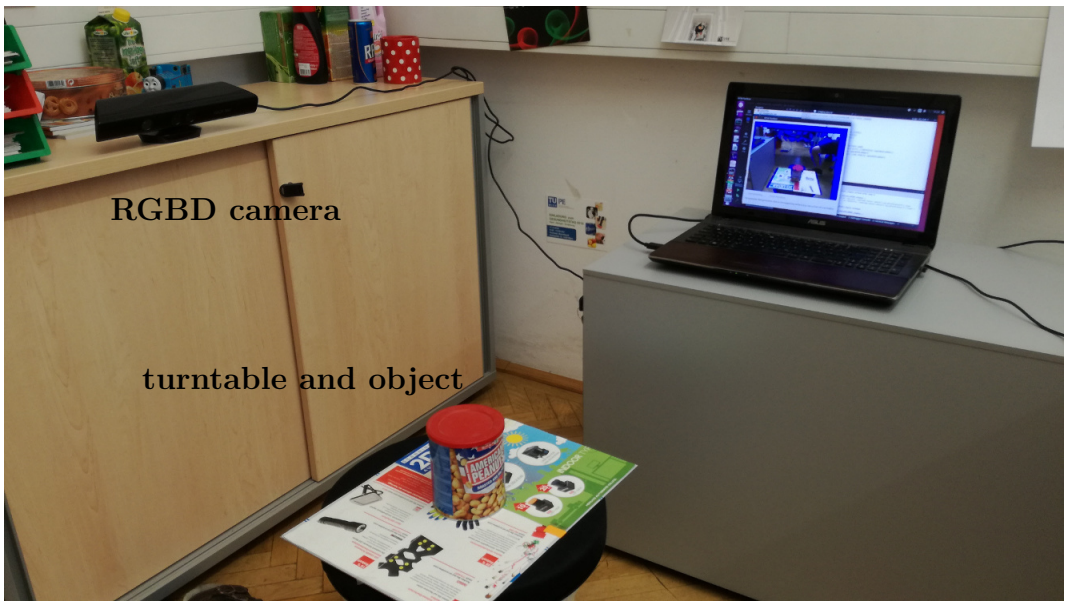


Figure A.1: V4R object modeling setup.

A.1 Object Modeling

First we have to model all the objects we want to recognize later. We use an offline tool for this, the RTM Toolbox [1].

Place the object on a flat surface on a newspaper or something similar. This allows us to rotate the object without touching it. The texture on the

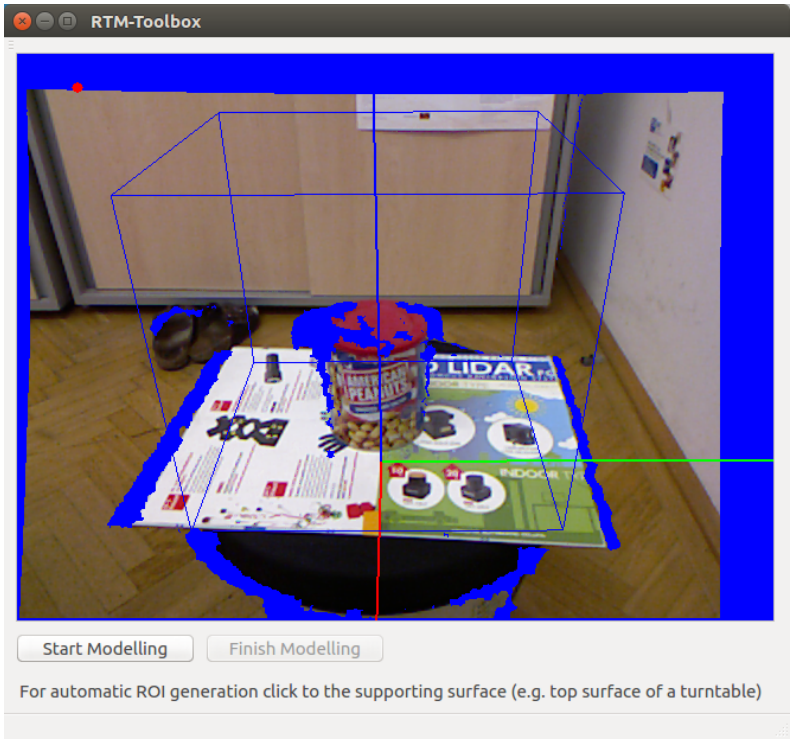
[1]The manual for the RTM Tool is based on the tutorial given in here: https://github.com/strands-project/lamor15/wiki/Tutorial-materials-3

newspaper also helps to keep track of the orientation of the object. The pictures below were taken with the object on a turn table, which is the most convenient way of rotating the object.

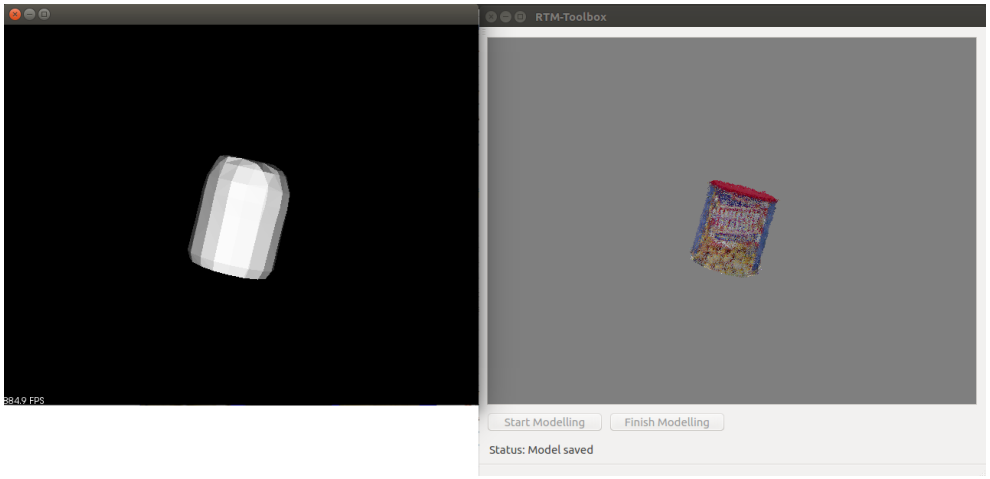
Start the modeling tool:

```
~/somewhere/v4r/bin/RTMT_simple
```

- Click on the flat surface next to the object to generate a ROI (region of interest). The tool considers everything inside the blue grid box as part of the model.



- Rotate 360 degrees, the program will generate a number of key frames. IMPORTANT: Do not violate the ROI (e.g. with our hand) during the modeling process.
- Press "Finnish modeling" (be patient! All post-processing is done after clicking this button.)
- Finally, choose a model name and store the created point cloud and mesh files.



Configuration options:

- Set data folder and model name:
  (File -> Preferences -> Settings -> Path and model name)
- Configure number of key frames to be selected using a camera rotation and a camera translation threshold:
  (File -> Preferences -> Settings -> Min. delta angle, Min. delta camera distance)
- Configure the level of detail of the mesh model:
  (File -> Preferences -> Postprocessing -> Poisson depth)

A.2 Object Recognition

After all objects are stored in the model database, the actual recognition stage can start. The recognizer itself is implemented in a ROS action server. The action client is implemented as a Blockly block.

Configuration Options:

Before starting the recognizer, some ROS launch parameters may be set. All parameter have default values and it is not necessary to set values manually. Path to the launch file:

```
~/somewhere/sr_recognizer/launch/recognition_demo.launch
```

(m) We can choose a different path to our model database.
(t) If we additionally specify a path to a point cloud file the recognizer uses

this file instead of acquiring a point cloud from the Kinect.
(cfg) We can choose a different path to our configuration files folder.
(recParam) We can choose a parameter set which changes the accuracy and calculation effort of the recognizer: 1: low calculation effort, but inaccurate results, 2: balanced between calculation effort and accuracy. 3: high calculation effort, but high accuracy of the results.
(arg) additional parameters for the recognizer can be chosen, mainly used for testing/debugging purposes.

Starting the recognizer:

```
$ roslaunch sr_recognizer recognition_demo.launch
```

The launch file runs all necessary ROS nodes, including the Blockly web server. Open this link in a web browser.
http://localhost:8000/pages/blockly.html

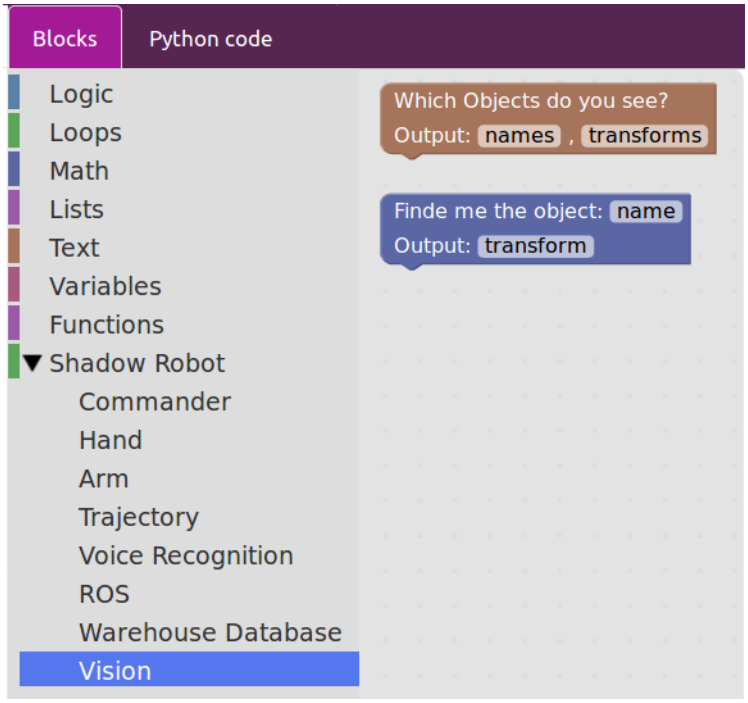


Figure A.2: Blockly Interface.

We should see the Blockly interface. Click on the toolbox on the left side, choose Vision and drag a vision block into the editor (see Figure A.2). Launch the

Blockly code and wait till the recognizer has finished the calculations. Finally, the visualization window shows the object recognition results.

B Recognition System Parameters

The following sections give an overview of the data type and functionality of parameters which can be found in the configuration files of the recognizer. These explanations are just a short summary to give an idea how certain parameters influence the results of the recognizer. The descriptions are mainly summaries from comments in the source code of this PhD thesis [8].

B.1 multipipeline_config.xml

float cg_size_: size for correspondence grouping.
int cg_thresh_: threshold for correspondence grouping. The lower the more hypotheses are generated, the higher the more confident and accurate. Minimum 3.
bool use_graph_based_gc_grouping_: if true, uses graph-based geometric consistency grouping

B.2 sift_config.xml /shot_config.xml

parameters for feature matching:
int kdtree_splits_: kdtree splits
int kdtree_num_trees_: number of trees for FLANN approximate nearest neighbor search
size_t knn_: nearest neighbors to search for when checking feature descriptions of the scene
float max_descriptor_distance_: maximum distance of the descriptor in the respective norm (L1 or L2) to create a correspondence
float correspondence_distance_weight_: weight factor for correspondences distances This is done to favor correspondences from different pipelines that are more reliable than others (SIFT and SHOT correspondences simultaneously fed into CG)
int distance_metric_: defines the norm used for feature matching (1... L1 norm, 2... L2 norm, 3... ChiSquare, 4... Hellinger)

float max_keypoint_distance_z_: Maximum distance of an extracted key point to be accepted

parameters for plane filter:
bool filter_planar_: Filters key points with a planar surface
int min_plane_size_: Minimum number of points for a plane to be checked if filter only points above table plane
int planar_computation_method_: Defines the method used to check for planar points. 0... based on curvature value after normalestimationomp, 1... with eigenvalue check of scatter matrix
float planar_support_radius_: Radius used to check key points for planarity.
float threshold_planar_: Threshold ratio used for deciding if patch is planar. Ratio defined as largest eigenvalue to all others.

parameters for depth-discontinuity filter:
int filter_border_pts_: Filters key points at the boundary (value according to the edge types defined in pcl::OrganizedEdgeBase:
EDGELABEL_NAN_BOUNDARY = 1, EDGELABEL_OCCLUDING = 2, EDGELABEL_OCCLUDED = 4.
int boundary_width_: Width in pixel of the depth discontinuity
float required_viewpoint_change_deg_: required viewpoint change in degree for a new training view to be used for feature extraction. Training views will be sorted incrementally by their file name and if the camera pose of a training view is close to the camera pose of an already existing training view, it will be discarded for training.
bool train_on_individual_views_: if true, extracts features from each view of the object model. Otherwise, will use the full 3D cloud

B.3 hv_config.xml

int resolution_mm_: The resolution of models and scene used to verify hypotheses (in millimeters)
float inliers_threshold_: inlier distance in meters between model and scene point
float inliers_surface_angle_thres_: inlier distance in radiant between model and scene surface normal
float occlusion_thres_: Threshold for a point to be considered occluded when model points are back-projected to the scene (depends e.g. on sensor noise)

int smoothing_radius_: radius in pixel used for smoothing the visible image mask of an object hypotheses (used for computing pairwise intersection)
bool do_smoothing_: if true, smoothes the silhouette of the re-project object hypotheses (used for computing pairwise intersection)
bool do_erosion_: if true, performs erosion on the silhouette of the re-project object hypotheses. This should avoid a pairwise cost for touching objects (used for computing pairwise intersection)
int erosion_radius_: erosion radius in px (used for computing pairwise intersection)
int icp_iterations_: number of icp iterations for pose refinement
float w_normals_: weighting factor for normal fitness
float w_color_: weighting factor for color fitness
float w_3D_: weighting factor for 3D fitness
float color_sigma_l_: allowed illumination (L channel of LAB color space) variance for a point of an object hypotheses to be considered explained by a corresponding scene point (between 0 and 1, the higher the fewer objects get rejected)
float color_sigma_ab_: allowed chrominance (AB channel of LAB color space) variance for a point of an object hypotheses to be considered explained by a corresponding scene point (between 0 and 1, the higher the fewer objects get rejected)
float sigma_normals_deg_: variance for normals between model and scene
float regularizer_: represents a penalty multiplier for model outliers. In particular, each model outlier associated with an active hypothesis increases the global cost function.
int normal_method_: method used for computing the normals of the down sampled scene point cloud (defined by the V4R Library)
bool ignore_color_even_if_exists_: if true, only checks 3D Euclidean distance of neighboring points
int max_iterations_: max iterations the optimization strategy explores local neighborhoods before stopping because the cost does not decrease.
float clutter_regularizer_: The penalty multiplier used to penalize unexplained scene points within the clutter influence radius
radius_neighborhood_clutter_ of an explained scene point when they belong to the same smooth segment.
bool use_replace_moves_: parameter for optimization. If true, local search uses replace moves (deactivates one hypothesis and activates another one). Otherwise, it only searches locally by enabling/disabling hypotheses one at a time.
int opt_type_: defines the optimization method. 0: Local search (converges quickly, but can easily get trapped in local minima), 1: Tabu Search, 2:

Tabu Search + Local Search (Replace active hypotheses moves), 3: Simulated Annealing
bool use_histogram_specification_: if true, tries to globally match brightness (L channel of LAB color space) of visible hypothesis cloud to brightness of nearby scene points. It does so by computing the L channel histograms for both clouds and shifting it to maximize histogram intersection.
bool initial_status_: sets the initial activation status of each hypothesis to this value before starting optimization. E.g. If true, all hypotheses will be active and the cost will be optimized from that initial status.
int color_comparison_method_: method used for color comparison (0... CIE76, 1... CIE94, 2... CIEDE2000)
float min_visible_ratio_: defines how much of the object has to be visible in order to be included in the verification stage
bool check_smooth_clusters_: Euclidean smooth segmentation. Ff true, checks if hypotheses explain whole smooth regions of input cloud (if they only partially explain one smooth region, the solution is rejected)
float eps_angle_threshold_deg_: angle threshold in degree to cluster two neighboring points together
float curvature_threshold_: curvature threshold to allow clustering of two points (points with surface curvatures higher than this threshold are skipped)
float cluster_tolerance_: cluster tolerance in meters for point to be clustered together
int min_points_: minimum number of points for a smooth region to be extracted
float min_ratio_cluster_explained_: defines the minimum ratio a smooth cluster has to be explained by the visible points (given there are at least 100 points)
bool z_adaptive_: if true, scales the smooth segmentation parameters linear with distance (constant till 1m at the given parameters)
size_t min_pts_smooth_cluster_to_be_explained_: minimum number of points a cluster need to be explained by model points to be considered for a check (avoids the fact that boundary points of a smooth region can be close to an object)
float min_fitness_: hypotheses which have a lower fitness score will be defined as "outlier"
float min_dotproduct_model_normal_to_viewray_: surfaces which point are oriented away from the view ray will be discarded if the absolute dot product between the surface normal and the view ray is smaller than this threshold. This should ignore points for further fitness check which are very sensitive to small rotation changes.

float min_px_distance_to_image_boundary_: minimum distance in pixel a re-projected point needs to have to the image boundary

B.4 esf_config.xml

bool check_elongations_: if true, checks if the elongation of the segmented cluster fits approximately the elongation of the matched object hypothesis
float max_elongation_ratio_: if the elongation of the segment w.r.t. to the matched hypotheses is above this threshold, it will be rejected (used only if check_elongations_ is true.
float min_elongation_ratio_: if the elongation of the segment w.r.t. to the matched hypotheses is below this threshold, it will be rejected (used only if check_elongations_ is true).
bool use_table_plane_for_alignment_: if true, aligns the matched object model such that the centroid corresponds to the centroid of the segmented cluster downprojected onto the found table plane. The z-axis corresponds to the normal axis of the table plane and the remaining axis build an orthonormal system. Rotation is then sampled in equidistant angles around the z-axis. ATTENTION: This assumes the models are in a coordinate system with the z-axis aligning with the typical upright position of the object.
float z_angle_sampling_density_degree_: if use_table_plane_for_alignment_, this value will generate object hypotheses at each multiple of this value.
float required_viewpoint_change_deg_: required viewpoint change in degree for a new training view to be used for feature extraction. Training views will be sorted incrementally by their file name and if the camera pose of a training view is close to the camera pose of an already existing training view, it will be discarded for training.
bool estimate_pose_: if true, tries to estimate a coarse pose of the object based on the other parameters
bool classify_instances_: if true, classifier learns to distinguish between model instances instead of categories

Bibliography

[1] D. Fischinger, *Enabling autonomous robotic grasping based on topographic features*, Parallelt. [Ãœbers. des Autors] Enabling Autonomous Robotic Grasping based on Topographic Features; Wien, Techn. Univ., Diss., 2014, 2014.

[2] J. Prankl, A. Aldoma, A. Svejda, and M. Vincze, „Rgb-d object modelling for object recognition and tracking," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2015, pp. 96–103.

[3] J. Prankl, T. Mörwald, M. Zillich, and M. Vincze, „Probabilistic cue integration for real-time object pose tracking," in *Computer Vision Systems: 9th International Conference, ICVS 2013, St. Petersburg, Russia, July 16-18, 2013. Proceedings*, M. Chen, B. Leibe, and B. Neumann, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 254–263, ISBN: 978-3-642-39402-7. [Online]. Available: `http://dx.doi.org/10.1007/978-3-642-39402-7_26`.

[4] B. Calli, A. Walsman, A. Singh, S. Srinivasa, P. Abbeel, A. Dollar, „Benchmarking in manipulation research," *IEEE Robotics and Automation Magazine*, vol. 22, no. 3, pp. 36–52, 2015.

[5] ——, „The ycb object and model set towards common benchmarks for manipulation research," in *International Conference on Advanced Robotics (ICAR)*, 2015.

[6] T. Weise, T. Wismer, B. Leibe, and L. V. Gool, „In-hand scanning with online loop closure," in *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops*, 2009, pp. 1630–1637.

[7] M. M. Torres, A. C. Romea, and S. Srinivasa, „Moped: A scalable and low latency object recognition and pose estimation system," in *Proceedings of ICRA 2010*, Pittsburgh, PA, 2010.

[8] T. Fäulhammer, M. Zillich, J. Prankl, and M. Vincze, „A multi-modal rgb-d object recognizer," in *Pattern Recognition (ICPR), 2016 23rd International Conference on*, IEEE, 2016, pp. 733–738.

[9] W. Wohlkinger and M. Vincze, „Ensemble of shape functions for 3d object classification,“ in *2011 IEEE International Conference on Robotics and Biomimetics*, 2011, pp. 2987–2992.

[10] D. G. Lowe, „Distinctive image features from scale-invariant keypoints,“ *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[11] D. Lowe, *Method and apparatus for identifying scale invariant features in an image and use of same for locating an object in an image*, US Patent 6,711,293, 2004. [Online]. Available: `https://www.google.com/patents/US6711293`.

[12] D. G. Lowe, „Object recognition from local scale-invariant features,“ in *International Conference on Computer Vision, 1999*, IEEE, 1999, pp. 1150–1157.

[13] L. Assirati N. Silva L. Berton A. Lopes O.Bruno, „Performing edge detection by difference of gaussians using q-gaussian kernels,“ in *2nd International Conference on Mathematical Modeling in Physical Sciences*, 2013.

[14] F. Tombari, S. Salti, L. Di Stefano, „Unique signatures of histograms for local surface description,“ in *11th European Conference on Computer Vision (ECCV)*, 2010.

[15] J. Beis, D. Lowe, „Shape indexing using approximate nearest-neighbour search in high-dimensional spaces,“ in *Conference on Computer Vision and Pattern Recognition*, 1997, pp. 1000–1006.

[16] F. Tombari and L. D. Stefano, „Object recognition in 3d scenes with occlusions and clutter by hough voting,“ in *2010 Fourth Pacific-Rim Symposium on Image and Video Technology*, 2010, pp. 349–355.

[17] H. Chen and B. Bhanu, „3d free-form object recognition in range images using local surface patches,“ in *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004.*, vol. 3, 2004, 136–139 Vol.3.

[18] A. Aldoma Buchaca, F. Tombari, J. Prankl, A. Richtsfeld, L. di Stefano, M. Vincze, „Multimodal cue integration through hypotheses verification for rgb-d object recognition and 6dof pose estimation,“ in *IEEE International Conference on Robotics and Automation (ICRA)*, 2013.

[19] A. Aldoma, F. Tombari, L. Di Stefano, and M. Vincze, „A global hypotheses verification method for 3d object recognition," in *Computer Vision – ECCV 2012: 12th European Conference on Computer Vision, Florence, Italy, October 7-13, 2012, Proceedings, Part III*, A. Fitzgibbon, S. Lazebnik, P. Perona, Y. Sato, and C. Schmid, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 511–524, ISBN: 978-3-642-33712-3. [Online]. Available: `https://doi.org/10.1007/978-3-642-33712-3_37`.

[20] J. Schmidhuber, „Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85 –117, 2015, ISSN: 0893-6080. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/S0893608014002135`.

[21] A. Kumar, „Artificial neural networks," 2012.

[22] Z. Zhang, in *Microsoft Kinect sensor and its effect*. 19th ed. IEEE MultiMedia, 2012, pp. 4–10.

[23] H. Sarbolandi, D. Lefloch, and A. Kolb, „Kinect range sensing: Structured-light versus time-of-flight kinect," May 2015.

[24] I. Anwar and S. Lee, „High performance stand-alone structured light 3d camera for smart manipulators," in *2017 14th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, 2017, pp. 192–195.

[25] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon, „Bundle adjustmentâ€"a modern synthesis," in *International workshop on vision algorithms*, Springer, 1999, pp. 298–372.

[26] C. Wu, S. Agarwal, B. Curless, and S. M. Seitz, „Multicore bundle adjustment," in *CVPR 2011*, 2011, pp. 3057–3064.

[27] J. Nocedal and S. J. Wright, *Numerical Optimization*, 2nd. New York: Springer, 2006.

[28] P. J. Besl, N. D. McKay, *et al.*, „A method for registration of 3-d shapes,"

[29] M. B. M. Kazhdan and H. Hoppe, „Poisson surface reconstruction," in *Proceedings of the 4th Eurographics Symposium on Geometry Processing*, 2006, pp. 61–70.

[30] T. Fäulhammer, „From the lab to the wild: learning and recognizing objects in cluttered environments on a mobile robot," PhD thesis, TU Wien (TUW), 2017.

[31] M. Vincze, M. Zillich, D. Wolf, „Machine vision and cognitive robotics," in *Lecture on Machine Vision and Cognitive Robotics*, 2015.

[32] K. Khoshelham, S. Elberink, „Accuracy and resolution of kinect depth data for indoor mapping applications,“ *Sensors*, no. 12, pp. 1437–1454, 2012.

[33] K. Khoshelham and S. O. Elberink, „Accuracy and resolution of kinect depth data for indoor mapping applications,“ *Sensors*, vol. 12, no. 2, pp. 1437–1454, 2012, ISSN: 1424-8220. [Online]. Available: `http://www.mdpi.com/1424-8220/12/2/1437`.

[34] T. Rabbania, F. A. van den Heuvelb, and G. Vosselmanc, „Segmentation of point clouds using smoothness constraint,“ 2006.

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Wien, September 26, 2017

Thomas Muttenthaler