Contents

I Introduction **1**

1 Abstract **2**

2 Purpose of This Document **3**

3 Setup **4**

4 ROS Overview **5**

II Object Recognition **9**

5 Preliminary Considerations **10**
5.1 Target Groups 11
5.2 Message Types for Detected Objects 12
5.2.1 2D Message Type 12
5.2.2 3D Message Type 13
5.3 Parametrization and Parameter Files 14

6 Object Recognition Algorithms **16**
6.1 HSV Detection 16
6.2 Feature Detection 18
6.3 Shape Detection 20

7 Individual Programs **22**
7.1 HSV Detection 23
7.2 Feature Detection 25
7.3 Shape Detection 27

8 Library Solution **29**
8.1 Classes and their Relationships 30
8.1.1 Processing data and detecting objects 30
8.1.2 Monitoring and Setting Parameters 31
8.1.3 Saving and Loading Parameters 35
8.2 Nodes 37
8.2.1 Creating Parameter Files 38
8.2.2 Publishing Detected Objects 41
8.3 Layers and Packages 42
8.4 Implemented Object Recognition Systems 44
8.4.1 HSV Detection 44
8.4.2 Feature Detection 46
8.4.3 Shape Detection 47
8.5 2D and 3D Specifics 47
8.5.1 Saving and Loading Parameters 47
8.5.2 Visualization 51

9 ROS-based Solution **54**
9.1 Fundamental Ideas 54
9.2 Filter and Detector Nodes 56
9.3 Parameter Management 58
9.3.1 Alternatives 58
9.3.2 The Chosen Solution 59
9.3.3 A Minimal Example 60
9.3.4 Using the System 63
9.3.5 Encapsulating the Parameter Management within Nodes 64
9.4 2D and 3D Modules 66
9.4.1 2D Modules 67
9.4.2 3D Modules 68
9.5 Implemented Object Recognition Systems 68
9.5.1 HSV Detection 69
9.5.2 Feature Detection 70
9.5.3 Shape Detection 74

10 Evaluation of the Solutions **77**

III Navigation **80**

11 Experimental Setup **81**

12 Mapping **82**
12.1 Creating the Map 82
12.2 Viewing and Editing the Map 83

13 Simulation and Determining Goal Poses **85**
13.0.1 Preparing the Map 85
13.0.2 Creating the World File 85
13.0.3 Launching the Simulation and Navigating the Robot . 86
13.0.4 Determining Goal Poses 86

14 The Navigation Node **88**
14.1 Navigating in Code 88
14.2 State Machines and Smach 89
14.3 The Node 89
14.4 Launching the Node 92

IV Conclusion **95**

15 Summary **96**

16 Outlook **98**

A PCL Cylinder Segmentation Tutorial **99**

B `MouseEvent` Message Definition **104**

C Simulation World File **105**

D Launch File for Simulation **107**

E File of Goal Poses **108**

Bibliography **110**

List of Figures

4.1 ROS graph of a system that comprises four nodes and two topics 6
4.2 ROS graph of the system shown in figure 4.1 after mapping the name `topic2` to `topic3` on start-up of `publisher2` 6

5.1 Graph of a modular object recognition system 11

6.1 Image before (left) and after (right) applying the `cv::inRange` function to it 17
6.2 Top: Original image with detected objects. Bottom left: Result of applying thresholds to the image in HSV color space. Bottom right: Result of additional opening transformation. . . 18

7.1 User interface of the `hsv_definer` node; the button for opening the properties window is marked with a red ellipse 24
7.2 User interface of the `feature_definer` node 26
7.3 User interface of the `feature_definer` node after selecting a target 27

8.1 Class diagram [Boo07, chapter 5.7] of classes that participate in the creation of parameter files (created with Dia [McC17]) . 30
8.2 Class diagram of classes that enable the user to monitor and set parameters 32
8.3 Sequence diagram [Boo07, chapter 5.8] (created with Dia [McC17]) of the interactions that are triggered by a user who changes the value of a parameter in the GUI (compare with [Gam95, p. 295]) 33
8.4 Diagram of classes that are involved in the creation of parameter files 35

8.5 GUI for creating parameter files for the shape detection system. The original point cloud is rendered in white, the filtered point cloud is rendered in red. The detected object is a cylindrical garbage can. 40
8.6 Seperation of classes and nodes into layers and packages . . . 43
8.7 GUI for creating parameter files for the HSV detection system 45

9.1 ROS graph of a general object recognition system 55
9.2 Classes that are used in filter and detector nodes for 2D (top) and 3D (bottom) systems 57
9.3 `rqt_reconfigure` GUI for managing the parameters of the `my_module` node 63
9.4 ROS graph of the HSV detection system 69
9.5 `rviz` and `rqt_reconfigure` provide the interface to the HSV detection system 70
9.6 ROS graph of the feature detection system 71
9.7 `rqt_reconfigure` GUI for managing the parameters of the `feature_detector` node 73
9.8 ROS graph of the shape detection system 74
9.9 `rviz` and `rqt_reconfigure` provide the interface to the shape detection system. The detected object is a cylindrical garbage can on an office chair. 76

12.1 Map of the test environment 84

13.1 Stage simulation (left) and defining a goal pose in rviz (right) 87

14.1 State machine of the simulation program 90

List of Tables

10.1 Pros (+) and cons (-) of each solution 78

List of Acronyms and Abbreviations

API	Application programming interface
GUI	Graphical user interface
HSV	Hue, saturation, value
ID	Identifier
MVC	Model-View-Controller
PCL	Point Cloud Library
RANSAC	Random sample consensus
RGB	Red, green, blue
ROS	Robot Operating System
RPC	Remote procedure call
ssh	Secure Shell
SURF	Speeded-Up Robust Features
TCP	Transmission Control Protocol
UI	User interface
URI	Uniform Resource Identifier
XML	Extensible Markup Language

Part I

Introduction

Chapter 1

Abstract

Recently, the faculty for mechanical engineering of the Westphalian University of Applied Sciences aquired a TurtleBot2, a mobile robot equipped with a 3D sensor. This robot is meant to be used as a platform for learning to implement object recognition algorithms and to develop applications that make use of these algorithms.

To facilitate the implementation of these algorithms and their usage in robot software, three solutions for a modular framework for 2D and 3D object recognition systems were developed. Part II of this document presents and evaluates these solutions and gives recommendations for further steps.

Part III presents a basic program for autonomous navigation, which can be adapted to specific tasks.

Chapter 2

Purpose of This Document

The supreme purpose of this document is to enable the reader to explore the described solutions on his own and to use, extend, and modify them.

To this end, it describes the fundamental ideas, the implementation, and the usage of each solution. The descriptions focus on the solutions in their current states. Only if it improves the understanding of an actual solution are considered alternatives mentioned.

The descriptions are not complete; it is assumed that the reader has access to the source code and is able to compile and run the programs.

Chapter 3

Setup

The mobile robot that is used in this project is a TurtleBot2 [Turtle17]. It consists of a Kobuki differential-drive mobile base [Yujin17] and an Astra 3D sensor [Orb17], which delivers 2D color images and 3D point clouds. The notebook that controls the TurtleBot runs Ubuntu 16.04 [Ubu17] and ROS Kinetic [ROS17a].

Chapter 4

ROS Overview

ROS, the Robot Operating System, is an open-source framework for developing robot software. It provides tools, libraries, and conventions that facilitate this task.

This chapter provides the information that is absolutely necessary to understand this document. The book "Programming Robots with ROS" [Qui15] is an excellent introduction to ROS. The ROS wiki [ROS17b] contains the documentation of all ROS features.

ROS Graph

A *ROS system* is made up of multiple simultaneously running processes, called *ROS nodes*. ROS nodes communicate with each other, mainly by using another concept called *ROS topics*. One or multiple nodes can publish *messages* on a topic. Other nodes can receive these messages by subscribing to the topic. A *ROS graph* renders these correlations for a specific ROS system.

As an example, figure 4.1 shows the ROS graph of a simple system. The two nodes `publisher1` and `publisher2` publish messages on the topic called `topic1`, to which the node `subscriber1` subscribes. `publisher2` also publishes messages on the topic `topic2`, to which `subscriber1` and `subscriber2` subscribe.

All messages that are published on and expected from a specific topic have to be of the same type. ROS provides many standard types—e.g. integers, strings, images, and point clouds. If these types are not sufficient, new types can be defined.

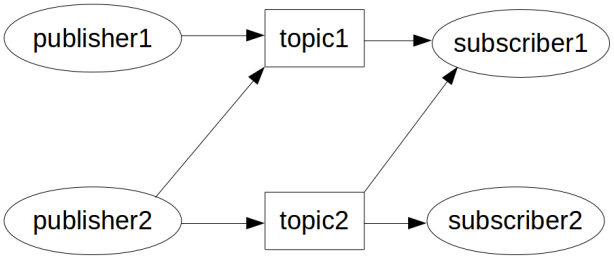

Figure 4.1: ROS graph of a system that comprises four nodes and two topics

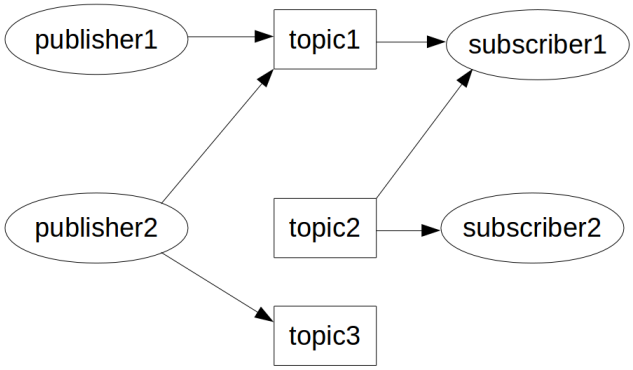

Figure 4.2: ROS graph of the system shown in figure 4.1 after mapping the name `topic2` to `topic3` on start-up of `publisher2`

An important feature of ROS is the ability to remap the names of topics when launching nodes. This allows to connect nodes that use the same message type, even if, according to their source code, they publish and subscribe to differently named topics. When, in the example above, the name `topic2` is mapped to `topic3` on start-up of node `publisher2`, the ROS graph of the system looks like figure 4.2.

Packages and Workspaces

ROS software is organized into *packages*. A package is a collection of resources—i.e. code, data, and documentation—that are built and distributed together.

New packages are created within a *workspace*. A workspace is a set of directories and contains a related set of ROS code. The complete code of this project resides in a single workspace, a folder called `turtlebot_ws`. Inside

this workspace is a folder called `src`, which contains the packages of this project, each one represented by a seperate folder.

Facilities for Launching ROS Systems

rosrun is a command-line tool for launching single ROS nodes. It is invoked with the name of the package that contains the node, the name of the executable that represents the node, and the command-line arguments that are passed to the node:

```
$ rosrun <package> <executable> [args]
```

Seperately launching every node of a large ROS system is cumbersome and error-prone. For this reason, ROS provides the ***roslaunch*** command-line tool. It is invoked with the name of a *launch file* and the name of the package that contains that file:

```
$ roslaunch <package> <launch-file>
```

Launch files are XML files that describe the nodes that are launched by `roslaunch`, together with their parameters (described later), remappings, and command-line arguments. Several examples of launch files are presented in this document.

Other Communication Mechanisms

Besides topics, ROS nodes use two main communication mechanisms.

ROS services are synchronous remote procedure calls. This means that nodes can call a function that executes in another node, the service provider. This function is executed whenever the service provider receives a service request.

ROS actions are asynchronous and build on top of ROS messages, i.e. implemented using topics. Actions are typically used to provide time-extended, goal-oriented behaviour. An action is triggered when a client sends a goal to the corresponding action server. While the server tries to reach the goal, it can send feedback to the client, and server as well as client can abort the action. Servers and clients react to goals, feedback, results, and other incoming messages by executing corresponding callback functions. In this project, actions are used for the navigation, described in part III.

The ROS Master

Every ROS system has a *ROS master*, which is a process with a known URI, determined by the environment variable `ROS_MASTER_URI`. Nodes register themselves with the master at start-up, and they inform the master when they publish or subscribe to a topic or when they advertise or call a service. The master provides each node with the relevant addresses of other nodes so that the nodes can establish peer-to-peer connections. The communication with the master and the negotiation of peer-to-peer connections are based on XML-RPC.[ROS17c] After such a peer-to-peer connection has been established, the connected nodes communicate directly with one another, normally via TCP, although other types of connections are possible.

Programming Languages

ROS nodes can be written in any programming language for which a so-called *client library* has been written. Client libraries for multiple languages exist, but the two best-supported APIs, which are also exclusively used in this project, are the ones for C++ and Python.

Part II

Object Recognition

Chapter 5

Preliminary Considerations

This chapter states general demands on the object recognition framework. It describes the framework in rather abstract terms. The subsequent chapters concretize these descriptions by presenting different implementations of the framework.

To make the discussion a bit more concrete, figure 5.1 shows an example of an object recognition system that has been implemented in accordance with the framework. It comprises five modules, each depicted as an ellipse. It is assumed that the system takes images as its input, which are generated by the drivers of a camera, depicted as another ellipse at the top of the figure. After being modified by the `image-preprocessor` module, the images are passed to the two detection modules, `detector1` and `detector2`. The outputs of these modules, i.e. the detected objects, are passed to the `object-counter` module, which counts the total number of detected objects and writes its results to disk. The `visualizer` module visualizes the original and the preprocessed images.

A key attribute of the framework is that similar modules have identical interfaces. In figure 5.1, both detection modules have inputs of the same type and outputs of the same type; this makes it possible to exchange one for the other or to add a third detection module to the system without having to modify other modules. Using the same interface also enables the user of the framework to write modules that rely on that interface. In figure 5.1, the `object-counter` module can operate on the outputs of any detection module, and the `visualizer` module can visualize the original and the preprocessed images.

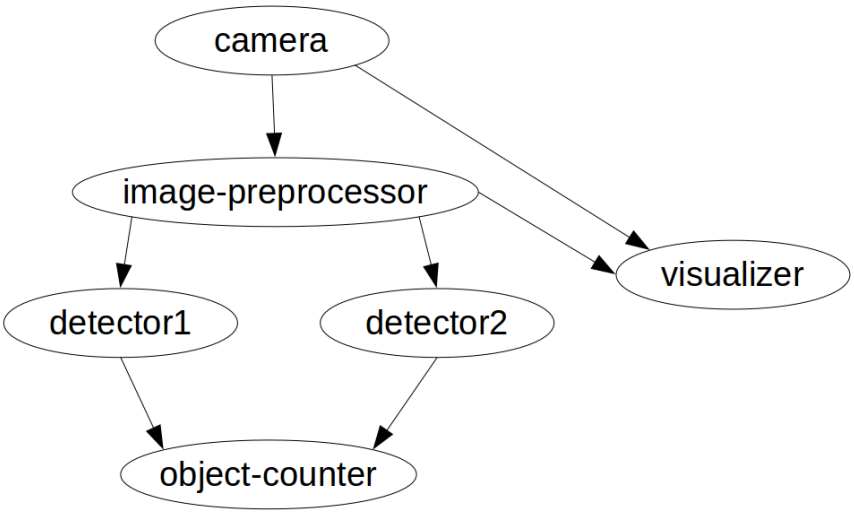

Figure 5.1: Graph of a modular object recognition system

5.1 Target Groups

The framework mainly tries to address the needs of two groups of people. Looking at their use cases reveals the major demands on the framework.

The *first* group consists of people who want to implement object recognition algorithms that are not already part of the framework. They should be able to implement an algorithm as a module that can be combined with existing modules, and they should not have to worry about the implementation of auxiliary tasks. Features that are needed by many object recognition systems or within many modules, e.g. the visualization of the inputs and outputs of each module, should be provided by the framework in the form of libraries and general-purpose modules. In addition, the integration of a new module should not require any modifications of the underlying framework and the existing modules; this can be considered as an application of the Open/Closed Principle [Wiki17a].

The *second* target group develops software that operates on the outputs of existing object recognition modules. This software could be a general-purpose module that implements some higher-level logic, e.g. counting the number of detected objects or visualizing them. It could also be robot software for performing a specific task, e.g. navigating around a test environment and checking the presence of a specific object; ideally, the developers of such

an application can choose from a large pool of existing object recognition modules and pick and combine those that best fit their problem. This target group relies on the common interface of the detection modules, which makes it possible to process the outputs of arbitrary detection modules.

5.2 Message Types for Detected Objects

The importance of uniform, well-defined interfaces for specific kinds of modules has already been pointed out. Obviously, detection modules have to use images, point clouds, or similar data as their input. They also have to publish the detected objects in some way. This section describes the message types that were defined for this purpose.

The message type that is used for detected objects has to be sufficiently general to be useable by a large number of detection modules. The type should allow to describe the pose and size of a detected object accurately enough for most applications. On the other hand, the description should be simple enough to be easy to use. To keep the network load low, the reprentation of objects also should not require an unnecessary amount of data.

The main difficulty in defining the message type is that different detection modules have different needs. Some detectors are able to determine the pose of a detected object within the input data, others are not. Some detectors can detect multiple objects in a single input message, others cannot. The most significant difference, however, is that some detectors operate on two-dimensional data and others on three-dimensional data. This makes it impossible to use the same interface for 2D and 3D detection modules and led to the creation of two independent message types.

5.2.1 2D Message Type

2D detection modules operate on color images. The message type that they use to publish detected objects can be printed with the `rosmsg show` command as shown in listing 5.1.

Listing 5.1: The `DetectedObject2DArray` message type from the `object_detection_2d_msgs` package

```
$ rosmsg show \
```

```
> object_detection_2d_msgs/DetectedObject2DArray
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
object_detection_2d_msgs/DetectedObject2D[] objects
  string name
  geometry_msgs/Polygon polygon
    geometry_msgs/Point32[] points
      float32 x
      float32 y
      float32 z
```

To enable detection modules to publish multiple detected objects in a single message, each message contains an array of detected objects. To make it possible to distinguish between multiple detected objects, each one is given a name. A polygon describes the outline of an object within the corresponding image. In general, it is recommended to use existing ROS message types where possible.[Qui15, p. 47] For this reason, the polygon is of type `geometry_msgs/Polygon`, which is capable of describing polygons in three-dimensional space. `x` and `y` represent the coordinates of a vertex of a polygon in the image; `z` is not used.

All objects of a message share a single header, which contains a sequence ID, a timestamp, and the name of the coordinate frame that is associated with the message. The standard message type for headers in ROS, `std_msgs/Header`, is used for that purpose.

5.2.2 3D Message Type

3D detection modules operate on point clouds. The message type that they use to publish detected objects is shown in listing 5.2.

Listing 5.2: The `DetectedObject3DArray` message type from the `object_detection_3d_msgs` package

```
$ rosmsg show \
> object_detection_3d_msgs/DetectedObject3DArray
std_msgs/Header header
  uint32 seq
  time stamp
```

```
  string frame_id
object_detection_3d_msgs/DetectedObject3D[] objects
  string name
  object_detection_3d_msgs/OrientedBox box
    geometry_msgs/Pose pose
      geometry_msgs/Point position
        float64 x
        float64 y
        float64 z
      geometry_msgs/Quaternion orientation
        float64 x
        float64 y
        float64 z
        float64 w
    float64 width
    float64 height
    float64 depth
```

The message type is very similar to the one in listing 5.1. The only difference is that the location of a detected object is represented by a rectangular prism. This prism is described by the pose of its center and by its dimensions, i.e. its width, height and depth.

5.3 Parametrization and Parameter Files

A fundamental idea that underlies the framework is that each specific object detection system detects objects based on a specific criterion, e.g. color or shape, and manages parameters whose values determine which specific objects are detected, e.g. red objects or cylindrical objects. Each system should also provide a user interface for monitoring, setting, saving, and loading the values of its parameters.

The user of the framework will typically first choose an object recognition system and set its parameters according to his problem. Then he will request the system to save the values of its parameters to one or multiple files, which we call "parameter files". The user should be able to launch the system later and pass it handles to these files in a convenient way. In particular, he should not have to know the internals of the system to set it up for the specific task at hand.

In summary, each object detection system should provide facilities for the following tasks:

- Monitoring and setting the parameters of the system.
- Providing feedback by visualizing the results of parameter changes.
- Creating parameter files.
- Passing handles to parameter files to the system at start-up.

Chapter 6

Object Recognition Algorithms

During the development of the framework, three specific object recognition systems were taken as examples. They represent use cases for the integration of systems into the framework, and their integration reveals major demands on the framework.

The first two systems operate on color images and mainly use facilities of the OpenCV library [OpenCV17]. The last system operates on point clouds and mainly uses facilities of the PCL library [PCL17a].

This chapter describes the underlying algorithm of each of the three systems without going into non-essential implementation details. The subsequent chapters present different solutions for creating a framework and describe the implementations of the systems for each solution.

6.1 HSV Detection

The HSV detection system detects objects with noticeable colors. The naive approach, filtering images on RGB values, is a poor way to find a particular color in an image, since the RGB values are strongly affected by the overall brightness.[Qui15, pp. 200–201] Therefore, the images are first transformed into HSV color space, meaning that the red, green, and blue values of each pixel are transformed into a corresponding set of hue, saturation, and value values. This is done with the `cv::cvtColor` function [Kae17, pp. 117–119].

The actual filtering is done with the `cv::inRange` function [Kae17, pp. 124–125], which takes a lower and an upper limit for each channel of the image as its arguments. These limits are parameters of the system and are called

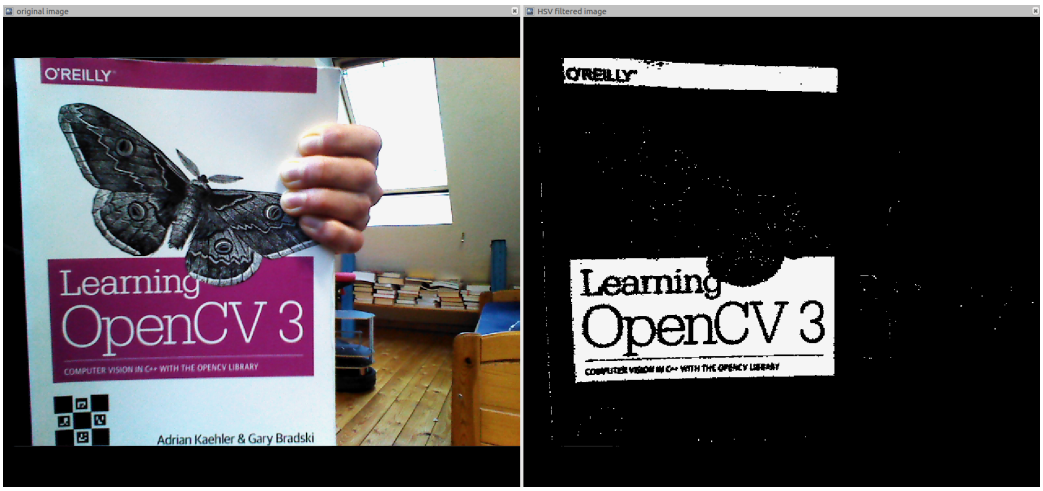

Figure 6.1: Image before (left) and after (right) applying the `cv::inRange` function to it

`h_min`, `h_max`, `s_min`, `s_max`, `v_min`, and `v_max`. Only pixels whose hue (`H`), saturation (`S`), and value (`V`) values fulfill the following conditions pass through the filter:

$$\begin{aligned} \text{h\_min} &<= \text{H} <= \text{h\_max} \\ \text{s\_min} &<= \text{S} <= \text{s\_max} \\ \text{v\_min} &<= \text{V} <= \text{v\_max} \end{aligned}$$

The result is a one channel image that contains the value 255 where these conditions are fulfilled and zero where they are not. As an example, figure 6.1 shows an image on the left and the result of applying the filter to it on the right.

Carefully inspecting the figure reveals that the result of the filter is not perfect, and adjusting the parameters reveals that it is hard to make the result much better under varying lightning conditions. To remove the white speckle noise in the image, a morphological transformation is applied to it; the *erosion* and *opening* transformations are suited for this task. The `cv::morphologyEx` function [Kae17, pp. 275–284] is capable of applying these transformations to the image. Parameters of this function, which should also be parameters of the object recognition system, are the kind of operation (erode, dilate, open, etc.) and the number of iterations.

The external contours of the detected objects, i.e. of the contiguous areas

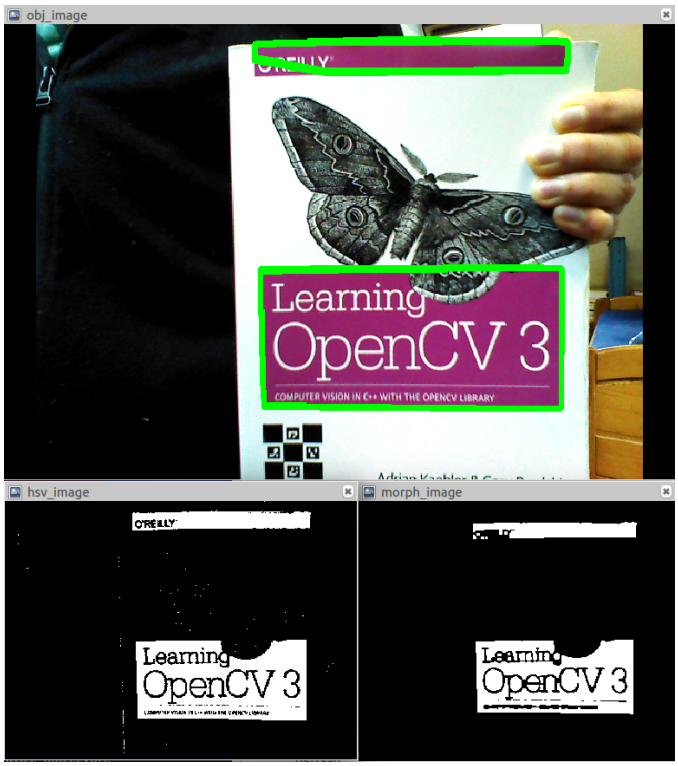

Figure 6.2: Top: Original image with detected objects. Bottom left: Result of applying thresholds to the image in HSV color space. Bottom right: Result of additional opening transformation.

of pixels with non-zero values, are computed with the `cv::findContours` function [Kae17, pp. 407–413]. Contours whose length does not lie within specific limits, which are additional parameters of the system, are filtered out. Lastly, the number of vertices is reduced by applying the `cv::convexHull` function [Kae17, pp. 426–428] to each contour.

Figure 6.2 shows a complete example.

6.2 Feature Detection

The feature detection system detects objects based on keypoints and descriptors. A *keypoint* is a small patch of an image that is rich in local information. A *descriptor* contains the descriptive information about a keypoint and can

be used to determine whether two keypoints are "the same". One major motivation for computing keypoints and descriptors is to represent an object in an invariant form that will be similar in similar views of the object.[Kae17, p. 493 and pp. 511–513]

In OpenCV, keypoints are represented by instances of the `cv::KeyPoint` class [Kae17, pp. 514–516]. Each keypoint contains the x- and y-coordinates of the patch that it represents. The abstract `cv::Feature2D` class provides member functions for computing keypoints and descriptors from a region of interest within an image.[Kae17, pp. 516–519] Many methods for doing this exist, and each of them is implemented by a concrete subclass of the `cv::Feature2D` class. The method that is used in this project is called SURF (Speeded-Up Robust Features) and is implemented by the `cv::xfeatures2d::SURF` class [Kae17, pp. 545–551].

To match two sets of descriptors, OpenCV provides the abstract `cv::DescriptorMatcher` class [Kae17, pp. 520–526]. Different algorithms for matching descriptors are implemented by different derived classes. In this project, "brute force matching" is used, which is implemented by the `cv::BFMatcher` class [Kae17, pp. 573–574]. The descriptor matcher is trained with one set of descriptors, which we call training descriptors. After that, matches between the training descriptors and a second set of descriptors, which we call query descriptors, are computed with the `match` member function of the matcher. A match between a training descriptor and a query descriptor is represented by an instance of the `cv::DMatch` class [Kae17, pp. 519–520].

Before the program can detect an object, the user has to select an area within an image. This area should contain many noticeable keypoints that belong to the object that the system should recognize. For simplicity of implementation, it is assumed that the area is rectangular. When a rectangular area is selected, the coordinates of the rectangle are saved, and the keypoints and descriptors within that area are computed. Since it is assumed that the area contains the object that should be recognized, we call them target keypoints and target descriptors. After that, the descriptor matcher is trained with the target descriptors.

As soon as a valid selection has been made, the system tries to detect the target in images that it receives from the camera. From each image, keypoints and descriptors are computed; we call them query keypoints and query descriptors. Then the system computes matches between the query descriptors and the target descriptors. Optionally, weak matches can be fil-

tered out. If enough matches have been found, the system assumes that the target is present in the newly received image and computes the homography that transforms the points of the target keypoints into the points of the matched query keypoints; this is done with the `cv::findHomography` function [Kae17, pp. 661–665]. Then the polygon that represents the detected object is computed by transforming the rectangle of the user selection using the homography.

6.3 Shape Detection

The shape detection system detects objects that have great similarities to common shapes. A typical use case is the detection of a mug with a cylindrical form.

Because of its robustness, RANSAC (random sample consensus) is used, which is a method for estimating parameters of a mathematical model from a set of data points that contains outliers.[Wiki17b] With regard to this project, the data points are the points of a point cloud, and the mathematical model describes a sphere or a cylinder, depending on the shape that the user selects. The RANSAC method is implemented by the `pcl::SACSegmentation` class.

To reduce the computing time, each point cloud is filtered first. All points whose distance from the camera does not lie in a range that is specified by the parameters of the system are filtered out. After that, the normals of the filtered point cloud are computed using the `pcl::NormalEstimation` class. Then a `pcl::SACSegmentation` object is created and a set of arguments is passed to it; among them are the shape that is segmented and the lower and upper limits for the radius of the segmented shape. These arguments should also be parameters of the object recognition system. Given the point cloud and the computed normals, this object segments the cloud and returns the indices of the inlier points and the coefficients of the model. If any inliers have been found, the object is computed from the given values. As described in section 5.2, the location of each detected object is represented by the pose of its center and by its width, depth, and height. The computation of these values depends on the selected shape.

For a sphere, the model coefficients that are returned by the segmentation function completely describe the sphere. They include its radius and the coordinates of its center so that it is trivial to determine the rectangular prism that represents the object.

For a cylinder, the model coefficients only contain the radius of the cylinder and its axis, given by the direction of the axis and *any* point on the axis. What is missing to fully describe a cylinder with finite length in three-dimensional space are the center and the height of the cylinder. The center of the cylinder is approximated by projecting the centroid of the inlier points onto the cylinder axis. The height of the cylinder is approximated by the maximum distance between two inlier points after projecting all inliers onto the cylinder axis.

Chapter 7

Individual Programs

Writing a self-contained program for each object recognition system is a natural starting point. The term "individual programs" expresses that each object recognition system is running as a single node and is independent from other object recognition systems. To keep each executable simple, each system is actually implemented as two nodes, one for creating parameter files and one for detecting objects based on an existing parameter file. Each system resides in its own package and only depends on standard ROS packages, e.g. `cv_bridge` and `pcl_ros`.

Even though this solution is very naive, it can be considered as some kind of framework and as a possible solution to our problem. By implementing all detection systems as ROS nodes that subscribe and publish to the same topics, it is possible to exchange one detection node for another, to run multiple detection nodes in parallel, and to write nodes that process the outputs of arbitrary detection nodes.

Nevertheless, we are striving for a more integrated and uniform solution that reduces the effort for developing new object recognition modules significantly. So why waste time with this approach? The major motivation is to gain a clear understanding of the demands that the detection systems put on the framework. Writing these programs reveals how they interact with the user. Doing this before designing the framework avoids that the designs and implementations of the detection systems are, conciously or subconciously, influenced by the constraints of the framework. Later, these programs are used as informal requirements; each system should behave similarly when being part of a framework.

The implementations of the systems are intentionally kept simple and

premature. Since it is already assumed that this approach will not be carried on in the future, the implementations are restricted to things that are absolutely necessary for the programs to work or that incur significantly new requirements on the framework.

7.1 HSV Detection

The HSV detection system resides in the `hsv_detector` package. It comprises the `hsv_definer` node, which is used to create parameter files, and the `hsv_detector` node, which publishes objects that are detected based on such a file. Facilities that are used by both nodes are defined in the `hsv_utils.h` header file.

The implementations of both nodes are straightforward. Each one creates a single object before entering an infinite loop, waiting for incoming events. This object does all the work; it sets up the system in its constructor, and it provides the callback functions for reacting to user inputs and to incoming messages. This object is of type `HSVDefiner` in the `hsv_definer` node and of type `HSVDetector` in the `hsv_detector` node. The callback function that receives the images does all the processing by calling ordinary functions. Therefore, it is fair to say that both nodes are procedural programs in disguise.

`hsv_definer`

The `hsv_definer` node provides a GUI that is implemented with OpenCV's HighGUI module [Kae17, chapter 9]. HighGUI enables developers to quickly prototype simple GUIs for OpenCV applications. Its set of UI-features is not complete; it only comprises trackbars and the capturing of keypresses and mouse clicks. By compiling OpenCV with Qt-support, it is possible to attach buttons to the so-called *properties window*.

Launching the node pops up three windows, as shown in figure 7.1. The window titled "Unmodified RGB Image" shows the most recently received, unmodified color image; the polygons that represent detected objects are drawn into it. The window titled "Filtered Image" renders the completely filtered image; white pixels have passed the filter, black ones have been filtered out. The window titled "Trackbars" contains the trackbars for adjusting the thresholds for hue, saturation, and value. By adjusting these values

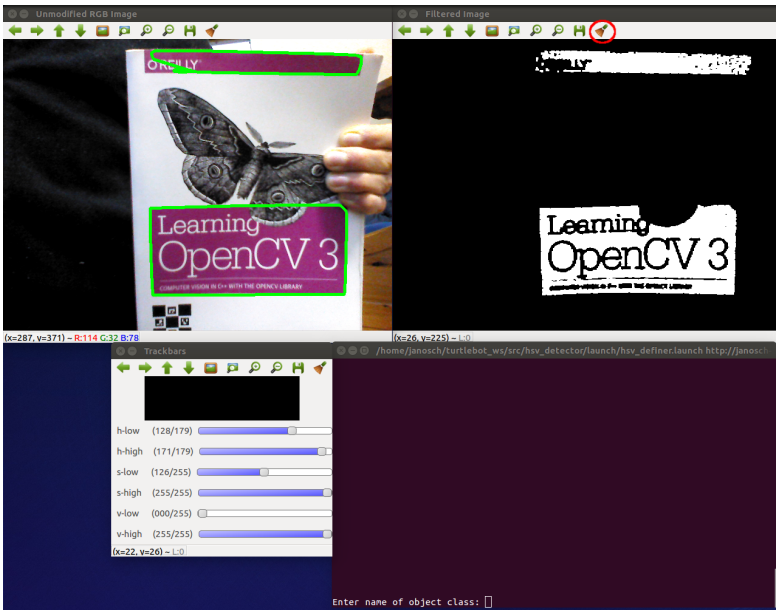

Figure 7.1: User interface of the `hsv_definer` node; the button for opening the properties window is marked with a red ellipse

and observing the results in the other two windows, the user determines the right values for detecting the color of his choice. The parameter values are saved by clicking the Save-button in the properties window, which is opened by clicking on the rightmost icon in the toolbar. The program then prompts the user to enter the name of detected objects in the console. The parameter file is stored in the `objects` folder of the package, and its name equals the entered object name followed by the file extension `hsv`.

`hsv_detector`

The `hsv_detector` node can be launched with the following command:

```
$ rosrun hsv_detector hsv_detector bradski_book \
> bgr_image:=camera/rgb/image_raw
```

The parameters of the system are loaded from the file in the `objects` folder whose name equals the given object name, `bradski_book` in the example above, followed by the file extension `hsv`. In addition, the name of the topic `bgr_image` is mapped to the name of the topic over which images are

published by the camera.

The detected objects are published over the `detected_objects` topic. This can be verified by using the `rostopic echo` command as shown in listing 7.1.

Listing 7.1: Printing messages of the `/detected_objects` topic

```
$ rostopic echo /detected_objects
header:
  seq: 71
  stamp:
    secs: 1510846457
    nsecs: 930620939
  frame_id: camera_depth_frame
objects:
  -
    name: bradski_book
    polygon:
      points:
        -
          x: 7.0
          y: 479.0
          z: 0.0
        -
          x: 0.0
          y: 479.0
          z: 0.0
        -
          ...
```

Each object is given the name `bradski_book`, which has been passed to the node at start-up.

7.2 Feature Detection

The implementation of the feature detection system is similar to the one of the HSV detection system. The system resides in the `feature_detector` package and comprises the `feature_definer` node, which is used to create parameter files, and the `feature_detector` node, which publishes objects

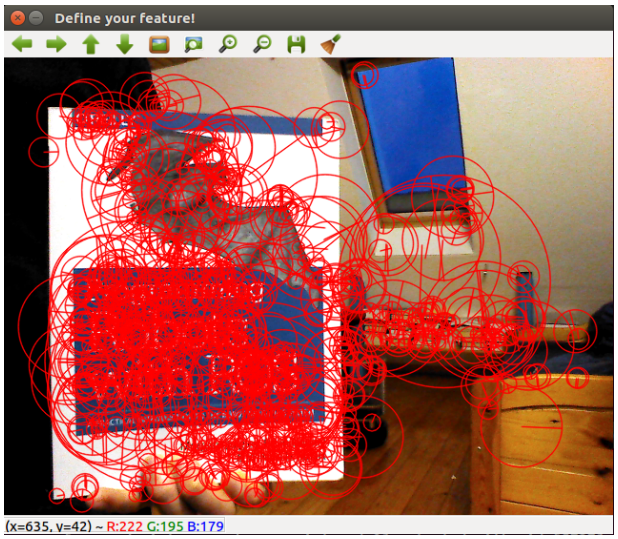

Figure 7.2: User interface of the `feature_definer` node

that are detected based on such a file. Facilities that are used by both nodes are defined in the `feature_utils.h` header file.

Launching the `feature_definer` nodes opens the window that is shown in figure 7.2. Keypoints are depicted as red circles with radial lines, which indicate their orientation.

To select target keypoints, the user draws a rectangular area into the rendered image with his mouse. When the selection has been completed, the window expands as shown in figure 7.3. The most recently received image is rendered on the left; the image from which the target has been selected is rendered on the right. If a keypoint from the right image is recognized in the left image, the two keypoints are connected by a line. If an object has been detected, its polygon is drawn in blue color into the left image; this polygon is computed by transforming the green selected rectangle that is shown in the right image.

Saving the current selection and launching the system with an existing parameter file is done in the exact same way as for the HSV detection system.

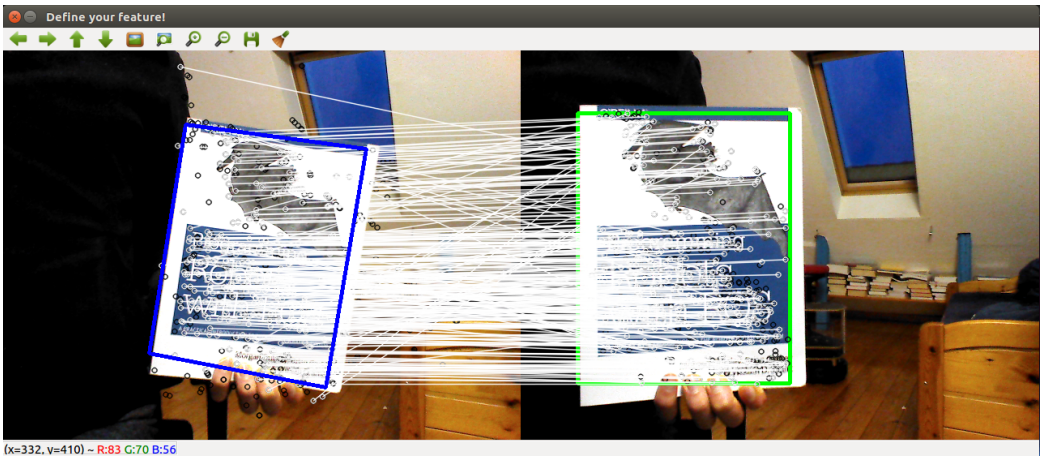

Figure 7.3: User interface of the `feature_definer` node after selecting a target

7.3 Shape Detection

After the individual program for the shape detection system was written in a very rudimentary form, it was developed further until it became what we call the "library solution", which is described in the next chapter. The original program does not exist anymore, but it was very similar to a tutorial from the PCL website that is listed in appendix A.

As mentioned in the introduction of this chapter, the main purpose of creating individual programs is to gain a clear understanding of each system and of its requirements. These requirements, which also apply to the solutions that are described in the following chapters, are listed below:

- The most recently received point cloud should be visualized. The visualization should have similar capabilities as the `PCLVisualizer` class [PCL17c], which is provided by the PCL library. More precisely, it should be possible to zoom in and out and to rotate the view of the point cloud.

- The rectangular prism that represents a detected object should be visualized.

- The user should be able to adjust the most important parameters at run time. At a minimum, these parameters should include the detected

shape (cylinder or sphere), the thresholds for filtering the points based on their distances, and the minimum and maximum radii of the shape (see chapter 6.3).

- As always, the system must provide means for saving and loading parameter files.

Chapter 8

Library Solution

Critically investing the programs from the previous chapter reveals that many concepts occur in all of them and would presumably occur in many other object recognition systems as well. Examples are the user interface, the parameter management, and the overall workflow.

The core idea of the approach described in this chapter is to extract as many of the commonly needed facilities into a library—i.e. a collection of functions, classes, and other definitions—that is used by all object recognition systems. This library also suggests a general structure and a general workflow for all object recognition systems. This workflow describes the order in which ROS messages are received, converted, and processed and in which results are converted to ROS messages and published. This approach still uses two nodes for each object recognition system, one for creating parameter files and one for detecting objects based on such files, but both nodes use the same set of classes. Creating the library is the first attempt to create a modular framework for object recognition; in this case, each module is implemented as a class.

It was necessary to create two seperate libraries, one for 2D and one for 3D object recognition. Nevertheless, the fundamental ideas that underlie both libraries are identical. For that reason, this chapter describes as much as possible in common terms. The specifics of each library are mentioned at the end of the chapter. The names of the classes in the two libraries are identical where possible; they differ only in their namespace. When speaking of a class named `Detector`, we mean the class `object_detection_2d::Detector` for the 2D library and `object_detection_3d::Detector` for the 3D library.

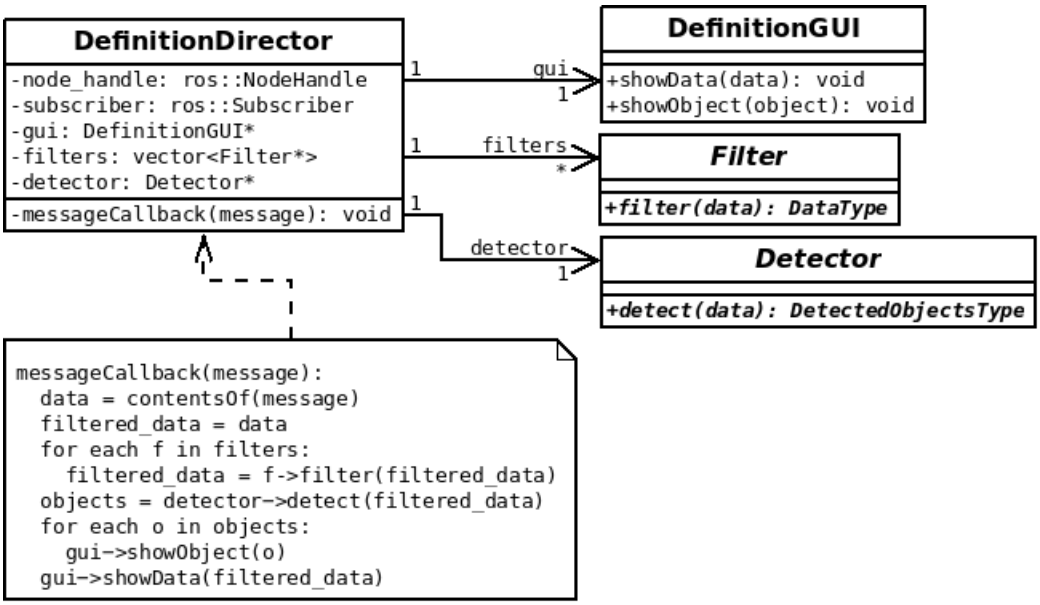

Figure 8.1: Class diagram [Boo07, chapter 5.7] of classes that participate in the creation of parameter files (created with Dia [McC17])

8.1 Classes and their Relationships

This section describes the classes of the created libraries and the relationships between these classes. It investigates the libraries from three point of views.

The descriptions use general terms; the actual solutions for the 2D and 3D libraries, described in section 8.5, are slightly different from each other, and none of them matches the descriptions given in this section exactly.

8.1.1 Processing data and detecting objects

This section describes the workflow of an object recognition system without user interaction. This includes receiving data over the ROS network, optionally preprocessing this data, detecting objects within this data, and visualizing or publishing the results.

Classes that participate in the creation of parameter files are shown in figure 8.1. A `DefinitionDirector` object handles the ROS communication and directs the components of the system that do the actual work. To this end, it has references to these components, namely to a `DefinitionGUI`

object, to zero or more `Filter` objects, and to a `Detector` object. To receive ROS messages, it has a `ros::NodeHandle` and a `ros::Subscriber` as data members. The messages that it receives contain images for 2D detection and point clouds for 3D detection. Since figure 8.1 tries to capture the correlations of the 2D and the 3D library, it calls instances of these data types collectively `data`. The `DefinitionGUI` is a Qt-based [Qt17a] GUI that contains a widget for rendering images or point clouds and detected objects, and it provides methods for doing so. It also contains widgets for setting the parameters of the object recognition system. `Filter` and `Detector` are abstract classes that provide pure virtual functions for filtering data and detecting objects, respectively. The most important member function of the `DefinitionDirector` is the callback function `messageCallback`, which handles incoming data; this function is called `imageCallback` for 2D and `cloudCallback` for 3D object recognition. Within this method, data is first extracted from the received ROS message. Then it is filtered by all registered filters. The filtered data is rendered in the GUI and passed to the detector's `detect` method. If any objects are detected, they are rendered by the GUI as well.

In the node that publishes objects that are detected based on an existing parameter file, no GUI is used and the instance of the `DefinitionDirector` class is replaced by an instance of the `DetectionDirector` class. The only differences between both classes are that the latter has no reference to a GUI, has a `ros::Publisher` as a data member, and publishes detected objects instead of visualizing them.

The chosen structure leads to an encapsulation of the ROS communication and the general workflow and makes it possible to reuse the `*Director` classes for different object recognition systems. In addition, it leads to a better encapsulation of dependencies: only the `*Director` classes depend significantly on ROS; only the `DefinitionGUI` class depends on Qt.

8.1.2 Monitoring and Setting Parameters

Since the GUIs of the object recognition systems treated in this document all have a similar structure and many common facilities, the creation of a common GUI class seems appropriate. On the other hand, each GUI has to contain widgets that are highly specific to the particular object recognition system.

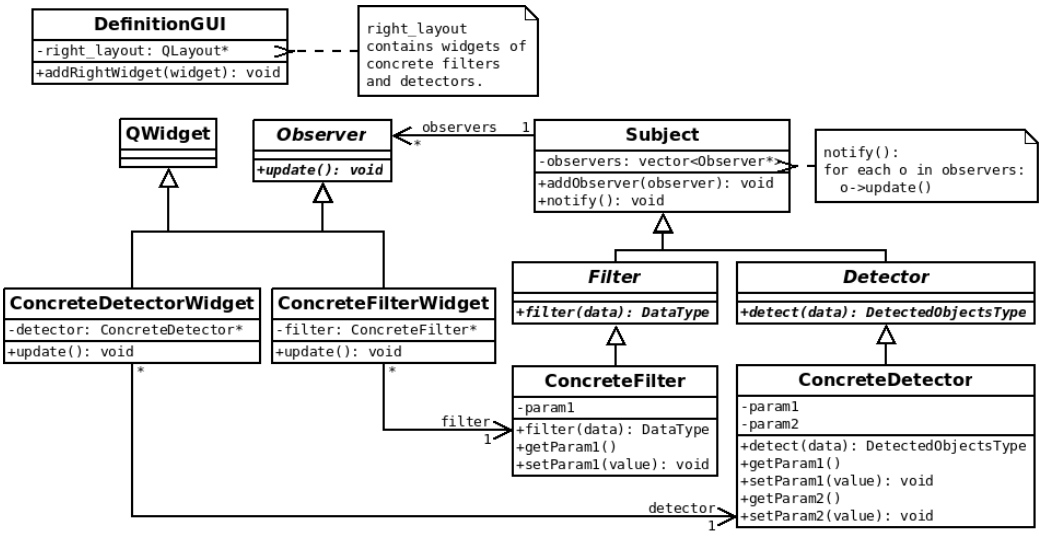

Figure 8.2: Class diagram of classes that enable the user to monitor and set parameters

Class Diagram

This conflict was solved by creating a common GUI class, called `DefinitionGUI`, that provides a container for other widgets. An instance of this class can be configured for a specific object recognition system by attaching system-specific widgets to the GUI. The class diagram in figure 8.2 depicts the involved classes and their relationships.

Filters and detectors have private data members that represent their parameters and provide public methods for accessing these data members.

Normally, a widget, which inherits from `QWidget` [Qt17b], exists for each filter and each detector. This means that writing a new filter or detector module always includes writing the corresponding widget, which provides access to the parameters of that filter or detector. When a specific filter or detector is used in a program with a GUI, the widget of that filter or detector is attached to the GUI. To this end, each `DefinitionGUI` instance maintains a `QLayout` [Qt17c] reference, called `right_layout`, and provides the `addRightWidget` method that inserts a widget into that layout.

The communication between a specialized widget and the corresponding filter or detector is based on the *Document-View* pattern, explained below. The abstract `Filter` and `Detector` classes inherit from the `Subject` class.

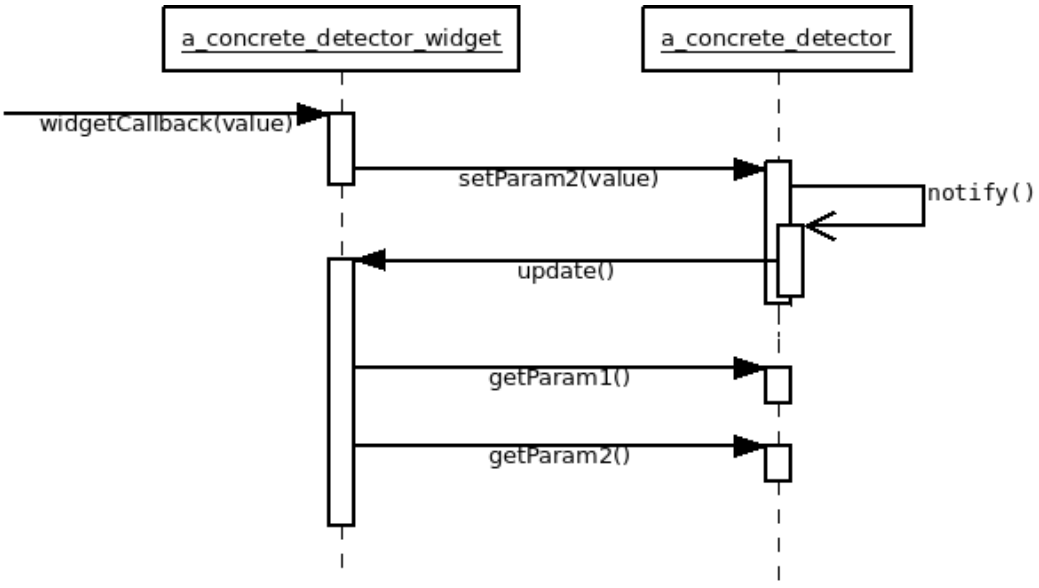

Figure 8.3: Sequence diagram [Boo07, chapter 5.8] (created with Dia [McC17]) of the interactions that are triggered by a user who changes the value of a parameter in the GUI (compare with [Gam95, p. 295])

Each `Subject` maintains a `vector` of `Observer` references and provides the `addObserver` method for adding a reference to that vector. The `notify` method of a `Subject` instance should be called if any parameter of that instance changes its value; this method calls the `update` method of each registered `Observer`. Each specialized widget keeps a reference to an instance of the corresponding filter or detector class; in figure 8.2, for example, each `ConcreteFilterWidget` instance has a reference to a `ConcreteFilter` instance. Each of these widgets also inherits from the abstract `Observer` class and overrides the `update` method of that class.

Sequence Diagram

A typical interaction between a `Subject` and its widget is depicted in figure 8.3. When the user tries to change the value of a parameter by using the widget, a callback function of the widget calls the corresponding setter method of the subject. In this method, the subject changes the value of its parameter and calls its `notify` method. This method calls the `update`

method of the observer, i.e. the widget. Within this method, the widget updates its view by calling the getter methods of the subject.

Patterns

As mentioned briefly, the implementation of the communication between widgets and their subjects is based on the Document-View pattern [Busch96, pp. 140–141]. This pattern is a variant of the Model-View-Controller (MVC) pattern [Busch96, pp. 125–144], which is based on the Observer pattern [Gam95, pp. 293–303]. In this case, an instance of the `Subject` class represents a *Document*; the specialized widget of that subject, i.e. the instance of the `Observer` class, represents the *View* of that Document.

Evaluation

Each subject has private data members that represent its parameters. It is the solely owner of these parameters and has full control over which values are assigned to them. This makes sense because it is the only instance that knows the meaning of these parameters and can therefore decide if their values are valid. This structure also leads to quick access to these values; this is important because they are normally accessed every time a message is received over the ROS network. The drawback of this structure is that the parameter management can constitute a major part of the subject's implementation; this is a violation of the Single Responsibility Principle [Mar09, pp. 138–140] if parameter management is considered a seperate responsibility. It would be better to encapsulate this responsibility in some way, e.g. by putting a dedicated data member in charge of the parameter management.

Using the Document-View pattern guarantees that the view of the widget is consistent with the state of the subject. This way of communicating is not as efficient as a simple one-way communication, i.e. as sending commands from the widget to the subject, relying on the subject to adjust its parameters accordingly. But since user interactions occur infrequently, this is not a problem.

Another advantage of this approach is that it leads to only one direct coupling; each specialized widget has a direct reference to the specific subject, not just to some base class. This dependency is unavoidable, since the widget has to provide access to each parameter of that subject.

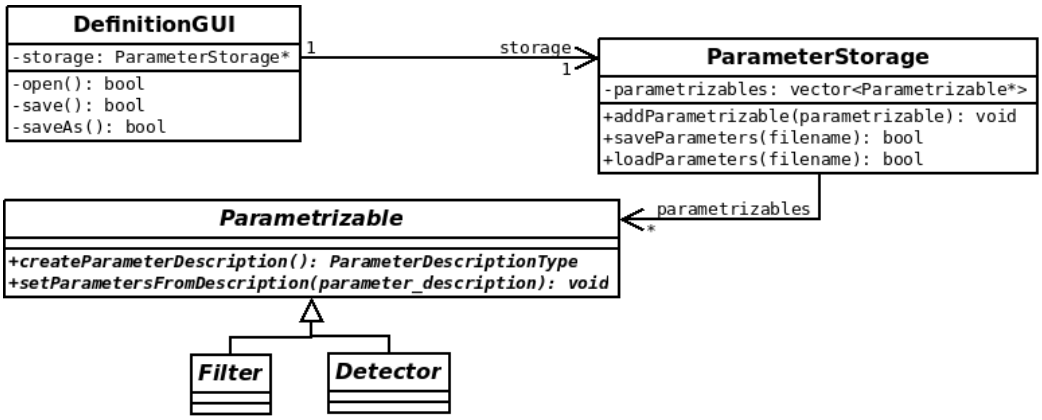

Figure 8.4: Diagram of classes that are involved in the creation of parameter files

8.1.3 Saving and Loading Parameters

Saving parameters to a file and loading them from a file are features that are needed by all object recognition systems. Therefore, they should be part of the libraries, usable by all systems. The difficulty of centralizing this functionality is that the number of parameters and their names and types depend on the specific object recognition system.

It should also be possible to trigger the saving and loading of parameters from the GUI, but the saving and loading should not be part of the GUI's implementation. They represent a seperate responsibility and should therefore, according to the Single Responsibility Principle, be encapsulated in a seperate class.

Class Diagram

The solution that is used to resolve these conflicts is depicted in the class diagram in figure 8.4. It applies to the node that is used to create parameter files.

The saving and loading of parameters is encapsulated in the `ParameterStorage` class. Every object recognition system has one instance of that class. This instance has a data member called `parametrizables`, which is a vector of references to instances of the `Parametrizable` class, and it provides the `addParametrizable` method for adding a reference to that vector.

`Parametrizable` is an abstract class that represents any object that has

parameters. All filters and detectors inherit from that class. It provides two pure virtual member functions. The first function, `createParameterDescription`, creates a parameter description. The second one, `setParametersFromDescription`, takes such a parameter description, parses it, and sets the parameters of the object accordingly. "Parameter description" is the term that is used to refer to a representation of parameter values that can be written to a file by the `ParameterStorage` instance. The `ParameterStorage` instance also knows how to create a parameter description from the contents of such a file. The parameter description has to contain the values of some parameters, as well as identifiers of these parameters so that a `Parametrizable` object can parse the description and find the values of its parameters within it. A string is a simple representation that could have been used for parameter descriptions.

The `DefinitionGUI` of a system has a reference to the `ParameterStorage` instance of that system. The private `save`, `saveAs`, and `open` methods of the `DefinitionGUI` are Qt slots; clicking on the corresponding entries in the menu bar of the GUI emits Qt signals that lead to the invocation of these methods.[Bla08, p. 5] If necessary, these methods get the filename from the user before passing it to the `loadParameters` or to the `saveParameters` method of the `ParameterStorage` instance.

The pseudocode of the `saveParameters` method is as follows:

```
saveParameters(filename):
  file = openForWriting(filename)
  for each p in parametrizables:
    writeParameterDescriptionToFile(
      p->createParameterDescription(), file)
```

After having opened the file with the given name, the method gets the parameter description of each registered `Parametrizable` object and writes it to the file.

The pseudocode of the `loadParameters` method is as follows:

```
loadParameters(filename):
  file = openForReading(filename)
  description = readParameterDescriptionFromFile(file)
  for each p in parametrizables:
    p->setParametersFromDescription(description)
```

First, the complete parameter description is read from the file with the given

name. Then, this description is passed to each registered `Parametrizable` object. Each of those searches for its parameters in this description and sets its parameters accordingly.

There is no GUI in the node that is responsible for detecting and publishing objects based on an existing parameter file. In that node, the `loadParameters` method of the `ParameterStorage` instance is invoked by the `main` function.

Evaluation

The chosen solution fulfills most of our requirements. Saving and loading of parameters can be triggered from the GUI, and it is encapsulated in the `ParameterStorage` class.

Nevertheless, a significant part of the work remains to be done by the `Parametrizable` objects of a system. Each `Parametrizable` object has to implement the functions for creating and parsing parameter descriptions.

8.2 Nodes

A fundamental assumption is that each user of the framework tries to solve a specific task and needs a specific object recognition system offered by the framework to do so. With a "specific object recognition system", we mean a set, containing filters and a detector, chosen from the filters and detectors offered by the framework. It is also assumed that the user will write two nodes to use the specific system. The first node provides the interface for creating parameter files; the second node detects and publishes objects based on such a parameter file.

For both nodes, the main part of the implementation consists of setting up the system according to the class diagrams shown in the previous section. First, the required components of the library have to be created. These include objects that are needed by every system, e.g. an instance of the `ParameterStorage` class, as well as the specific filters and detectors that the user needs. Second, these components have to be connected; this is done by passing references to components to other components. The objects provide methods for doing this; an example is the `addFilter` method that is used to pass a `Filter` reference to a `DefinitionDirector` object.

This section describes how each node is created and used. The shape

detection system is taken as an example. As described in section 8.4.3, it has been divided into a distance filter and a shape detector.

8.2.1 Creating Parameter Files

The node that is described in this section is the `object_definition` node from the `shape_detection_3d_test` package. This node is used to create parameter files for the shape detection system.

Setup

Listing 8.1 lists the `main` function of the node.

Listing 8.1: `try`-block of the `main` function in the `object_definition.cpp` file from the `shape_detection_3d_test` package

```
1  int main(int argc, char** argv)
2  try {
3    // Initialize Qt and ROS.
4    QApplication q_app {argc, argv};
5    ros::init(argc, argv, "object_definition");
6
7    //
8    // Setup detection-system.
9    //
10   DefinitionGUI gui;
11   gui.setWindowTitle("Shape Detection - Object
       Definition");
12   DistanceFilterGroupBox* filter_groupbox {new
       DistanceFilterGroupBox};
13   ShapeDetectorGroupBox* detector_groupbox {new
       ShapeDetectorGroupBox};
14
15   gui.addRightWidget(filter_groupbox);
16   gui.addRightWidget(detector_groupbox);
17   gui.show();
18
19   DistanceFilter filter;
20   filter.addObserver(filter_groupbox);
```

```
21    filter_groupbox->setFilterSubject(&filter);
22
23    ShapeDetector detector;
24    detector.addObserver(detector_groupbox);
25    detector_groupbox->setDetectorSubject(&detector);
26
27    ParameterStorage storage;
28    storage.addParametrizable(&filter);
29    storage.addParametrizable(&detector);
30    gui.setParameterStorage(&storage);
31
32    DefinitionDirector director;
33    director.setGUI(&gui);
34    director.addFilter(&filter);
35    director.setDetector(&detector);
36
37    //
38    // Event loop.
39    //
40    while (ros::ok()) {
41      ros::spinOnce();
42      q_app.processEvents();
43
44      // Shutdown ROS node if user closes the gui.
45      if (!gui.isVisible())
46        ros::shutdown();
47    }
48
49    return 0;
50  }
```

The call to `ros::init` is obligatory for every ROS node. Similar, a `QApplication` instance has to be created in every executable that uses Qt. After that, in lines 12 to 15, the GUI and the widgets of the distance filter and the shape detector are created. The widgets are attached to the GUI in lines 17 and 18; from then on, the GUI is responsible for deleting these objects on destruction. Then, in lines 21 to 23, the distance filter is created and connected to its widget. The same thing is done for the shape detector in

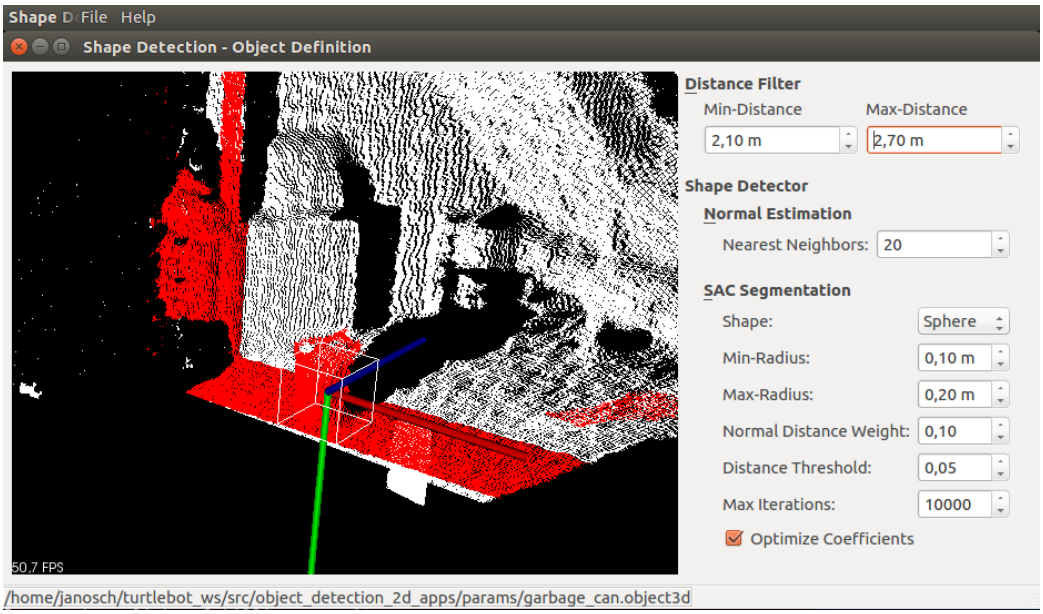

Figure 8.5: GUI for creating parameter files for the shape detection system. The original point cloud is rendered in white, the filtered point cloud is rendered in red. The detected object is a cylindrical garbage can.

lines 25 to 27. Lines 29 to 32 create the `ParameterStorage` instance, pass it references to all objects that have parameters, and register it with the GUI. At the end of the setup, the `DefinitionDirector` instance is created and references to the GUI, the filter, and the detector are passed to it. The event loop processes ROS and Qt events. Incoming point cloud messages are handled by the callback function of the `DefinitionDirector` object. The event loop is exited when the GUI is closed or when the user tries to shut down the node by pressing `Ctrl-C` in the terminal that runs this node.

Usage

Launching the node opens the window shown in figure 8.5. The "Help|About" entry in the menu bar displays a short explanation of the GUI, similar to the one given in this section. The widget on the left renders the most recently received point cloud and the boxes that represent detected objects. On the right are the widgets that are used for monitoring and setting the parameters of the distance filter and the shape detector. The parameter values are

adjusted until the result of the detection is satisfying. Then these values are stored in a parameter file by clicking on the "File|Save" or "File|SaveAs" entry. The status bar, at the bottom of the GUI, displays a status message for a short period of time, informing the user if the parameters have been saved successfully. The GUI remembers the path to the parameter file and displays it in the status bar. Clicking the "File|Save" entry again overrides this file, while "File|SaveAs" always prompts the user for the filename. An existing parameter file can be opened and modified by clicking on the "File|Open" entry in the menu bar and selecting the file in the file dialog that pops up. Again, the status bar informs the user if loading the parameters has succeeded.

8.2.2 Publishing Detected Objects

The node that is described in this section is the `object_detection` node from the `shape_detection_3d_test` package. This node detects objects based on an existing parameter file and publishes them.

Setup

Listing 8.2 lists the `main` function of the node.

Listing 8.2: `try`-block of the `main` function in the `object_detection.cpp` file from the `shape_detection_3d_test` package

```
1  int main(int argc, char** argv)
2  try
3  {
4    ros::init(argc, argv, "object_detection");
5
6    // Check command-line arguments.
7    if (argc != 2) {
8      printUsage(argv[0]);
9      return 0;
10   }
11
12   // Setup object recognition system.
13   DistanceFilter filter;
14   ShapeDetector detector;
```

```
15
16    ParameterStorage storage;
17    storage.addParametrizable(&filter);
18    storage.addParametrizable(&detector);
19    if (!storage.loadParameters(argv[1])) {
20      std::cerr << "Cannot load parameters from file "
            << argv[1] << '\n';
21      return 1;
22    }
23
24    DetectionDirector director;
25    director.addFilter(&filter);
26    director.setDetector(&detector);
27
28    // Event loop.
29    ros::spin();
30    return 0;
31  }
```

In contrast to the previous listing, this node does not use a GUI. Another major difference is the handling of parameters. This node requires that the name of a parameter file is passed to the node as the first and only command-line argument. Therefore, it checks the correct number of command-line arguments in line 7 and tries to load the parameters from the file in line 19. The last difference in the setup of the system is that a `DetectionDirector` instance is used instead of the `DefinitionDirector` instance.

Usage

To publish the objects that are detected according to a parameter file, the `object_detection` node is launched and the absolute path of the file is passed to it as a command-line argument. Detected objects are then published on the `detected_objects_3d` topic.

8.3 Layers and Packages

This section presents an alternative, higher-level view on the previously described classes and nodes by seperating them into three layers, as depicted

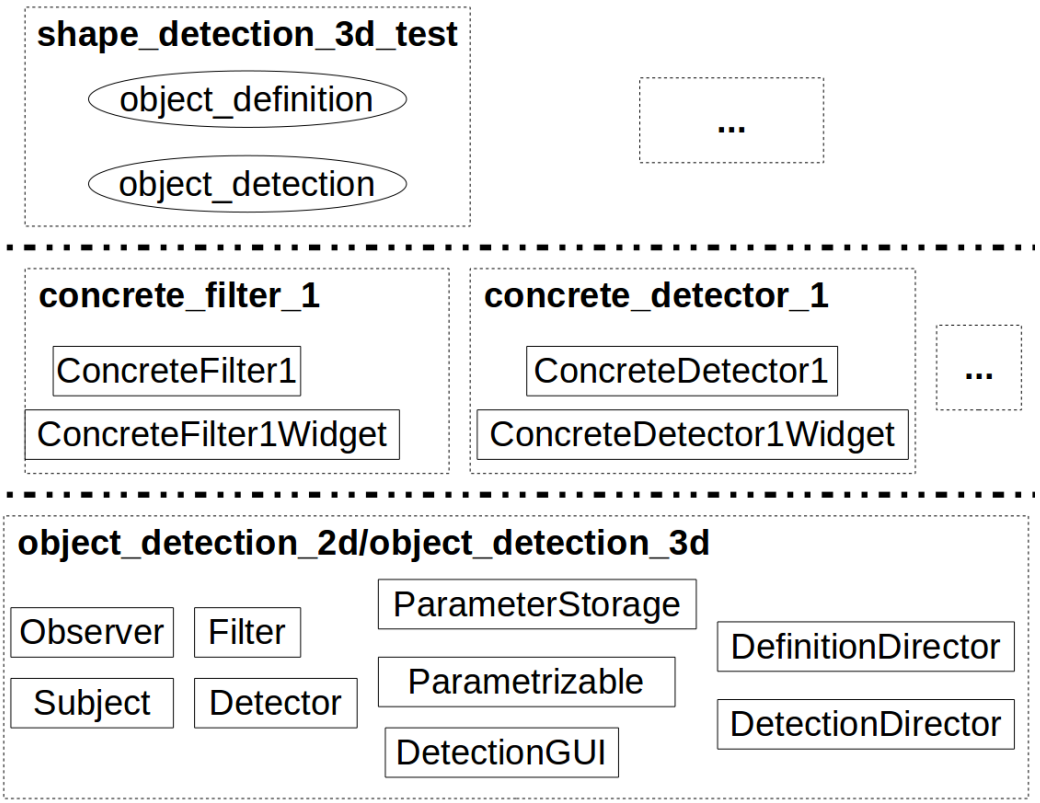

Figure 8.6: Seperation of classes and nodes into layers and packages

in figure 8.6.

The bottom layer comprises the classes that are used by all object recognition systems. These classes define the basic structure and the general workflow of all systems. For both libraries, this layer is provided as a single shared object, which is defined in the `object_detection_2d` package for the 2D library and in the `object_detection_3d` package for the 3D library.

The layer in the middle comprises concrete filters and detectors that have been added to the framework. Each concrete filter or detector module, including its widget (see section 8.1.2), resides in its own package and is provided in the form of a shared object.

The top layer comprises ROS nodes that make use of the library. These are two nodes for each object recognition system, as described in the last section. Users of a specific object recognition system will have to write these

nodes if they do not exist already. The distribution of these nodes into packages does not matter. As an example, the top layer in figure 8.6 includes the `shape_detection_3d_test` package, which contains the `object_definition` and `object_detection` nodes described in the previous section.

Mentally dividing the classes and nodes into these three layers helps to clearly understand and minimize the dependencies between packages. A package should only depend on packages in the lower layers. In addition, the nodes in the top layer only depend on those filters and detectors they use. Following these rules supports the Open/Closed Principle; the libraries can be extended by adding new filters and detectors to it without having to modify any existing classes.

8.4 Implemented Object Recognition Systems

This section describes how each of the three object recognition systems that are used as examples throughout the document are integrated into the framework. Among other things, it is shown how each system is divided into filter and detector modules; each module resides in its own package and is part of the middle layer that is shown in figure 8.6. More implementation details are described in section 8.5.

8.4.1 HSV Detection

The HSV detection system has been divided into three parts, the HSV filter, the morphology filter, and the binary detector.

The HSV filter resides in the `hsv_filter` package and is implemented as the `object_detection_2d::HSVFilter` class, which inherits from `object_detection_2d::Filter`. Its parameters are called `h_min`, `h_max`, `s_min`, `s_max`, `v_min`, and `v_max`. The widget of the HSV filter is implemented as the `object_detection_2d::HSVFilterGroupBox` class. The `HSVFilter`'s `filter` method returns an RGB image whose pixels have the maximum red, green, and blue values where the hue, saturation, and value values of the input image are within the ranges defined by the parameters, as described in section 6.1. All other pixels of the output image are set to zero.

The morphology filter resides in the `morphology_filter` package and is implemented as the `object_detection_2d::MorphologyFilter` class, which inherits from `object_detection_2d::Filter`. Its first parameter is the op-

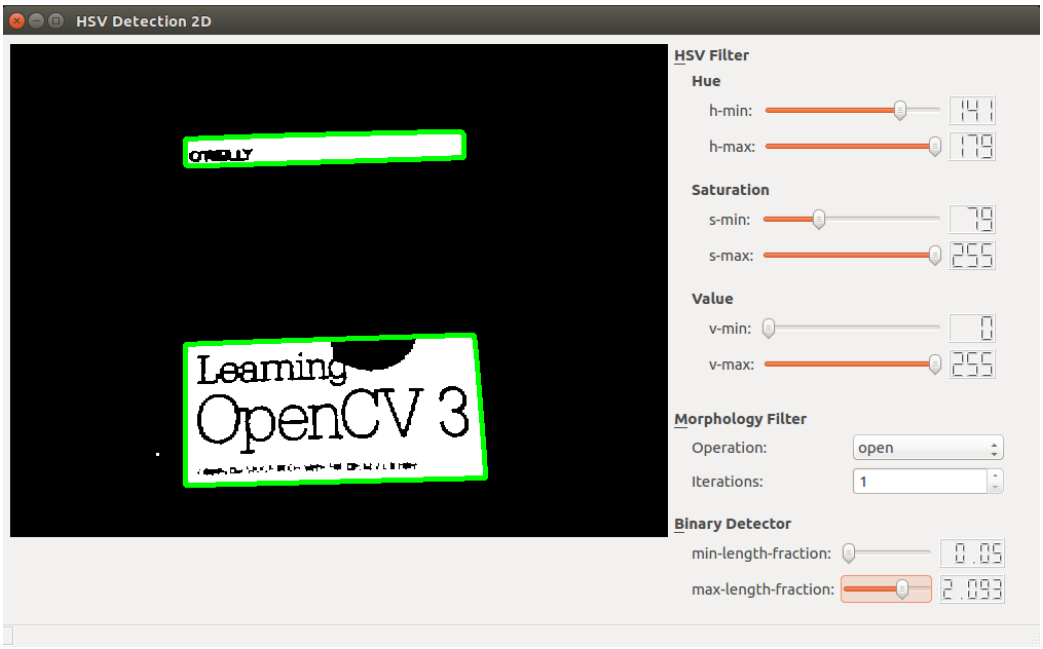

Figure 8.7: GUI for creating parameter files for the HSV detection system

eration that the filter applies to its input image; this operation can be erosion, dilation, opening, or closing. The second parameter is the number of iterations of the morphological transformation.

The binary detector resides in the `binary_detector` package and is implemented as the `object_detection_2d::BinaryDetector` class, which inherits from `object_detection_2d::Detector`. Every pixel whose value—computed by converting its red, green, and blue values to a corresponding grayscale value—differs from zero is considered a potential part of an object. A contiguous area of such non-zero pixels is declared an object if the length of its contour lies in the range that is determined by the parameters of the detector.

Figure 8.7 shows the GUI that is provided by the `hsv_object_definition` node from the `object_detection_2d_apps` package, which is used to create parameter files for that system. On the right are the widgets that are used to adjust the parameters of the two filters and the detector. Rendered on the left is the filtered image with the detected objects drawn into it. A limitation of the framework is that only the completely filtered image is displayed.

Splitting the system into two filters and one detector improves the modu-

larity of the system and the reusability of its components. But this seperation is also necessary to integrate the system into the framework. If the filtering of the image would have been done within the detector, there would have been no seperate filter component in the system. Since only the output image of the last filter is rendered, the rendered image would have been equal to the original, unmodified image. This would have made it almost impossible for the user to find appropriate values for the parameters.

Another disadvantage that results from integrating the system into the framework is a reduced efficiency, caused by unnecessary conversions. To be able to exchange filters and detectors and to use the same detector with other filters or without any filter, each filter and detector assumes a 3-channel image as its input. For this reason, the HSV filter converts the grayscale image that it computes to a RGB image before returning it. After passing the morphology filter, this image is converted back to a grayscale image at the beginning of the `detect` method of the binary detector. These two conversions are not necessary for the HSV detection system to work; they merely satisfy the constraints of the framework.

Launch files for both nodes of the system can be found in the `object_detection_2d_apps` package; they are called `hsv_object_definition.launch` and `hsv_object_detection.launch`.

8.4.2 Feature Detection

The feature detector resides in the `feature_detector_lib` package and is implemented as the `object_detection_2d::FeatureDetector` class, which inherits from `object_detection_2d::Detector`. A target can be selected by drawing a rectangular area the mouse into the image that is rendered in the GUI. The detector processes such a selection in its `processAreaSelection` method, which is called by the GUI; this mechanism is described in greater detail in section 8.5.2. When a parameter file is created after a valid target selection has been made, the selection is written to that file, together with the parameters of the system. Once a valid selection has been made or loaded from a parameter file, the detector tries to detect the target in its `detect` method.

Conforming to the framework inhibits the rendering of keypoints and matches, as done in the individual program (see section 7.2). The integration of the feature detector heavily influenced the design of the 2D library; section 8.5 describes this in greater detail.

Launch files for both nodes of the system can be found in the `object_detection_2d_apps` package; they are called `feature_object_definition.launch` and `feature_object_detection.launch`.

8.4.3 Shape Detection

The shape detection system has been divided into the distance filter and the shape detector modules.

The distance filter resides in the `distance_filter` package and is implemented as the `object_detection_3d::DistanceFilter` class, which inherits from `object_detection_3d::Filter`. Its parameters are the minimum and maximum distances of points that pass the filter.

The shape detector resides in the `shape_detector` package and is implemented as the `object_detection_3d::ShapeDetector` class, which inherits from the `object_detection_3d::Detector` class. The parameters of the detector are the parameters of the segmentation process described in section 6.3. They include the shape that is detected and the minimum and maximum radii of detected objects.

Section 8.2 describes the two nodes of the system in detail. Implementation details concerning the visualization and parameter storage are described in section 8.5.

8.5 2D and 3D Specifics

Besides the type of the input data, RGB images for the 2D library and point clouds for the 3D library, the two libraries only differ significantly in their implementations of the parameter storage and the visualization. For each of these aspects, the specifics of the 3D library are desribed first, since it was developed before the 2D library.

8.5.1 Saving and Loading Parameters

3D

As described in section 8.1.3, saving and loading of parameters is done by the `ParameterStorage` instance of the application, which exchanges parameter descriptions with all `Parametrizable` objects, i.e. with all filters and

detectors. This section points out two things that have not been stated yet and explains how each one is implemented for the 3D library.

First, the format of parameter files has to be defined. It was decided that each line of a parameter file describes the value of one parameter, according to the following format:.

```
<class_name>/<parameter_name> : <parameter_value>.
```

For example, when the `min_distance_` parameter of a `DistanceFilter` object has the value 0.5 (given in meters), the following line is written to the parameter file:

```
DistanceFilter/min_distance_ : 0.5
```

An obvious drawback of this solution is that there are name clashes if multiple filters of the same class are used in a system. Since there has not been any use for that yet, this format is kept for the time being.

Second, the type of the parameter description has to be defined. Section 8.1.3 only stated that a parameter description is some representation of parameter values that can be read from and written to a file. It was decided to use a dedicated type for the parameter description. In the source code of the `Parametrizable` class, this type is defined as follows:

```
using MultiClassParameterMap =
  std::map<std::string, SingleClassParameterMap>;
```

`SingleClassParameterMap` is defined in the same file as follows:

```
using SingleClassParameterMap =
  std::map<std::string, std::string>;
```

The `SingleClassParameterMap` type is used for storing all the parameters of a single class. A key within the `SingleClassParameterMap` is the name of a parameter of the class; the corresponding value is the string representation of that parameter's value. As an example, the parameters of a `DistanceFilter` object are called `min_distance_` and `max_distance_`; therefore, a `SingleClassParameterMap` of such an object might have the following form:

```
{"min_distance_": "0.5",
 "max_distance_": "1.5"}
```

The key of the `MultiClassParameterMap` is the name of a class. The value of this map is the `SingleClassParameterMap` that represents the parameters

of an object of this class. So a `MultiClassParameterMap` that represents the parameters of a system that comprises a distance filter and a shape detector might have the following form:

```
{"DistanceFilter":
    {"max_distance_" : "1.5"},
 "ShapeDetector":
    {"max_iterations_": "1000",
     "min_radius_": "0.1"}}
```

A simpler solution would have been to pass the complete contents of a parameter file as a string to all `Parametrizable` objects. The advantage of using the `MultiClassParameterMap` is that the parsing of the file contents can be centralized in the `ParameterStorage` class. Extracting the parameters of a specific `Parametrizable` object from an object of type `MultiClassParameterMap` is trivial. Each `Parametrizable` object searches for the name of its class first, thereby extracting the `SingleClassParameterMap` that contains its parameters if it is available. After that, it searches for the names of all of its parameters in the `SingleClassParameterMap` object. The value of each parameter that is found in the map is converted from its string representation to the right type and assigned to the corresponding data member of the `Parametrizable` object. It is hard to avoid doing the conversion of each parameter within the corresponding `Parametrizable` object because that object is the only one that knows the types of its parameters.

2D

In general, when creating a parameter file, it is not sufficient to store the parameters that can be set from the GUI. The parameter file has to contain the complete state of each `Parametrizable` object in the system so that loading the file later results in detecting the same objects. In the case of the feature detector, target keypoints and target descriptors are data members of the detector (compare with section 6.2). When one of the parameters that relate to the computation of the SURF features changes, the target keypoints and descriptors have to be recomputed. For that reason, the image from which the target was selected, or at least the selected area within it, has to be stored in the parameter file.

Unfortunately, it is inconvenient and unnatural to convert images to a string representation and back again, as done with parameters of 3D object

recognition systems. Since it is assumed that many other filters and detectors for 2D object recognition could have similar requirements as the feature detection system, a different implementation was chosen for the 2D library.

The chosen solution is based on the `cv::FileStorage` class [Kae17, pp. 198–204]. The example in listing 8.3 illustrates the capabilities of this class and how it is used.

Listing 8.3: Usage of the `cv::FileStorage` class

```
1  cv::FileStorage fs {"parameter_file.yml", cv::
       FileStorage::WRITE};
2  fs << "double_value" << 5.2;
3  fs << "identity_matrix" << cv::Mat::eye(3, 3, CV_32F
       );
4  fs.release();
5
6  cv::FileStorage ifs {"parameter_file.yml", cv::
       FileStorage::READ};
7  double d;
8  ifs["double_value"] >> d;
9  cv::Mat m;
10 ifs["identity_matrix"] >> m;
```

In the first part of the listing, two values are written to a file called `parameter_file.yml`. Each value is given a name that serves as a handle to retrieve the value from that file later. In the second part of the listing, the same file is opened for reading, and the two values are retrieved from it and stored in local variables. In general, each value that is written to a `FileStorage` object has to have a unique name. Values of different types can be stored; these types can be any OpenCV types, e.g. `cv::Mat` and `cv::Rect`, as well as any built-in types, e.g. `int` and `double`.

Within the `ParameterStorage`'s `saveParameters` method, a `FileStorage` object is created for writing to the file whose name has been passed to the method. Then a reference to this object is passed to the `writeParameters-ToStorage` method of each registered `Parametrizable` object. This method replaces the `createParameterDescription` method that is mentioned in section 8.1.3, and it writes all parameters of a `Parametrizable` object to the given `FileStorage` object, giving each one a distinguishing name.

The `ParameterStorage`'s `loadParameters` method works similarly. The only differences are that the `FileStorage` object is created for reading the

file with the given filename and that a reference to it is passed to the `setParametersFromStorage` method of each `Parametrizable` object. The latter method replaces the `setParametersFromDescription` method that is mentioned in section 8.1.3, and it tries to retrieve all of its parameters from the `FileStorage` object and to set the values of its data members accordingly.

8.5.2 Visualization

3D

For rendering point clouds and detected objects, the `DefinitionGUI` uses an object of the `QVTKWidget` class [VTK17]. This widget is connected to an object of the `PCLVisualizer` class [PCL17c] in such a way that the presentation of the visualizer is seen in the widget and the user interactions with the widget are forwarded to the visualizer. The `PCLVisualizer` class provides methods for visualizing three-dimensional data, e.g. point clouds, coordinate frames, and shapes.

The `DefinitionGUI` provides the following methods for visualization:

- `showCloud` adds a white point cloud to the visualizer; it is used by the `DefinitionDirector` to render the original point cloud.
- `showColoredCloud` adds a red point cloud to the visualizer; it is used by the `DefinitionDirector` to render the filtered point cloud.
- `showObject` renders an object by adding the coordinate frame that represents the center of the object and the bounding box of the object to the visualizer.
- `clearRenderWindow` removes all point clouds, shapes, and coordinate frames from the visualizer; it is used by the `DefinitionDirector` to clear the visualizer before adding new data to it.

The major limitation of the chosen implementation is that there are only two colors to choose from for rendering point clouds. This is sufficient when, as done at the time of writing, only the original and the completely filtered point clouds are rendered. Nevertheless, in the more general case of having multiple filters, being able to render the result of each filter in a different color would be useful.

2D

To enable the user to set the parameters of a 2D object recognition system appropriately, the completely filtered image with the detected objects drawn into it is rendered by the GUI. Drawing the polygons that represent the detected objects into the filtered image is done with OpenCV's `cv::polylines` function [Kae17, pp. 163–164]. Rendering these images, which are passed to the `DefinitionGUI`'s `showImage` method as objects of type `cv::Mat`, is done by copying their contents into a `QImage` widget [Qt17e].

Investigating the needs of the feature detector reveals some additional requirements. It should be possible to select a rectangular area within the image by using the mouse. While the selection is active, i.e. as long as the user is pressing the mouse button, the selection should be visualized as a rectangle that is drawn into the image. When the user releases the mouse button, thereby committing the selection, the selection has to be passed to the feature detector.

To facilitate the task, a widget class called `InteractiveImage` was created. This class inherits from `QWidget` and has a `QImage` object as a data member. It provides a `showImage` method that takes an OpenCV image, copies its contents into the `QImage` member, and paints the `QImage`. In addition, it reacts to mouse events in various ways. First, it emits Qt signals when the mouse is pressed, moved, or released. Second, it visualizes active area selections by drawing a rectangle that represents the active selection into the image that is passed to its `showImage` method. Third, when the selection has been completed, it emits a signal called `areaSelected` and passes the selected area to any slots that are connected to that signal.

The `DefinitionGUI` has a data member of type `InteractiveImage`. To display an image, the `DefinitionGUI` forwards the image to the `showImage` method of the widget. To react to area selections, the `DefinitionGUI` has a slot that is connected to the `areaSelected` signal of the widget. In addition, it maintains a vector of callback functions; each of these functions is invoked from within the slot and takes the selected area as its argument. Area selections can be processed by a `FeatureDetector` object by passing its `processAreaSelection` method to the `registerAreaSelectionCallback` method of the GUI, thereby adding the function to the vector of callback functions.

Besides being able to react to user interactions, the creation of the `InteractiveImage` widget yields additional advantages. First, all details of dis-

playing an OpenCV image in a Qt widget have been encapsulated in that class, keeping the implementation of the `DefinitionGUI` focused. Second, as described in section 9.4.1, the `InteractiveImage` is of general utility, i.e. reusable by other applications.

The major drawback of the chosen solution is that only the completely filtered image is displayed. It would be useful to also see the original image and the results of each filter.

Chapter 9

ROS-based Solution

This chapter presents an alternative way of dividing object recognition systems into modules. Similar to the library solution, two frameworks were created, one for 2D and one for 3D object recognition systems. Since the fundamental ideas of both are the same, this chapter describes as many aspects as possible in general terms.

The first section of this chapter describes the fundamental ideas of this approach. After that, some general design decisions are explained, namely how filter and detector modules are implemented, and how parameters are managed in a uniform way. Then, modules that are specific to 2D or 3D systems are described. At the end of the chapter, it is shown how the three systems that serve as examples have been integrated into the framework. Along the way, it is described how the framework is used for creating parameter files and publishing detected objects based on existing parameter files.

9.1 Fundamental Ideas

The objective of the approach described in this chapter is the same as the objective of the library solution described in the previous chapter. Object recognition systems are divided into modules to obtain a set of modules and a general arrangement of these modules that can be used by a large number of object recognition systems. The difference compared to the library solution is that each module is represented by a node, not by an object of a class.

Figure 9.1 shows a ROS graph of a general object recognition system.

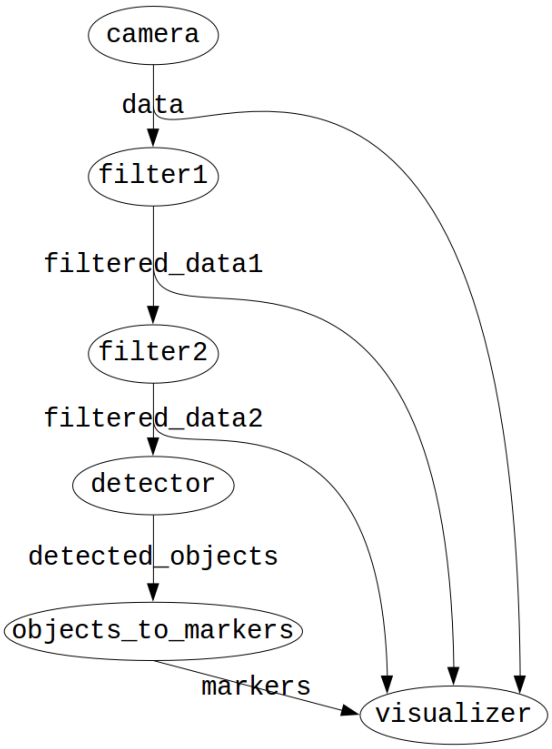

Figure 9.1: ROS graph of a general object recognition system

The system comprises six nodes, which communicate via ROS topics. The `camera` node publishes data over the `data` topic. The data represents images in 2D systems and point clouds in 3D systems. In general, the data is processed by an arbitrary number of filters. In the graph of figure 9.1, the data is first filtered by the node `filter1`, which publishes the filtered data on the topic `filtered_data1`. This data is then processed by the node `filter2`, which publishes its result on `filtered_data2`. After having passed all of the filters, the data is processed by the detector node, called `detector` in figure 9.1. If any objects are detected, they are published by the `detector` as `DetectedObject2DArray` or `DetectedObject3DArray` messages (see section 5.2) on the `detected_objects` topic. Other nodes can subscribe to the results of the filter and detector nodes. As an example, the graph in figure 9.1 contains the `objects_to_markers` and `visualizer` nodes. The `objects_to_markers` node publishes markers that represent the detected objects it receives; these markers can be rendered by the `visualizer`. The

`visualizer` renders the original data, the filtered data, and the markers.

One ideal is to keep each module minimal, i.e. each node should do only one thing. This yields maximum reusability of modules. Secondary tasks can be encapsulated in general-purpose helper modules, reusable by a large number of object recognition systems. Examples for such helper modules are the `objects_to_markers` and `visualizer` nodes in figure 9.1.

In addition to providing a set of reusable modules, the framework suggests a general structure for all systems by specifying interfaces for specific kinds of modules. As an example, all detector nodes in 3D systems subscribe to messages of type `sensor_msgs/PointCloud2` and publish messages of type `object_detection_3d_msgs/DetectedObject3DArray`; this makes these nodes interchangeable. This convention, together with the ability to remap the names of topics that nodes subscribe and publish to, makes it possible to create new systems from existing modules.

As mentioned in the introduction of this chapter, there are actually two frameworks, one for 2D systems and one for 3D systems. This means that modules written for one of the two groups are normally not usable in the other group of systems.

In contrast to the previous solutions, there are no seperate programs for creating parameter files and for detecting objects based on such files. In both cases, the fundamental modules of the system are launched, including filters and detectors. When creating parameter files, additional nodes for visualization and for setting and saving parameters are launched. When detecting objects based on existing parameter files, nodes for loading the parameters from these files are launched instead. Instructions for doing both tasks are presented in section 9.3.

9.2 Filter and Detector Nodes

As described in the previous section, object recognition systems contain filter and detector modules, and modules of the same kind have the same interface. As a consequence, all filter and detector nodes need to receive and publish messages of certain types and have to do conversions between these message types and corresponding types that are used internally. This section describes the design that encapsulates these common tasks as well as the common interface, thereby making it more explicit.

The basic idea is to seperate the ROS communication within filter and

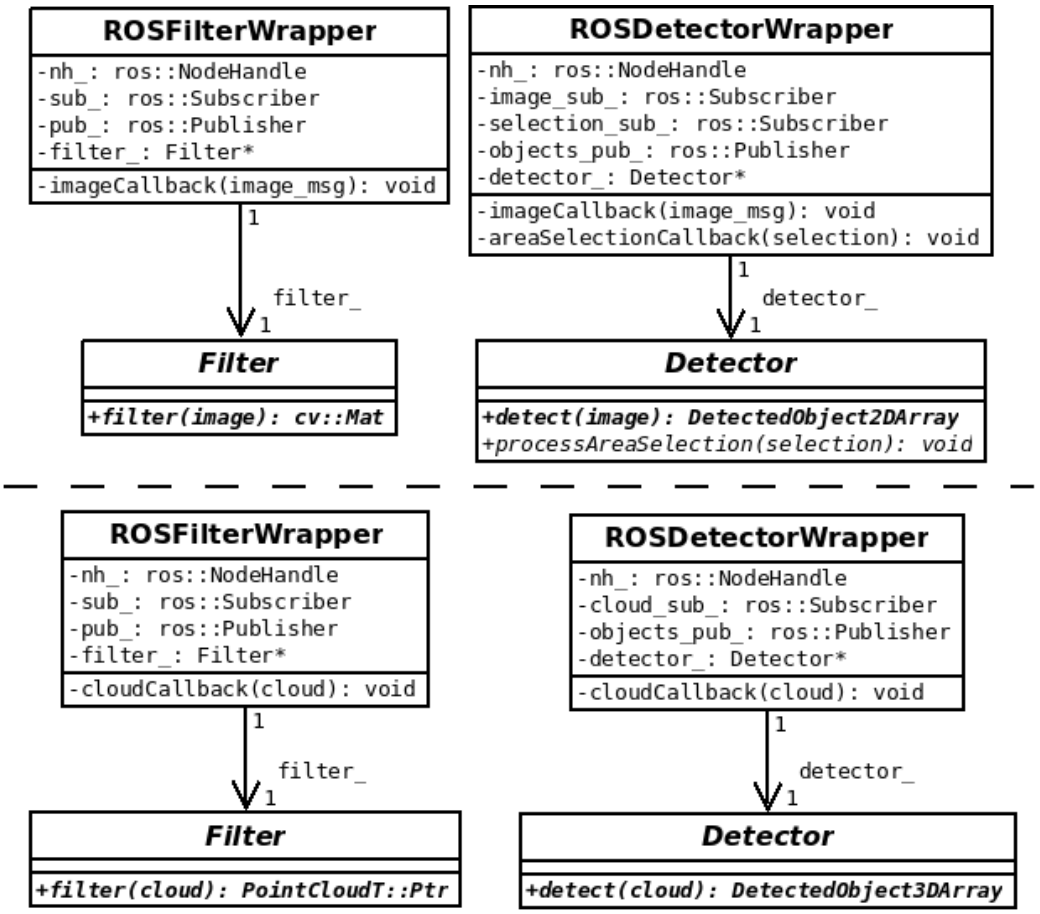

Figure 9.2: Classes that are used in filter and detector nodes for 2D (top) and 3D (bottom) systems

detector modules from their core functionality. The ROS communication is encapsulated in classes called `ROSFilterWrapper` and `ROSDetectorWrapper`, while the core functionality is represented by abstract classes called `Filter` and `Detector`. All of these classes are defined in the `object_detection_2d_nodes` and `object_detection_3d_nodes` packages. The relationships between these classes are depicted in figure 9.2.

A `ROSFilterWrapper` object has a reference to a `Filter` object. It provides a callback function, which is called `imageCallback` or `cloudCallback`, to receive messages. Within the callback function, the contents of a message are passed to the pure virtual `filter` method of the `Filter` object, and the

returned data is published. If necessary, conversions are done in between these steps.

A `ROSDetectorWrapper` object behaves similarly, but it keeps a reference to a `Detector` object and passes data to the pure virtual `detect` method of that object. As already discussed in previous chapters, the feature detector needs to receive rectangular selections within the current image to compute target keypoints and descriptors. To this end, the `ROSDetectorWrapper` of the 2D framework receives such selections in its `areaSelectionCallback` method and passes them to the `processAreaSelection` method of the Detector object. The latter method is implemented as an empty method so that only detectors that need this information have to override it. This mechanism is discussed further in section 9.4.1.

Major advantages of this design are a better seperation of concerns and the reusability of the `*Wrapper` classes. In addition, the `Filter` and `Detector` classes become relatively independent of ROS. The reduction of efficiency that is caused by the additional redirections is small because internally, data can be passed by reference and returned using move semantics.

9.3 Parameter Management

In general, all modules of a system have parameters and have to provide ways for monitoring, changing, saving, and loading these parameters. More precisely, it should be possible to save the whole state of a module in a file so that the state can be restored later by loading that file. Again, the ideal is to encapsulate as many of those common tasks into generally usable facilities like nodes or classes. This increases the uniformity among different object recognition systems and reduces the development effort for new modules.

9.3.1 Alternatives

Looking at how parameters are managed in the library solution, a natural approach is to use a centralized parameter management. This could be implemented as a single node that provides a central user interface for monitoring, setting, saving, and loading the parameters of a complete system. When saving the parameters to disk, this node would gather the parameters of the system and write them into a file. When loading the parameters from disk, the node would read the parameters from such a file and distribute them

among the nodes of the system. Since the components of the system are seperate nodes, not just objects within a single node, this approach bears some difficulties:

- How does the node that manages the parameters communicate with the other nodes in the system?
- How can the view that this node provides on the parameters be kept accurate at all times?
- Which node is responsible for checking that the values that the user tries to set are valid?
- How are parameters of different types handled without knowing the number and types of the parameters in advance?

Therefore, this approach was abandoned for the time being.

The approach on the other end of the spectrum decentralizes the parameter management by letting each module manage its own parameters and provide its own interface for accessing them. A major advantage is that each node knows the meaning of its parameters. Therefore, handling parameters of different types and checking the validity of each parameter are trivial. The major disadvantages of this approach are reduced uniformity among modules of the framework and an unnecessary amount of work for implementing new modules.

9.3.2 The Chosen Solution

The chosen solution is based on three existing ROS facilities, the ROS parameter server and the `dynamic_reconfigure` and `rqt_reconfigure` packages.

The ROS parameter server is in the same process as the ROS master and maintains a dictionary of key/value pairs, in which the keys are strings and the values can be of a variety of types, including `bool`, `int`, `double`, and `string`.[Qui15, pp. 361–362][ROS17d] The ROS parameter server is used as an interface for the parameters of the object recognition modules. This means that a set of ROS parameters corresponds to a set of variables within the modules, and both sets are synchronized. Using the ROS parameter server in this way makes it possible to create nodes that access the parameters of all modules in a system.

Using `dynamic_reconfigure` [ROS17f] allows to synchronize variables within a node with a corresponding set of ROS parameters. To use it, a configuration file has to be written that contains a description of this set of ROS parameters, including their names, types, default values, and ranges. Assuming that this file is called `Parameters.cfg`, executing it will generate a header file called `ParametersConfig.h`. Including this header file in the source code of a node makes it possible to register a callback function that is called whenever one of the ROS parameters defined in `Parameters.cfg` changes. The parameters are passed to that callback function by reference so that the function can read and modify their values.

Lastly, `rqt_reconfigure` [ROS17g] provides a GUI for monitoring, setting, saving, and loading the parameters of active `dynamic_reconfigure` servers. This GUI serves as a common interface for adjusting the parameters of object recognition modules and for saving their parameters to disk.

All of these facilities are of high quality, since they are used by a major part of the ROS community. Using them fits the overall philosophy of the ROS-based solution well. The common task of parameter management has been partially moved out of the modules. This corresponds to a better seperation of concerns, to uniform user interfaces among modules, and to a reduction of the development effort that the creation of new modules requires.

A limitation of this approach is that only values of specific types can be managed with `dynamic_reconfigure`. These types are `bool`, `int`, `double`, and `string`.[ROS17e] If the state of a module includes more complex data, storing this data as part of the parameter files is tricky. An example of this can be found in the implementation of the feature detection system, discussed in section 9.5.2.

9.3.3 A Minimal Example

A minimal example should clarify how the modules of the framework use `dynamic_reconfigure`. The example is a node called `my_module` that resides in a package called `my_package` and that has one parameter of type `int`, called `my_module/my_param`.

As a prerequisite, the configuration file in listing 9.1 has to be created and included into the `CMakeLists.txt` [ROS17h] file of the package.

Listing 9.1: `Parameters.cfg` for configuring the parameters of the `my_module`

node from the `my_package` package

```
#!/usr/bin/env python

from dynamic_reconfigure.parameter_generator_catkin import *

gen = ParameterGenerator()
gen.add("my_param", int_t, 0,
  "An integer parameter with default value 3, " + \
    "minimum value 0, and maximum value 10",
  3, 0, 10)
exit(gen.generate("my_package", "my_module", "Parameters"))
```

This configuration file defines the parameter `my_module/my_param`, which has a default value of 3, a minimum value of 0, and a maximum value of 10. This Python script generates a header file called `ParametersConfig.h`.

The source code of the `my_module` node is listed in listing 9.2.

Listing 9.2: Source code of the `my_module` node from `my_package`

```
#include <ros/ros.h>
#include <dynamic_reconfigure/server.h>
#include <my_package/ParametersConfig.h>

using namespace std;
using my_package::ParametersConfig;
using dynamic_reconfigure::Server;

class MyModule {
public:
  MyModule()
  {
    callback_ = bind(&MyModule::myCallback, this,
                     std::placeholders::_1, std::
                       placeholders::_2);
    server_.setCallback(callback_);
  }

```

```
18 private:
19   void myCallback(ParametersConfig& config, uint32_t
       )
20   {
21     my_param_ = config.my_param;
22   }
23
24   dynamic_reconfigure::Server<ParametersConfig>
       server_;
25   dynamic_reconfigure::Server<ParametersConfig>::
       CallbackType callback_;
26
27   int my_param_;
28 };
29
30 int main(int argc, char** argv)
31 {
32   ros::init(argc, argv, "my_module");
33   MyModule module;
34   ros::spin();
35   return 0;
36 }
```

The `main` function creates an object of the `MyModule` class before waiting for ROS events in the event loop, which is represented by the call to `ros::spin()`. The constructor of the object initializes a `dynamic_reconfigure` server for the previously created configuration called `ParametersConfig` and registers the method `myCallback` with it. The `MyModule` object has one data member called `my_param_`, which is synchronized with the ROS parameter called `my_module/my_param`. Whenever the value of the ROS parameter `my_module/my_param` changes, the callback function is invoked. This function sets the data member `my_param_` to the same value as the ROS parameter. Optionally, the value of the ROS parameter could be modified in the callback function by assigning a value to `config.my_param`.

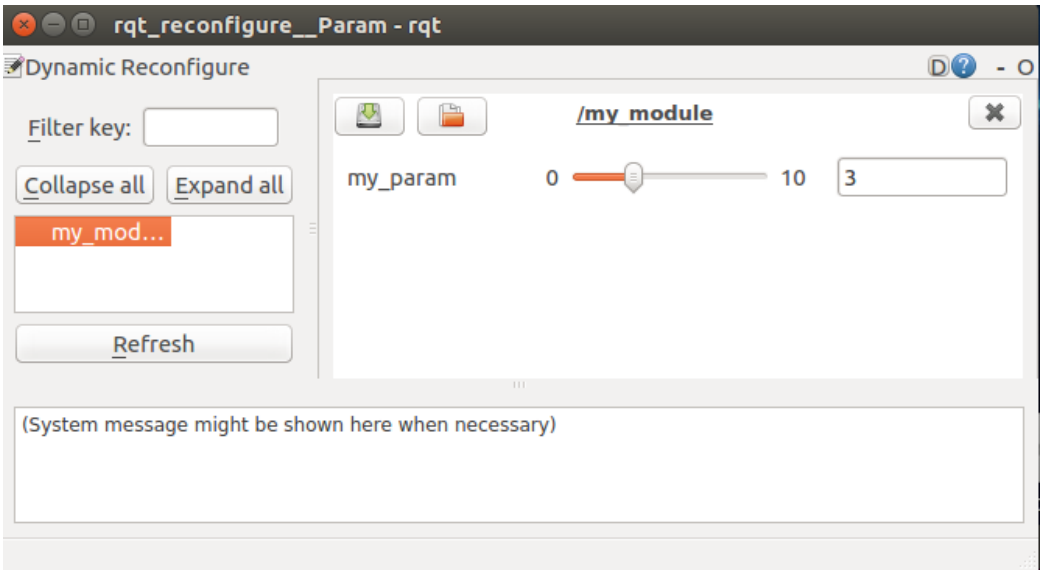

Figure 9.3: `rqt_reconfigure` GUI for managing the parameters of the `my_module` node

9.3.4 Using the System

This section uses the previous example to give instructions on how to create parameter files for a module and how to load such files when launching the module.

First, the `rqt_reconfigure` node from the equally named package is launched. This opens the GUI shown in figure 9.3. If `my_module` is running, its `dynamic_reconfigure` server can be selected from the left panel of the GUI. The right panel of the GUI provides widgets for accessing the parameters of the selected servers. It also contains two buttons for each server: the button on the left is used to save the current parameter values of the server, the button on the right is used to load the parameters of the server from such a file. The parameter files for an object recogition system are created by setting the parameters of all modules to the desired values first and then saving the parameters of each module to a seperate file.

The launch file in listing 9.3 is used to launch the `my_module` node and to load its parameter from a previously created parameter file; the file is called `params.yaml` and is located in the `params` folder of the `my_package` package.

Listing 9.3: File for launching the `my_module` node and loading its parameters

```
<launch>
  <node name="dynamic_reconfigure_load"
    pkg="dynamic_reconfigure"
    type="dynparam"
    args="load /my_module
      $(find my_package)/params/params.yaml" />
  <node name="my_module"
    pkg="my_package"
    type="my_module" />
</launch>
```

Besides launching the `my_module` node, this file launches the `dynparam` node of the `dynamic_reconfigure` package with the `load` command and passes the path of the parameter file to it. The parameters defined in that file will be loaded into the `/my_module` namespace, which is passed to the `dynparam` node as another command-line argument. This will assign the value found in `params.yaml` to the ROS parameter `/my_module/my_param`. If an object recognition system contains multiple modules, one `dynparam` instance is launched for each of them.

9.3.5 Encapsulating the Parameter Management within Nodes

To seperate the parameter management from the core functionality of a module, the same pattern is used for all modules; the parameter management of a module is encapsulated in a seperate class, normally called `ParameterServer`. Similar to the `MyModule` class in listing 9.2, this class has a `dynamic_reconfigure::Server` object and the internal variables as data members, and it defines the callback function that is registered with the server object. The internal variables are synchronized with corresponding `dynamic_reconfigure` parameters and can be accessed by calling public getter methods.

The core functionality of a module is encapsulated in a seperate class. An object of this class has a `ParameterServer` object as a data member, and it accesses the parameters of the module by calling the getter methods of that member.

Applying this pattern to the minimal example leads to the source code in listing 9.4.

Listing 9.4: Source code of the `my_module` node after encapsulating the parameter management

```
1  #include <ros/ros.h>
2  #include <dynamic_reconfigure/server.h>
3  #include <my_package/ParametersConfig.h>
4
5  using namespace std;
6  using my_package::ParametersConfig;
7  using dynamic_reconfigure::Server;
8
9
10 class ParameterServer {
11 public:
12   ParameterServer()
13   {
14     callback_ = std::bind(&ParameterServer::
         myCallback, this,
15                           std::placeholders::_1, std::
                                placeholders::_2);
16     server_.setCallback(callback_);
17   }
18
19   int myParam() const { return my_param_; }
20
21 private:
22   void myCallback(ParametersConfig& config, uint32_t
       )
23   {
24     my_param_ = config.my_param;
25   }
26
27   dynamic_reconfigure::Server<ParametersConfig>
       server_;
28   dynamic_reconfigure::Server<ParametersConfig>::
       CallbackType callback_;
29
30   int my_param_;
```

```
31 };
32
33 class MyModule {
34 public:
35   void printMyParam() const
36   {
37     cout << "The value of my parameter is " <<
          param_server_.myParam()
38          << '\n';
39   }
40
41 private:
42   ParameterServer param_server_;
43 };
44
45 int main(int argc, char** argv)
46 {
47   ros::init(argc, argv, "my_module");
48   MyModule module;
49   ros::spin();
50   return 0;
51 }
```

In this case, the core functionality, printing the value of the parameter, is encapsulated in the `MyModule` class. The created object of this class has a `ParameterServer` object as a data member, which manages the parameter of the module. The value of that parameter is only changed within the callback function `myCallback`, and it is accessed by the `MyModule` object by using the `myParam` method of the `ParameterStorage` member.

9.4 2D and 3D Modules

This section covers general-purpose modules that have been developed for 2D or 3D object recognition systems. These nodes mainly correspond to the visualization of images or point clouds and detected objects.

9.4.1 2D Modules

We prefer to use `rviz` [Qui15, pp. 126–133][ROS17i], a general-purpose visualization tool that is part of ROS, for all of our visualization tasks. Unfortunately, `rviz`'s capabilities for visualizing 2D data are very limited. `rviz` provides a display for rendering images that are published on a specific topic, but it provides no features for interacting with that display. It also does not provide any features for displaying shapes within the image, as would be needed for rendering polygons that represent detected objects. Therefore, custom visualization tools were developed for the 2D framework.

`object_painter`

The `object_painter` node, which resides in the `object_painter_2d` package, is responsible for drawing polygons that represent detected objects into images. To this end, the node subscribes to images and to detected objects. It has a parameter called `max_age_s` and it maintains a vector of the received detected objects whose age in seconds, according to the time stamp in their message header, does not exceed the value of the parameter. Whenever the node receives an image, it draws the polygons of the objects contained in its vector into the image before publishing the modified image.

`visualizer`

As already mentioned several times in this document, the feature detector requires that the user can select a rectangular area within an image, from which target keypoints and target descriptors are computed. Preferably, the user can do so by drawing a rectangle into the rendered image with his mouse.

For this purpose, two additional modules were created. The first one is the `visualizer` node from the `object_detection_2d_vis` package. This node displays the `InteractiveImage` widget, which has already been described in section 8.5.2, in a window. As mentioned before, the `InteractiveImage` widget emits a signal when the mouse is pressed, moved, or released, and it draws a rectangle into the image that represents an active area selection. Corresponding slots in the `visualizer` node react to these signals by publishing messages of type `object_detection_helpers/MouseEvent`. The definition of this type is listed in appendix B. The information in such a message includes which button has been used, the type of the event (pressed, moved, or

released), and the coordinates of the mouse cursor in the coordinate system of the image.

`area_selection`

To keep the `visualizer` node minimal, it only publishes primitive mouse events. The `area_selection` node from the same package subscribes to these events and publishes rectangles that represent area selections that have been made with the mouse. Such a selection begins when the user presses the left mouse button and is completed when this button is released. The published rectangle has the start and end coordinates as its vertices. The message type for these rectangles is `object_detection_helpers/Rect2D`. Section 9.2 describes the mechanism that detector modules use to process this information.

9.4.2 3D Modules

Similar to the 2D framework, the 3D framework should provide tools for displaying the original and filtered point clouds and the detected objects.

In this case, `rviz` can be used for all visualization tasks. `rviz` already provides displays for rendering point clouds. To be able to display detected objects, i.e. messages of type `object_detection_3d_msgs/DetectedObject3DArray` (see section 5.2), the `objects_to_markers` node from the equally named package was created. This node subscribes to detected objects and publishes messages of type `visualization_msgs/MarkerArray`. Each message contains an array of markers; each marker represents the bounding box of a detected object. `MarkerArray` is a standard message type of the ROS framework, and `rviz` provides a display for visualizing messages of this type.

A screenshot of `rviz` visualizing the data of the shape detection system is shown in figure 9.9.

9.5 Implemented Object Recognition Systems

This section describes how each of the three object recognition systems that are used as examples throughout this document is integrated into the ROS-based framework. For each system, it is described which nodes are used and how these nodes interact. If a system has any peculiarities, they are mentioned as well.

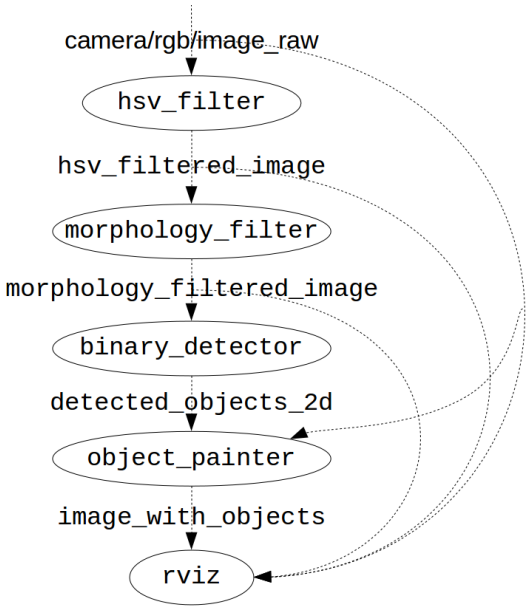

Figure 9.4: ROS graph of the HSV detection system

In principal, the described modules do the same thing and have the same parameters as the corresponding, and similar named, modules in the library solution. The major difference is that instead of being an object that provides some method, each module is a node that subscribes to at least one topic and typically responds to an incoming message by publishing the result of a computation.

9.5.1 HSV Detection

The ROS graph of the HSV detection system is shown in figure 9.4. As already done for the library solution, the system was divided into the `hsv_filter`, the `morphology_filter`, and the `binary_detector` modules. These nodes reside in the `hsv_filter_node`, `morphology_filter_node`, and `binary_detector_node` packages, respectively. The images of the camera are first filtered by the `hsv_filter`, then by the `morphology_filter`. The `binary_detector` publishes objects that it detects in the filtered images. The `object_painter` draws the polygons that represent these objects into the original image and publishes the result over the `image_with_objects` topic. The filtered images and the original image with the objects drawn into it are rendered in

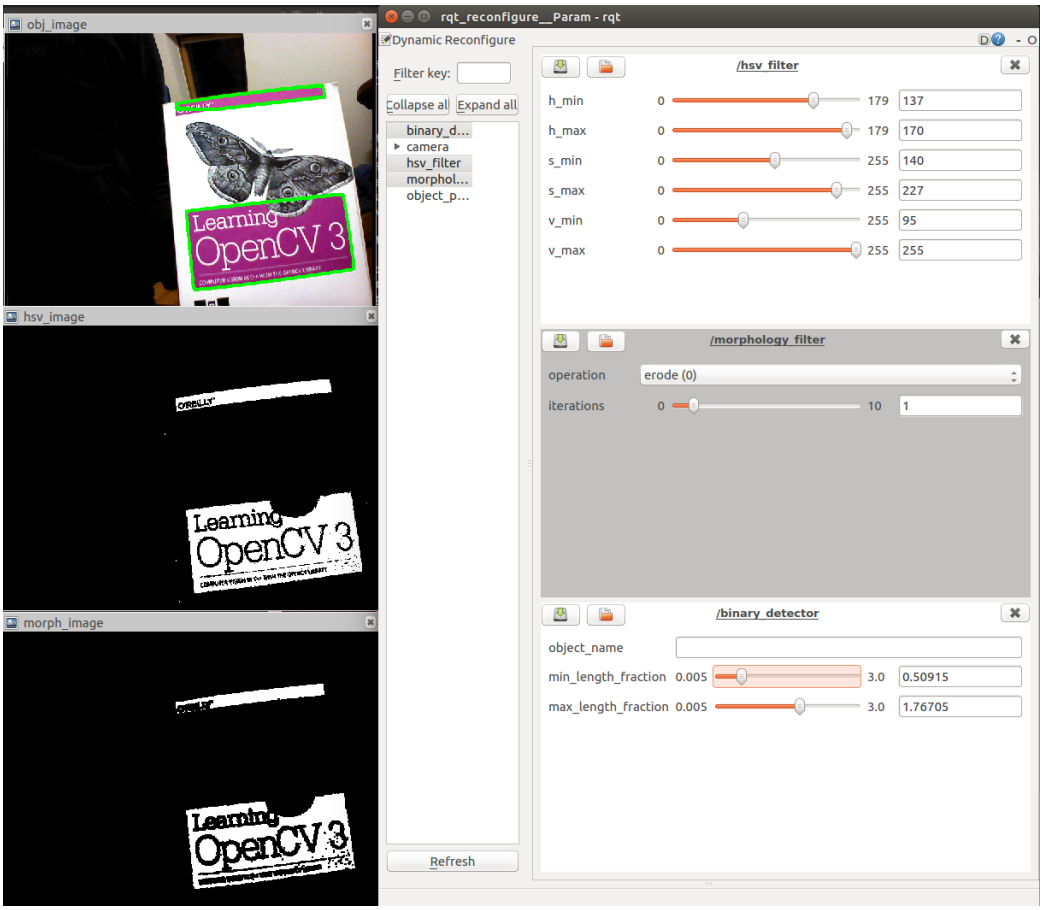

Figure 9.5: `rviz` and `rqt_reconfigure` provide the interface to the HSV detection system

`rviz`.

Figure 9.5 shows `rviz` and the `rqt_reconfigure` GUI after the system has been set up. Creating the parameter files for the system and launching it with existing parameter files can be done by following the general instructions from section 9.3.4.

9.5.2 Feature Detection

The feature detector is implemented as the `feature_detector` node from the `feature_detector_node` package.

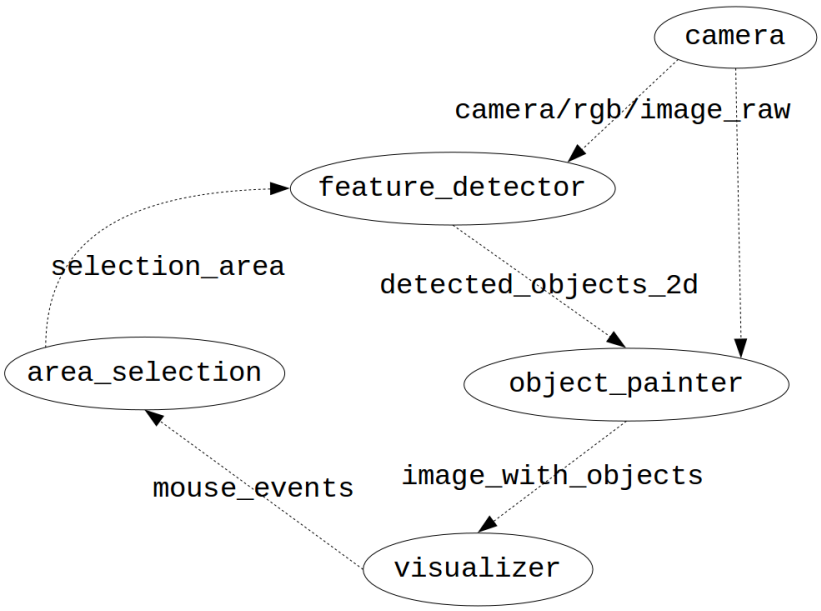

Figure 9.6: ROS graph of the feature detection system

ROS Graph

The ROS graph of the feature detection system is shown in figure 9.6. As described in section 9.4.1, the `visualizer` renders images and publishes mouse events that occur in the widget. The `area_selection` node subscribes to these mouse events and publishes rectangles that represent selected areas within the image. The `feature_detector` node subscribes to these rectangles as well as to the images of the camera. It publishes detected objects, and the `object_painter` node uses these messages to draw the object polygons into the original images. The modified images, published on the `image_with_objects` topic, are the images that are rendered by the `visualizer`.

Parameter Management

As already discussed, saving the state of the `feature_detector` requires to save the rectangle that represents the target and the image from which the user selected the target. In section 9.3, it was decided to use `dynamic_reconfigure` for saving and loading the parameters of all modules to the greatest

extent possible, but saving and loading images on the ROS parameter server is not possible.

One solution to this problem would be to use `dynamic_reconfigure` only for a part of the data that constitutes the state of the `feature_detector`. The detector would then have to provide some other interface for saving and loading the data that corresponds to the selection of the target, e.g. in the form of ROS services (see chapter 4). This solution was abandoned because it does not fit the overall philosophy of the ROS-based solution and the general strategy for managing parameters.

In the chosen solution, the target selection is saved in a seperate file, and saving and loading it are done with a `cv::FileStorage` object (see section 8.5.1) within the `feature_detector` node. Nevertheless, `dynamic_reconfigure` and `rqt_reconfigure` are still used as an interface for triggering these processes. This was implemented by adding three `dynamic_reconfigure` parameters to the `feature_detector` module. The first one is a string that represents the path of the file to which the selection data is saved. The second one is a bool called `feature_detector/save_selection`. When this parameter is set to true, the `feature_detector` saves its current selection to the file of the given name; then it resets the parameter to false. The third one, called `feature_detector/load_selection`, has a similar role. When it is set to true, the `feature_detector` loads the selection data from the given file before resetting the parameter.

Figure 9.7 shows the `rqt_reconfigure` GUI for managing the parameters of the `feature_detector`. To save the state of the `feature_detector`, the following steps have to be executed:

1. Enter the absolute path of the file to which the selection data should be saved into the field labeled `filename`.

2. Check the `save_selection` check box. The detector will save the selection data to the file with the given path, and it will reset the parameter so that the checkbox is unchecked again.

3. Press the normal save button to store the `dynamic_reconfigure` parameters in a file of your choice.

The steps for loading a state of the `feature_detector` from the `rqt_reconfigure` GUI are similar:

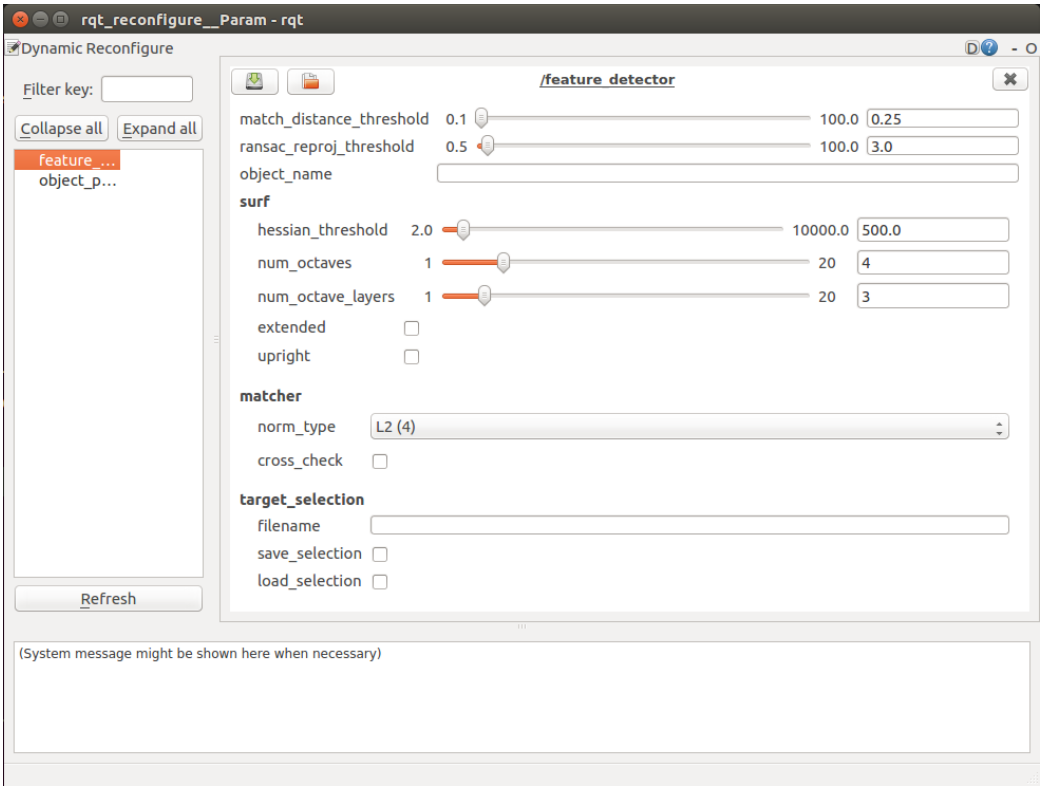

Figure 9.7: `rqt_reconfigure` GUI for managing the parameters of the `feature_detector` node

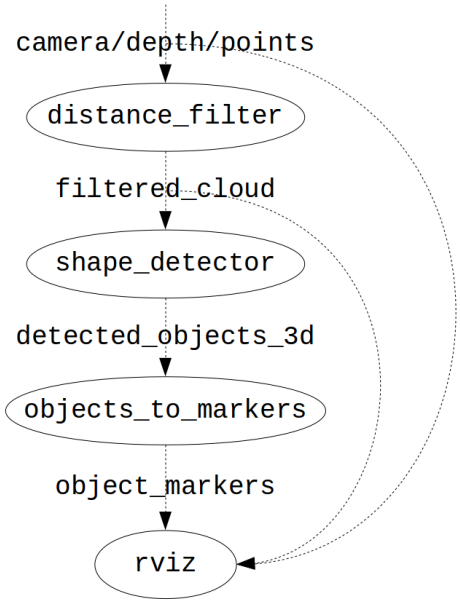

Figure 9.8: ROS graph of the shape detection system

1. Press the load-button to load the `dynamic_reconfigure` parameters from a file. This will also load the path of the file with the selection data.

2. Check the checkbox labeled `load_selection`. Again, the `feature_detector` will reset that boolean parameter after having loaded the selection data from the file with the given path.

As usually, to load the `dynamic_reconfigure` parameters when launching the node, the `dynparam` node with the `load` command is included in the same launch file, as described in section 9.3.4. To load the selection data as well, the parameter file that is passed to the `dynparam` node has to be manually edited by changing the value of the `load_selection` parameter to `true`.

9.5.3 Shape Detection

The graph of the shape detection system is shown in figure 9.8. As already done for the library solution, the system was divided into the `distance_filter` and the `shape_detector` modules. These nodes reside in the `distance_filter_node` and `shape_detector_node` packages, respectively. The point

cloud from the camera is filtered by the `distance_filter`. The `shape_detector` publishes objects that it detects in the filtered point cloud. As described in section 9.4.2, the `objects_to_markers` node publishes markers that represent the bounding boxes of the detected objects. `rviz` visualizes the original point cloud, the filtered point cloud, and the markers.

Figure 9.9 shows `rviz` and the `rqt_reconfigure` GUI after the system has been set up. The original point cloud is rendered in white, the filtered point cloud in red. The markers are rendered as green transparent boxes. Creating the parameter files for the system and launching it with existing parameter files can be done by following the general instructions from section 9.3.4.

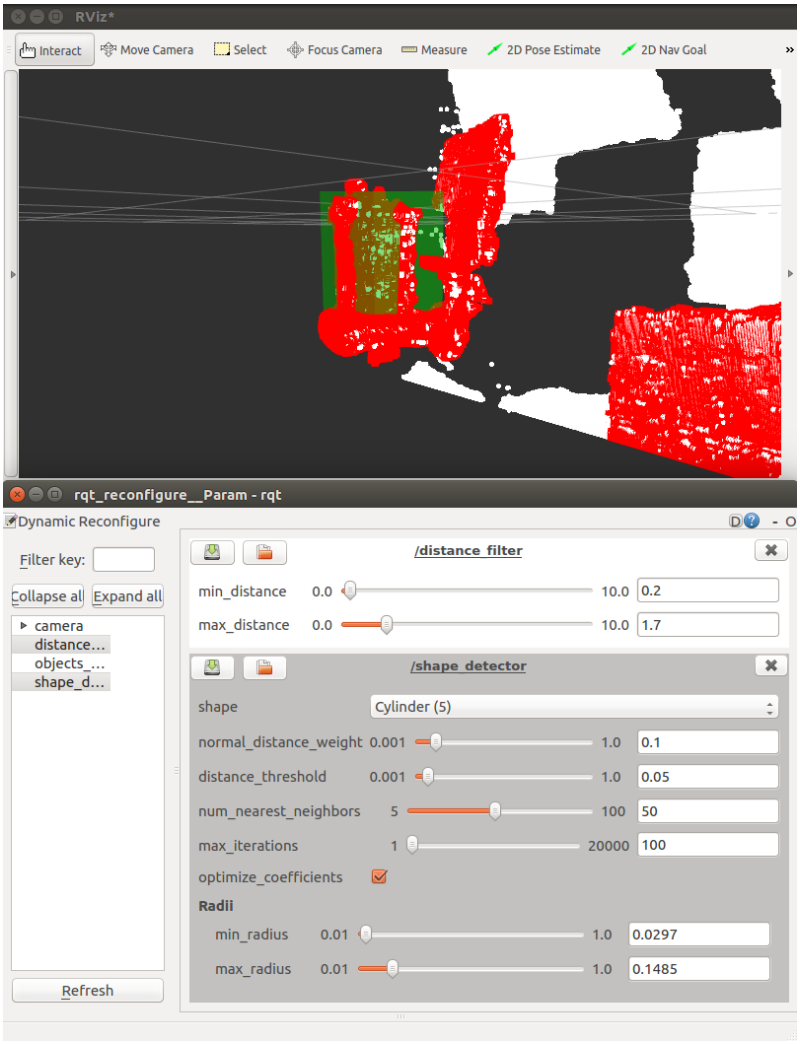

Figure 9.9: `rviz` and `rqt_reconfigure` provide the interface to the shape detection system. The detected object is a cylindrical garbage can on an office chair.

Chapter 10

Evaluation of the Solutions

While the previous three chapters mainly discribe each of the three solutions, this chapter evaluates and compares them. A recommendation for which solution should be pursued further is given at the end of the chapter.

Table 10.1 summarizes the pros and cons of each solution.

Individual Programs

An individual program is not subject to constraints of any framework. This gives the developer of the program full freedom, enabling him to make it fit the specific object recognition system perfectly. Among other things, this results in a highly specialized user interface and in maximum efficiency. The disadvantage of the individual programs is that each system is created from scratch. Not having any reusable modules, leads to higher development effort for new systems and to a lack of uniformity among systems.

Commonalities between the Two Frameworks

The other two solutions are modular frameworks. Being able to reuse existing modules reduces the effort to create new systems. Encapsulating commonly needed features in general-purpose modules also reduces redundancy of code and the effort needed for developing new modules. Reusing modules also justifies to put more effort into their creation, leading to higher quality modules.

On the other hand, creating a framework and adapting it to unforeseen use cases while it is still premature takes a lot of effort in itself. In addition,

Individual Programs	**Library Solution**	**ROS-based Solution**
+ no constraints	+ modularity	
+ specialized UI	+ reusability of modules	
+ high efficiency	+ less development effort for new systems & modules	
		+ small & simple nodes
		+ integration of ROS tools
- no reusability	- initial development of framework	
- no uniformity	- constraints of framework	
- high development effort	- inflexible	- high latency
		- ROS dependency
		- asynchronism

Table 10.1: Pros (+) and cons (-) of each solution

the constraints of a framework lead to suboptimal systems. Conversions and redirections that are needed to satisfy the interfaces defined by the framework are one example. Another example is the abandonment of drawing matches between keypoints in the feature detection system.

Differences between the Two Frameworks

In contrast to the library solution, every system in the ROS-based framework comprises multiple simultaneously running nodes. This can lead to a noticeable latency. In addition, the asynchronism of the nodes can complicate matters; in the `object_painter` node (see section 9.4.1), this fact is compensated by storing the received objects up to a maximum age. On the other hand, the structure of the ROS-based framework reduces the complexity of each executable and weakens the dependencies among modules.

Creating new systems from existing modules is relatively easy in both frameworks. For the library solution, this is done by writing two nodes as described in section 8.2. In the ROS-based framework this is even simpler, since it only requires to write two launch files as indicated in section 9.3.4.

Both frameworks suggest a general structure for the systems that are implemented in them. For the library solution, this structure is relatively rigid. It is mainly encapsulated in the `DefinitionDirector` and `DetectionDirector` classes, and integrating new kinds of modules into the framework probably requires to modify these classes. An example of this is the fixed number of images that is rendered in the GUI of the 2D library. Changing this number

requires to modifiy the `DefinitionGUI` and the `DefinitionDirector` classes. The possibility to remap the names of topics makes the structure of the ROS-based framework less rigid. Examples for this are the `object_painter` and `area_selection` modules (see section 9.4.1), whose creation did not incur any modifications of the existing modules.

Lastly, the ROS-based framework facilitates the integration of existing ROS tools, e.g. `rviz`, `dynamic_reconfigure`, and `rqt_reconfigure`. This advantage is impeded by the fact that these tools do not always fit the needs of a particular system exactly. This can require the use of adapter modules such as the `objects_to_markers` node from section 9.4.2. The ROS-based framework is also highly dependent on ROS. In contrast to the library solution, adapting the framework to a usage outside of ROS would require a major rewrite.

Conclusion

Creating individual programs is not the primary goal of this project. Nevertheless, creating prototypes for new systems in this way turned out to be useful to gain a better understanding of these systems and to ease their integration into a framework.

The library solution and the ROS-based solution both have their merits. If we had to choose one, we would take the ROS-based solution. The major reason for that is that the simplicity of the executables makes this solution easier to understand and to use. Other reasons are the abovementioned higher flexibility and the possibility to include existing ROS tools into a system.

Apart from that, most disadvantages of the ROS-based solution are not crucial for this project. As long as the software is based on a TurtleBot, it will depend on ROS anyways. Latency is not a problem either, since the TurtleBot is only used as a platform for learning to write software.

Part III

Navigation

Chapter 11

Experimental Setup

Computers

When working with the TurtleBot, we actually use two computers. We use the term "notebook" for the one that is carried by the robot and controls it. The term "workstation" is used for the second, stationary PC, which is operated by the developer. Both computers run the same Ubuntu and ROS versions and are connected via a wireless network. `ssh` [Kirk17] is used to launch nodes that run on the notebook from the workstation. The ROS wiki describes in detail how this system is set up.[ROS17j]

In general, the nodes of a ROS system can be distributed among multiple computers. In this project, the complete ROS system comprises the nodes on the notebook and the nodes on the workstation, and the ROS master runs on the notebook.

Environment

A room of the Westphalian University of Applied Sciences is used as the test environment. Irrespective of that, the general mapping procedure described in chapter 12 and the navigation program described in chapter 14 can be used for arbitrary environments.

Chapter 12

Mapping

Having a map of an environment is a prerequisite for navigating in that environment. This chapter describes how to create such a map.

Maps are build with the `slam_gmapping` node from the `gmapping` package [ROS17k]. This node uses the GMapping algorithm, which is based on a Rao-Blackwellized particle filter.[Qui15, p. 141]

12.1 Creating the Map

To create the map, the following files are launched on the notebook:

- `minimal.launch` from the `turtlebot_bringup` package. This file primarily launches the drivers of the mobile base. Besides motors, the mobile base also comprises sensors for odometry measurements.
- `gmapping_demo.launch` from the `turtlebot_navigation` package. Among other things, this starts the drivers of the 3D sensor and the abovementioned `slam_gmapping` node.

On the workstation, the following files are launched:

- `view_navigation.launch` from the `turtlebot_rviz_launchers` package. This file launches `rviz`, set up for visualizing the sensor data and the current state of the map.
- `keyboard_teleop.launch` from the `turtlebot_teleop` package. This file launches a node that provides an interface for controlling the TurtleBot with the keyboard.

Optionally, sensor data can be recorded with the following command, preferably executed on the notebook to avoid transferring the data over the wireless network:

```
$ rosbag record –O data.bag /scan /tf /tf_static
```

This will save the messages of the `/scan`, `/tf`, and `/tf-static` topics to the file `data.bag` in the current working directory. This data can be replayed with the following command:

```
$ rosbag play —clock data.bag
```

Recording the data makes it possible to build the map multiple times from the same sensor data using different parameters for the `slam_gmapping` node in each iteration.[Qui15, pp. 143–144]

After the abovementioned files have been launched, the TurtleBot is slowly navigated around the world with the keyboard. The `slam_gmapping` node builds the map from the sensor data and publishes it on the `/map` topic, which is visualized by `rviz`. Once the map is complete, it is saved by running the `map_saver` node:

```
$ rosrun map_server map_saver
```

This command creates two files: `map.pgm`, which contains the map, and `map.yaml`, which contains metadata of the map. In this project, these files are called `b5203.pgm` and `b5203.yaml`[1] and are contained in the `maps` folder of the `whs_navigation` package.

12.2 Viewing and Editing the Map

Maps are represented as pgm files. Therefore, they can be viewed with any image viewing software. As an example, figure 12.1 shows the map that was build from the test environment. Open space is rendered as white areas in the image, occupied space as black areas, and unknown space as gray areas.

Similarly, maps can be edited with any image editing software. Obstacles that should not be in the map can be removed, and fake obstacles can be inserted; this will influence the path planning of the autonomous navigation.

[1] B5.2.03 is the name of the room that is used as a test environment.

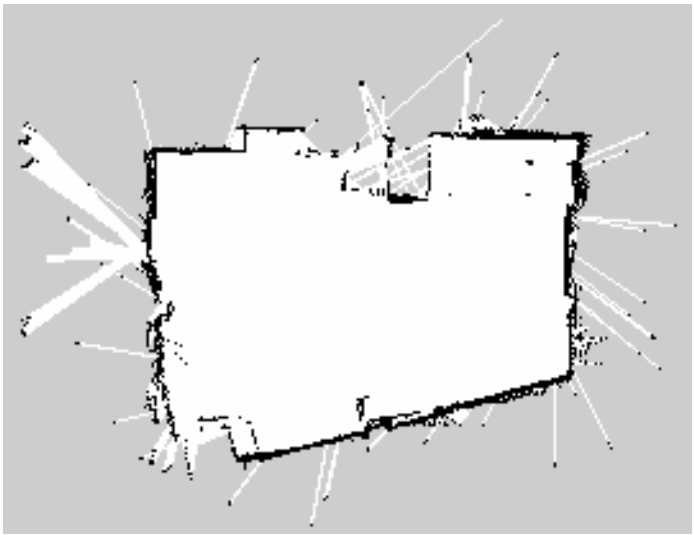

Figure 12.1: Map of the test environment

Chapter 13

Simulation and Determining Goal Poses

Testing the navigation program in a simulator is reproduceable and obviously quicker, cheaper, and safer than doing it in the real world. It also enables people without access to the robot or the environment to write and test navigation programs. The 2D simulator Stage [ROS17l] is used for that purpose because it is well-integrated into ROS.

13.0.1 Preparing the Map

An image, such as the map that has been created in the previous chapter, can be used as the basis for creating a simulation world; black pixels in the image represent walls in the simulation world. When creating the simulation world from the map, Stage trims the gray areas that surround the map (see figure 12.1). When this happens, the map and the simulation world have different sizes and unaligned coordinate frames.[Gav17] To avoid this, a thin border is drawn around the map. Again, this can be done with any image editing program.

13.0.2 Creating the World File

The world file is listed in appendix C. It has been created by modifying the existing world file `maze.world` from the publicly available `turtlebot_stage` package according to instructions from Gaitech [Gai17]. The only things that were modified in this file are the path of the image that represents the

simulation world, its size and pose, and the initial pose of the TurtleBot within that world.

13.0.3 Launching the Simulation and Navigating the Robot

The launch file that is used to launch the simulation is listed in appendix D. It includes the existing file `turtlebot_in_stage.launch` from the `turtlebot_stage` package and sets the arguments for the map file, the world file, and the initially estimated pose of the TurtleBot. The initially estimated pose is set to the same value as the actual initial pose of the TurtleBot, which is stated in the world file (see appendix C).

Besides starting the Stage simulator, this file launches the ROS navigation stack and `rviz`. The navigation stack comprises the following nodes:[Qui15, p. 308]

- `map_server` reads a map from a file and publishes it and its metadata.
- `amcl` localizes the robot and publishes the estimated pose of the robot over the `amcl_pose` topic.
- `move_base` receives goal poses, does the path-planning, and controls the robot.

Figure 13.1 shows a screenshot after launching the simulation. On the left is the Stage simulator showing the simulation world and the TurtleBot. On the right is `rviz` rendering the map and the TurtleBot at its estimated pose. Goal poses can be defined in `rviz` by clicking on the button labeled "2D Nav Goal" in the toolbar, and then clicking and dragging in the map. The goal pose is depicted as a red arrow, and the path that has been planned by `move_base` is depicted as a green line.

13.0.4 Determining Goal Poses

Besides testing the navigation program, the simulation can be used to determine the goal poses that are fed to the program.

The currently estimated pose can be monitored by printing the messages of the `amcl_pose` topic. The poses of the goals are determined by navigating

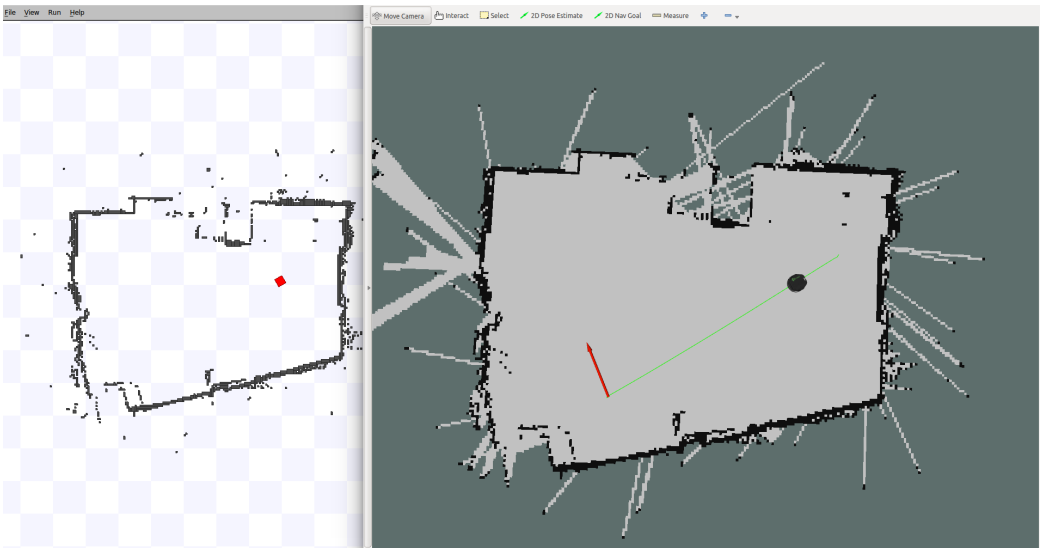

Figure 13.1: Stage simulation (left) and defining a goal pose in rviz (right)

the TurtleBot to each goal, and reading its estimated pose when the goal has been reached.

As required by the navigation program, each goal pose is written on a seperate line of a text file, using the following format:

$$pos_x, pos_y, pos_z, ori_x, ori_y, ori_z, ori_w$$

pos_i represents the i-coordinate of the position, ori_i represents the i-coordinate of the orientation. Orientations are represented by quaternions. The order of the poses in the file corresponds to the order in which the TurtleBot drives to them. Optionally, the pose that is roughly one meter in front of the docking station, prefixed with `station_pose:`, can be inserted into that file. When this pose is given, the TurtleBot will try to dock at the station when its battery is low. As an example, appendix E lists the text file that contains the goal poses of this project.

Chapter 14

The Navigation Node

14.1 Navigating in Code

The previous chapter mentioned that the `move_base` node is responsible for receiving and acting upon goal poses. Besides using `rviz`, sending goals to `move_base` can also be done in code. Listing 14.1 lists a minimal program, which sends one goal pose to `move_base`.

Listing 14.1: `minimal_navigation.py`, which sends one goal to the `move_base` action server

```
1  #!/usr/bin/env python
2
3  import rospy
4  from geometry_msgs.msg import Pose
5  import actionlib
6  from move_base_msgs.msg import MoveBaseAction,
       MoveBaseGoal
7
8
9  def createGoal():
10     goal = MoveBaseGoal()
11
12     goal.target_pose.header.frame_id = "map"
13     goal.target_pose.pose.position.x = 14.4
14     goal.target_pose.pose.position.y = 12.8
15     goal.target_pose.pose.orientation.w = 1.
```

```
16
17      return goal
18
19  if __name__ == "__main__":
20      rospy.init_node("minimal_navigation")
21      goal = createGoal()
22      client = actionlib.SimpleActionClient("move_base
            ", MoveBaseAction)
23      client.wait_for_server()
24      client.send_goal(goal)
25      client.wait_for_result()
```

A pose is created, its non-zero components are set, and it is send to the `move_base` action server as soon as the server is active. The program exits when the server sends back a result, i.e. when the goal is reached or aborted.

14.2 State Machines and Smach

The navigation program is based on a state machine. This facilitates adapting the program to specific use cases by inserting, removing, or modifying states.

For the implementation of the state machine, the *Smach* Python library is used, which is provided by the `smach` [ROS17m] and `smach_ros` [ROS17n] packages.[Qui15, chapter 13] In Smach, each state represents an action, implemented by a function. The return value of the function is the so-called outcome of the action and determines to which state the state machine transitions next.

14.3 The Node

The executable of the navigation program is called `smach_patrol_main.py` and resides in the `whs_navigation` package. The state machine, representing the overall logic of that program, is shown in figure 14.1. The drawing has been created by the `smach_viewer` node from the `smach_viewer` package [ROS17o]. States are depicted as ellipses, state machines, which are also states, as boxes, transitions as arrows, conditions as labels next to the transitions, and outcomes as red boxes. Currently active states have

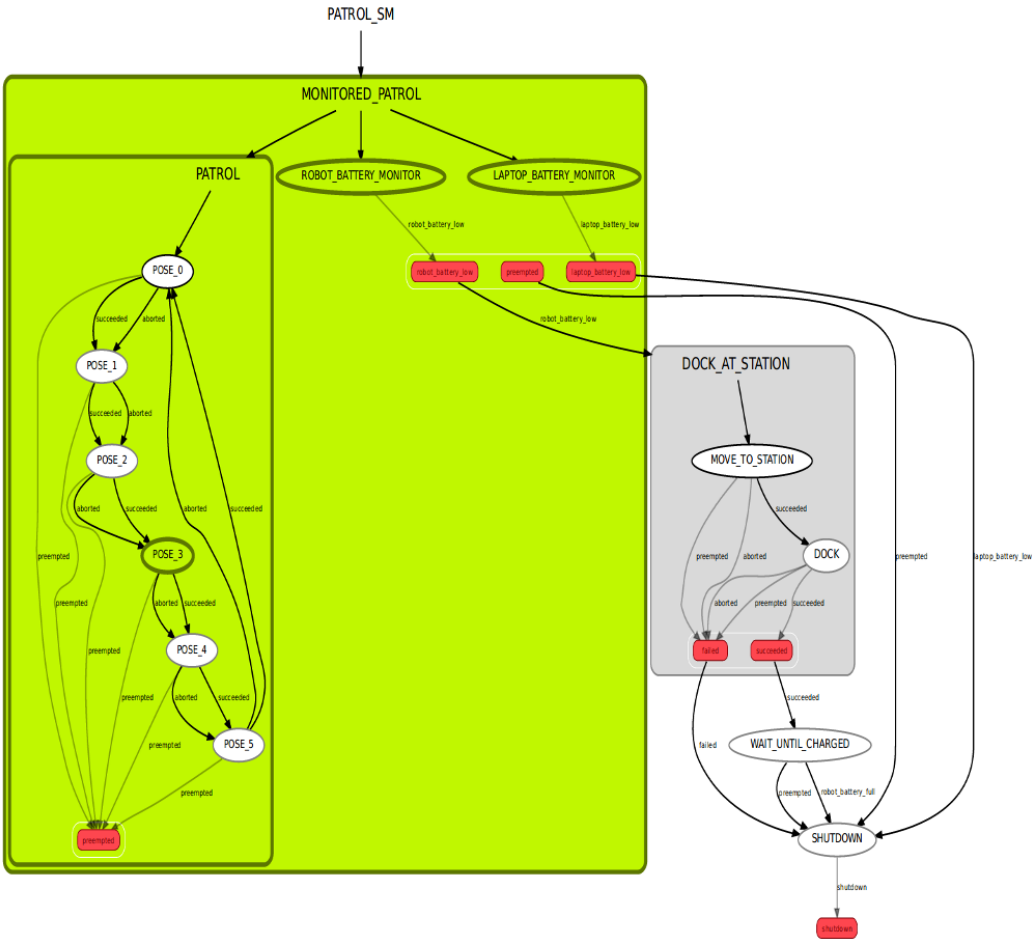

Figure 14.1: State machine of the simulation program

thick borders; in figure 14.1, these are MONITORED_PATROL, PATROL, POSE_3, ROBOT_BATTERY_MONITOR, and LAPTOP_BATTERY_MONITOR.

MONITORED_PATROL is the initial state. It contains the three states PATROL, LAPTOP_BATTERY_MONITOR, and ROBOT_BATTERY_MONITOR, and it is a smach.Concurrence, meaning that these three states are executed concurrently. As soon as one of the three states terminates, the two remaining active states are preempted.

The ROBOT_BATTERY_MONITOR and LAPTOP_BATTERY_MONITOR states continuously check if the batteries of the robot and the laptop are low. When the battery of the robot is low, ROBOT_BATTERY_MONITOR's outcome is robot_battery_low, which leads to the equally named outcome of MONITORED_PATROL. When the battery of the laptop is low, LAPTOP_BATTERY_MONITOR's outcome is laptop_battery_low, which leads to the equally named outcome of MONITORED_PATROL.

PATROL is a state machine that contains one state per goal pose. It starts with the state POSE_0, continues with POSE_1, and so on, and it repeats the procedure when the last substate terminates. In each of those states, the corresponding goal pose is send to the move_base action server. If the action succeeds or is aborted because the goal cannot be reached, PATROL transitions to the next substate. If the action is preempted, e.g. because one of the other states in MONITORED_PATROL has terminated, or because an interrupt signal has been received, PATROL terminates with its preempted outcome.

The default outcome of MONITORED_PATROL is preempted. Therefore, whenever it terminates without one of the batteries being low, preempted will be its outcome.

If the MONITORED_PATROL's outcome is preempted, the top-level state machine transitions into the SHUTDOWN state. The same is true for the laptop_battery_low outcome, since we do not have the equipment to charge the laptop battery from the robot battery and therefore cannot provide a more useful behaviour.

If MONITORED_PATROL's outcome is robot_battery_low, and if the pose in front of the docking station is known to the program, the top-level state machine transitions into the state DOCK_AT_STATION. The first state within that state machine is MOVE_TO_STATION. In this state, the pose in front of the docking station is send to the move_base action server. If the action succeeds, DOCK_AT_STATION transitions to the DOCK state, which calls the dock_drive_action action. The server of that action is started in the launch file listed below. When this action is executed, the TurtleBot tries

to dock at the station. If the action succeeds, the `DOCK_AT_STATION` state will terminate with the `succeeded` outcome, which leads to a transition to the `WAIT_UNTIL_CHARGED` state. If one of the actions executed within `DOCK_AT_STATION` is aborted or preempted, the state's outcome will be `failed`, leading to a transition to the `SHUTDOWN` state.

The `WAIT_UNTIL_CHARGED` state simply waits until the battery of the robot is completely charged or the process is preempted. In both cases, the state machine transitions to the `SHUTDOWN` state. The `SHUTDOWN` state initiates the shutdown of the node, which also leads to stopping the robot. After that, the top-level state machine terminates with the `shutdown` outcome.

14.4 Launching the Node

Before launching the navigation node, the system, including the drivers of the TurtleBot and the navigation stack, has to be set up. When the program is tested in the simulator, this means simply starting the simulation as described in chapter 13.

When running the navigation node in the real world, setting up the system is more laborious. On the notebook, the launch file from listing 14.2 is launched.

Listing 14.2: Contents of the `b5203_real.launch` file from the `whs_navigation` package

```
<launch>
        <include file="$(find turtlebot_bringup)/
            launch/minimal.launch"/>
        <include file="$(find kobuki_auto_docking)/
            launch/minimal.launch"/>
        <include file="$(find turtlebot_navigation)/
            launch/amcl_demo.launch">
            <arg name="map_file"
                    value="$(find whs_navigation
                        )/maps/b5203.yaml"/>
            <arg name="initial_pose_x" value
                ="14.4"/>
            <arg name="initial_pose_y" value
                ="12.8"/>
```

```
            <arg name="initial_pose_a" value
                ="0.0"/>
        </include>
</launch>
```

This file includes the following launch files:

- `minimal.launch` from the `turtlebot_bringup` package. As mentioned in chapter 12, this file primarily launches the drivers of the mobile base.
- `minimal.launch` from the `kobuki_auto_docking` package. This file starts the action server for the `dock_drive_action`, mentioned above.
- `amcl_demo.launch` from the `turtlebot_navigation` package. This file starts the drivers of the 3D sensor and the navigation stack (see chapter 13). It also passes the correct map file to the `map_server` and a reasonable initial pose estimate to `amcl`, although it is recommended to make a better pose estimation as described below.

On the workstation, the `view_navigation.launch` file from the `turtlebot_rviz_launchers` package is launched. This starts `rviz`, displaying, among other things, the map and the model of the TurtleBot at its estimated pose, similar to what is shown in figure 13.1. Then the initially estimated pose of the TurtleBot has to be set with the "2D Pose Estimate" button of `rviz`'s toolbar; this button is used similarly to the "2D Nav Goal" button, which is described in chapter 13. In addition, it is recommended to drive the TurtleBot around for a bit, using the keyboard as described in chapter 12; this will improve the quality of the pose estimate.

After the system has been set up, either in simulation or real world, the navigation node is launched. This can be done conveniently with the launch file listed in listing 14.3.

Listing 14.3: Contents of the `patrol_b5203.launch` file from the `whs_navigation` package

```
<launch>
        <node name="patrol"
                pkg="whs_navigation"
                type="smach_patrol_main.py"
                args="$(find whs_navigation)/poses/
                    b5203.txt"
```

```
                output="screen">
                <remap from="laptop_charge" to="
                    laptop_charge"/>
                <remap from="power_system_event" to
                    ="mobile_base/events/power_system
                    "/>
                <remap from="sound_command" to="
                    mobile_base/commands/sound"/>
        </node>
</launch>
```

Besides launching the navigation node, it passes the text file that contains the goal poses to it and remaps the names of the topics that the node subscribes and publishes to.

Part IV

Conclusion

Chapter 15

Summary

Part I of this document started by explaining the overall goal of this thesis, developing a modular framework for 2D and 3D object recognition and creating a navigation program that can be adapted to specific needs. *Chapter* 3 mentioned that the setup comprised a TurtleBot2 mobile robot with an Astra 3D sensor. *Chapter* 4 gave an overview of ROS, a framework for developing robot software, which runs on the computer that controls the TurtleBot.

Part II dealt with the object recognition framework. *Chapter* 5 explained preliminary considerations that apply to all solutions. These include the demands on and the target groups of the framework, the message types that are used for publishing detected objects, and the role of parametrization and parameter files. *Chapter* 6 explained the algorithms of three specific object recognition systems, which were used as examples throughout the document. These comprise the HSV detection, the feature detection, and the shape detection systems.

After that, *part* II described three different solutions, i.e. approaches for implementing the framework. *Chapter* 7 presented the first solution, writing individual programs. This naive approach implements each system as an isolated set of two, mainly procedural, executables, one for creating parameter files and one for publishing objects that are detected based on such files. *Chapter* 8 presented the second solution, libraries that provide classes and a general structure for usage by all object recognition systems. In this case, modules are instances of these classes, and each specific system is implemented by two nodes that make use of the module. *Chapter* 9 presented the third solution, a ROS-based framework that provides a set of ROS nodes and suggests a general structure for arranging these nodes. The nodes represent

the modules of the framework and are connected to create specific object recognition systems.

Chapter 10 compared the solutions and exposed the pros and cons of each one. While, with regard to the objectives of this thesis, writing individual programs is merely useful for prototyping, the other two solutions both have their merits. In the end, the ROS-based solution was chosen as the best approach because it is easier to understand and more flexible, and it facilitates the integration of existing ROS tools.

Part III dealt with the navigation program. *Chapter* 12 gave instructions for creating a map of the robot's environment and for viewing and editing this map. *Chapter* 13 showed how to create a simulation world from the map and how to determine goal poses within the Stage simulator. Finally, *chapter* 14 presented the node that implements the higher logic of the navigation and that is based on the Smach library for creating state machines and on the ROS navigation stack for moving the robot to specific goal poses.

Chapter 16

Outlook

At the time of writing, no additional projects that concern themselves with the TurtleBot or the object recognition framework are scheduled. The source code of the project will be made publicly available for any purpose, and a video that gives an introduction to the framework will be posted on the Internet. This is done in the hope that the source code will be useful and that someone might even develop the framework further.

Appendix A

PCL Cylinder Segmentation Tutorial

Source code of the "cylinder model segmentation" tutorial from the PCL website:[PCL17d]

```
1  #include <pcl/ModelCoefficients.h>
2  #include <pcl/io/pcd_io.h>
3  #include <pcl/point_types.h>
4  #include <pcl/filters/extract_indices.h>
5  #include <pcl/filters/passthrough.h>
6  #include <pcl/features/normal_3d.h>
7  #include <pcl/sample_consensus/method_types.h>
8  #include <pcl/sample_consensus/model_types.h>
9  #include <pcl/segmentation/sac_segmentation.h>
10
11 typedef pcl::PointXYZ PointT;
12
13 int
14 main (int argc, char** argv)
15 {
16   // All the objects needed
17   pcl::PCDReader reader;
18   pcl::PassThrough<PointT> pass;
19   pcl::NormalEstimation<PointT, pcl::Normal> ne;
```

```
20    pcl::SACSegmentationFromNormals<PointT, pcl::
          Normal> seg;
21    pcl::PCDWriter writer;
22    pcl::ExtractIndices<PointT> extract;
23    pcl::ExtractIndices<pcl::Normal> extract_normals;
24    pcl::search::KdTree<PointT>::Ptr tree (new pcl::
          search::KdTree<PointT> ());
25
26    // Datasets
27    pcl::PointCloud<PointT>::Ptr cloud (new pcl::
          PointCloud<PointT>);
28    pcl::PointCloud<PointT>::Ptr cloud_filtered (new
          pcl::PointCloud<PointT>);
29    pcl::PointCloud<pcl::Normal>::Ptr cloud_normals (
          new pcl::PointCloud<pcl::Normal>);
30    pcl::PointCloud<PointT>::Ptr cloud_filtered2 (new
          pcl::PointCloud<PointT>);
31    pcl::PointCloud<pcl::Normal>::Ptr cloud_normals2 (
          new pcl::PointCloud<pcl::Normal>);
32    pcl::ModelCoefficients::Ptr coefficients_plane (
          new pcl::ModelCoefficients),
          coefficients_cylinder (new pcl::
          ModelCoefficients);
33    pcl::PointIndices::Ptr inliers_plane (new pcl::
          PointIndices), inliers_cylinder (new pcl::
          PointIndices);
34
35    // Read in the cloud data
36    reader.read ("table_scene_mug_stereo_textured.pcd"
          , *cloud);
37    std::cerr << "PointCloud has: " << cloud->points.
          size () << " data points." << std::endl;
38
39    // Build a passthrough filter to remove spurious
          NaNs
40    pass.setInputCloud (cloud);
41    pass.setFilterFieldName ("z");
42    pass.setFilterLimits (0, 1.5);
```

```
43    pass.filter (*cloud_filtered);
44    std::cerr << "PointCloud after filtering has: " <<
          cloud_filtered->points.size () << " data
          points." << std::endl;
45
46    // Estimate point normals
47    ne.setSearchMethod (tree);
48    ne.setInputCloud (cloud_filtered);
49    ne.setKSearch (50);
50    ne.compute (*cloud_normals);
51
52    // Create the segmentation object for the planar
          model and set all the parameters
53    seg.setOptimizeCoefficients (true);
54    seg.setModelType (pcl::SACMODEL_NORMAL_PLANE);
55    seg.setNormalDistanceWeight (0.1);
56    seg.setMethodType (pcl::SAC_RANSAC);
57    seg.setMaxIterations (100);
58    seg.setDistanceThreshold (0.03);
59    seg.setInputCloud (cloud_filtered);
60    seg.setInputNormals (cloud_normals);
61    // Obtain the plane inliers and coefficients
62    seg.segment (*inliers_plane, *coefficients_plane);
63    std::cerr << "Plane coefficients: " << *
          coefficients_plane << std::endl;
64
65    // Extract the planar inliers from the input cloud
66    extract.setInputCloud (cloud_filtered);
67    extract.setIndices (inliers_plane);
68    extract.setNegative (false);
69
70    // Write the planar inliers to disk
71    pcl::PointCloud<PointT>::Ptr cloud_plane (new pcl
          ::PointCloud<PointT> ());
72    extract.filter (*cloud_plane);
73    std::cerr << "PointCloud representing the planar
          component: " << cloud_plane->points.size () <<
          " data points." << std::endl;
```

```
74    writer.write ("
         table_scene_mug_stereo_textured_plane.pcd", *
         cloud_plane, false);
75
76    // Remove the planar inliers, extract the rest
77    extract.setNegative (true);
78    extract.filter (*cloud_filtered2);
79    extract_normals.setNegative (true);
80    extract_normals.setInputCloud (cloud_normals);
81    extract_normals.setIndices (inliers_plane);
82    extract_normals.filter (*cloud_normals2);
83
84    // Create the segmentation object for cylinder
         segmentation and set all the parameters
85    seg.setOptimizeCoefficients (true);
86    seg.setModelType (pcl::SACMODEL_CYLINDER);
87    seg.setMethodType (pcl::SAC_RANSAC);
88    seg.setNormalDistanceWeight (0.1);
89    seg.setMaxIterations (10000);
90    seg.setDistanceThreshold (0.05);
91    seg.setRadiusLimits (0, 0.1);
92    seg.setInputCloud (cloud_filtered2);
93    seg.setInputNormals (cloud_normals2);
94
95    // Obtain the cylinder inliers and coefficients
96    seg.segment (*inliers_cylinder, *
         coefficients_cylinder);
97    std::cerr << "Cylinder coefficients: " << *
         coefficients_cylinder << std::endl;
98
99    // Write the cylinder inliers to disk
100   extract.setInputCloud (cloud_filtered2);
101   extract.setIndices (inliers_cylinder);
102   extract.setNegative (false);
103   pcl::PointCloud<PointT>::Ptr cloud_cylinder (new
         pcl::PointCloud<PointT> ());
104   extract.filter (*cloud_cylinder);
105   if (cloud_cylinder->points.empty ())
```

```
106       std::cerr << "Can't find the cylindrical
              component." << std::endl;
107     else
108     {
109             std::cerr << "PointCloud representing the
                    cylindrical component: " <<
                    cloud_cylinder->points.size () << "
                    data points." << std::endl;
110             writer.write ("
                    table_scene_mug_stereo_textured_cylinder
                    .pcd", *cloud_cylinder, false);
111     }
112     return (0);
113 }
```

Appendix B

`MouseEvent` Message Definition

`MouseEvent` message definition from the file `~/turtlebot_ws/src/object_detection_helpers/msg/MouseEvent.msg`:

```
# Types.
uint8 PRESS      = 0
uint8 MOVE       = 1
uint8 RELEASE    = 2

# Buttons.
uint8 NO_BUTTON          = 0
uint8 LEFT_BUTTON        = 1
uint8 MIDDLE_BUTTON      = 2
uint8 RIGHT_BUTTON       = 3
uint8 UNKNOWN_BUTTON     = 4

uint8 type
uint8 button    # button that caused the event (== NO_BUTTON for MOVE)

# Mouse position when event was generated.
object_detection_helpers/Point2D position
```

Appendix C

Simulation World File

World file
`~/turtlebot_ws/src/whs_navigation/maps/stage/b5203.world` for the Stage simulator:

```
include "/opt/ros/kinetic/share/turtlebot_stage/maps
    /stage/turtlebot.inc"

define floorplan model
(
  # sombre, sensible, artistic
  color "gray30"

  # most maps will need a bounding box
  boundary 1

  gui_nose 0
  gui_grid 0
  gui_outline 0
  gripper_return 0
  fiducial_return 0
  laser_return 1
)

resolution 0.02
interval_sim 100  # simulation timestep in
    milliseconds
```

```
21
22 window
23 (
24   size [ 600 700 0.0 ]
25   center [ 0.0 0.0 ]
26   rotate [ 0.0 0.0 ]
27   scale 60
28 )
29
30 floorplan
31 (
32   name "b5203_bordered"
33   bitmap "../b5203_bordered.pgm"
34   size [ 28.8 25.6 2.0 ] #the real size in meters (
       from yaml and pgm, resolution*pixels)
35   pose [ 14.4 12.8 0.0 0.0 ]    # center = size
       divided by two
36 )
37
38 # throw in a robot
39 turtlebot
40 (
41   pose [14.4 12.8 0.0 0.0 ]
42   name "turtlebot"
43   color "red"
44 )
```

Appendix D

Launch File for Simulation

File `~/turtlebot_ws/src/whs_navigation/launch/b5203_stage.launch` for launching the simulation:

```
1  <launch>
2        <include file="$(find turtlebot_stage)/launch/
             turtlebot_in_stage.launch">
3              <arg name="map_file"
4                       value="$(find whs_navigation)/
                            maps/b5203.yaml"/>
5              <arg name="world_file"
6                       value="$(find whs_navigation)/
                            maps/stage/b5203.world"/>
7              <arg name="initial_pose_x" value
                   ="14.4"/>
8              <arg name="initial_pose_y" value
                   ="12.8"/>
9              <arg name="initial_pose_a" value
                   ="0.0"/>
10       </include>
11 </launch>
```

Appendix E

File of Goal Poses

File `~/turtlebot_ws/src/whs_navigation/poses/b5203.txt`, which contains the goal poses for the test environment of this project:

```
1  # Each pose is given as position (x,y,z) and
       orientation (x,y,z,w) in the
2  # following form:
3  # x, y, z, x, y, z, w
4
5  # About 1 meter in front of the docking station.
6  # This is where the robot drives to when its battery
       -level is low.
7  # You can leave the pose empty or remove the line if
        you don't use
8  # auto-docking.
9  station_pose: 14.4, 12.8, 0.0, 0.0, 0.0, 0.0, 1.0
10
11 # Driving one circle, clockwise, starting about one
       meter left from the door,
12 # after entering it.
13
14 10.6897900314, 14.1520320024, 0.0, 0.0, 0.0,
       0.0547466180359, 0.998500279326
15 11.0041180086, 13.350122288, 0.0, 0.0, 0.0,
       -0.100977534052, 0.994888706146
16 11.5688588117, 13.3086513764, 0.0, 0.0, 0.0,
       -0.0597612667342, 0.998212698275
```

```
17 12.4120555185, 13.2870717652, 0.0, 0.0, 0.0,
       -0.0328873590553, 0.999459064502
18 13.8678938173, 13.2435042537, 0.0, 0.0, 0.0,
       0.0252025030913, 0.999682366473
19 15.2674508993, 13.3011369944, 0.0, 0.0, 0.0,
       0.082274725463, 0.996609687666
20 16.2565594058, 12.9133816457, 0.0, 0.0, 0.0,
       -0.220465767845, 0.975394712518
21 16.7598204749, 11.898318164, 0.0, 0.0, 0.0,
       -0.829100810015, 0.559099138644
22 16.0036199677, 11.5767658905, 0.0, 0.0, 0.0,
       -0.999087469055, 0.0427109959804
23 14.3191292045, 11.2729175032, 0.0, 0.0, 0.0,
       -0.994535582141, 0.104398160209
24 12.9513965112, 11.0261445805, 0.0, 0.0, 0.0,
       -0.986468310383, 0.163952043626
25 11.6297515701, 10.7445547853, 0.0, 0.0, 0.0,
       0.997988232181, 0.0633994355452
26 11.1683314998, 11.080445797, 0.0, 0.0, 0.0,
       0.915775004152, 0.401691600323
27 10.7400558102, 12.3083205933, 0.0, 0.0, 0.0,
       0.781362391416, 0.624077569923
28 10.5085637881, 13.1479526766, 0.0, 0.0, 0.0,
       0.806924012491, 0.590655261608
```

Bibliography

[Bla08] Blanchette, Jasmin; Summerfield, Mark.
C++ GUI Programming with Qt4.
Second edition.
Boston: Prentice Hall, 2008.

[Boo07] Booch, Grady; et al.
Object-Oriented Analysis and Design with Applications.
Third edition.
Boston: Addison-Wesley, 2007.

[Busch96] Buschmann, Frank; et al.
Pattern-Oriented Software Architecture.
Chichester, West Sussex: Wiley, 1996.

[McC17] McCann, William Jon.
Dia.
URL: https://wiki.gnome.org/Apps/Dia.
Accessed: 20 November 2017.

[Gai17] Gaitech.
Map-Based Navigation.
URL: edu.gaitech.hk/turtlebot/map-navigation.html.
Accessed: 17 November 2017.

[Gam95] Gamma, Erich; et al.
Design Patterns: Elements of Reusable Object-Oriented Software
Boston: Addison-Wesley, 1995.

[Gav17] Gavran, Ivan.
ROS—creating .world file from existing .yaml.

URL: https://medium.com/@ivangavran/ros-creating-world-file-from-existing-yaml-5b553d31cc53.
Accessed: 17 November 2017.

[Kae17] Kaehler, Adrian; Bradski, Gary.
Learning OpenCV 3.
Sebastopol: O'Reilly, 2017.

[Kirk17] Kirkland, Dustin.
ubuntu manuals: ssh.
URL: manpages.ubuntu.com/manpages/xenial/man1/ssh.1.html.
Accessed: 17 November 2017.

[Mar09] Martin, Robert C.; et al.
Clean Code: A Handbook of Agile Software Craftsmanship
Boston: Prentice Hall, 2009.

[OpenCV17] OpenCV team.
OpenCV.
URL: https://opencv.org.
Accessed: 17 November 2017.

[Orb17] Orbbec 3D.
Orbbec Astra, Astra S & Astra Pro.
URL: https://orbbec3d.com/product-astra.
Accessed: 17 November 2017.

[PCL17a] Open Perception, Inc.
PCL.
URL: pointclouds.org.
Accessed: 17 November 2017.

[PCL17b] Open Perception, Inc.
Documentation.
URL: pointclouds.org/documentation. Accessed: 17 November 2017.

[PCL17c] Open Perception, Inc.
`pcl::visualization::PCLVisualizer` class reference.

URL: docs.pointclouds.org/1.7.0/classpcl_1_1visualization_-1_1_p_c_l_visualizer.html.
Accessed: 17 November 2017.

[PCL17d] Open Perception, Inc.
Cylinder model segmentation.
URL: pointclouds.org/documentation/tutorials/-cylinder_segmentation.php.
Accessed: 17 November 2017.

[Qt17a] The Qt Company, Ltd.
Qt.
URL: https://www.qt.io.
Accessed: 20 November 2017.

[Qt17b] The Qt Company, Ltd.
`QWidget` class.
URL: doc.qt.io/qt-5/qwidget.html.
Accessed: 17 November 2017.

[Qt17c] The Qt Company, Ltd.
`QLayout` class.
URL: doc.qt.io/qt-5/qlayout.html.
Accessed: 17 November 2017.

[Qt17d] The Qt Company, Ltd.
`QGroupBox` class.
URL: doc.qt.io/qt-5/qgroupbox.html.
Accessed: 17 November 2017.

[Qt17e] The Qt Company, Ltd.
`QImage` class.
URL: doc.qt.io/qt-5/qimage.html.
Accessed: 17 November 2017.

[Qui15] Quigley, Morgan; Gerkey, Brian; Smart, William.
Programming Robots with ROS.
Sebastopol: O'Reilly, 2015.

[ROS17a] Open Source Robotics Foundation, Inc.
Documentation.

URL: www.ros.org.
Accessed: 18 November 2017.

[ROS17b] Open Source Robotics Foundation, Inc.
Documentation.
URL: wiki.ros.org.
Accessed: 17 November 2017.

[ROS17c] Open Source Robotics Foundation, Inc.
Technical Overview.
URL: wiki.ros.org/ROS/Technical Overview.
Accessed: 17 November 2017.

[ROS17d] Open Source Robotics Foundation, Inc.
Parameter Server.
URL: wiki.ros.org/ROS/Parameter Server.
Accessed: 17 November 2017.

[ROS17e] Open Source Robotics Foundation, Inc.
How to Write Your First .cfg File.
URL: wiki.ros.org/dynamic_reconfigure/Tutorials/-HowToWriteYourFirstCfgFile.
Accessed: 17 November 2017.

[ROS17f] Open Source Robotics Foundation, Inc.
dynamic_reconfigure.
URL: wiki.ros.org/dynamic_reconfigure.
Accessed: 17 November 2017.

[ROS17g] Open Source Robotics Foundation, Inc.
rqt_reconfigure.
URL: wiki.ros.org/rqt_reconfigure.
Accessed: 17 November 2017.

[ROS17h] Open Source Robotics Foundation, Inc.
catkin/ CMakeLists.txt.
URL: wiki.ros.org/catkin/CMakeLists.txt.
Accessed: 17 November 2017.

[ROS17i] Open Source Robotics Foundation, Inc.
rviz.

URL: wiki.ros.org/rviz.
Accessed: 17 November 2017.

[ROS17j] Open Source Robotics Foundation, Inc.
Robots/ TurtleBot.
URL: wiki.ros.org/Robots/TurtleBot.
Accessed: 17 November 2017.

[ROS17k] Open Source Robotics Foundation, Inc.
gmapping.
URL: wiki.ros.org/gmapping.
Accessed: 17 November 2017.

[ROS17l] Open Source Robotics Foundation, Inc.
stage.
URL: wiki.ros.org/stage.
Accessed: 17 November 2017.

[ROS17m] Open Source Robotics Foundation, Inc.
smach.
URL: wiki.ros.org/smach.
Accessed: 17 November 2017.

[ROS17n] Open Source Robotics Foundation, Inc.
smach_ros.
URL: wiki.ros.org/smach_ros.
Accessed: 17 November 2017.

[ROS17o] Open Source Robotics Foundation, Inc.
smach_viewer.
URL: wiki.ros.org/smach_viewer.
Accessed: 17 November 2017.

[Turtle17] Open Source Robotics Foundation, Inc.
TurtleBot2.
URL: www.turtlebot.com/turtlebot2.
Accessed: 17 November 2017.

[Ubu17] Canonical Ltd.
Ubuntu.

URL: https://www.ubuntu.com.
Accessed: 18 November 2017.

[VTK17] Kitware Inc.
`QVTKWidget` class reference.
URL: https://www.vtk.org/doc/nightly/html/-classQVTKWidget.html.
Accessed: 17 November 2017.

[Wiki17a] Wikipedia.
Open/closed principle.
URL: https://en.wikipedia.org/wiki/Open/closed_principle.
Accessed: 17 November 2017.

[Wiki17b] Wikipedia.
Random sample consensus.
URL: https://en.wikipedia.org/wiki/Random_sample_consensus.
Accessed: 17 November 2017.

[Yujin17] Yujin Robot Co., Ltd.
Kobuki.
URL: kobuki.yujinrobot.com.
Accessed: 17 November 2017.