Hands-on Learning of ROS Using Common Hardware

Andreas Bihlmaier and Heinz Wörn

Abstract Enhancing the teaching of robotics with hands-on activities is clearly beneficial. Yet at the same time, resources in higher education are scarce. Apart from the lack of supervisors, there are often not enough robots available for undergraduate teaching. Robotics simulators are a viable substitute for some tasks, but often real world interaction is more engaging. In this tutorial chapter, we present a hands-on introduction to ROS, which requires only hardware that is most likely already available or costs only about 150$. Instead of starting out with theoretical or highly artificial examples, the basic idea is to work along tangible ones. Each example is supposed to have an obvious relation to whatever real robotic system the knowledge should be transfered to afterwards. At the same time, the introduction covers all important aspects of ROS from sensors, transformations, robot modeling, simulation and motion planning to actuator control. Of course, one chapter cannot cover any subsystem in depth, rather the aim is to provide a big picture of ROS in a coherent and hands-on manner with many pointers to more in-depth information. The tutorial was written for ROS Indigo running on Ubuntu Trusty (14.04). The accompanying source code repository is available at https://github.com/andreasBihlmaier/holoruch.

Keywords General introduction · Hands-on learning · Education

1 Introduction

Individual ROS packages are sometimes well documented and sometimes not. However, the bigger problem for somebody working with ROS for the first time is not the poor documentation of individual packages. Instead the actual problem

A. Bihlmaier (✉) · H. Wörn
Institute for Anthropomatics and Robotics (IAR), Intelligent Process Control and Robotics Lab (IPR), Karlsruhe Institute of Technology (KIT),
76131 Karlsruhe, Germany
e-mail: andreas.bihlmaier@kit.edu

H. Wörn
e-mail: woern@kit.edu

is to understand the big picture-to understand how all the various pieces of ROS come together.[1] Although the ROS community provides tutorials,[2] in our experience undergraduates struggle to transfer what they have learned in the sandbox tutorials to real systems. At the same time, it is difficult to start out with real ROS robots for two reasons. First, real robots are expensive, easily broken and often require significant space to work with. Therefore they are often not available in sufficient quantity for undergraduate education. Second, real robots can be dangerous to work with. This holds true especially if the goal is to provide a hands-on learning experience for the students, i.e. allow them to explore and figure out the system by themselves.

Ideally, each student would be provided with a simple robot that is safe and yet capable enough to also learn more advanced concepts. To our knowledge no such device is commercially available in the range of less than 200$. We will not suggest how one could be built, since building it for each student would be too time consuming. Instead, the goal of this tutorial is to detail a hands-on introduction to ROS on the basis of commonly available and very low-cost hardware, which does not require tinkering. The main hardware components are one or two webcams, a Microsoft Kinect or Asus Xtion and two or more low-power Dynamixel servos. The webcams may also be laptop integrated. Optionally, small embedded computers such as Raspberry Pis or BeagleBone Blacks can be utilized. We assume the reader to understand fundamental networking and operating system concepts, to be familiar with the basics of Linux including the command line and to know C++ or Python. Furthermore, some exposure to CMake is beneficial.

The remainder of the chapter is structured as follows:

- First, a brief background section on essential concepts of ROS. It covers the concepts of the ROS master, names, nodes, messages, topics, services, parameters and launch files. This section should be read on a first reading. However, its purpose is also to serve as glossary and reference for the rest of the chapter.
- Second, a common basis in terms of the host setup is created.
- Third, working with a single camera, e.g. a webcam, under ROS serves as an example to introduce the computation graph: nodes, topics and messages. In addition, rqt and tools of the image_pipeline stack are introduced.
- Fourth, a custom catkin package for filtering sensor_msgs/Image is created. Names, services, parameters and launch files are presented. Also, the definition of custom messages and services as well as the dynamic_reconfigure and vision_opencv stacks are shown.
- Fifth, we give a short introduction on how to use RGB-D cameras in ROS, such as the Microsoft Kinect or Asus Xtion. Point clouds are visualized in rviz and pointers for the interoperability between ROS and PCL are provided.
- Sixth, working with Dynamixel smart servos is explained in order to explain the basics of ros_control.

---

[1]At least this has been the experience in our lab, not only for undergraduates but also for graduate students with a solid background in robotics, who had never worked with ROS before.

[2]http://wiki.ros.org/ROS/Tutorials.

- Seventh, a simple robot with two joints and a camera at the end effector is modelled as an URDF robot description. URDF visualization tools are shown. Furthermore, tf is introduced.
- Eighth, based on the URDF model, a MoveIt! configuration for the robot is generated and motion planning is presented exemplary.
- Ninth, the URDF is extended and converted to SDF in order to simulate the robot with Gazebo.

2 Background

ROS topics are an implementation of the publish-subscribe mechanism, in which the ROS Master serves as a well-known entry point for naming and registration. Each ROS node advertises the topics it publishes or subscribes to the ROS Master. If a publication and subscription exist for the same topic, a direct connection is created between the publishing and subscribing node(s), as shown in Fig. 1. In order to have a definite vocabulary, which may also serve the reader as glossary or reference, we give a few short definitions of ROS terminology:

- **Master**[3]: Unique, well-known (`ROS_MASTER_URI` environment variable) entry point for naming and registration. Often referred to as roscore.
- **(Graph Resource) Name**[4]: A name of a resource (node, topic, service or parameter) within the ROS computation graph. The naming scheme is hierarchical and has many aspects in common to UNIX file system paths, e.g. they can be absolute or relative.
- **Host**: Computer within the ROS network, identified by its IP address (`ROS_IP` environment variable).
- **Node**[5]: Any process using the ROS client API, identified by its graph resource name.
- **Topic**[6]: A unidirectional, asynchronous, strongly typed, named communication channel as used in the publish-subscribe mechanism, identified by its graph resource name.
- **Message**[7]: A specific data structure, based on a set of built-in types,[8] used as type for topics. Messages can be arbitrarily nested, but do not offer any kind of is-a (inheritance) mechanism.

---

[3]http://wiki.ros.org/Master.

[4]http://wiki.ros.org/Names.

[5]http://wiki.ros.org/Nodes.

[6]http://wiki.ros.org/Topics.

[7]http://wiki.ros.org/Messages.

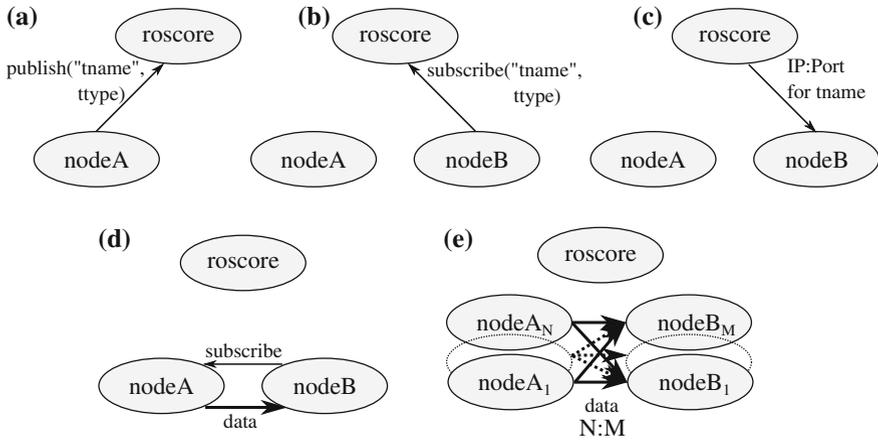[8]http://wiki.ros.org/msg.

Fig. 1 Overview of the ROS topic mechanism. When a publisher (**a**) and subscriber (**b**) are registered to the same topic, the subscriber receives the network address and port of all publishers (**c**). The subscriber continues by directly contacting each publisher, which in return starts sending data directly to the subscriber (**d**). Many nodes can publish and subscribe to the same topic resulting in a $N : M$ relation (**e**). On the network layer there are $N \cdot M$ connections, one for each (publisher, subscriber) tuple. The nodes can be distributed over any number of hosts within the ROS network

- **Connection** A connection between a (publisher, subscriber) tuple carrying the data of a specific topic (cf. Fig. 1), identified by the tuple (publisher-node, topic, subscriber-node).
- **Service**[9]: A synchronous remote procedure call, identified by its graph resource name.
- **Action**[10]: A higher-level mechanism built on top of topics and services for long-lasting or preemptable tasks with intermediate feedback to the caller.
- **Parameters**[11]: "A shared, multi-variate dictionary that is accessible via network APIs." Its intended use is for slow changing data, such as initialization arguments. A specific parameter is identified by its graph resource name.
- **roslaunch**[12]: A command line tool and XML format to coherently start a set of nodes including remapping of names and setting of parameters.

Topics are well suited for streaming data, where each subscriber is supposed to get as much of the data as he can process in a given time interval and the network can deliver. The network overhead and latency is comparatively low because the connections between publisher and subscriber remain open. However, it is important to remember that messages are automatically dropped, if the subscriber queue becomes full. In contrast, services establish a new connection per call and cannot

---

[9]http://wiki.ros.org/Services.

[10]http://wiki.ros.org/actionlib.

[11]http://wiki.ros.org/ParameterServer.

[12]http://wiki.ros.org/roslaunch.

fail without notice on the caller side. The third mechanism built on top of topics and services are actions. An action uses services to initiate and finalize a (potentially) long-lasting task, thereby providing definite feedback. Between start and end of the action, continuous feedback is provided via the topics mechanism. This mapping to basic communication mechanisms is encapsulated by the actionlib.

3 ROS Environment Configuration

For the rest of this chapter, we assume a working standard installation[13] of ROS Indigo-on Ubuntu Trusty (14.04). Furthermore, everything shown in this chapter can be done on a single host. Therefore, a localhost setup is assumed, i.e. roscore and all nodes run on localhost:

```
echo 'export ROS_MASTER_URI=http://127.0.0.1:11311' >> ~/.bashrc
echo 'export ROS_IP=127.0.0.1' >> ~/.bashrc

# run "source ~/.bashrc" or open a new terminal
echo $ROS_MASTER_URI
# should: http://127.0.0.1:11311
echo $ROS_IP
# should: 127.0.0.1
```

From here on it is presumed that `roscore` is always running. The second part of setup requires a working catkin build system.[14] In case no catkin workspace has been initialized, this can be achieved with

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/src
catkin_init_workspace
```

Afterwards all packages in the catkin workspace source directory (`~/catkin_ws/src`) can be built with

```
cd ~/catkin_ws
catkin_make
```

The `catkin_make` command should run without errors for an empty workspace. In the rest of this chapter, the instruction "Install the `REPOSITORY` package from source" refers to cloning the repository into the `~/catkin_ws/src` directory, followed by running `catkin_make`. Start by installing https://github.com/andreasBihlmaier/holoruch, which will be used throughout this chapter. After these preliminaries, we can start the hands-on introduction to the various aspects of ROS.

---

[13]A desktop-full installation according to http://wiki.ros.org/indigo/Installation/Ubuntu.

[14]http://wiki.ros.org/catkin.

4 Camera Sensors: Driver, Use and Calibration

The first example is a camera sensor. A USB webcam or integrated camera must be attached to the computer. If multiple cameras are connected to the computer, the first one is used here.[15] Where these settings are stored and how to change them will be shown later in this section. Use guvcview to make sure the camera is working correctly and also have a look at the "Image Controls" and "Video" tab for supported resolutions/framerates and control settings.[16]

First, install https://github.com/ktossell/camera_umd from source:

```
cd ~/catkin_ws/src ; git clone https://github.com/ktossell/
    ↪ camera_umd.git
cd ~/catkin_ws ; catkin_make
```

Second, start the camera driver node

```
roslaunch holoruch_camera webcam.launch
```

Third, use the command line utilities to check if the camera node is running as expected:

```
rosnode list
```

The output should contain /webcam/uvc_camera_webcam. We also want to check if the expected topics have been created:

```
rostopic list
```

Here, /webcam/camera_info and /webcam/image_raw is expected. If either the node or the topics do not show up, look at the output of roslaunch and compare possible errors to the explanation of the launch file in this section. Next, start rqt to visualize the image data of the webcam

```
rosrun rqt_gui rqt_gui
```

Start the "Image View" plugin through the menu: "Plugins" → "Visualization" → "Image View". Use the upper left drop-down list to select /webcam/image_raw. Now the webcam's live image should appear. Note that everything is already network transparent. If we would not be using localhost IP addresses, the camera node and rqt could be running on two different hosts without having to change anything.

After a simple data source and consumer have been setup and before we add more nodes, let's look at all of the involved ROS components. First the roslaunch file webcam.launch:

---

[15]The order is determined by the Linux kernel's Udev subsystem.

[16]Note: While it does not matter for this tutorial, it is essential for any real-world application that the webcam either has manual focus or the auto focus can be disabled. Very cheap or old webcams have the former and better new ones usually have the latter. For the second kind try v4l2-ctl -c focus_auto=0.

```
<launch>
  <node ns="/webcam"
        pkg="uvc_camera" type="uvc_camera_node" name="uvc_camera_webcam"
        output="screen">
    <param name="width" type="int" value="640" />
    <param name="height" type="int" value="480" />
    <param name="fps" type="int" value="30" />
    <param name="frame" type="string" value="wide_stereo" />

    <param name="auto_focus" type="bool" value="False" />
    <param name="focus_absolute" type="int" value="0" />
    <!-- other supported params: auto_exposure,
         exposure_absolute, brightness, power_line_frequency -->

    <param name="device" type="string" value="/dev/video0" />
    <param name="camera_info_url" type="string"
           value="file://$(find holoruch_camera)/webcam.yaml" />
  </node>
</launch>
```

The goal here is to introduce the major elements of a roslaunch file, not to detail all features, which are described in the ROS wiki.[17] Each `<node>` tag starts one node, in our case `uvc_camera_node` from the `uvc_camera` package we installed earlier. The name, `uvc_camera_webcam` can be arbitrarily chosen, a good convention is to combine the executable name with a task specific description. If `output` was not set to `screen`, the node's output would not be shown in the terminal, but sent to a log file.[18] Finally, the namespace tag, `ns`, allows to prefix the node's graph name, which is in analogy with pushing it down into a subdirectory of the filesystem namespace. Note that the nodes are not started in any particular order and there is no way to enforce one. The `<param>` tags can be either direct children of `<launch>` or within a `<node>` tag.[19] Either way they allow to set values on the parameter server from within the launch file before any of the nodes are started. In the latter case, which applies here, each `<param>` tag specifies a private parameter[20] for the parent `<node>` tag. The parameters are node specific.

In case of the `uvc_camera_node`, the parameters pertain to camera settings. This node uses the Linux Video4Linux2 API[21] to retrieve images from any video input device supported by the kernel. It then converts each image to the ROS image format `sensor_msgs/Image` and published them over a topic, thereby making them available to the whole ROS system.[22] If the launch file does not work as it is, this is most likely related to the combination of `width`, `height` and `fps`. Further

---

[17]http://wiki.ros.org/roslaunch/XML.

[18]See http://wiki.ros.org/roslaunch/XML/node.

[19]See http://wiki.ros.org/roslaunch/XML/param.

[20]Cf. http://wiki.ros.org/Names.

[21]http://lwn.net/Articles/203924/.

[22]See also http://wiki.ros.org/image_common.

information on the `uvc_camera` package is available on the ROS wiki[23] and in the repository's example launch files.[24]

In order to get a transformation between the 3D world and the 2D image of the camera, an instrinsic calibration of the camera is required. This functionality is available, for mono and stereo cameras, through the `cameracalibrator.py` in the `camera_calibration` package. Here one essential feature and design pattern of ROS comes into play. The `cameracalibrator.py` node does not require command line arguments for changing the relevant ROS names, such as the image topic. Instead ROS provides runtime name remapping. Any ROS graph name within a node's code can be changed by this ROS mechanism on startup of the node. In our case, the `cameracalibrator.py` code subscribes to a `"image"` topic, but we want it to subscribe to the webcam's images on `/webcam/image_raw`. The launch syntax provides the `<remap>` tag for this purpose:

```
<launch>
  <node pkg="camera_calibration" type="cameracalibrator.py"
        name="calibrator_webcam" output="screen"
        args="--size 8x6 --square 0.0255">
    <remap from="image" to="/webcam/image_raw" />
    <remap from="camera" to="/webcam" />
  </node>
</launch>
```

The same can be achieved on the command line by the `oldname:=newname` syntax:

```
rosrun camera_calibration cameracalibrator.py \
  --size 8x6 --square 0.0255 \
  image:=/webcam/image_raw camera:=/webcam
```

In both cases, the same node with the same arguments and remappings is started, albeit with a different node name. The monocular calibration tutorial[25] explains all steps to calibrate the webcam using a printed checkerboard. We continue with the assumption this calibration has been done and "commited".[26]

Commonly required image processing tasks, such as undistorting and rectification of images, are available in the `image_proc` package.[27] Due to the flexibility of the topic mechanism, we do not have to restart the camera driver, rather we just add further nodes to the ROS graph.[28] Run

```
roslaunch holoruch_camera proc_webcam.launch
```

---

[23]http://wiki.ros.org/uvc_camera.

[24]~/catkin_ws/src/camera_umd/uvc_camera/launch/example.launch.

[25]http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration.

[26]Check that the values of `rostopic echo -n 1 /webcam/camera_info` correspond to those printed to the terminal by `cameracalibrator.py`.

[27]http://wiki.ros.org/image_proc.

[28]If the camera driver and image processing is running on the same host, it is good practice to use nodelets instead of nodes for both-in order to reduce memory and (de)serialization overhead. However, this does not substantially change anything.
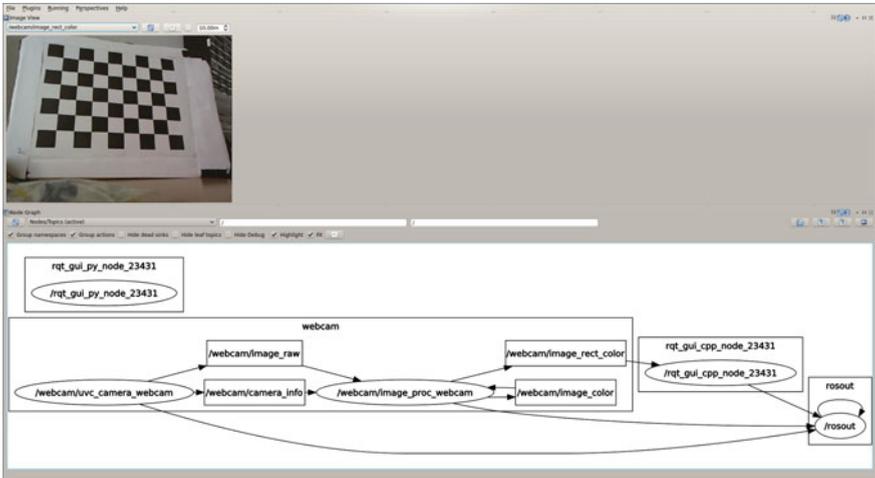
Fig. 2 The screenshot shows `rqt` with the "Image View" plugin on the *top* and the "Node Graph" on the *bottom*. The graph visualizes the simple image processing pipeline consisting of `uvc_camera` and `image_proc`

Afterwards `rostopic list` should show additional topics in the `/webcam` namespace. For a live view of the undistorted image, update the drop-down list in the "Image View" `rqt` plugin and select `/webcam/image_rect_color`.

So far the ROS graph is simple. There only exists the `uvc_camera_node`, whose `image_raw` and `camera_info` topics are subscribed by the `image_proc_webcam` node, whose `image_rect_color` topic is in turn subscribed by the `rqt` plugin. However, graphs of real systems often contain dozens of nodes with hundreds of topics. Fortunately, ROS provides introspection capabilities. That is, ROS provides mechanisms to acquire some key information about the current system state. One important information is the structure of the computation graph, i.e. which nodes are running and to which topics is each one publishing or subscribing. This can be visualized with the "Node Graph" `rqt` plugin (under "Plugins" → "Introspection"). The current ROS graph can be seen in Fig. 2. Next, we will write a custom ROS node that does custom image processing on the webcam image stream.

5 Custom Node and Messages for Image Processing with OpenCV

In this section we will create a custom catkin package for image processing on a `sensor_msgs/Image` topic.[29] Also, the definition of custom messages and services as well as the `dynamic_reconfigure` and `vision_opencv` stacks

---

[29]The goal of this section is to provide an example which is short, but at the same time very close to a real useful node. Standalone examples of how to create ROS publishers and subscribers in C++

are shown. First, create a new package, here `holoruch_custom`, and specify all dependencies that are already known[30]:

```
cd ~/catkin_ws/src
catkin_create_pkg holoruch_custom roscpp dynamic_reconfigure
cv_bridge
```

The slightly abbreviated code for the core node functionality in `holoruch_custom/src/holoruch_custom_node.cpp` is shown below

```
1   int main(int argc, char **argv) {
2     ros::init(argc, argv, "holoruch_custom");
3     ros::NodeHandle n;
4
5     sub = n.subscribe("image_raw", 1, imageCallback);
6     pub = n.advertise<sensor_msgs::Image>("image_edges", 1);
7
8     ros::spin();
9     return 0;
10  }
11
12  void imageCallback(const sensor_msgs::ImageConstPtr& msg) {
13    cv_bridge::CvImagePtr cvimg = cv_bridge::toCvCopy(msg, "bgr8");
14
15    cv::Mat img_gray;
16    cv::cvtColor(cvimg->image, img_gray, CV_BGR2GRAY);
17    cv::Canny(img_gray, img_gray, cfg.threshLow, cfg.threshHigh);
18    cv::Mat img_edges_color(cvimg->image.size(), cvimg->image.type(),
19                          cv::Scalar(cfg.edgeB, cfg.edgeG, cfg.edgeR
20  ));
21    img_edges_color.copyTo(cvimg->image, img_gray); // use img_gray as
22  mask
23
24    pub.publish(cvimg->toImageMsg());
25  }
```

After initializing the node (2), we create a subscriber to receive the images (5) and a publisher to send the modified ones (6). The remaining work will be done in the subscriber callbacks, which are handled by the ROS spinner (8). Each time a new image arrives, the `imageCallback` function is called. It converts the ROS image format to the OpenCV format (13) using the `cv_bridge` package.[31] Afterwards some OpenCV functions are applied to the data (15–20), the resulting image is converted back to ROS and published (22). In the example, we apply an edge filter

---

(Footnote 29 continued)

and Python are available in the ROS wiki: http://wiki.ros.org/ROS/Tutorials. Very basic standalone packages with examples can also be found at: https://github.com/andreasBihlmaier/ahb_rospy_example and https://github.com/andreasBihlmaier/ahb_roscpp_example.

[30]These steps are for illustration only, the full `holoruch_custom` and `holoruch_custom_msgs` package is already contained in the `holoruch` repository.

[31]See http://wiki.ros.org/cv_bridge. We will not go into any OpenCV details such as color encodings here. For more information see http://wiki.ros.org/cv_bridge/Tutorials/UsingCvBridgeToConvertBetweenROSImagesAndOpenCVImages and the OpenCV documentation at http://opencv.org/documentation.html.

to the image and overlay the edges on the original one. There is much more to be learned about working with images in ROS. However, since the goal of this tutorial is to give the big picture, we cannot go into further details here.[32]

Now we add the functionality to change the filter settings during runtime using `dynamic_reconfigure`. If the goal would be to change the settings only on startup (or very seldom), we would use the parameter server instead.[33] First, we need to define the parameters, which is accomplished by creating a `holoruch_custom/cfg/filter.cfg` file:

```
# imports omitted from listing
g = ParameterGenerator()
#     Name      Type   Rcfg-lvl Description           Default Min Max
g.add("edgeR", int_t, 0,        "Edge Color Red",     255,    0,  255)
g.add("edgeG", int_t, 0,        "Edge Color Green",   0,      0,  255)
# further parameters omitted from listing
exit(g.generate(PACKAGE, "holoruch_custom", "filter"))
```

Second, we need to initialize a `dynamic_reconfigure` in `main`. Therefore, we add the following in line 7 above

```
  dynamic_reconfigure::Server<holoruch_custom::filterConfig> server;
  server.setCallback(boost::bind(&dynamic_reconf_cb, _1, _2));
```

and a new callback function, e.g. after line 23, for reconfigure

```
void dynamic_reconf_cb(holoruch_custom::filterConfig &ncfg, uint32_t lvl
)
{ // cfg is a global variable: holoruch_custom::filterConfig cfg
  cfg = ncfg;
}
```

We refer to the `holoruch_custom` repository regarding the required additions in `CMakeLists.txt`.

After compiling the workspace with `catkin_make`, we can now run our custom node. This time we use the `ROS_NAMESPACE` environment variable to put `holoruch_custom_node`, and thereby its depend names, into `/webcam`.

```
env ROS_NAMESPACE=/webcam rosrun holoruch_custom holoruch_custom_node \
  image_raw:=image_rect_color
```

Again, the same could be achieved by remapping alone or by the roslaunch `ns` tag as shown in the previous section. The result is shown in Fig. 3, the filter settings can be continuously changed while the node is running by use of the "Dynamic Reconfigure" `rqt` plugin.

So far we have only used predefined ROS message types. Let's assume we want to publish the number of connected edge pixel components, which are detected by our node, together with the number of pixels in each one. No adequate predefined

---

[32]Good documentation and tutorials to delve into this topic can be found at http://wiki.ros.org/image_common, http://wiki.ros.org/camera_calibration and http://wiki.ros.org/cv_bridge. For more information on ROS supported sensors, see "Cameras" and "3D sensors" at http://wiki.ros.org/Sensors.

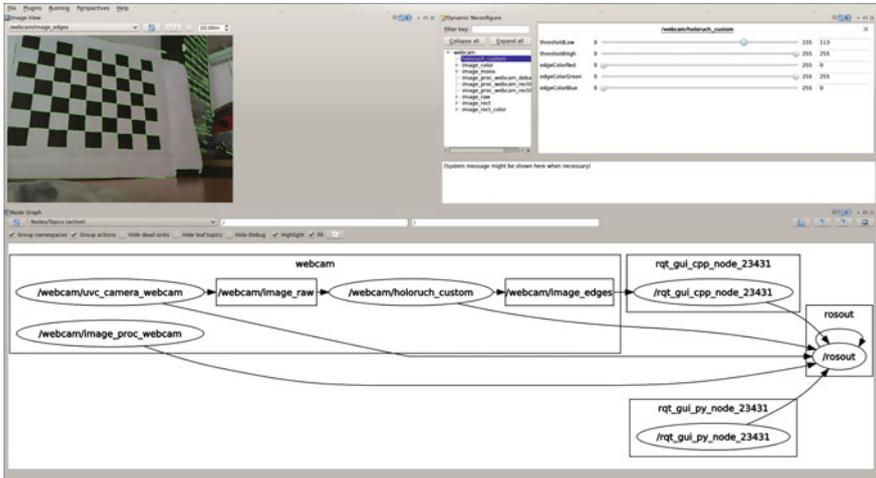[33]See http://wiki.ros.org/roscpp/Overview/ParameterServer. for an example.

Fig. 3 A screenshot of the custom image processing node's output `/webcam/image_edges` in `rqt`. Compared to Fig. 2, the "Dynamic Reconfigure" plugin was added in the *upper right* and the "Node Graph" in the *bottom* shows the additional node `holoruch_custom` and its connections

message exists for this purpose, thus we will define a custom one.[34] We could put the message definitions into `holoruch_custom`, but it is favorable to have them in a separate package. This way other nodes that want to use the custom message only have to depend on the message package and only compile it. Thus we create it

```
catkin_create_pkg holoruch_custom_msgs \
  message_generation message_runtime std_msgs
```

Our custom message `holoruch_custom_msgs/EdgePixels`, is defined by the file `holoruch_custom_msgs/msg/EdgePixels.msg`[35]:

```
std_msgs/Header header
int32[] edge_components
```

Again, see the repository for details about message generation with `CMakeLists. txt`. After running `catkin_make`, the custom message is available in the same manner as predefined ones:

```
rosmsg show holoruch_custom_msgs/EdgePixels
```

The custom message can now be used in `holoruch_custom_node.cpp` to communicate information about edge components in the image. We would create a second publisher in `main` at line 6:

```
compPub = n.advertise<holoruch_custom_msgs::EdgePixels>("edges", 1);
```

---

[34] It is highly advisable to make sure that no matching message type already exists before defining a custom one.

[35] For available elementary data types see http://wiki.ros.org/msg.

and the code to calculate and afterwards publish the components would be inserted at line 21 in `imageCallback`. Without going into the details of catkin dependencies, please refer to the `holoruch` repository and wiki documentation,[36] it is necessary to add a dependency to `holoruch_custom_msgs` into `holoruch_custom/package.xml` and `holoruch_custom/CMakeLists.txt`. This concludes the introduction to creating a custom ROS node. Next, we look a different kind of optical sensor.

6 RGB-D Sensors and PCL

RGB-D sensors such as the Microsoft Kinect or the Asus Xtion provide two kinds of information: First, a RGB image just like a normal camera. Second, a depth image usually encoded as a grayscale image, but with each gray value measuring distance to the camera instead of light intensity. Separately, the RGB and depth image can be used as shown in the previous sections. However, the depth image can also be represented as a point cloud, i.e. an ordered collection of 3D points in the camera's coordinate system. If the RGB and depth image is registred to each other, which means the external camera calibration is known, a colored point cloud can be generated. The ROS community provides a stack for OpenNI-compatible devices,[37] which contains a launch file to bring up the drivers together with the low-level processing pipeline[38]:

```
roslaunch openni2_launch openni2.launch
```

Use dynamic reconfigure, e.g. the "Dynamic Reconfigure" plugin, to activate "depth registration" and "color_depth_synchronisation" for the `driver` node. Because by default no RGB point clouds are generated.

The most interesting new ROS subsystems that become relevant when working with RGB-D sensors, or with the `sensor_msgs/PointCloud2` message, are `rviz` and the Point Cloud Library (PCL). If no RGB-D sensor is available, but two (web)cameras, have a look at the `stereo_image_proc` stack[39], which can calculate stereo disparity images and process these to point clouds, as well. We want to stress this point once more, since it is one of the big benefits when-properly-using ROS: Message types, sent over topics, represent an abstract interface to the robot system. They abstract not only on which machine data is generated, transformed or consumed, but also how this data has been acquired. In the case at hand, once a `sensor_msgs/PointCloud2` has been created, it does not matter whether it

---

[36]http://wiki.ros.org/ROS/Tutorials/CreatingPackage.

[37]Since the end of 2014 there also exists a similar stack for the Kinect One (aka Kinect v2): https://github.com/code-iai/iai_kinect2.

[38]If `openni2` does not work, also try the older `openni` stack.

[39]In case of using two webcams, do not start a seperate driver node for each, instead use `uvc_stereo_node`. For stereo calibration refer to http://wiki.ros.org/camera_calibration/Tutorials/StereoCalibration.
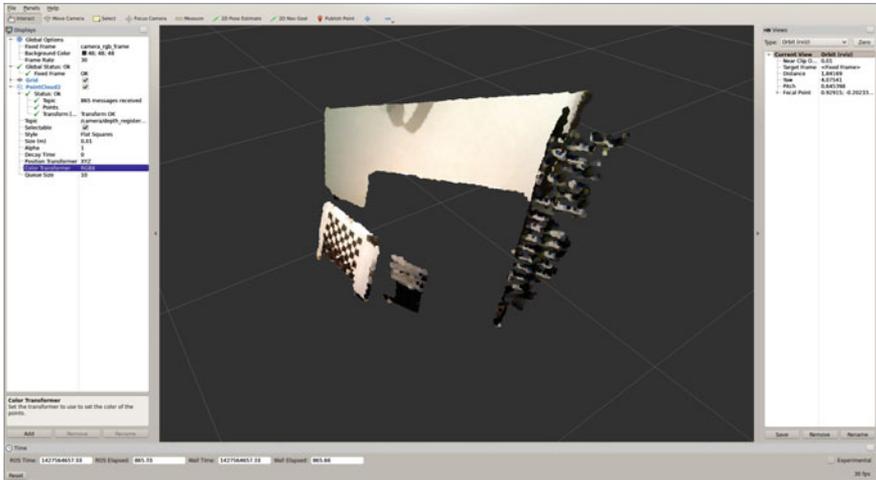
Fig. 4 Live visualization of RGB point clouds (`sensor_msgs/PointCloud2`) with `rviz`. In order to show the colored point cloud, the "Color Transformer" of the "PointCloud2" plugin must be set to RGB8

originates from a stereo camera or a RGB-D sensor or from a monocular structure from motion algorithm and so forth.

The `rviz` "PointCloud2" plugin provides a live 3D visualization of a point cloud topic. We start `rviz` using `rosrun`

```
rosrun rviz rviz
```

and use the "Add" button in order to create a new instance of a visualization plugin. Next we select the "By topic" tab and select the `depth_registered` PointCloud2 entry. The resulting display can be seen in Fig. 4. Furthermore, as will become evident till the end of this chapter, `rviz` is able to combine a multitude of sensor and robot state information into a single 3D visualization.

A custom ROS node for processing point clouds with PCL is very similar to the custom OpenCV node shown in the previous section. Put differently, it is simple and only requires a few lines of code to wrap existing algorithms-based on frameworks such as OpenCV or PCL-for ROS. Thereby benefiting from network transparency and compatibility with the large amount of software available by the ROS community. Again working with stereo cameras or RGB-D sensors is a large topic in its own right and we cannot go into further details here.[40] Having gained experience with two different optical sensor types and a lot of fundamental knowledge in working with ROS, we turn our attention to actuators.

---

[40]We refer to the excellent documentation and tutorials in the ROS wiki (http://wiki.ros.org/pcl/Overview) as well as those for OpenCV (http://opencv.org/documentation.html) and PCL (http://pointclouds.org/documentation/).

Fig. 5 A simple pan-tilt unit is shown, it consists of two Dynamixel AX-12A actuators together with a USB2Dynamixel adapter. Any small webcam or RGB-D camera can be attached to the pan-tilt unit, in the image a Logitech C910 is used. Together the pan-tilt unit and the camera can be used for tabletop robotics experiments. They provide a simple closed sensor actuator loop, allowing to teach advanced robotics concepts

7 Actuator Control: Dynamixel and ROS Control

The Robotis Dynamixel actuators are integrated position or torque controlled "smart servo" motors and can be daisy chained. It is sufficient to connect them on one end of the daisy chain to a computer via an USB adapter and to a 9-12 V power supply. For this chapter we use two Dynamixel AX-12A and one USB2Dynamixel adapter. Each AX-12A has a stall torque of 1.5 Nm, smooth motion is possible up to about 1/5th of the stall torque.[41] As of 2015 the overall cost of the three items is less than 150$. The Dynamixel actuators are popular in the ROS community because-among other things-they can be used out-of-the-box with the `dynamixel_motor` stack.[42] We built a simple pan-tilt unit, which is depicted in Fig. 5. The goal here is not an introduction to `dynamixel_motor`,[43] rather we provide a hands-on example of working with actuators under ROS.

First we need to start the node, which directly accesses the Dynamixel motors via USB:

```
roslaunch holoruch_pantilt pantilt_manager.launch
```

Before proceeding make sure that motor data is received:

```
rostopic echo /motor_states/pan_tilt_port
```

Now we spawn a joint controller for each axis in the pan-tilt unit:

---

[41] http://www.robotis.com/xe/dynamixel_en.

[42] http://wiki.ros.org/dynamixel_motor.

[43] In depth tutorials are available in the ROS wiki: http://wiki.ros.org/dynamixel_controllers/Tutorials.

```
roslaunch holoruch_pantilt pantilt_controller_spawner.launch
```

Note that the above call will terminate after spawning the joint controllers. Before moving the motors, one can set the positioning speed (default is 2.0):

```
rosservice call /pan_controller/set_speed 'speed: 0.5'
rosservice call /tilt_controller/set_speed 'speed: 0.1'
```

To check whether everything worked, try to move each axis:

```
rostopic pub -1 /pan_controller/command std_msgs/Float64 -- 0.4
rostopic pub -1 /tilt_controller/command std_msgs/Float64 -- 0.1
```

Now we could control our pan-tilt unit through a custom node that publishes to each axis' command topic. However, ROS provides higher level mechanisms to work with robots. The next step will thus be to create a model of our two axis "robot".

8 Robot Description with URDF

In ROS robots are described, in terms of their rigid parts (= links) and (moveable) axis (= joints), by the URDF format. The complete holoruch_pantilt.urdf can be found in the holoruch repository. Only the general URDF structure is described here:

```
1   <robot name="holoruch_pantilt">
2     <link name="base_link">
3       <visual>
4         <geometry>
5           <box size="0.15 0.13 0.01" />
6         </geometry>
7       </visual>
8       ...
9     </link>
10
11    <joint name="base_to_pan_joint" type="fixed">
12      <origin xyz="0 0 0.021" rpy="0 0 0"/>
13      <parent link="base_link"/>
14      <child link="pan_link"/>
15    </joint>
16
17    <link name="pan_link">
18      <visual>
19        <geometry>
20          <mesh filename="package://holoruch_pantilt_description/
21           meshes/AX-12A.stl" />
22        </geometry>
23      </visual>
24      <collision>
25        <geometry>
26          <mesh filename="package://holoruch_pantilt_description/
27           meshes/AX-12A_convex.stl" />
28        </geometry>
```

```
29       </collision>
30       <inertial>
31         <origin rpy="0 0 0" xyz="0 0 0"/>
32         <mass value="0.055"/>
33         <inertia ixx="1" ixy="0" ixz="0" iyy="1" iyz="0" izz="1"/>
34       </inertial>
35     </link>
36
37     <joint name="pan_joint" type="revolute">
38       <origin xyz="0 0 0.06" rpy="0 0 0"/>
39       <parent link="pan_link"/>
40       <child link="tilt_link"/>
41       <limit lower="-1.57079" upper="1.57079" effort="1.0" velocity=
42       "1.0" />
43       <axis xyz="0 0 1" />
44     </joint>
45
46     <link name="tilt_link"> ... </link>
47     <joint name="tilt_joint" type="revolute"> ... </joint>
48     <link name="camera_link"> ... </link>
```

Each `<link>` tag has at least a `<visual>`, `<collision>` and `<inertial>` child tag. The visual elements can be either geometric primitives (5) or mesh files (20). For the collision element the same is true with one very important exception: Collision meshes must always be *convex*. If this is not minded, collision checking might not work correctly when using the URDF in combination with MoveIt! (cf. the following section). Each `<joint>` tag has an `origin` relative to its `parent` link and connects it with a `child` link. There are several different joint types, e.g. `fixed` (11) and `revolute` (35). Furthermore, if the joint is moveable, its `axis` of motion has to be specified together with a `limit`. To understand how coordinate systems are specified in URDF please see the relevant drawings in the ROS wiki.[44]

At any time the XML is valid, the current appearance of the robot can be tested by a utility from the `urdf_tutorial` package:

```
roslaunch urdf_tutorial display.launch gui:=True model:=some_robot.urdf
```

This starts `rviz` including the required plugins together with a `joint_state_publisher`, which allows to manually set the joint positions through a GUI. Figure 6 depicts a screenshot for the pan-tilt unit.

Once a robot description has been created it can be used by the `rviz` "Robot-Model" plugin to visualize the current robot state. Furthermore, the URDF is the basis for motion planning, which will be covered next.
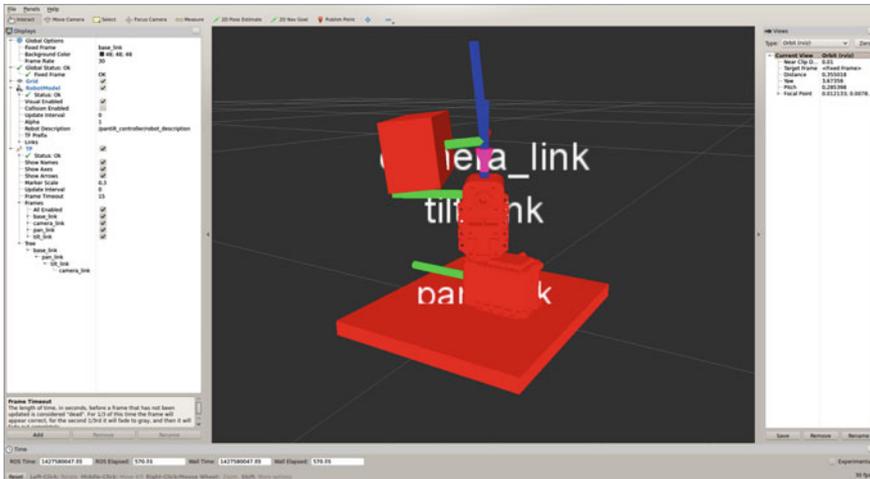
---

[44]http://wiki.ros.org/urdf/XML/model, http://wiki.ros.org/urdf/XML/link and http://wiki.ros.org/urdf/XML/joint.

Fig. 6 `rviz` displaying the live state of the pan-tilt unit. The fixed frame is the base link of the pan-tilt unit (`base_link`). The "RobotModel" plugin uses the URDF robot description from the parameter server (`/pantilt_controller/robot_description`) to visualize the robot links according to the current joint position. The "TF" plugin shows all `tf` frames. In the screenshot these are only the frames of the pan-tilt joints. These joint frames are provided by the `robot_state_publisher` based on the robot description and a `sensor_msgs/JointState` topic (`joint_states`)

9 Motion Planning with MoveIt!

The MoveIt! framework[45] gives ROS users an easy and unified access to many motion planning algorithms. This does not only include collision free path planning according to meshes, but also works in combination with continuously updated obstacles. Furthermore, it provides `rviz` plugins to interactively view trajectory execution and specify motion targets with live pose visualization. In order to use MoveIt!, a lot of additional configuration is required. Fortunately, a graphical setup assistant is provided, which takes care of (almost) all required configuration:

```
roslaunch moveit_setup_assistant setup_assistant.launch
```

We will not go through the configuration process here, the completly configured `holoruch_pantilt_moveit_config`, however, is contained in the repository. For more details and background information, we refer to the online available MoveIt! setup assistant tutorials.[46] After completion of the setup, we can directly use the generated `demo.launch` to test all planning features but without actually sending the trajectory goals to the robot (cf. Fig. 7). Although having a generic motion planner can be demonstrated by means of more impressive examples than

---

[45]http://moveit.ros.org/.

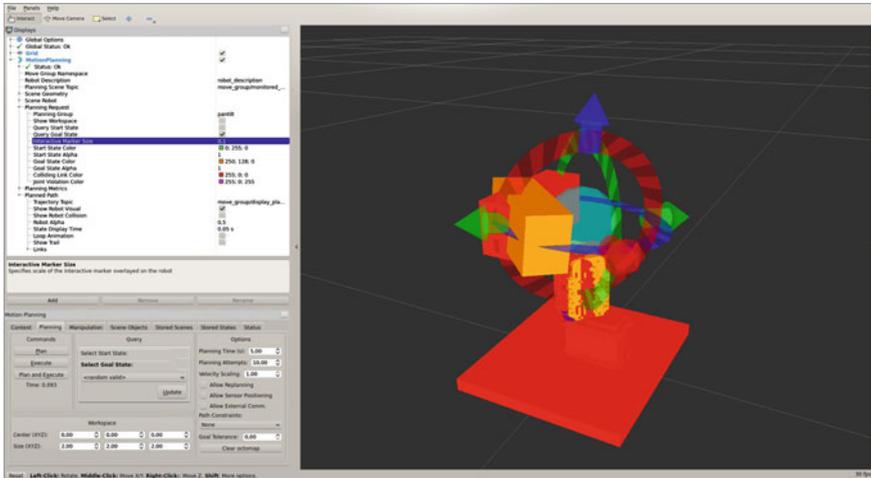[46]http://docs.ros.org/indigo/api/moveit_setup_assistant/html/doc/tutorial.html.

Fig. 7 The "MotionPlanning" `rviz` plugin provided by MoveIt! is shown in combination with the pan-tilt unit. Target positions can be specified by moving the interactive marker. Once the desired target is set, a trajectory from the current pose to the target can be planned by pressing the "Plan" button. If a motion plan is found, it can be directly visualized in different ways. However, this barely scratches the surface of the available "MotionPlanning" functionality

by a two axis pan-tilt unit, MoveIt! also works in this case and could be even useful in applications such as visual servoing. The last point is especially relevant when considering that no modification of code is required when switching between robots with completely different kinematics if the MoveIt! API-or at least the `actionlib` together with `trajectory_msgs/JointTrajectory`-is used instead of custom solutions.

At this point, we have seen how to work with sensor data and control actuators using ROS. In addition, it should have become clear, at least the big picture of it, how one can write custom nodes that process incoming sensor data in order to plan and finally take an action by sending commands to the actuators. The last section of this tutorial will show how the Gazebo simulator can be used together with ROS in order to work without access to the real robot.[47] Everything written so far about ROS, processing sensor data as well as moving robots, can be done without any modification with simulated sensors and simulated actuators. Nodes do not even have to be recompiled for this. Most of the time, even the same launch files can be used.

---

[47]There are also many other important uses for simulated robot instances. One of these is "Robot Unit Testing", which brings unit and regression testing for robotics to a new level, for more information see [1].

10   Robot Simulation with Gazebo

Due to historical reasons, ROS and Gazebo use different formats to describe robots.
The former, URDF, has been introduced in the previous section. The Gazebo for-
mat is called SDF and has a similar structure to URDF, i.e. there are links which
are connected by joints-however, there are also some important differences on the
conceptual level, for example how coordinate systems are represented. Thankfully
it is possible to automatically convert from URDF to SDF[48]:

```
gz sdf --print robot.urdf > model.sdf
```

We have to convert `holoruch_pantilt.urdf` in this manner and also add a
`model.config`.[49] The `model.sdf` and `model.config` file must be copied
into their own subdirectory of `~/.gazebo/models/`. Afterwards Gazebo can be
started and we can add our simulated pan-tilt unit via the "Insert" tab in the left side
menu:

```
rosrun gazebo_ros gazebo
```

After adding the model to the world, the simulated gravity will slowly pull the tilt
axis down until the camera rests on the motor. This happens for two different reasons:
First, we did not specify a `<dynamics>` tag, which contains `friction`, for our
model. Second, the SDF does not contain any sensor or model plugins.

   We will add two plugins to the SDF file: First, a simulated camera sensor[50]:

```
<link name='camera_link'>
     ...
  <sensor name="cam_sensor" type="camera">
    <always_on>1</always_on>
    <visualize>0</visualize>
    <pose>
      0 0 0.116
      3.14159 3.14159 -1.5708
    </pose>
    <update_rate>30</update_rate>
    <camera>
      <horizontal_fov>1.0</horizontal_fov>
      <image>
        <width>640</width>
        <height>480</height>
        <format>R8G8B8</format>
      </image>
      <clip>
        <near>0.0100000</near>
        <far>100.000000</far>
```

---

[48] In case one starts with the SDF description, this can also be converted to an URDF using
`sdf2urdf.py model.sdf robot.urdf`, see http://wiki.ros.org/pysdf. This option might
become even more attractive with the graphical model editor in Gazebo 6, cf. http://gazebosim.org/
tutorials?tut=model_editor.

[49] Cf. http://gazebosim.org/tutorials?tut=build_robot\&cat=build_robot.

[50] http://gazebosim.org/tutorials?tut=ros_gzplugins.

```
        </clip>
      </camera>
      <plugin name="cam_sensor_ros" filename="libgazebo_ros_camera.so"
       >
        <alwaysOn>true</alwaysOn>
        <cameraName>/simcam</cameraName>
        <imageTopicName>image_raw</imageTopicName>
        <cameraInfoTopicName>camera_info</cameraInfoTopicName>
        <frameName>sim_cam</frameName>
      </plugin>
    </sensor>
  </link>
```

Second, a plugin to control the joints of our simulated model[51]:

```
...
  <plugin name="joint_pos_control" filename=
  "libRobotJointPositionControlPlugin.so">
    <robotNamespace>/pantilt</robotNamespace>
    <jointsReadTopic>get_joint_positions</jointsReadTopic>
    <jointsWriteTopic>set_joint_positions</jointsWriteTopic>
  </plugin>
</model>
```
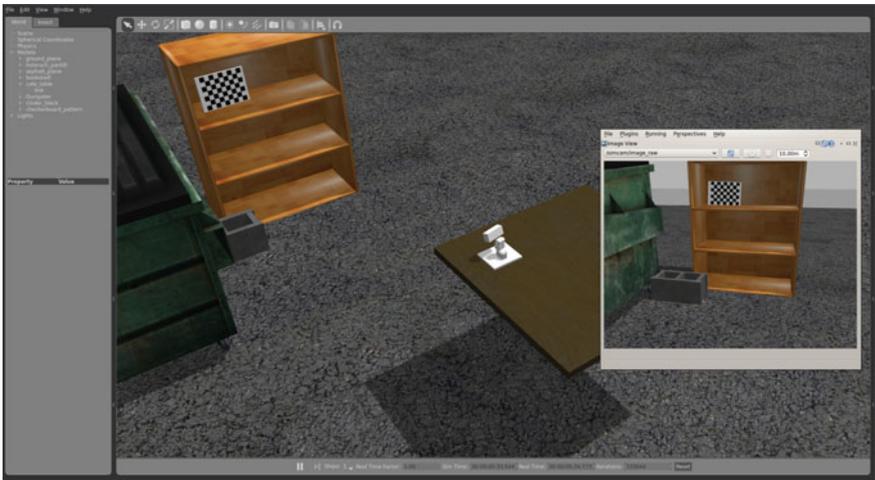

Fig. 8 The screenshot shows a simulated world containing the pan-tilt unit in Gazebo together with the live image provided by the simulated camera in `rqt`. Since the camera images are sent over a standard `sensor_msgs/Image` topic, we could run the simulated images through our common node created in an earlier section

---

[51]For ease of use, we use a custom plugin here. It must be installed from source https://github.com/andreasBihlmaier/robot_joint_position_controller_gazebo. Alternatively Gazebo also provides ROS control interfaces, however they require the robot to be externally spawned via `spawn_model`, see http://gazebosim.org/tutorials/?tut=ros_control.

Both plugins provide the usual ROS interface as would be expected from real hardware. The complete `model.sdf` is available in `holoruch_gazebo`. Figure 8 shows the simulated pan-tilt unit in Gazebo together with the simulated camera image shown in `rviz`. This concludes the last part of our hands-on introduction to ROS.

Reference

1. A. Bihlmaier, H. Wörn, Automated endoscopic camera guidance: a knowledge-based system towards robot assisted surgery, in *Proceedings for the Joint Conference of ISR 2014 (45th International Symposium on Robotics) and ROBOTIK 2014 (8th German Conference on Robotics)*, pp. 617-622 (2014)

Authors' Biography

Andreas Bihlmaier Dipl.-Inform., obtained his Diploma in computer science from the Karlsruhe Institute of Technology (KIT). He is a Ph.D. candidate working in the Transregional Collaborative Research Centre (TCRC) "Cognition-Guided Surgery" and is leader of the Cognitive Medical Technologies group in the Institute for Anthropomatics and Robotics-Intelligent Process Control and Robotics Lab (IAR-IPR) at the KIT. His research focuses on cognitive surgical robotics for minimally-invasive surgery, such as a knowledge-based endoscope guidance robot.

Heinz Wörn Prof. Dr.-Ing., studied electronic engineering at the University of Stuttgart. He did his Phd thesis on "Multi Processor Control Systems". He is an expert on robotics and automation with 18 years of industrial experience. In 1997 he became professor at the University of Karlsruhe, now the KIT, for "Complex Systems in Automation and Robotics" and also head of the Institute for Process Control and Robotics (IPR). Prof. Wörn performs research in the fields of industrial, swarm, service and medical robotics.