ROS-Based Object Recognition Framework - Project Description

Janosch Hoffmann

January 1, 2018

Contents

1 Preface **2**
1.1 About This Project . . . 2
1.2 Purpose of This Document . . . 2
1.3 Contact Me . . . 3
1.4 Setup . . . 3

2 Preliminary Considerations **3**
2.1 Goals and Ideals . . . 3
2.2 Parametrization . . . 4
2.3 2D and 3D Systems . . . 4
2.4 Message Types for Detected Objects . . . 5

3 Example Systems **6**
3.1 HSV Detection . . . 6
3.2 Feature Detection . . . 7
3.3 Shape Detection . . . 10

4 Architecture and Design **10**
4.1 Filters and Detectors . . . 10
4.2 How Nodes Constitute Object Recognition Systems . . . 12
4.2.1 Modules as Objects . . . 12
4.2.2 Modules as Nodes . . . 13
4.3 Parameter Management . . . 15
4.3.1 `ReconfigureParameterManager` . . . 15
4.3.2 `ObservedParameterManager` . . . 16

5 Navigation **18**

6 Packages **18**
6.1 binary_detector 19
6.2 distance_filter 19
6.3 feature_detector 19
6.4 hsv_filter 19
6.5 morphology_filter 19
6.6 object_detection 20
6.7 object_detection_2d 20
6.8 object_detection_2d_msgs 20
6.9 object_detection_2d_nodes 20
6.10 object_detection_2d_vis 20
6.11 object_detection_3d 21
6.12 object_detection_3d_msgs 21
6.13 object_detection_3d_nodes 21
6.14 object_painter 21
6.15 objects2d_to_objects3d 22
6.16 objects_to_markers 22
6.17 shape_detector 22
6.18 whs_navigation 22

7 TODOs **22**

1 Preface

1.1 About This Project

The main purpose of this project is to develop a ROS-based framework that facilitates the development of 2D and 3D object recognition software. The source code mainly comprises ROS nodes and classes that belong to that framework.

In addition, a basic navigation program has been written, which is described in section 5.

The project was created as part of my master thesis at the Westphalian University of Applied Sciences in Gelsenkirchen, Germany (https://www.w-hs.de). The thesis was supervised by Prof. Jürgen Dunker.

1.2 Purpose of This Document

This document should enable you to ...

- ... create ROS systems that use the modules (nodes and classes) of this project.
- ... integrate your own modules into the framework.
- ... explore, understand, and modify the source code.

1.3 Contact Me

I'm happy about anyone who can make use of the code, for whatever purpose. If you have any questions or suggestions, feel free to e-mail me (janosch.hoffmann@hotmail.de); I will help you as good as I can.

Most of the code and the documentation (including this document) are pretty premature. I'm willing to improve them, but only if someone (i.e. at least one) is actually interested in that. Therefore, don't be shy of contacting me.

I think it would be reasonable to use the project as a starting point for your own (school) project or just as an idea that could influence your project. Whatever you do with it, please let me know.

1.4 Setup

This section describes the hardware and software setup that was used during development. Regardless of that, adapting the code to other robots, cameras, or ROS versions should require only minor changes. The object recognition software can also be used without any robot.

The software was written for a TurtleBot2 robot with an Orbbec Astra camera (https://orbbec3d.com/product-astra). I think the `objects2d_to_objects3d` package is the only package that uses this information by having the height and the intrinsics of the camera hard-coded into it.

The notebook of the TurtleBot was running Ubuntu 16.04 and ROS Kinetic.

The libraries that were mainly used are OpenCV 3, PCL 1.7, Qt 5, and Eigen. In addition, for the navigation program, the Smach library/package (wiki.ros.org/smach) was used.

2 Preliminary Considerations

2.1 Goals and Ideals

The goal of this project is to create a ROS-based framework for 2D and 3D object recognition software. Figure 1 illustrates how an object recognition system that uses this framework should look like. The system is composed of modules, each depicted as an ellipse. It's useful to group these modules into filters, which preprocess the data (images or point clouds), detectors, which detect objects within the filtered data, and auxiliary modules, e.g. for visualization.

All filters should have the same interface, i.e. the same inputs and outputs; this should also be the case for all detectors. This makes these modules interchangeable and enables the development of modules that rely on that common interface.

It should be easy to create new filters and detectors. Developers should be able to concentrate on the main algorithm; auxiliary functionality should be provided by general-purpose modules. The development of one module should not require to modify existing modules.

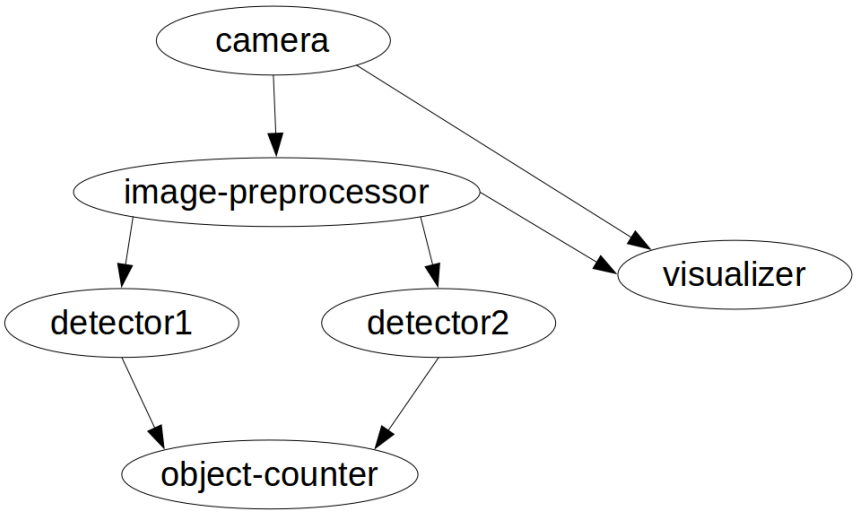

Figure 1: Graph of a modular object recognition system

It should also be easy to create systems from existing modules. Ideally, the user can choose from a pool of existing modules and combine those that best suit his particular problem.

2.2 Parametrization

Each detector, or more generally each system, offers some criterion for detecting objects. One system might use the color of objects, another their shapes. Each system should provide parameters that allow the user to configure the system for the detection of specific objects, e.g. objects with a specific color or with a specific shape. After setting the parameters of a system to the desired values, it has to be possible to store their values into one or more files. It should be possible to pass such files to a system at start-up.

In summary, each 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.

2.3 2D and 3D Systems

We call object recognition systems that operate on images '2D systems' and those that operate on point clouds '3D systems'. Most modules can only be

used in one of the two groups, since they work on data types that are specific to that group. As an example, 2D filters work on images, 3D filters on point clouds.

2.4 Message Types for Detected Objects

To be able to process detected objects independently of the detector that detected them, a common message type for detected objects was defined. More precisely, two message types were defined, one for 2D object recognition and one for 3D object recognition. These types contain a header and an array of detected objects. The description of each object comprises an identifier (name) and a description of the object's location within the data (image or point cloud).

The message type for 2D systems is listed in listing 1. The location of the object within the image that contains it is represented by a polygon. The x- and y-coordinates of the polygon are given in the coordinate frame of the image; the z-coordinate is not used.

Listing 1: The `DetectedObject2DArray` message type

```
$ 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
```

The message type for 3D systems is listed in listing 2. In this message type, the location of each object is represented by a rectangular prism, which is described by the pose of its center and by its dimensions, i.e. its width, height, and depth.

Listing 2: The `DetectedObject3DArray` message type

```
$ 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
```

3 Example Systems

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. The last system operates on point clouds and mainly uses facilities of the PCL library.

Feel free to add your own systems or modules. This will enhance the usability of the framework.

3.1 HSV Detection

The HSV detection system detects objects with noticeable colors.

Each image is 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.

Then the image is filtered using the `cv::inRange` function, 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 `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:

```
h_min <= H <= h_max
s_min <= S <= s_max
v_min <= V <= v_max
```

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 2 shows an image on the left and the result of applying the filter to it on the right. This filtering in HSV space is done by a module that we call HSV-filter.

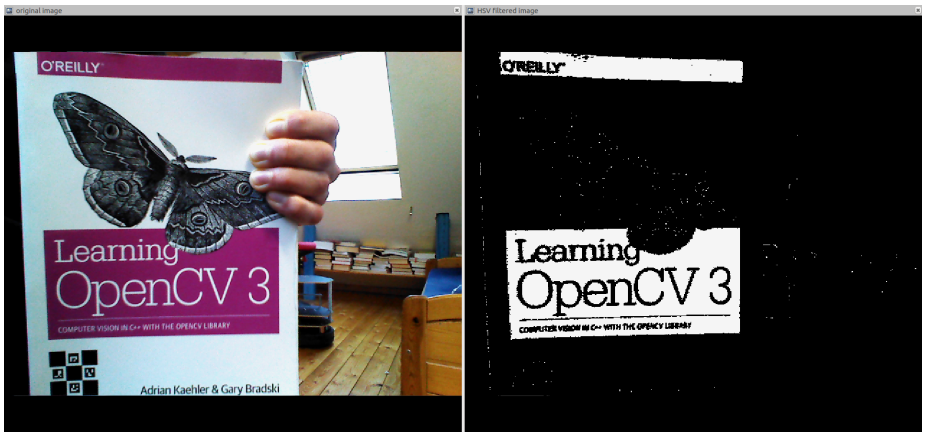

Figure 2: Image before (left) and after (right) applying the `cv::inRange` function to it

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. Another module, called morphology-filter, was created for this job. It uses the `cv::morphologyEx` function to do this. Parameters of this function, which are also parameters of the morphology-filter module (and therefore of the object recognition system), are the kind of operation (erode, dilate, open, etc.) and the number of iterations.

The detection of objects in the filtered images is done by the binary-detector module. The external contours of the detected objects, i.e. of the contiguous areas of pixels with non-zero values, are computed with the `cv::findContours` function. Contours whose length does not lie within specific limits, which are parameters of the binary-detector, are filtered out. Lastly, the number of vertices is reduced by applying the `cv::convexHull` function to each contour.

Figure 3 shows a complete example.

3.2 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.

In OpenCV, keypoints are represented by instances of the `cv::KeyPoint` class. Different methods for computing keypoints and descriptors exist; the method that is used in this project is called SURF (Speeded-Up Robust Features). There are also different methods for matching two sets of descriptors;

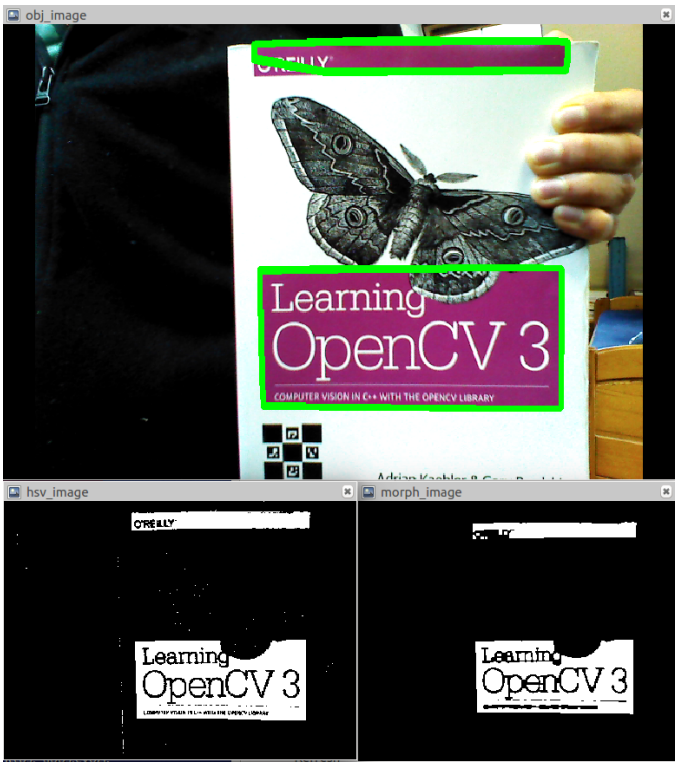

Figure 3: Top: Original image with detected objects. Bottom left: Result of applying thresholds to the image in HSV color space. Bottom right: Result of additional opening transformation.

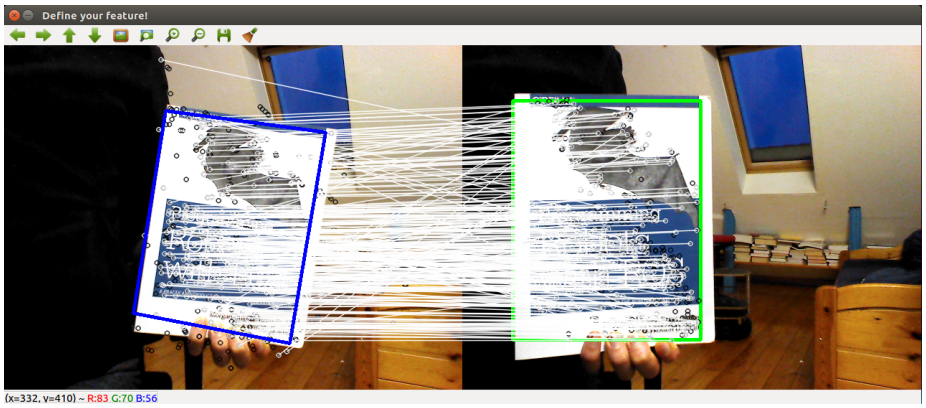

Figure 4: The green rectangle represents the target-selection; the blue polygon represents the detected object. Keypoints are drawn as little circles, matches as connecting lines between keypoints.

"brute force matching" is used in this project.

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. 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. Then the polygon that represents the detected object is computed by transforming the rectangle of the user selection using the homography.

Figure 4 shows an example. The most recently received image is rendered on the left; the image from which the target has been selected is rendered on the right. The rectangle that the user created with his mouse to select the target is green. Keypoints are drawn as little circles. If a keypoint within the target-selection (in 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. Note that figure 4

was not created with any of the programs that you find in the source code.

3.3 Shape Detection

The shape detection system detects cylindrical or spherical objects.

RANSAC (random sample consensus) is used as a method for estimating parameters of a mathematical model from a set of data points that contains outliers (see Wikipedia for details). 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 shape is a parameter of the system. The RANSAC method is implemented by the `pcl::SACSegmentation` class.

To reduce the computing time, each point cloud is filtered first, which is done by the distance-filter module. All points whose distance from the camera does not lie in a range that is specified by the parameters of the module are filtered out.

The detection is done by the shape-detector module. First it computes the normals of the filtered point cloud 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 are also parameters of the shape-detector module. 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 an example, Figure 5 shows the detection of a cylindrical garbage can that is placed on the seat of an office chair. The original point cloud is rendered in white, the filtered point cloud in red. The box that represents the detected object is rendered as a green, transparent box.

4 Architecture and Design

This section describes the main design decisions that were made during development. The basic ideas behind each decision are outlined without going into the details.

4.1 Filters and Detectors

`Filter` and `Detector` are abstract base classes for all filters and detectors. `Filter` provides the pure virtual (i.e. abstract) member function `filter`, `Detector` provides the pure virtual member function `detect`.

Since we distinguish 2D and 3D systems, there are actually different classes for both cases. For 2D systems, the `object_detection_2d::Filter` and `object_detection_2d::Detector` classes are used. For 3D systems, the `object_detection_3d::Filter` and `object_detection_3d::Detector` classes are used.

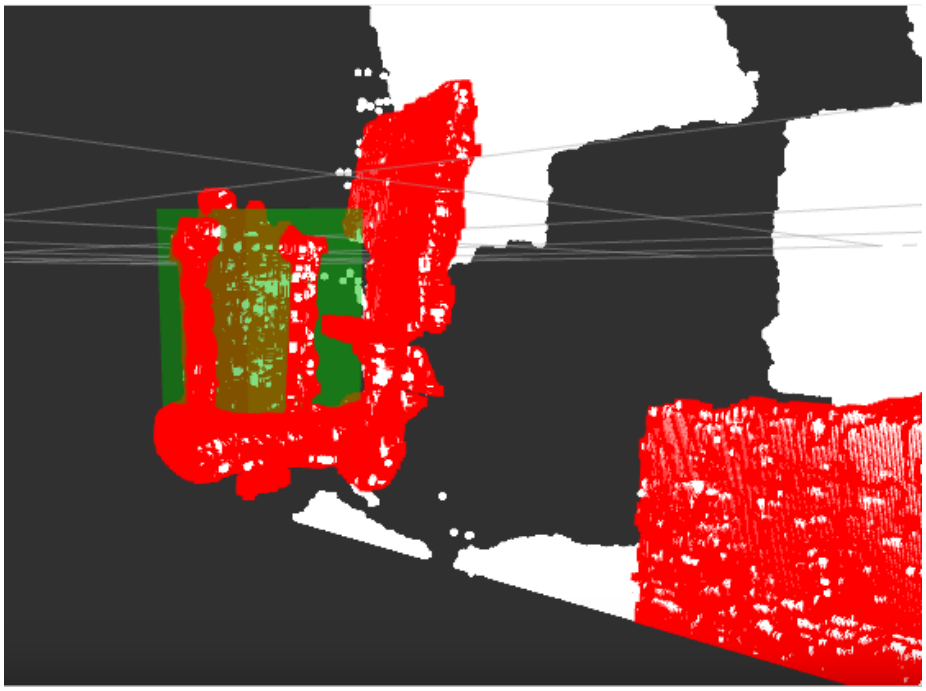

Figure 5: Detection of a cylindrical garbage can on an office chair

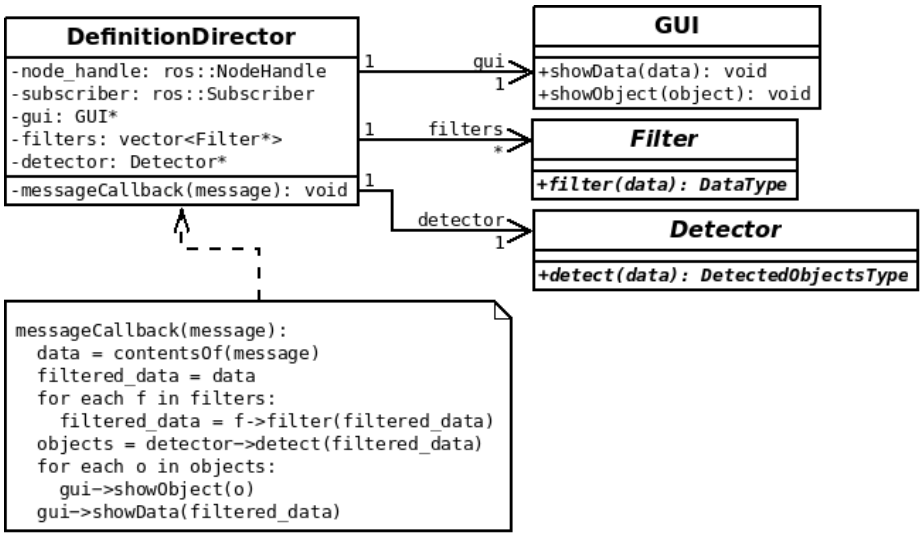

Figure 6: The `DefinitionDirector` object receives incoming ROS messages and passes data to the gui, filters, and detector of the system (diagram created with Dia)

These abstract classes are defined in the `object_detection_2d` and `object_detection_3d` packages.

4.2 How Nodes Constitute Object Recognition Systems

There are currently two ways for using filters and detectors in object recognition systems. The approach that is described first was also developed first. Currently I prefer the second approach because it is simpler and more flexible.

4.2.1 Modules as Objects

The first approach combines the most important modules of the system in a single ROS node. Two of these nodes exist for every object recognition system, one for creating parameter files and one for detecting and publishing objects based on such a file. Each module is represented by an object and the objects communicate with each other by keeping references to each other.

For the node that is used to create parameter files, the classes that process the incoming ROS messages are shown in figure 6. 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 `GUI` 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 6 tries to

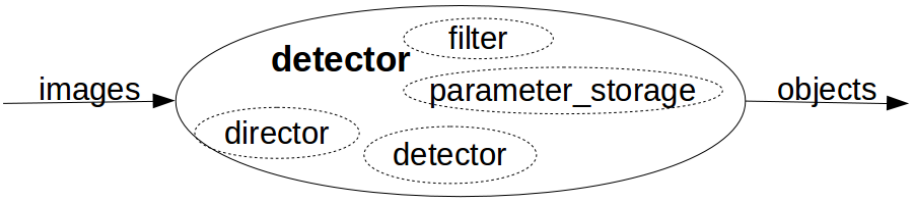

Figure 7: Node for publishing detected objects. Modules are instances of classes and are depicted as dashed ellipses.

capture the correlations of 2D and 3D systems, it calls instances of these data types collectively `data`.

The `GUI` is a Qt-based 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.

A system is created by writing these two nodes. In each node, the appropriate classes are instantiated and the objects are connected according to figure 6.

Figure 7 presents a simplified visualization of the running system. The system is represented by a single ROS node that subscribes to images and publishes detected objects. The objects are detected according to a parameter file that has been passed to the node at start-up. Internally, the node is seperated into modules, which are instances of classes and depicted as dashed ellipses.

4.2.2 Modules as Nodes

The second approach implements modules as nodes.

Let's use the `hsv_filter::HSVFilter` class from the `hsv_filter` package as an example. An object of this class inherits from the `Filter` class and overrides the `filter` method. This method takes an image (object of the `cv::Mat` class) and returns a new `cv::Mat` object that contains the filtered image. "Using this

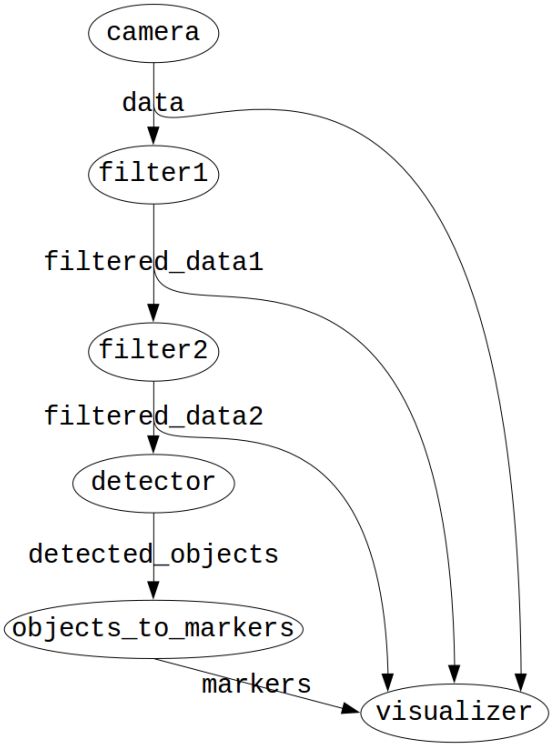

Figure 8: The ROS nodes represent modules of the framework and constitute an object recognition system

object as a node" means that a ROS node is created which has a similar function as the object; it subscribes to images and publishes new, filtered images. To this end, 2 (or rather 4; 2 for 2D systems and 2 for 3D systems) wrapper classes were created, `ROSFilterWrapper` and `ROSDetectorWrapper`. An object of the `ROSFilterWrapper` class has a reference to a `Filter` object. For 2D systems, it subscribes to images over the ROS network, converts these images to `cv::Mat` objects, passes these `cv::Mat`s to the `Filter`'s `filter` method, converts the return value back to the ROS image message type, and publishes the result over the ROS network. The `ROSDetectorWrapper` class behaves similarly, but its results are detected objects.

Figure 8 shows a general example of an object recognition system. Each module is implemented as a ROS node and the modules communicate with each other, mainly using ROS topics. Typically needed functionalities, e.g. visualization, are encapsulated in general-purpose modules. Existing ROS tools (e.g. rviz) are used wherever possible.

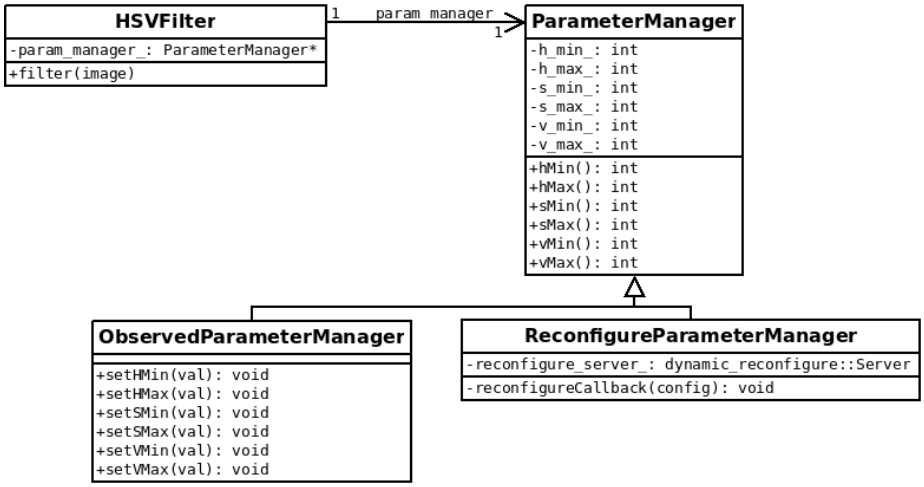

Figure 9: Parameter-Manager classes in the `hsv_filter` package

4.3 Parameter Management

As described in section 2.2, each module (especially filters and detectors) provides parameters for configuring the module.

To encapsulate the parameter management in each module, a common pattern is used. The class diagram in figure 9 illustrates this. Typically, each filter- and each detector-object has a reference to an object of some `ParameterManagement` class as a data member. As an example, an object of the `hsv_filter::HSVFilter` class keeps a pointer to an object of the `hsv_filter::ParameterManagement` class. Although the ParameterManagement class is not abstract (it probably should be; I'm not completely sure about that; see section 7), the pointer normally points to an instance of some subclass. The basic idea is that the `ParameterManager` has the parameters of the module as data members and provides getter methods for reading the values of these parameters. These getters are called by the filter or detector. How the parameters are set is determined by the subclass in use. The subclasses currently used are `ReconfigureParameterManager` and `ObservedParameterManager`; the use of each subclass corresponds to one of the two ways for creating systems from existing modules, as described in section 4.2.

4.3.1 `ReconfigureParameterManager`

`ReconfigureParameterManager` is the subclass that is used for systems whose modules are implemented as ROS nodes. An object of this class contains a `dynamic_reconfigure` server, which defines a set of ROS parameters that correspond to the parameters of the module. These ROS / `dynamic_reconfigure` parameters can be observed, manipulated, saved, and loaded with the existing ROS facilities like the `dynparam` node (from the `dynamic_reconfigure` pack-

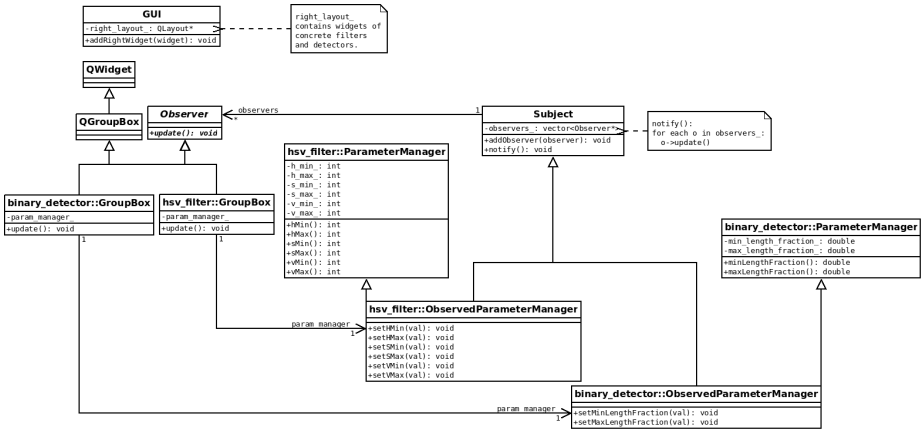

Figure 10: Classes that enable the user to monitor and set parameters

age) and `rqt_reconfigure`. Whenever the value of one of those ROS parameters changes, a callback function within the module (node) is invoked, and the values of the internal parameters are adjusted so that they match those of the corresponding ROS parameters. Therefore, the parameters of the module are manipulated by manipulating the corresponding ROS parameters. Saving/loading the parameters to/from file can be done by using the existing `dynparam dump` and `dynparam load` commands; these commands can also be triggered from `rqt_reconfigure`.

4.3.2 `ObservedParameterManager`

`ObservedParameterManager` is the subclass that is used for systems that are implemented as single nodes with modules represented by objects, as described in section 4.2.1. The classes that are involved in enabling the user to access the parameters of the system are shown in figure 10.

A general `GUI` class is defined in the `object_detection_2d` and `object_detection_3d` packages. The GUI of the HSV-detection system is shown in figure 11. The GUI provides methods for rendering images / point clouds and detected objects and a menu bar with entries for creating and loading parameter files (as described later). To allow the user to observe and modify the parameters of the specific filters and detectors of the system, the GUI provides the `addRightWidget` method, which adds a widget to the layout on the right of the GUI.

To use a module in this way, you have to create a widget that provides access to its parameters. As an example, `hsv_filter::GroupBox` is the widget that provides access to the parameters of the HSV-filter.

The communication between the widget and the `ObservedParameterManager` of a module is based on the Model-View-Controller pattern (see Wikipedia for a description); the `ObservedParameterManager` is the model, the widget acts as the view and the controller. The relationships between the participating

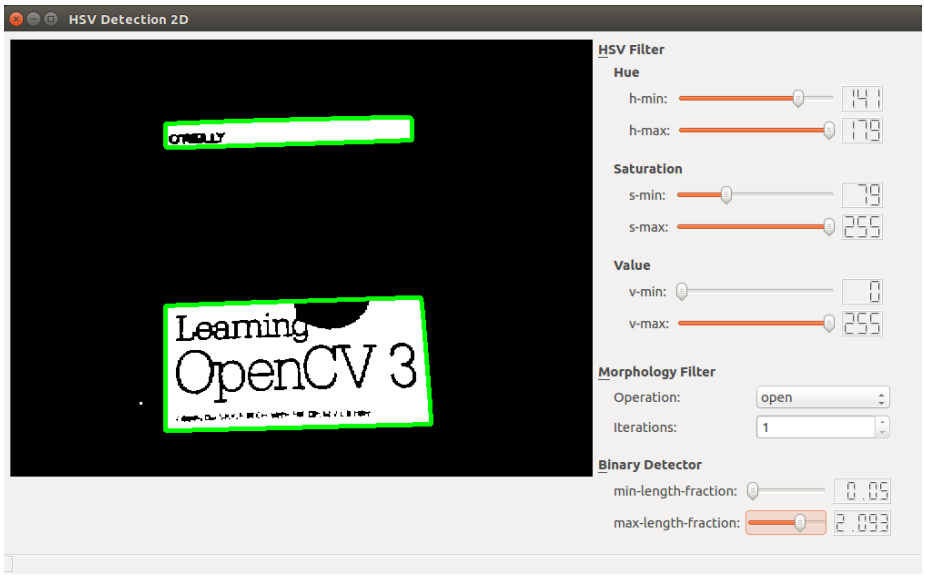

Figure 11: GUI for creating parameter files for the HSV detection system

classes are depicted in figure 10. Whenever the user changes some value in the widget, the widget sends a request to the parameter-manager by calling its setter-methods. Whenever a parameter of an `ObservedParameterManager` object changes, the object calls its `notify` method, which calls the `update` method of its observers (normally there is only one observer). The `update` method of a widget updates its view with the current values of the parameters by calling the getter-methods of the subject. According to figure 10, the widget and the `ObservedParameterManager` are connected by (1) adding the widget as an observer to the subject and (2) passing a reference (pointer) to the `ObservedParameterManager` to the widget. Then the widget is added to the GUI by calling its `addRightWidget` method.

The classes that participate in saving parameters to a file and loading them from such a file are depicted in figure 12.

In addition to inheriting from `Subject`, each `ObservedParameterManager` inherits from `Parametrizable`. The exact mechanisms differ for 2D and 3D systems, but the following general description conveys the basic idea. `Parametrizable` is an abstract class that provides methods for saving/loading parameters to/from file (for 2D systems) or for creating descriptions that can be written to disk by `ParameterStorage` and for parsing descriptions that have been read from such a file (for 3D systems). The `GUI` has a reference to a `ParameterStorage` object, which has a vector of references to `Parametrizables`. When the user clicks on the entry for saving the parameters in the menu bar of the GUI, the GUI gets the name of the parameter file from the user and calls the `saveParameters` method of the `ParameterStorage` object. In this method, the `ParameterStorage` ob-

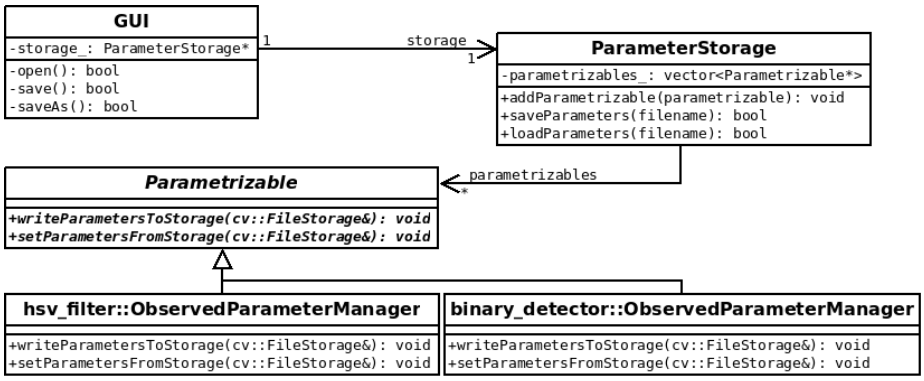

Figure 12: Classes that are involved in the creation of parameter files

ject opens the file with the given name and passes a handle to that file (a `cv::FileStorage` object) to each registered `Parametrizable` so that each one can write its parameters into the file. (In 3D systems, this method collects a parameter description from each `Parametrizable` and writes the descriptions into the parameter file.) When the user clicks on the entry for opening a parameter file in the menu bar of the GUI, the GUI gets the name of the file from the user and calls the `loadParameters` method of the `ParameterStorage` object. In this method, the file is opened for reading and a handle to the file is passed to each registered `Parametrizable` so that each one can read its parameters from the file. (In 3D systems, this method reads the parameter file, creates a parameter-description from it, and passes the description to each `Parametrizable`. Each `Parametrizable` then looks for the values of its parameters in the description and sets its parameters accordingly.)

5 Navigation

The navigation program resides in the `whs_navigation` package. It is based on a state machine and it's meant to be adaptable to specific problems. Look into part III of my thesis for the description of the navigation node and the preliminary process of creating a map of your environment and determining goal-poses. The description given there is still up-to-date.

6 Packages

This section gives a short description of the packages of the `tb_ws` workspace, which contains the source code of this project. For the C++ code, the facilities (classes, functions, enums, etc.) that are defined in a package normally use the name of the package as a namespace.

6.1 binary_detector

This package contains the definition of the `BinaryDetector` class and the corresponding widget (`GroupBox`) and parameter-managers (`ParameterManager`, `ReconfigureParameterManager`, and `ObservedParameterManager`).

The `BinaryDetector` is a 2D-detector that interprets images as binary, i.e. it converts images to grayscale and only distinguishes between pixels with value zero and pixels with non-zero values. A contiguous area of non-zero pixels is considered a detected object if the length of its contour lies within an adjustable range. The lower and upper limits of this range are the parameters of the module.

6.2 distance_filter

This package contains the definition of the `DistanceFilter` class and the corresponding widget (`GroupBox`) and parameter-managers (`ParameterManager`, `ReconfigureParameterManager`, and `ObservedParameterManager`).

The `DistanceFilter` filters (3D) point clouds. It filters out all points whose distance does not lie within an adjustable range. The lower and upper limits of this range are the parameters of the module.

6.3 feature_detector

This package contains the definition of the `FeatureDetector` class and the corresponding widget (`GroupBox`, which contains `DescriptorMatcherGroupBox`, `SurfGroupBox`, and `ThresholdsGroupBox`) and parameter-managers (`ParameterManager`, `ReconfigureParameterManager`, and `ObservedParameterManager`).

The `FeatureDetector` detects objects based on SURF features as described in section 3.2.

6.4 hsv_filter

This package contains the definition of the `HSVFilter` class and the corresponding widget (`GroupBox`) and parameter-managers (`ParameterManager`, `ReconfigureParameterManager`, and `ObservedParameterManager`).

As described in section 3.1, the `HSVFilter` filters images in HSV color space. Only pixels whose H-, S-, and V-values lie within adjustable ranges pass through the filter. The limits of these ranges are the parameters of the module.

6.5 morphology_filter

This package contains the definition of the `MorphologyFilter` class and the corresponding widget (`GroupBox`) and parameter-managers (`ParameterManager`, `ReconfigureParameterManager`, and `ObservedParameterManager`).

As described in section 3.1, the `MorphologyFilter` applies morphological transformations on images. The parameters of the module are the type of

transformation (currently this can be 'erode', 'dilate', 'open', or 'close') and the number of iterations for this transformation.

6.6 object_detection

This package comprises facilities that are used in 2D and 3D systems. Currently, this only includes the Observer and Subject classes, which were described in section 4.3.

6.7 object_detection_2d

This package defines the general classes that are used in most 2D object recognition systems. These classes include `DefinitionDirector`, `DetectionDirector`, `Detector`, `Filter`, `GUI`, `ParameterStorage`, `Parametrizable`, `ROSDetectorWrapper`, and `ROSFilterWrapper`.

In addition, it defines some message types that are only used in 2D systems.

6.8 object_detection_2d_msgs

This package defines the `DetectedObject2DArray` message type that is described in section 2.4.

6.9 object_detection_2d_nodes

This package contains the nodes that were described in section 4.2 for 2D systems.

There are two groups of nodes in this package. The first group comprises nodes that represent modules as described in section 4.2.2. The corresponding source files are `hsv_filter_node.cpp`, `morphology_filter_node.cpp`, `binary_detector_node.cpp`, and `feature_detector_node.cpp`. In each of these files the corresponding filter or detector object and an instance of the `ROSFilterWrapper` or `ROSDetectorWrapper` classes are created and connected. The second group comprises nodes that represent complete systems as described in section 4.2.1. These nodes are defined in the files `hsv_definition.cpp` (for creating parameter files for the hsv-detection system), `hsv_detection.cpp` (for publishing objects with the hsv-detection system), `feature_definition.cpp` (for creating parameter files for the feature-detection system), and `feature_detection.cpp` (for publishing objects with the feature-detection system).

6.10 object_detection_2d_vis

This package defines two nodes, `visualizer` and `area_selection`. The motivation for their creation is the need of the feature-detector node (with the source file `feature_detector_node.cpp` from the `object_detection_2d_nodes` package) to receive target-selections as described in section 3.2.

When `rviz` with its existing display for images is used, the user is not able to select an area within the image with his mouse. The `visualizer` node in contrast visualizes images that it receives over the ROS network and allows the user to click on the image with his mouse. Whenever a mouse event on the displayed image occurs, the `visualizer` node publishes it using the `object_detection_2d/MouseEvent` message type.

The `area_selection` node subscribes to these mouse events and publishes area-selections. An area-selection begins when the user presses the left mouse button within the displayed image and is completed when the user releases this button. The `object_detection_2d/Rect2d` message type is used to publish area-selections.

Running the `visualizer` and `area_selection` nodes allows the `feature_detector` node to subscribe and react to area-selections.

6.11 object_detection_3d

This package is very similar to `object_detection_2d` and defines the general classes that are used in most 3D object recognition systems.

6.12 object_detection_3d_msgs

This package is very similar to `object_detection_2d_msgs`; it defines the `DetectedObject3DArray` message type that is described in section 2.4.

6.13 object_detection_3d_nodes

This package is very similar to `object_detection_2d_nodes`. It contains the nodes that were described in section 4.2 for 3D systems.

Again, there are two groups of nodes in this package. The first group comprises nodes that represent modules as described in section 4.2.2. The corresponding source files are `distance_filter_node.cpp` and `shape_detector_node.cpp`. In each of these files the corresponding filter or detector object and an instance of the `ROSFilterWrapper` or `ROSDetectorWrapper` classes are created and connected. The second group comprises nodes that represent complete systems as described in section 4.2.1. The only system is the shape-detection system that is described in section 3.3. The node defined in `distance_shape_definition.cpp` is used to create parameter files for this system, the node from `distance_shape_detection.cpp` detects objects based on such a file and publishes them.

6.14 object_painter

This package defines the `object_painter` node. The purpose of this node is to visualize detected objects in 2D detection systems. It subscribes to images and messages of type `DetectedObject2DArray`, it draws the polygons that represent the detected objects into the image, and it publishes the resulting image.

6.15 objects2d_to_objects3d

This package defines the objects2d_to_objects3d node. This node can be used to transform messages of type DetectedObject2DArray into messages of type DetectedObject3DArray.

To do this, it uses the intrinsics of the camera and the following assumptions:

- The detected objects are standing on the floor.
- The camera is attached to the TurtleBot. This means that the optical axis is parallel to the floor and approximately 32 cm high.
- The depth of a detected object is equal to its width.

6.16 objects_to_markers

This package defines the objects_to_markers node. This node subscribes to messages of type object_detection_3d_msgs/DetectedObject3DArray and publishes corresponding messages of type visualization_msgs/MarkerArray, which can be visualized by rviz.

Each marker in a marker-array represents the bounding box of an object. This node is used to visualize the locations of detected 3D-objects in rviz.

6.17 shape_detector

This package contains the definition of the ShapeDetector class and the corresponding widget (GroupBox, which contains the NormalEstimationGroupBox and the SegmentationGroupBox) and parameter-managers (ParameterManager, ReconfigureParameterManager, and ObservedParameterManager).

As described in section 3.3, the ShapeDetector detects cylinders or spheres in point clouds by segmenting the clouds with RANSAC.

6.18 whs_navigation

This package contains the code that is related to the navigation program as described in section 5.

7 TODOs

- The shape-detection system is VERY slow and unreliable (i.e. has many false positives). Integrating the filters of the following two items should improve the system a lot.
- Creating a voxel grid filter that reduces the density of a point cloud and therefore the number of its points. This filter could be based on the pcl::VoxelGrid (pointclouds.org/documentation/tutorials/voxel_grid.php) and would reduce the computing time in 3D detection systems.

- Creating a segmentation filter for 3D systems. Similar to the shape-detector, this filter would be based on the `pcl::SACSegmentation` class. The parameters could include the shape (e.g. plane or cylinder) and a boolean for determining if the points that belong to the shape or the points that do not belong to it are filtered out. The main use of this filter would probably be to filter out planes (the floor and walls).

- The packages `object_detection_2d` and `object_detection_2d_msgs` both define custom message types. This is confusing; a clear seperation of concerns/responsibilities is missing. It's probably better to put all message type definitions into one of the two packages.

  The same is true for `object_detection_3d` and `object_detection_3d_msgs`, even though the former package doesn't define any message types yet.

- The seperation of concerns for the `ParameterManager` class and its subclasses is not clear. `ParameterManager` is not abstract and contains the implementation of functionalities that are common among the subclasses. This makes it hard to decide which implementation-details should be part of `ParameterManager` and which should be included in the subclasses.

- If possible, creating a plug-in for rviz would probably be better than using our own nodes for visualization (`visualizer` and `area_selection` from the `object_detection_2d_vis` package).