10
Programming Vision Sensors Using ROS, Open CV, and PCL

In the last chapter, we discussed the interfacing of sensors and actuators using I/O boards in ROS. In this chapter, we are going to discuss how to interface various vision sensors in ROS and program them using libraries such as **Open Source Computer Vision** (**OpenCV**) and **Point Cloud Library** (**PCL**). The vision of a robot is an important aspect of the robot for manipulating objects and navigation. There are lots of 2D/3D vision sensors available on the market, and most of the sensors have an interface driver package in ROS. We will discuss the interfacing of new vision sensors to ROS and programming them using OpenCV and PCL. Finally, we will discuss the use of fiducial marker libraries to develop vision-based robotic applications.

We will cover the following topics in this chapter:

- Integrating ROS, PCL, and OpenCV
- Using a USB webcam in ROS
- Learning how to calibrate a camera
- Using RGB-D sensors in ROS
- Using laser scanners in ROS
- Working with augmented-reality markers in ROS

Understanding ROS – OpenCV interfacing packages

OpenCV is one of the popular open source real-time computer vision libraries, which is mainly written in C/C++. OpenCV comes with a BSD license, and is free both for academic and commercial applications. OpenCV can be programmed using C/C++, Python, and Java, and it has multi-platform support, such as Windows, Linux, macOS X, Android, and iOS. OpenCV has tons of computer vision APIs that can be used for implementing computer vision applications. The web page of OpenCV library can be found at `http://opencv.org/`.

The OpenCV library is interfaced to ROS via a ROS stack, called `vision_opencv`. `vision_opencv` consists of two important packages for interfacing OpenCV to ROS. They are:

- `cv_bridge`: The `cv_bridge` package contains a library that provides APIs for converting the OpenCV image data type, `cv::Mat`, to the ROS image message called `sensor_msgs/Image` and vice versa. In short, it can act as a bridge between OpenCV and ROS. We can use OpenCV APIs to process the image and convert to ROS image messages whenever we want to send to another node. We will discuss how to do this conversion in the upcoming sections.
- `image_geometry`: One of the first processes that we should do before working with cameras is calibration. The `image_geometry` package contains libraries written in C++ and Python, which helps to correct the geometry of the image using calibration parameters. The package uses a message type called `sensor_msgs/CameraInfo` for handling the calibration parameters and feeding to the OpenCV image rectification function.

Understanding ROS - PCL interfacing packages

The point cloud data can be defined as a group of data points in some coordinate system. In 3D, it has x, y, and z coordinates. The PCL library is an open source project for handling 2D/3D images and point cloud processing.

Like OpenCV, it is under BSD license, and free for academic and commercial purposes. It is also a cross-platform packages that has support in Linux, Windows, macOS, and Android/iOS.

The library consists of standard algorithms for filtering, segmentation, feature estimation, and so on, which are required to implement different point cloud applications. The main web page of the point cloud library can be found at `http://pointclouds.org/`.

The point cloud data can be acquired by sensors such as Kinect, Asus Xtion Pro, Intel Real Sense, and others. We can use this data for robotic applications, such as robot object manipulation and grasping. PCL is tightly integrated into ROS for handling point cloud data from various sensors. The `perception_pcl` stack is the ROS interface for the PCL library. It consists of packages for pumping the point cloud data from ROS to PCL data types and vice versa. `perception_pcl` consists of the following packages:

- `pcl_conversions`: This package provides APIs to convert PCL data types to ROS messages and vice versa.
- `pcl_msgs`: This package contains the definition of PCL-related messages in ROS. The PCL messages are:
    - `ModelCoefficients`
    - `PointIndices`
    - `PolygonMesh`
    - `Vertices`
- `pcl_ros`: This is the PCL bridge of ROS. This package contains tools and nodes to bridge ROS messages to PCL data types and vice versa.
- `pointcloud_to_laserscan`: The main function of this package is to convert the 3D point cloud into a 2D laser scan. This package is useful for converting an inexpensive 3D vision sensor, such as Kinect and Asus Xtion Pro, to a laser scanner. The laser scanner data is mainly used for 2D-SLAM, for the purpose of robot navigation.

Installing ROS perception

We are going to install a single package called `perception`, which is a metapackage of ROS containing all the perception-related packages, such as OpenCV, PCL, and so on:

```
$ sudo apt-get install ros-kinetic-perception
```

The ROS perception stack contains the following ROS packages:

- `image_common`: This metapackage contains common functionalities to handle an image in ROS. The metapackage consists of the following list of packages (`http://wiki.ros.org/image_common`):
    - `image_transport`: This package helps to compress the image during publishing and subscribes the images to save the band width (`http://wiki.ros.org/image_transport`). The various compression methods are JPEG/PNG compression and Theora for streaming videos. We can also add custom compression methods to `image_transport`.
    - `camera_calibration_parsers`: This package contains a routine to read/write camera calibration parameters from an XML file. This package is mainly used by camera drivers for accessing calibration parameters.
    - `camera_info_manager`: This package consists of a routine to save, restore, and load the calibration information. This is mainly used by camera drivers.
    - `polled_camera`: This package contains the interface for requesting images from a polling camera driver (for example, `prosilica_camera`).
    - `Image_pipeline`: This metapackage contains packages to process the raw image from the camera driver. The various processing done by this metapackage include calibration, distortion removal, stereo vision processing, depth-image processing, and so on. The following packages are present in this metapackage for this processing (`http://wiki.ros.org/image_pipeline`).
- `camera_calibration`: One of the important tools for relating the 3D world to the 2D camera image is calibration. This package provides tools for doing monocular and stereo image calibration in ROS.
- `image_proc`: The nodes in this package act between the camera driver and the vision processing nodes. It can handle the calibration parameters, correct image distortion from the raw image, and convert to color images.
- `depth_image_proc`: This package contains nodes and `nodelets` for handling depth images from Kinect and 3D vision sensors. The depth image can be processed by these `nodelets` to produce point cloud data.

- `stereo_image_proc`: This package has nodes to perform distortion removal for a pair of cameras. It is the same as the `image_proc` package, except that it handles two cameras for stereo vision and for developing point cloud and disparity images.
- `image_rotate`: This package contains nodes to rotate the input image.
- `image_view`: This is a simple ROS tool for viewing ROS message topics. It can also view stereo and disparity images.
- `Image_transport_plugins`: These are the plugins of ROS image transport for publishing and subscribing the ROS images in different compression levels or different video codec to reduce the bandwidth and latency.
- `Laser_pipeline`: This is a set of packages that can process laser data, such as filtering and converting into 3D Cartesian points and assembling points to form a cloud. The `laser-pipeline` stack contains the following packages:
    - `laser_filters`: This package contains nodes to filter the noise in the raw laser data, remove the laser points inside the robot footprint, and remove spurious values inside the laser data.
    - `laser_geometry`: After filtering the laser data, we have to transform the laser ranges and angles into 3D Cartesian coordinates efficiently by taking into account the tilt and skew angle of laser scanner
    - `laser_assembler`: This package can assemble the laser scan into a 3D point cloud or 2.5D scan.
- `perception_pcl`: This is the stack of the PCL-ROS interface.
- `vision_opencv`: This is the stack of the OpenCV-ROS interface.

Interfacing USB webcams in ROS

We can start interfacing with an ordinary webcam or a laptop cam in ROS. Overall, there are no ROS-specific packages to install and use web cameras. If the camera is working in Ubuntu/Linux, it may be supported by the ROS driver too. After plugging in the camera, check whether a `/dev/videoX` device file has been created, or check with some application such as Cheese, VLC, or similar others. The guide to check whether the webcam is supported on Ubuntu is available at `https://help.ubuntu.com/community/Webcam`.

We can find the video devices present on the system using the following command:

```
$ ls /dev/ | grep video
```

If you get an output of `video`, you can confirm a USB cam is available for use.

After ensuring the webcam support in Ubuntu, we can install a ROS webcam driver called `usb_cam` using the following command:

```
$ sudo apt-get install ros-kinetic-usb-cam
```

We can install the latest package of `usb_cam` from the source code. The driver is available on GitHub, at `https://github.com/bosch-ros-pkg/usb_cam`.

The `usb_cam` package contains a node called `usb_cam_node`, which is the driver of USB cams. There are some parameters that need to be set before running this node. We can run the ROS node along with its parameters. The `usb_cam-test.launch` file can launch the USB cam driver with the necessary parameters:

```
<launch>
  <node name="usb_cam" pkg="usb_cam" type="usb_cam_node" output="screen" >
    <param name="video_device" value="/dev/video0" />
    <param name="image_width" value="640" />
    <param name="image_height" value="480" />
    <param name="pixel_format" value="yuyv" />
    <param name="camera_frame_id" value="usb_cam" />
    <param name="io_method" value="mmap"/>
  </node>
  <node name="image_view" pkg="image_view" type="image_view"
respawn="false" output="screen">
    <remap from="image" to="/usb_cam/image_raw"/>
    <param name="autosize" value="true" />
  </node>
</launch>
```

This launch file will start `usb_cam_node` with the video device `/dev/video0`, with a resolution of 640 x 480. The pixel format here is YUV (`https://en.wikipedia.org/wiki/YUV`). After initiating `usb_cam_node`, it will start an `image_view` node for displaying the raw image from the driver. We can launch the previous file by using the following command:

```
$ roslaunch usb_cam usb_cam-test.launch
```

We will get the following message with an image view, as shown next:

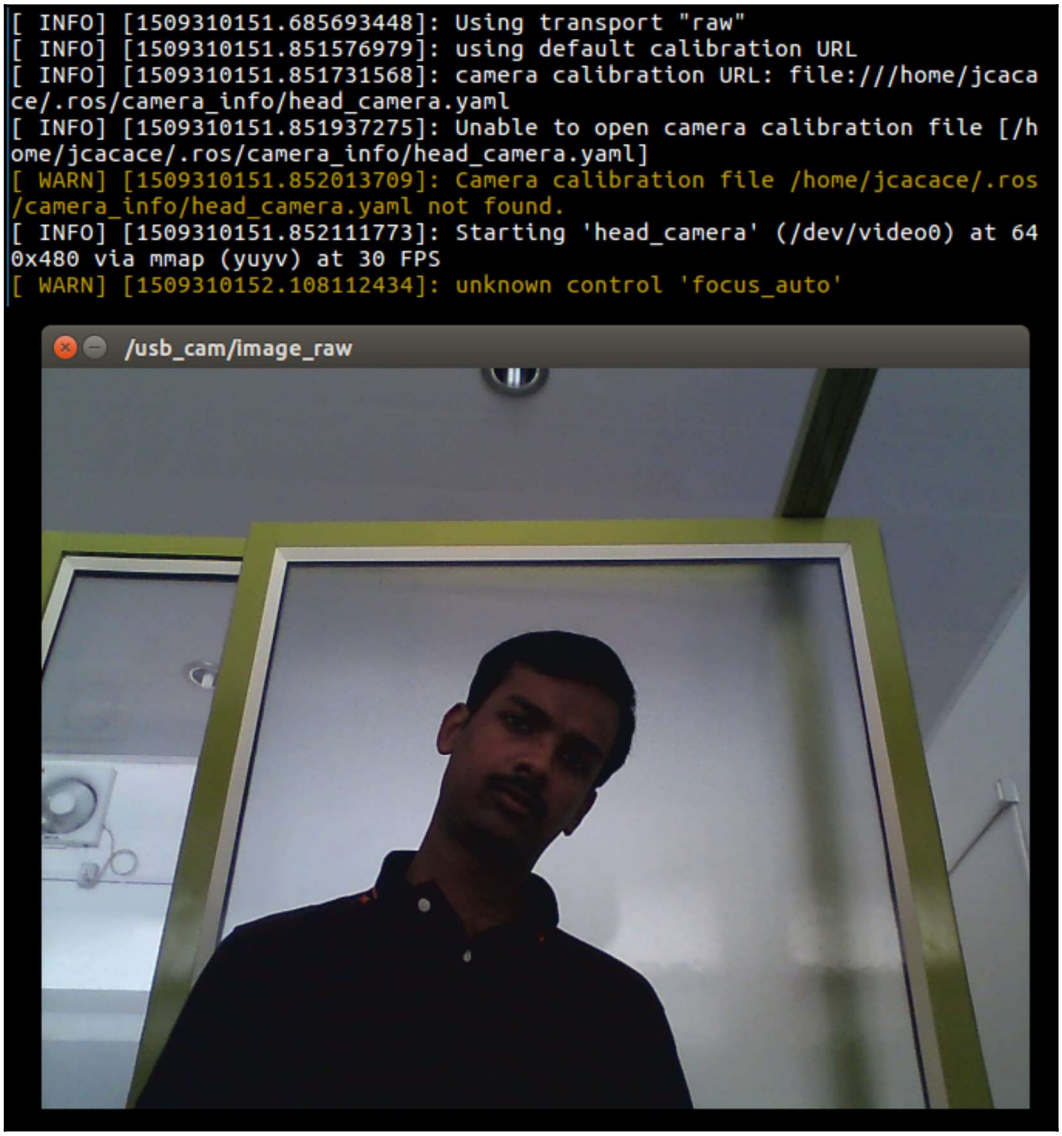

Figure 1: USB camera view using the image view tool

The topics generated by the driver are shown next. There are raw, compressed, and Theora codec topics generated by the driver:

```
/image_view/output
/image_view/parameter_descriptions
/image_view/parameter_updates
/rosout
/rosout_agg
/usb_cam/camera_info
/usb_cam/image_raw
/usb_cam/image_raw/compressed
/usb_cam/image_raw/compressed/parameter_descriptions
/usb_cam/image_raw/compressed/parameter_updates
/usb_cam/image_raw/compressedDepth
/usb_cam/image_raw/compressedDepth/parameter_descriptions
/usb_cam/image_raw/compressedDepth/parameter_updates
/usb_cam/image_raw/theora
/usb_cam/image_raw/theora/parameter_descriptions
/usb_cam/image_raw/theora/parameter_updates
```

Figure 2: List of topics generated by the USB camera driver

We can visualize the image in another window by using the following command:

```
$ rosrun image_view image_view image:=/usb_cam/image_raw
```

As you can see from the topic list, due to the installation of the `image_trasport` package, images are published in multiple ways, compressed and uncompressed. The latter format is useful to send images to other ROS nodes over the network or store video data of the topic into `bagfiles`, occupying little space on the hard disk. In order to use the compressed image from a `bagfiles` on a remote machine or from a `bagfile` , we need to republish it in an uncompressed format, using the republish node of the `image_transport` package:

```
$ rosrun image_transport republish [input format] in:=<in_topic_base>
[output format] out:=<out_topic>
```

For example:

```
$ rosrun image_transport republish compressed in:=/usb_cam/image_raw
[output format] out:=/usb_cam/image_raw/republished
```

Note that in the previous example, we have used the topic base name as input (`/usb_cam/img_raw`), and not its compressed version (`/usb_cam/image_raw/compressed`).

Now that we have learned how to acquire images from the camera, we have to work with the camera calibration.

Working with ROS camera calibration

Like all sensors, cameras also need calibration for correcting the distortions in the camera images due to the camera's internal parameters, and for finding the world coordinates from the camera coordinates.

The primary parameters that cause image distortions are radial distortions and tangential distortions. Using the camera calibration algorithm, we can model these parameters and also calculate the real-world coordinates from the camera coordinates by computing the camera calibration matrix, which contains the focal distance and the principle points.

Camera calibration can be done using a classic black-white chessboard, symmetrical circle pattern, or an asymmetrical circle pattern. According to each different pattern, we use different equations to get the calibration parameters. Using the calibration tools, we detect the patterns, and each detected pattern is taken as a new equation. When the calibration tool gets enough detected patterns it can compute the final parameters for the camera.

ROS provides a package named `camera_calibration` (`http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration`) to do camera calibration, which is a part of the image pipeline stack. We can calibrate monocular, stereo, and even 3D sensors, such as Kinect and Asus Xtion pro.

The first thing we have to do before calibration is to download the chessboard pattern mentioned in the ROS Wiki page, and print it and paste it onto a card board. This is the pattern we are going to use for calibration. This check board has 8x6 with 108 mm squares.

Run the `usb_cam` launch file to start the camera driver. We are going to run the camera calibration node of ROS using the raw image from the `/usb_cam/image_raw` topic. The following command will run the calibration node with the necessary parameters:

```
$ rosrun camera_calibration cameracalibrator.py --size 8x6 --square 0.108
image:=/usb_cam/image_raw camera:=/usb_cam
```

A calibration window will pop up, and when we show the calibration pattern to the camera, and the detection is made, we will see the following screenshot:

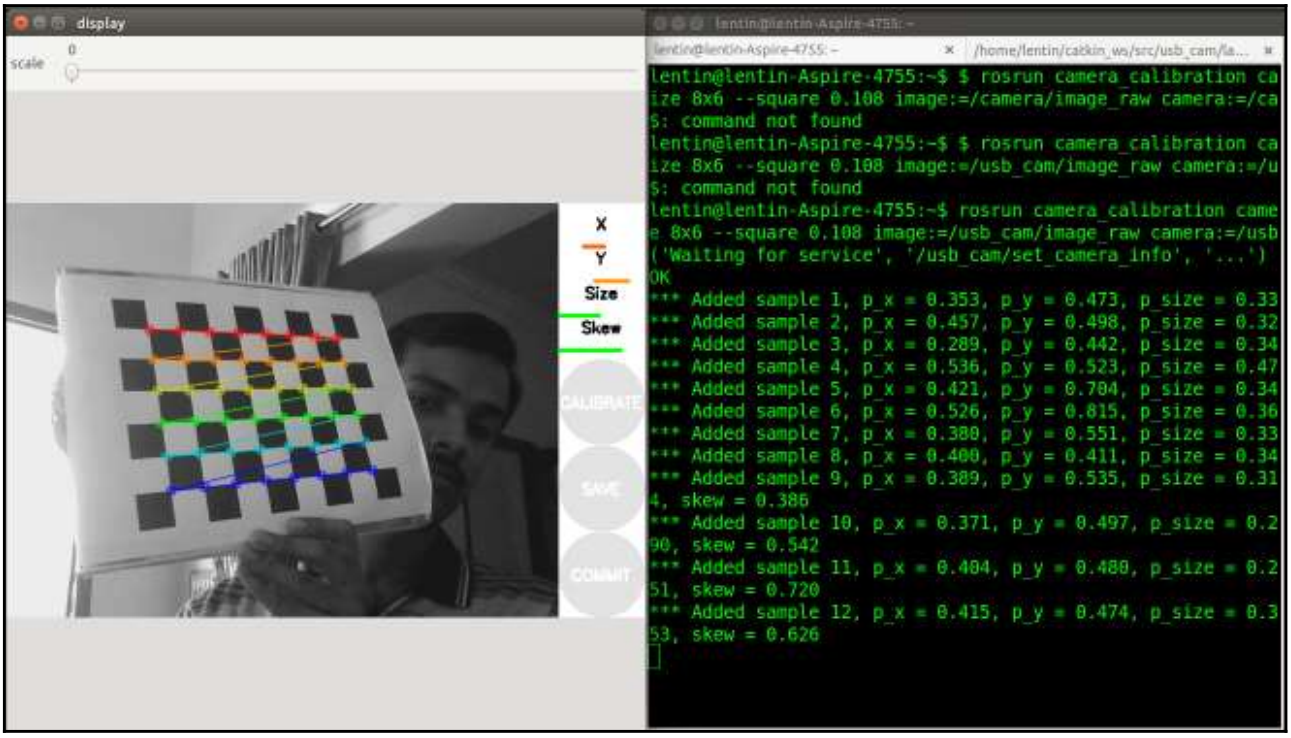

Figure 3: ROS camera calibration

Move the calibration pattern in the *X* direction and *Y* direction. If the calibrator node gets a sufficient amount of samples, a **CALIBRATE** button will become active on the window. When we press the **CALIBRATE** button, it will compute the camera parameters using these samples. It will take some time for the calculation. After computation, two buttons, **SAVE** and **COMMIT**, will become active inside the window, as shown in the following image. If we press the **SAVE** button, it will save the calibration parameters to a file in the `/tmp` folder. If we press the **COMMIT** button, it will save them to `./ros/camera_info/head_camera.yaml`:

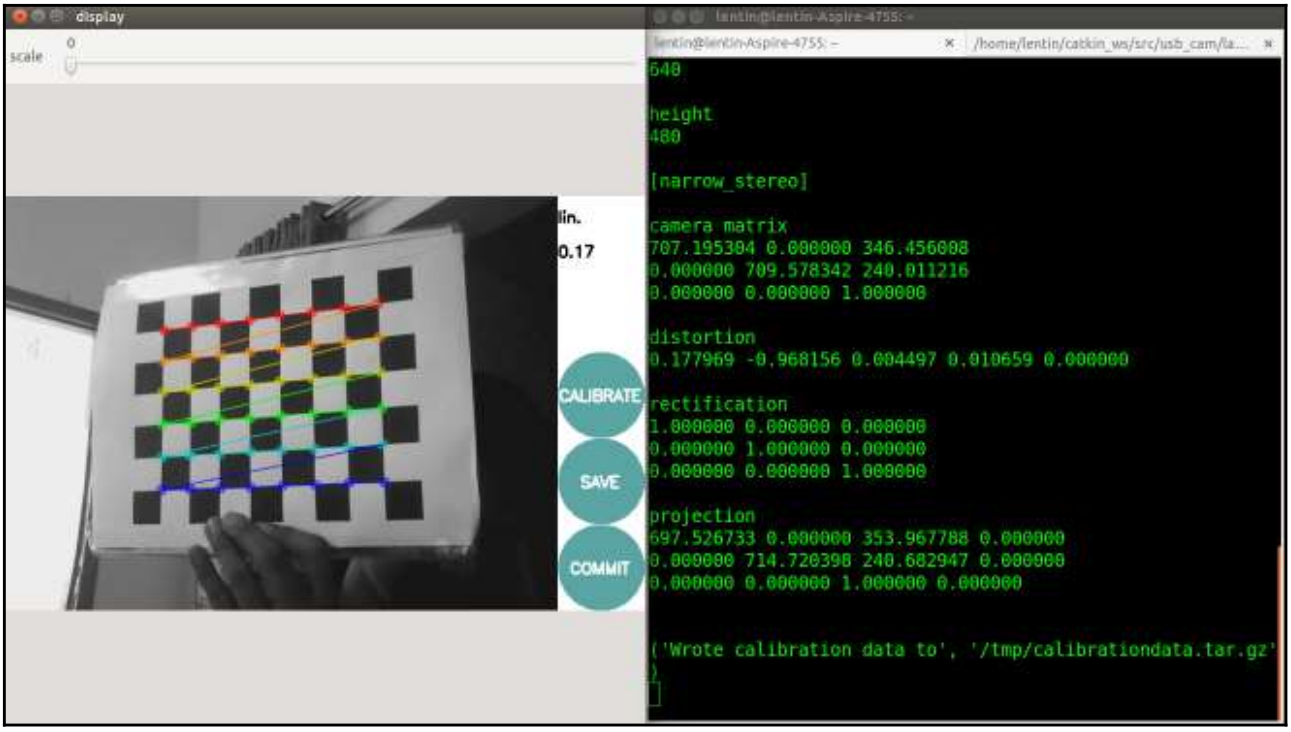

Figure 4: Generating the camera calibration file

Restart the camera driver, and we will see the YAML calibration file loaded along with the driver. The calibration file that we generated will look as follows:

```
image_width: 640
image_height: 480
camera_name: head_camera
camera_matrix:
rows: 3
cols: 3
data: [707.1953043273086, 0, 346.4560078627374, 0, 709.5783421541863,
240.0112155124814, 0, 0, 1]
distortion_model: plumb_bob
distortion_coefficients:
rows: 1
cols: 5
data: [0.1779688561999974, -0.9681558538432319, 0.004497434720139909,
0.0106588921249554, 0]
rectification_matrix:
rows: 3
cols: 3
data: [1, 0, 0, 0, 1, 0, 0, 0, 1]
projection_matrix:
```

```
  rows: 3
  cols: 4
  data: [697.5267333984375, 0, 353.9677879190494, 0, 0, 714.7203979492188,
240.6829465337159, 0, 0, 0, 1, 0]
```

Converting images between ROS and OpenCV using cv_bridge

In this section, we will see how to convert between the ROS image message (`sensor_msgs/Image`) and the OpenCV image data type (`cv::Mat`). The main ROS package used for this conversion is `cv_bridge`, which is part of the `vision_opencv` stack. The ROS library inside `cv_bridge`, called `CvBridge`, helps to perform this conversion. We can use the `CvBridge` library inside our code and perform the conversion. The following figure shows how the conversion is performed between ROS and OpenCV:

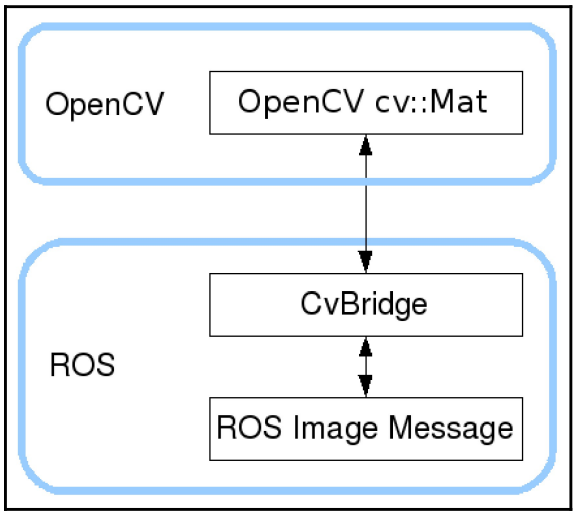

Figure 5: Converting images using CvBridge

Here, the `CvBridge` library acts as a bridge for converting the ROS messages to OpenCV images and vice versa. We will see how the conversion between ROS and OpenCV is performed in the following example.

Image processing using ROS and OpenCV

In this section, we will see an example of using `cv_bridge` for acquiring images from a camera driver, and converting and processing the images using OpenCV APIs. The following is how the example works:

1. Subscribe the images from the camera driver from the topic `/usb_cam/image_raw` (`sensor_msgs/Image`)
2. Convert the ROS images to the OpenCV image type using `CvBridge`
3. Process the OpenCV image using its APIs and find the edges on the image
4. Convert the OpenCV image type of the edge detection to the ROS image messages and publish into the topic `/edge_detector/processed_image`

The step-by-step procedure to build this example follows:

Step 1 – Creating a ROS package for the experiment

You can get the existing package `cv_bridge_tutorial_pkg` provided with this book, or you can create a new package, using the following command:

```
$ catkin_create_pkg cv_bridge_tutorial_pkg cv_bridge image_transport roscpp
sensor_msgs std_msgs
```

This package is mainly dependent on `cv_bridge`, `image_transport`, and `sensor_msgs`.

Step 2 – Creating source files

You can get the source code of the example `sample_cv_bridge_node.cpp` from the `cv_bridge_tutorial_pkg/src` folder.

Step 3 – Explanation of the code

The following is the explanation of the complete code:

```
#include <image_transport/image_transport.h>
```

We are using the `image_transport` package in this code for publishing and subscribing to an image in ROS:

```
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
```

This header includes the `CvBridge` class and image encoding related functions in the code:

```
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
```

These are main OpenCV image processing modules and GUI modules which provide image processing and GUI APIs in our code:

```
  image_transport::ImageTransport it_;
public:
  Edge_Detector()
    : it_(nh_)
  {
    // Subscribe to input video feed and publish output video feed
    image_sub_ = it_.subscribe("/usb_cam/image_raw", 1,
      &ImageConverter::imageCb, this);

    image_pub_ = it_.advertise("/edge_detector/processed_image", 1);
```

We will look in more detail at the line `image_transport::ImageTransport it_`. This line creates an instance of `ImageTransport`, which is used to publish and subscribe the ROS image messages. More information about the `ImageTransport` API is given next.

Publishing and subscribing images using image_transport

ROS image transport is very similar to ROS publishers and subscribers, and it is used to publish or subscribe the images along with the camera information. We can publish the image data using `ros::Publisher`, but image transport is a more efficient way of sending the image data.

The image transport APIs are provided by the `image_transport` package. Using these APIs, we can transport an image in different compression formats; for example, we can transport it as an uncompressed image, JPEG/PNG compression, or Theora compression in separate topics. We can also add different transport formats by adding plugins. By default, we can see the compressed and Theora transports:

```
image_transport::ImageTransport it_;
```

In the following line, we are creating an instance of the `ImageTransport` class:

```
  image_transport::Subscriber image_sub_;
  image_transport::Publisher image_pub_;
```

After that, we declare the subscriber and publisher objects for subscribing and publishing the images, using the `image_transport` object:

```
image_sub_ = it_.subscribe("/usb_cam/image_raw", 1,
      &ImageConverter::imageCb, this);
image_pub_ = it_.advertise("/edge_detector/processed_image", 1);
```

The following is how we subscribe and publish an image:

```
     cv::namedWindow(OPENCV_WINDOW);
  }
  ~Edge_Detector()
  {
    cv::destroyWindow(OPENCV_WINDOW);
  }
```

This is how we subscribe and publish an `image.cv::namedWindow()`, which is an OpenCV function to create a GUI for displaying an image. The argument inside this function is the window name. Inside the class destructor, we are destroying the named window.

Converting OpenCV to ROS images using cv_bridge

This is an image callback function, and it basically converts the ROS image messages into the OpenCV `cv::Mat` type using the `CvBridge` APIs. The following is how we can convert ROS to OpenCV, and vice versa:

```
  void imageCb(const sensor_msgs::ImageConstPtr& msg)
  {

    cv_bridge::CvImagePtr cv_ptr;
    namespace enc = sensor_msgs::image_encodings;

    try
    {
      cv_ptr = cv_bridge::toCvCopy(msg,
sensor_msgs::image_encodings::BGR8);
    }
    catch (cv_bridge::Exception& e)
    {
      ROS_ERROR("cv_bridge exception: %s", e.what());
```

```
        return;
      }
```

To start with `CvBridge`, we should start with creating an instance of a `CvImage`. Given next is the creation of the `CvImage` pointer:

```
cv_bridge::CvImagePtr cv_ptr;
```

The `CvImage` type is a class provided by `cv_bridge`, which consists of information such as an OpenCV image, its encoding, ROS header, and so on. Using this type, we can easily convert an ROS image to OpenCV, and vice versa:

```
cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
```

We can handle the ROS image message in two ways: either we can make a copy of the image or we can share the image data. When we copy the image, we can process the image, but if we use a shared pointer, we can't modify the data. We use `toCvCopy()` for creating a copy of the ROS image, and the `toCvShare()` function is used to get the pointer of the image. Inside these functions, we should mention the ROS message and the type of encoding:

```
if (cv_ptr->image.rows > 400 && cv_ptr->image.cols > 600){
detect_edges(cv_ptr->image);
    image_pub_.publish(cv_ptr->toImageMsg());
}
```

In this section, we are extracting the image and its properties from the `CvImage` instance, and accessing the `cv::Mat` object from this instance. This code simply checks whether the rows and columns of the image are in a particular range, and, if it is true, it will call another method called `detect_edges(cv::Mat)`, which will process the image given as an argument and display the edge-detected-image:

```
image_pub_.publish(cv_ptr->toImageMsg());
```

The preceding line will publish the edge-detected-image after converting to the ROS image message. Here we are using the `toImageMsg()` function for converting the `CvImage` instance to a ROS image message.

Finding edges on the image

After converting the ROS images to OpenCV type, the `detect_edges(cv::Mat)` function will be called for finding the edges on the image, using the following inbuilt OpenCV functions:

```
cv::cvtColor( img, src_gray, CV_BGR2GRAY );
cv::blur( src_gray, detected_edges, cv::Size(3,3) );
cv::Canny( detected_edges, detected_edges, lowThreshold,
lowThreshold*ratio, kernel_size );
```

Here, the `cvtColor()` function will convert an RGB image to a gray color space and `cv::blur()` will add blurring to the image. After that, using the Canny edge detector, we extract the edges of the image.

Visualizing raw and edge-detected-images

Here we are displaying the image data using the OpenCV function called `imshow()`, which consists of the window name and the image name:

```
cv::imshow(OPENCV_WINDOW, img);
cv::imshow(OPENCV_WINDOW_1, dst);
cv::waitKey(3);
```

Step 4 – Editing the CMakeLists.txt file

The definition of the `CMakeLists.txt` file is given next. In this example, we need OpenCV support, so we should include the OpenCV header path and also link the source code against the OpenCV library path:

```
include_directories(
  ${catkin_INCLUDE_DIRS}
  ${OpenCV_INCLUDE_DIRS}
)

add_executable(sample_cv_bridge_node src/sample_cv_bridge_node.cpp)

## Specify libraries to link a library or executable target against
 target_link_libraries(sample_cv_bridge_node
   ${catkin_LIBRARIES}
   ${OpenCV_LIBRARIES}
 )
```

Step 5 – Building and running an example

After building the package using `catkin_make`, we can run the node using the following command:

1. Launch the webcam driver:

```
$ roslaunch usb_cam usb_cam-test.launch
```

2. Run the `cv_bridge` sample node:

```
$ rosrun cv_bridge_tutorial_pkg sample_cv_bridge_node
```

3. If everything works fine, we will get two windows, as shown in the following image. The first window shows the raw image and the second is the processed edge-detected-image:

Figure 6: Raw image and edge-detected-image

Interfacing Kinect and Asus Xtion Pro in ROS

The webcams that we have worked with until now can only provide 2D visual information of the surroundings. For getting 3D information about the surroundings, we have to use 3D vision sensors or range finders, such as laser finders. Some of the 3D vision sensors that we are discussing in this chapter are Kinect, Asus Xtion Pro, Intel Real sense, and Hokuyo laser scanner:

Figure 7: Top: Kinect; bottom: Asus Xtion Pro.

The first two sensors we are going to discuss are Kinect and Asus Xtion Pro. Both of these devices need the **OpenNI** (**Open source Natural Interaction**) driver library for operating in the Linux system. OpenNI acts as a middleware between the 3D vision devices and the application software. The OpenNI driver is integrated to ROS and we can install these drivers by using the following commands. These packages help to interface the OpenNI supported devices, such as Kinect and Asus Xtion Pro:

```
$ sudo apt-get install ros-kinetic-openni-launch ros-kinetic-openni2-launch
```

The preceding command will install OpenNI drivers and launch files for starting the RGB/depth streams. After successful installation of these packages, we can launch the driver by using the following command:

```
$ roslaunch openni2_launch openni2.launch
```

This launch file will convert the raw data from the devices into useful data, such as 3D point clouds, disparity images, and depth, and the RGB images using ROS nodelets.

Other than the OpenNI drivers, there is another driver available called `lib-freenect`. The common launch files of the drivers are organized into a package called `rgbd_launch`. This package consists of common launch files that are used for the freenect and openni drivers.

We can visualize the point cloud generated by the OpenNI ROS driver by using RViz.

Run RViz, using the following command:

```
$ rosrun rviz rviz
```

Set the fixed frame to `/camera_depth_optical_frame`, add a **PointCloud2** display, and set the topic as `/camera/depth/points`. This is the unregistered point cloud from the IR camera, that is, it may have a complete match with the RGB camera and it only uses the depth camera for generating the point cloud:

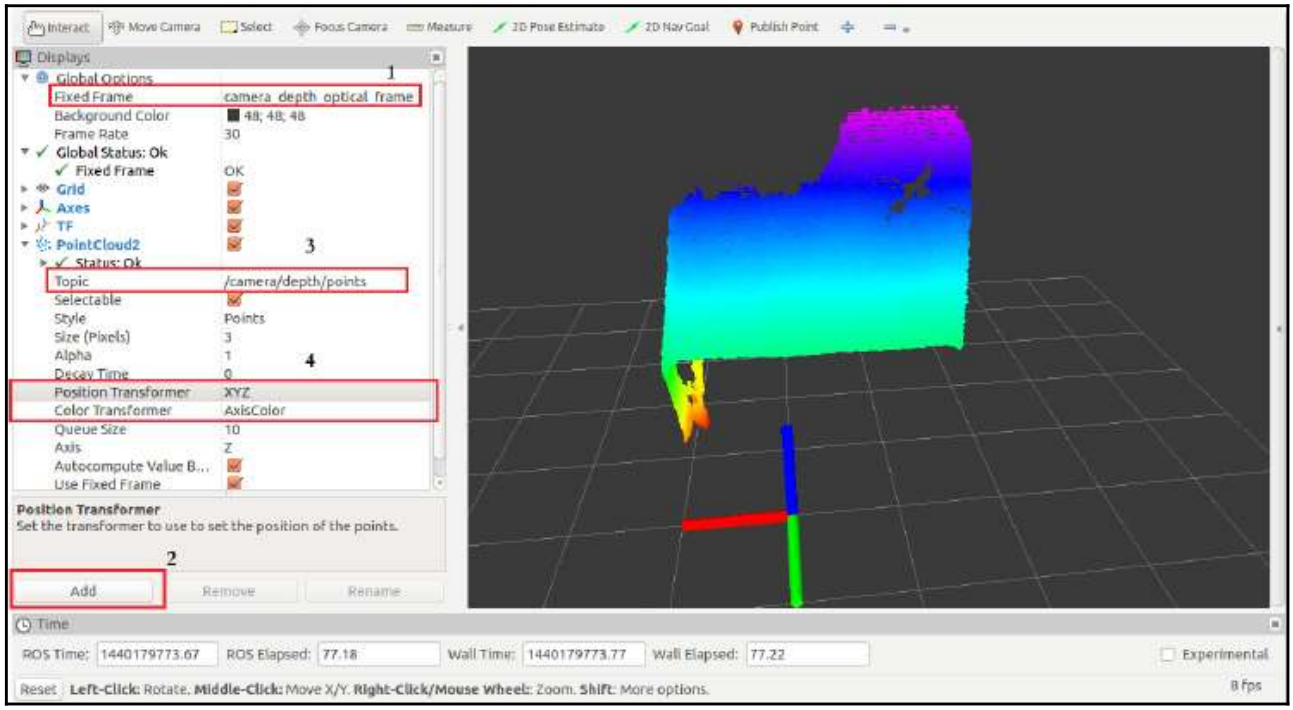

Figure 8: Unregistered point cloud view in RViz

We can enable the registered point cloud by using the **Dynamic Reconfigure** GUI, by using the following command:

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

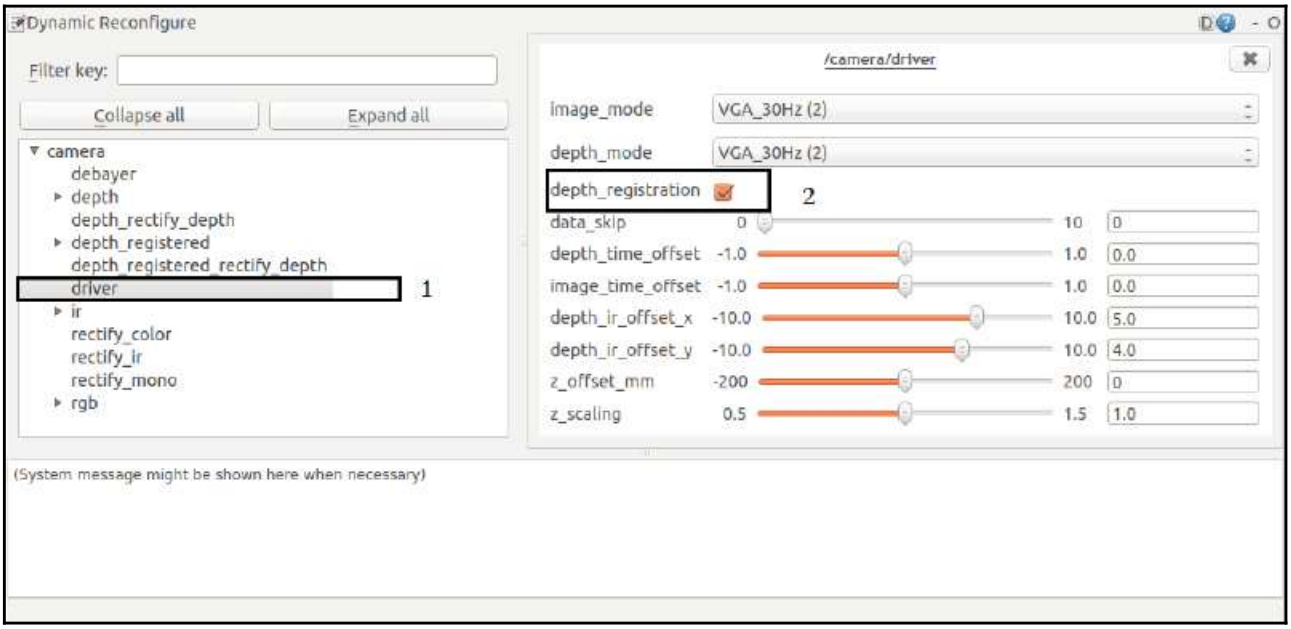

Figure 9: Dynamic Reconfigure GUI

Click on **camera** | **driver** and tick `depth_registration`. Change the point cloud to `/camera/depth_registered/points` and **Color Transformer** to `RGB8` in RViz. We will see the registered point cloud in RViz as it appears in the following image. The registered point cloud takes information from the depth and the RGB camera to generate the point cloud:

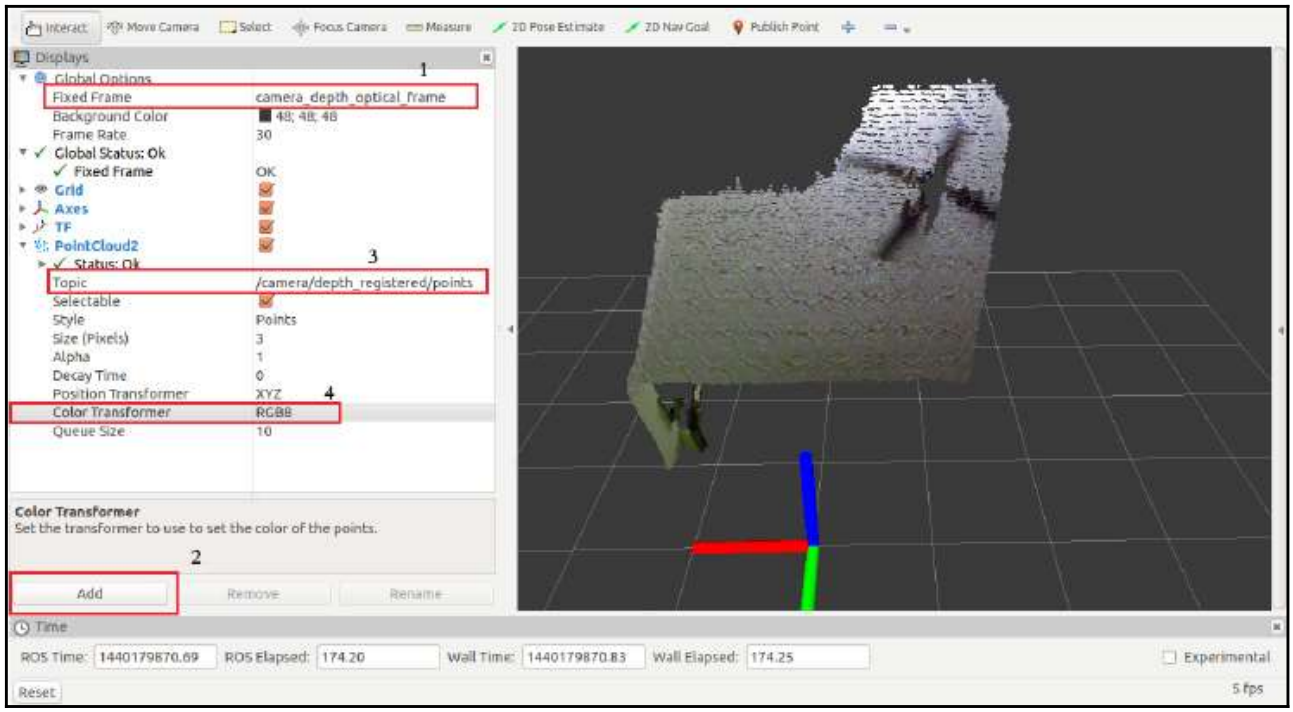

Figure 10: The registered point cloud

Interfacing Intel Real Sense camera with ROS

One of the new 3D depth sensors from Intel is Real Sense. Until now, different versions of this sensor have been released (F200, R200, SR30, ...). To interface Real Sense sensors with ROS, we first have to install the `librealsense` library.

Download the Real Sense library from the following link: `https://github.com/IntelRealSense/librealsense`, using the following code:

```
$ git clone https://github.com/IntelRealSense/librealsense.git
```

Then follow these steps:

1. Install `libudev-dev`, `pkg-config`, and `libgtk-3`:

```
$ sudo apt-get install libudev-dev pkg-config libgtk-3-dev
```

2. Install `glfw3`:

```
$ sudo apt-get install libglfw3-dev
```

3. Navigate to the `librealsense` root directory and run:

```
$ mkdir build && cd build
$ cmake ..
$ make
$ sudo make install
```

After installing the Real Sense library, we can install the ROS package to start sensor data streaming. We can install from the Ubuntu/Debian package manager by using the following command:

```
$ sudo make install ros-kinetic-realsense
```

Or we can directly clone the package from the Git repository and compile the workspace:

```
$ git clone https://github.com/intel-ros/realsense.git
```

Now we can start the sensor using the example launch file and open Rviz to visualize the color and depth data streamed by `realsense`:

```
roslaunch realsense_camera sr300_nodelet_rgbd.launch
```

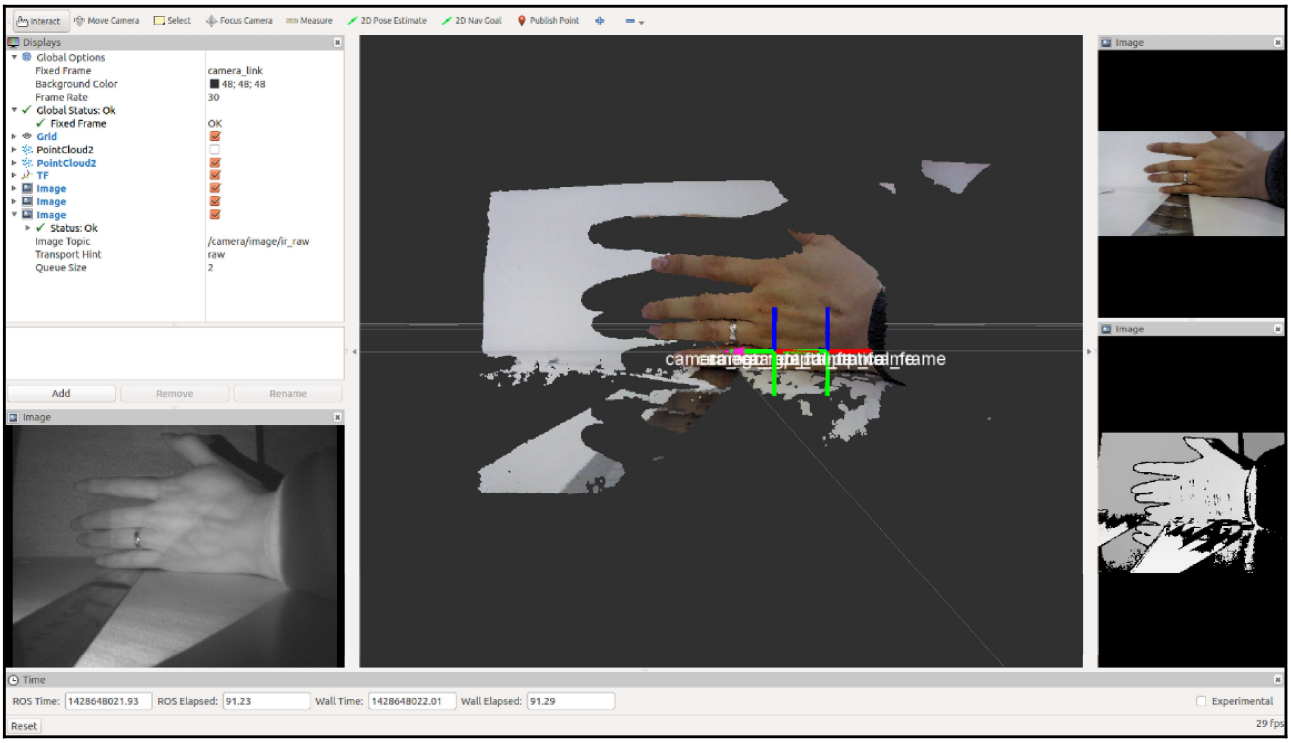

Figure 12: Intel Real Sense view in RViz

The following are the topics generated by the Real Sense driver:

```
sensor_msgs::PointCloud2
/camera/depth/points              point cloud without RGB
/camera/depth_registered/points   point cloud with RGB

sensor_msgs::Image
/camera/image/rgb_raw                   raw image for RGB sensor
/camera/image/depth_raw           raw image for depth sensor
/camera/image/ir_raw              raw image for infrared sensor
```

Working with a point cloud to a laser scan package

One of the important applications of 3D vision sensors is mimicking the functionalities of a laser scanner. We need the laser scanner data for working with autonomous navigation algorithms, such as SLAM. We can make a fake laser scanner using a 3D vision sensor. We can take a slice of the point cloud data/depth image and convert it to laser range data. In ROS, we have a set of packages to convert the point cloud to laser scans:

- `depthimage_to_laserscan`: This package contains nodes that take the depth image from the vision sensor and generate a 2D laser scan based on the provided parameters. The inputs of the node are depth image and camera info parameters, which include calibration parameters. After conversion to the laser scan data, it will publish laser scanner data in the `/scan` topic. The node parameters are `scan_height, scan_time, range_min, range_max`, and the output frame ID. The official ROS Wiki page of this package can be found at `http://wiki.ros.org/depthimage_to_laserscan`.
- `pointcloud_to_laserscan`: This package converts the real point cloud data into 2D laser scan, instead of taking a depth image as in the previous package. The official Wiki page of this package can be found at `http://wiki.ros.org/pointcloud_to_laserscan`.

The first package is suitable for normal applications; however, if the sensor is placed on an angle, it is better to use the second package. Also, the first package takes less processing than the second one. Here we are using the `depthimage_to_laserscan` package to convert a laser scan. We can install `depthimage_to_laserscan` and `pointcloud_to_laserscan` by using the following command:

```
$ sudo apt-get install ros-kinetic-depthimage-to-laserscan ros- kinetic-
pointcloud-to-laserscan
```

We can start converting from the depth image of the OpenNI device to the 2D laser scanner by using the following package.

Create a package for performing the conversion:

```
$ catkin_create_pkg fake_laser_pkg depthimage_to_laserscan nodelet roscpp
```

Create a folder called `launch`, and inside this folder create the following launch file called `start_laser.launch`. You will get this package and file from the `fake_laser_pkg/launch` folder:

```
<launch>
  <!-- "camera" should uniquely identify the device. All topics   are
pushed down
       into the "camera" namespace, and it is prepended to tf        frame
ids. -->
  <arg name="camera"      default="camera"/>
  <arg name="publish_tf"  default="true"/>

. . .
. . .
  <group if="$(arg scan_processing)">
    <node pkg="nodelet" type="nodelet"     name="depthimage_to_laserscan"
args="load     depthimage_to_laserscan/DepthImageToLaserScanNodelet $(arg
camera)/$(arg camera)_nodelet_manager">
      <!-- Pixel rows to use to generate the laserscan. For each
column, the scan willreturn the minimum value for those        pixels
centered vertically in the image. -->
      <param name="scan_height" value="10"/>
      <param name="output_frame_id" value="/$(arg
camera)_depth_frame"/>
      <param name="range_min" value="0.45"/>
      <remap from="image" to="$(arg camera)/$(arg       depth)/image_raw"/>
      <remap from="scan" to="$(arg scan_topic)"/>

. . .
. . .
</launch>
```

The following code snippet will launch the nodelet for converting the depth image to laser scanner:

```
<node pkg="nodelet" type="nodelet" name="depthimage_to_laserscan"
args="load depthimage_to_laserscan/DepthImageToLaserScanNodelet $(arg
camera)/$(arg camera)_nodelet_manager">
```

Launch this file and we can view the laser scanner in RViz.

Launch this file using the following command:

```
$ roslaunch fake_laser_pkg start_laser.launch
```

We will see the data in RViz, as shown in the following image:

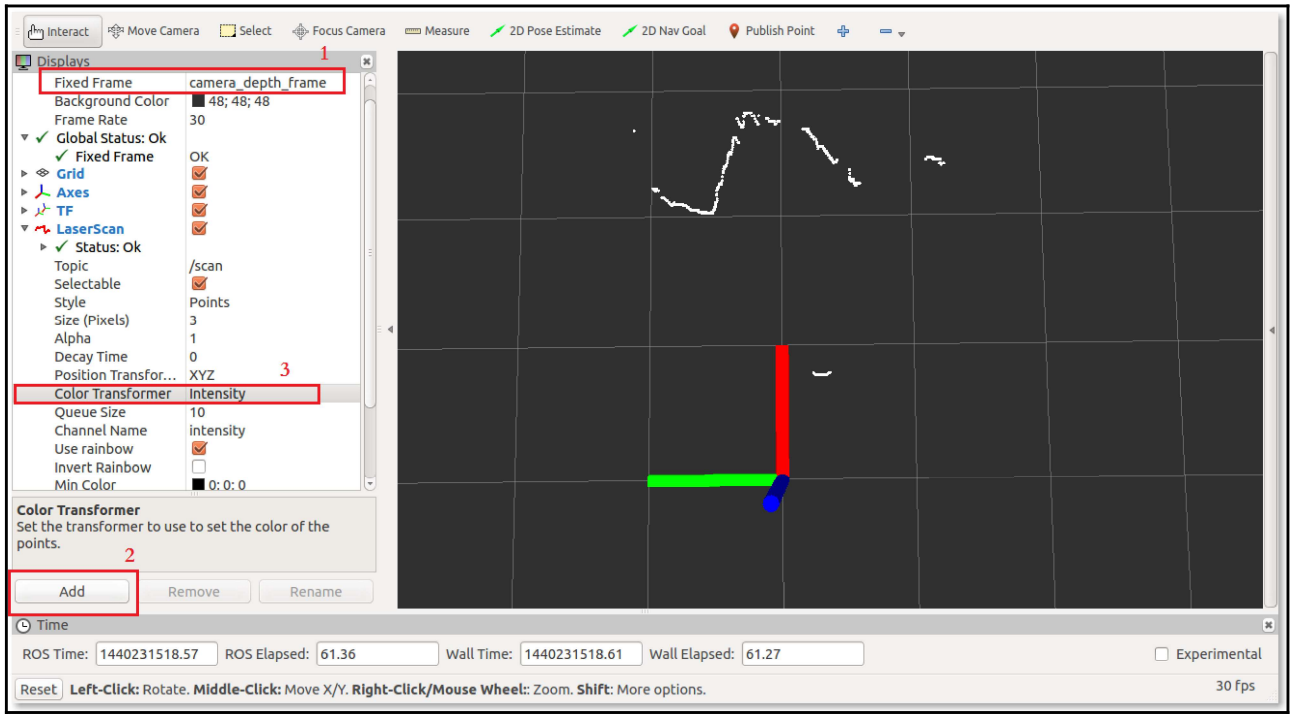

Figure 13: Laser scan in RViz

Set **Fixed Frame** as `camera_depth_frame` and **Add LaserScan** in the topic `/scan`. We can see the laser data in the view port.

Interfacing Hokuyo Laser in ROS

We can interface different ranges of laser scanners in ROS. One of the popular laser scanners available in the market is Hokuyo Laser scanner (`http://www.robotshop.com/en/hokuyo-utm-03lx-laser-scanning-rangefinder.html`):

Figure 14: Different series of Hokuyo laser scanners

One of the commonly used Hokuyo laser scanner models is `UTM-30LX`. This sensor is fast and accurate and is suitable for robotic applications. The device has a USB 2.0 interface for communication, and has up to a 30 meters range with a millimeter resolution. The arc range of the scan is about 270 degrees:

Figure 15: Hokuyo UTM-30LX

There is already a driver available in ROS for interfacing these scanners. One of the interfaces is called `urg_node` (`http://wiki.ros.org/urg_node`).

We can install this package by using the following command:

```
$ sudo apt-get install ros-kinetic-urg-node
```

When the device connects to the Ubuntu system, it will create a device called `ttyACMx`. Check the device name by entering the `dmesg` command in the Terminal. Change the USB device permission by using the following command:

```
$ sudo chmod a+rw /dev/ttyACMx
```

Start the laser scan device, using the following launch file called `hokuyo_start.launch`:

```
<launch>
 <node name="urg_node" pkg="urg_node" type="urg_node" output="screen">
         <param name="serial_port" value="/dev/ttyACM0"/>
    <param name="frame_id" value="laser"/>
    <param name="angle_min" value="-1.5707963"/>
         <param name="angle_max" value="1.5707963"/>
  </node>
  name="rviz" pkg="rviz" type="rviz" respawn="false" output="screen"
args="-d $(find hokuyo_node)/hokuyo_test.vcg"/>
</launch>
```

This launch file starts the node to get the laser data from the device `/dev/ttyACM0`. The laser data can be viewed inside the RViz window, as shown in the following image:

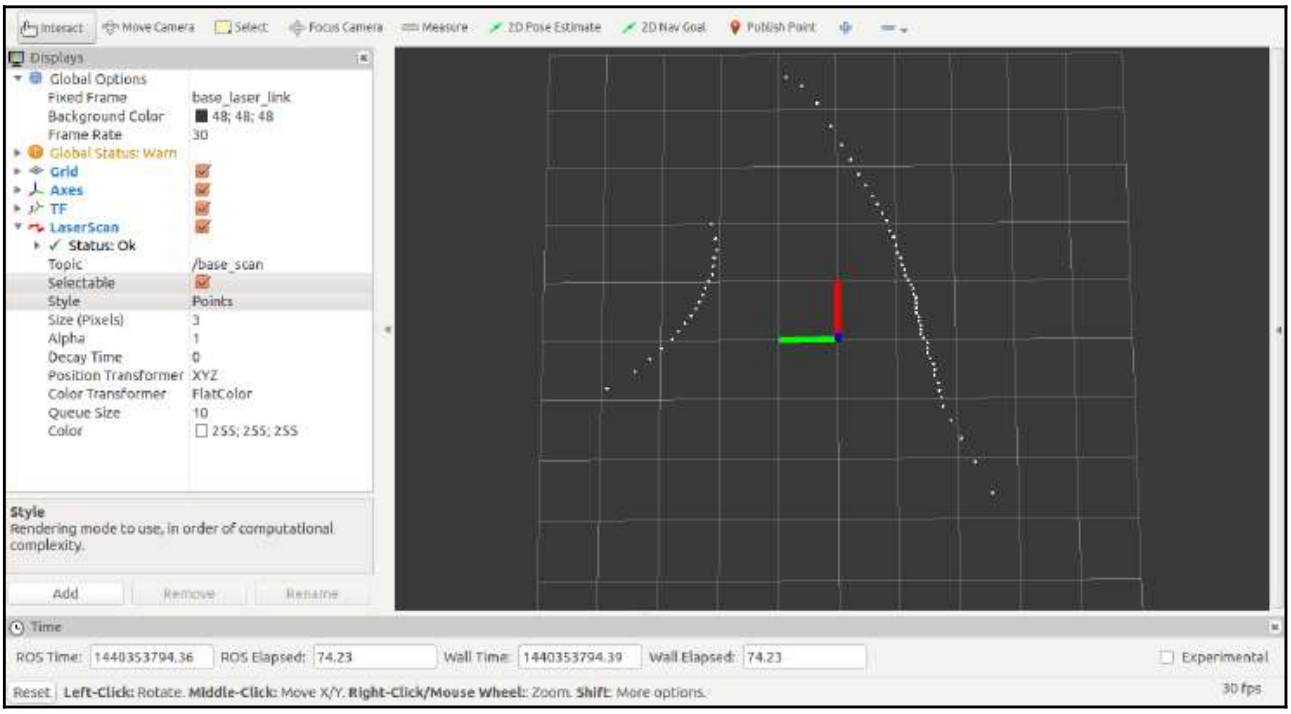

Figure 16: Hokuyo Laser scan data in RViz

Working with point cloud data

We can handle the point cloud data from Kinect or the other 3D sensors for performing a wide variety of tasks, such as 3D object detection and recognition, obstacle avoidance, 3D modeling, and so on. In this section, we will see some basic functionalities: using the PCL library and its ROS interface. We will discuss the following examples:

- How to publish a point cloud in ROS
- How to subscribe and process a point cloud
- How to write point cloud data to a PCD file
- How to read and publish a point cloud from a PCD file

How to publish a point cloud

In this example, we will see how to publish a point cloud data using the `sensor_msgs/PointCloud2` message. The code will use PCL APIs for handling and creating the point cloud, and converting the PCL cloud data to the PointCloud2 message type.

You will get the example code `pcl_publisher.cpp` from the `pcl_ros_tutorial/src` folder:

```
#include <ros/ros.h>

// point cloud headers
#include <pcl/point_cloud.h>
//Header which contain PCL to ROS and ROS to PCL conversion functions
#include <pcl_conversions/pcl_conversions.h>

//sensor_msgs header for point cloud2
#include <sensor_msgs/PointCloud2.h>

main (int argc, char **argv)
{
    ros::init (argc, argv, "pcl_create");


    ROS_INFO("Started PCL publishing node");

    ros::NodeHandle nh;

//Creating publisher object for point cloud

    ros::Publisher pcl_pub = nh.advertise<sensor_msgs::PointCloud2>
("pcl_output", 1);

//Creating a cloud object
    pcl::PointCloud<pcl::PointXYZ> cloud;

//Creating a sensor_msg of point cloud

    sensor_msgs::PointCloud2 output;

    //Insert cloud data
    cloud.width  = 50000;
    cloud.height = 2;
    cloud.points.resize(cloud.width * cloud.height);
```

```
//Insert random points on the clouds

    for (size_t i = 0; i < cloud.points.size (); ++i)
    {
        cloud.points[i].x = 512 * rand () / (RAND_MAX + 1.0f);
        cloud.points[i].y = 512 * rand () / (RAND_MAX + 1.0f);
        cloud.points[i].z = 512 * rand () / (RAND_MAX + 1.0f);
    }

    //Convert the cloud to ROS message
    pcl::toROSMsg(cloud, output);
    output.header.frame_id = "point_cloud";

    ros::Rate loop_rate(1);
    while (ros::ok())
    {
        //publishing point cloud data
      pcl_pub.publish(output);
        ros::spinOnce();
        loop_rate.sleep();
    }

    return 0;
}
```

The creation of the PCL cloud is done as follows:

```
//Creating a cloud object
pcl::PointCloud<pcl::PointXYZ> cloud;
```

After creating this cloud, we insert random points to the clouds. We convert the PCL cloud to a ROS message by using the following function:

```
//Convert the cloud to ROS message
pcl::toROSMsg(cloud, output);
```

After converting to ROS messages, we can simply publish the data on the topic `/pcl_output`.

How to subscribe and process the point cloud

In this example, we will see how to subscribe the generated point cloud on the topic `/pcl_output`. After subscribing the point cloud, we apply a filter called the `VoxelGrid` class in PCL to down-sample the input cloud by keeping the same centroid of the input cloud. You will get the example code `pcl_filter.cpp` from the `src` folder of the package:

```
#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud2.h>
//Vortex filter header
#include <pcl/filters/voxel_grid.h>

//Creating a class for handling cloud data
class cloudHandler
{
public:
    cloudHandler()
    {
//Subscribing pcl_output topics from the publisher
//This topic can change according to the source of point cloud

    pcl_sub = nh.subscribe("pcl_output", 10, &cloudHandler::cloudCB, this);
//Creating publisher for filtered cloud data
        pcl_pub = nh.advertise<sensor_msgs::PointCloud2>("pcl_filtered",
1);
    }
//Creating cloud callback
    void cloudCB(const sensor_msgs::PointCloud2& input)
    {
        pcl::PointCloud<pcl::PointXYZ> cloud;
        pcl::PointCloud<pcl::PointXYZ> cloud_filtered;

       sensor_msgs::PointCloud2 output;
       pcl::fromROSMsg(input, cloud);

     //Creating VoxelGrid object
      pcl::VoxelGrid<pcl::PointXYZ> vox_obj;
     //Set input to voxel object
     vox_obj.setInputCloud (cloud.makeShared());
     //Setting parameters of filter such as leaf size
    vox_obj.setLeafSize (0.1f, 0.1f, 0.1f);
    //Performing filtering and copy to cloud_filtered variable
    vox_obj.filter(cloud_filtered);
      pcl::toROSMsg(cloud_filtered, output);
      output.header.frame_id = "point_cloud";
```

```
        pcl_pub.publish(output);
    }

protected:
    ros::NodeHandle nh;
    ros::Subscriber pcl_sub;
    ros::Publisher pcl_pub;
};
main(int argc, char** argv)
{
    ros::init(argc, argv, "pcl_filter");
    ROS_INFO("Started Filter Node");
    cloudHandler handler;
    ros::spin();
    return 0;
}
```

This code subscribes the point cloud topic called `/pcl_output`, filters, using `VoxelGrid`, and publishes the filtered cloud through the `/cloud_filtered` topic.

Writing data to a Point Cloud Data (PCD) file

We can save the point cloud to a PCD file by using the following code. The filename is `pcl_write.cpp` inside the `src` folder:

```
#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud2.h>
//Header file for writing PCD file
#include <pcl/io/pcd_io.h>

void cloudCB(const sensor_msgs::PointCloud2 &input)
{
    pcl::PointCloud<pcl::PointXYZ> cloud;
    pcl::fromROSMsg(input, cloud);

//Save data as test.pcd file
    pcl::io::savePCDFileASCII ("test.pcd", cloud);
}

main (int argc, char **argv)
{
    ros::init (argc, argv, "pcl_write");
```

```
    ROS_INFO("Started PCL write node");

    ros::NodeHandle nh;
    ros::Subscriber bat_sub = nh.subscribe("pcl_output", 10, cloudCB);

    ros::spin();

    return 0;
}
```

Reading and publishing a point cloud from a PCD file

This code can read a PCD file and publish the point cloud in the `/pcl_output` topic. The code `pcl_read.cpp` is available in the `src` folder:

```
#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud2.h>
#include <pcl/io/pcd_io.h>

main(int argc, char **argv)
{
    ros::init (argc, argv, "pcl_read");

    ROS_INFO("Started PCL read node");

    ros::NodeHandle nh;
    ros::Publisher pcl_pub = nh.advertise<sensor_msgs::PointCloud2>
("pcl_output", 1);

    sensor_msgs::PointCloud2 output;
    pcl::PointCloud<pcl::PointXYZ> cloud;

//Load test.pcd file
    pcl::io::loadPCDFile ("test.pcd", cloud);

    pcl::toROSMsg(cloud, output);
    output.header.frame_id = "point_cloud";

    ros::Rate loop_rate(1);
    while (ros::ok())
    {
//Publishing the cloud inside pcd file
```

```
            pcl_pub.publish(output);
            ros::spinOnce();
            loop_rate.sleep();
        }

        return 0;
    }
```

We can create a ROS package called `pcl_ros_tutorial` for compiling these examples:

```
$ catkin_create_pkg pcl_ros_tutorial pcl pcl_ros roscpp sensor_msgs
```

Otherwise, we can use the existing package.

Create the preceding examples inside `src` as `pcl_publisher.cpp`, `pcl_filter.cpp`, `pcl_write.cpp`, and `pcl_read.cpp`.

Create `CMakeLists.txt` for compiling all the sources:

```
## Declare a cpp executable
add_executable(pcl_publisher_node src/pcl_publisher.cpp)
add_executable(pcl_filter src/pcl_filter.cpp)
add_executable(pcl_write src/pcl_write.cpp)
add_executable(pcl_read src/pcl_read.cpp)

target_link_libraries(pcl_publisher_node
   ${catkin_LIBRARIES}
 )
target_link_libraries(pcl_filter
   ${catkin_LIBRARIES}
 )
target_link_libraries(pcl_write
   ${catkin_LIBRARIES}
 )
target_link_libraries(pcl_read
   ${catkin_LIBRARIES}
 )
```

Build this package using `catkin_make`, and we can run `pcl_publisher_node` and the view point cloud inside RViz by using the following command:

```
$ rosrun rviz rviz -f point_cloud
```

A screenshot of the point cloud from `pcl_output` is shown in the following image:

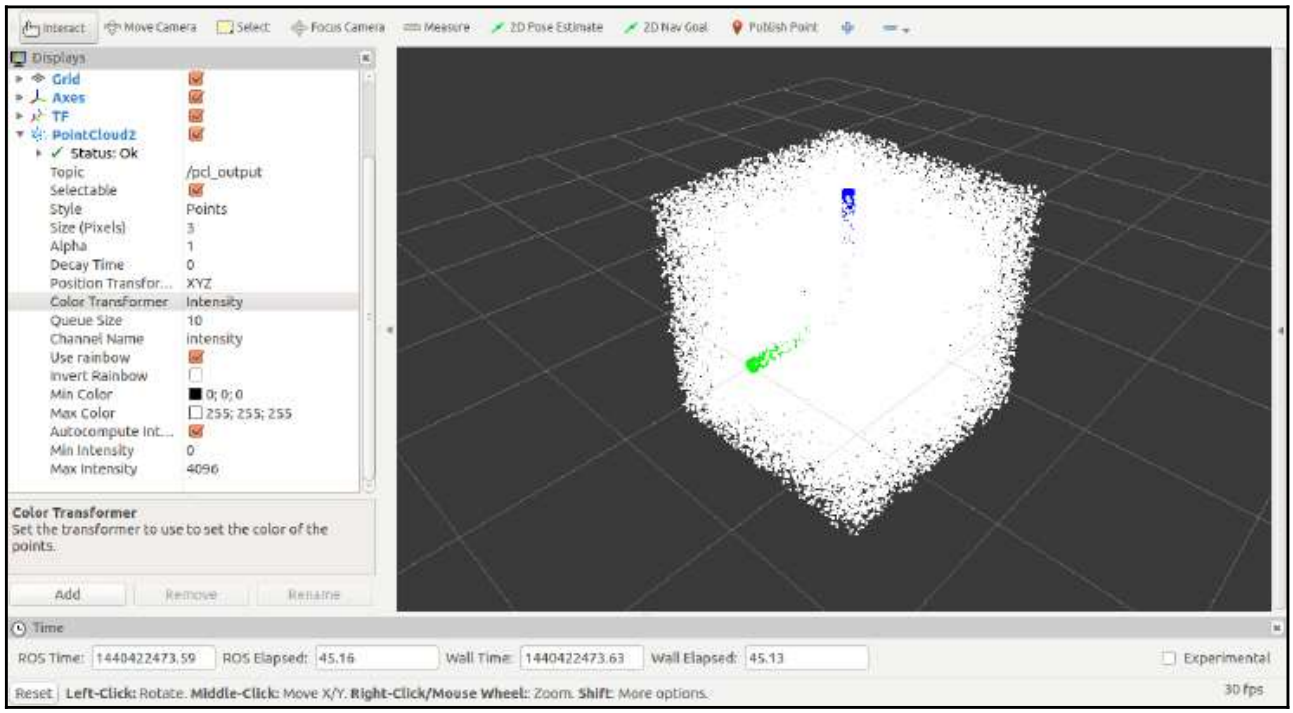

Figure 17: PCL cloud in RViz

We can run the `pcl_filter` node to subscribe this same cloud and do voxel grid filtering. The following screenshot shows the output from the `/pcl_filtered` topic, which is the resultant down-sampled cloud:

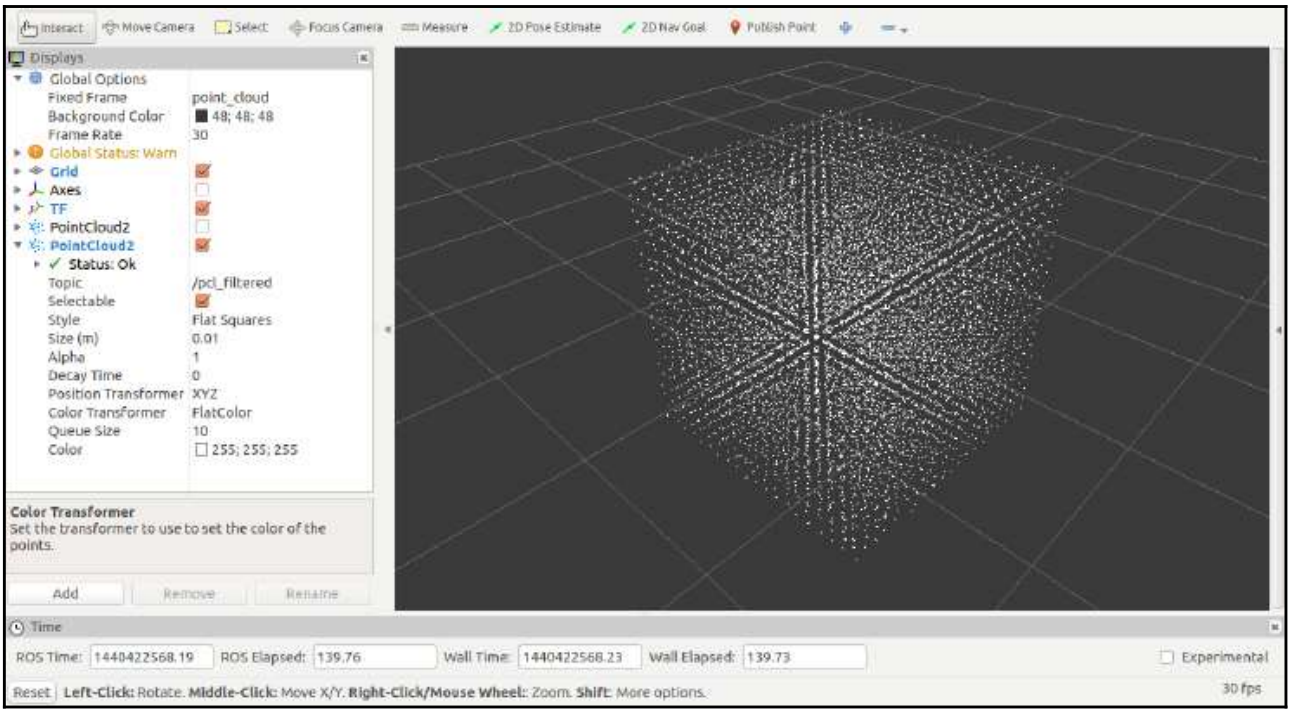

Figure 18: Filtered PCL cloud in RViz

We can write the `pcl_output` cloud by using the `pcl_write` node and read or publish by using the `pcl_read` nodes.

Working with AR Marker detection for object pose estimation

In this section, we will see how to use fiducial markers in order to enable a robot to easily interact with its environment. To interact with arbitrary objects, a robot should be able to recognize and localize them by relying on its vision sensors. Estimating the pose of an object represents an important feature of all robotic and computer-vision applications. However, efficient algorithms to perform object recognition and pose estimation working in real-world environments are difficult to implement, and in many cases one camera is not enough to retrieve the three-dimensional pose of an object.

More precisely, with the use of only one fixed camera, it is not possible to get spatial information about the depth of a framed scene. For this reason, object pose estimation is often simplified by exploiting AR markers. An AR marker is typically represented by a synthetic square image composed by a wide black border and an inner binary matrix which determines its unique identifier, as shown in Figure 19. The presence of a black border facilitates its fast detection in the image, and the binary codification allows its identification and the application of error detection and correction techniques:

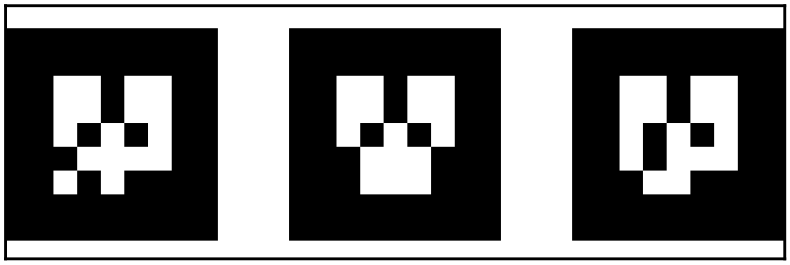

Figure 19: Augmented reality markers

The main benefit of these markers is that every fiducial image can be uniquely detected by our program, starting from a dictionary of markers, allowing the robot to identify and interact with a high number of objects. In addition, by configuring the size of the markers to detect, it is possible to estimate the distance of a given object with respect to the visual sensor. These markers are commonly called Augmented Reality markers because they have been widely used in augmented reality applications to display artificial information or artificial objects upon video frames. Different programs working with AR markers have been developed and many of these have been interfaced with ROS, such as, for example, ARToolKit (`https://github.com/ar-tools/ar_tools`) or ArUco: a minimal library for Augmented Reality applications (`https://www.uco.es/investiga/grupos/ava/node/26`). In the following section, we will discuss how to use the AprilTag visual fiducial system to detect and localize markers, interfacing it with ROS, and estimating the three-dimensional position and orientation of different objects.

Using AprilTag with ROS

AprilTag (`https://april.eecs.umich.edu/software/apriltag.html`) is a fiducial system particularly designed for robotics applications, thanks to its high level of precision computing the pose of AR markers, AprtilTag is particularly designed for robotics applications, you should clone the following repository:

```
$ git clone https://github.com/RIVeR-Lab/apriltags_ros.git
```

Now you can compile the ROS workspace in order to build the `apriltags` package and its ROS porting `apriltags_ros`. To use AprilTag, the following things are needed:

- **Video stream**: The video data received via `sensor_msgs/Image` is elaborated by searching for a list.
- **Camera calibration**: The calibration data received via `sensor_msgs/CameraInfo`, as shown in the previous sections.
- **Tag description***:* The configuration of the marker to detect. In particular its ID, the size of the markers, and the frames associated with its pose must be specified.
- **Tags**: A printed copy of the markers to detect. ApriTag already provides a complete set of `png` ready-to-print individual markers of five different encodings: 16h5, 25h7, 25h9, 36h9, or 36h11. These markers can be found in the `apriltags/tags` directory of `apriltags` package.

After it has been configured and launched, `apriltags_ros` will publish the pose of all the markers detected in the framed scene and configured in the launch file. In particular, the following topics will be published by the `apriltags` node:

```
/tag_detections
/tag_detections_image
/tag_detections_image/compressed
/tag_detections_image/compressed/parameter_descriptions
/tag_detections_image/compressed/parameter_updates
/tag_detections_image/compressedDepth
/tag_detections_image/compressedDepth/parameter_descriptions
/tag_detections_image/compressedDepth/parameter_updates
/tag_detections_image/theora
/tag_detections_image/theora/parameter_descriptions
/tag_detections_image/theora/parameter_updates
/tag_detections_pose
/tf
```

Figure 20: ROS topics published by AprilTag

The image elaboration process and the detected markers can be graphically seen displaying the image published on `/tag_detections_image`, where each fiducial marker is enlightened and labelled with its ID:

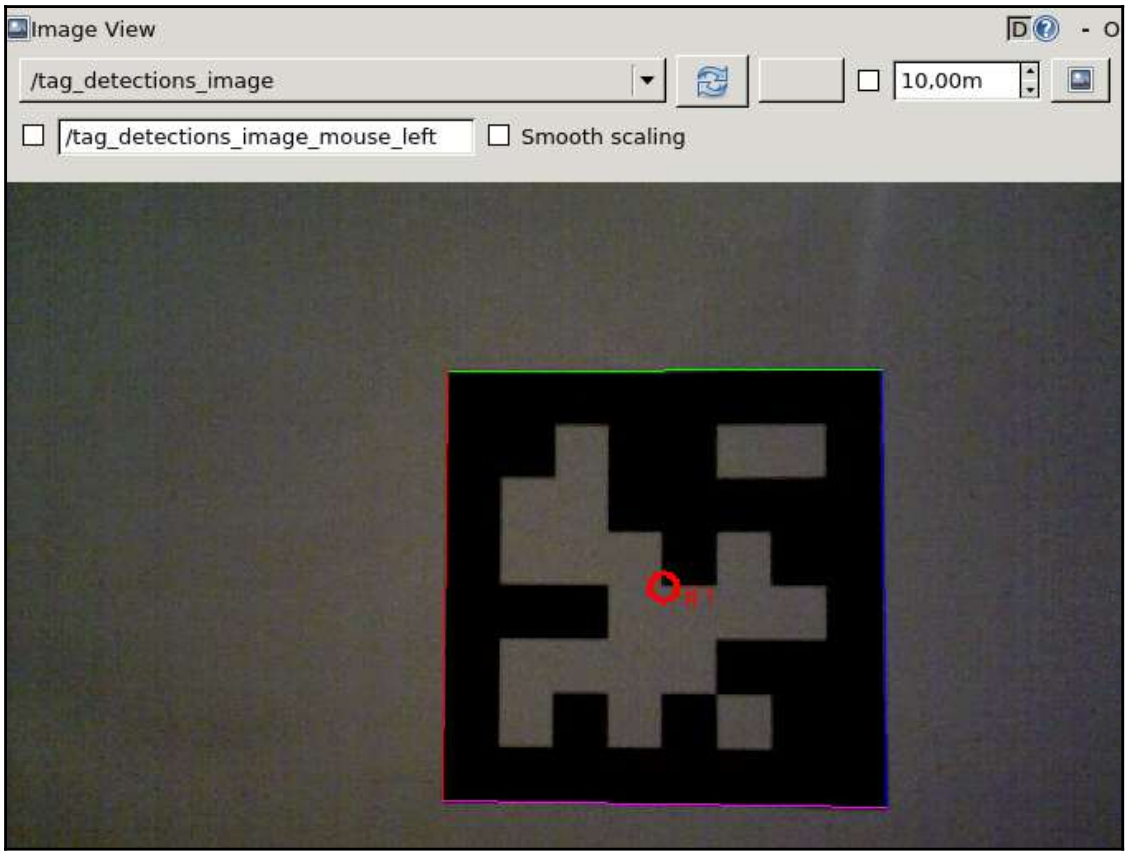

Figure 21: Graphical output of ROS AprilTag

Pose information about detected markers is published on `/tag_detections` of the `apriltags_ros/AprilTagDetectionArray` type. In this topic, the IDs and the poses of all detected markers are published. The content of this topic, associated to the frame of the previous figure, is shown following:

```
detections:
  -
    id: 1
    size: 0.08
    pose:
      header:
        seq: 55709
        stamp:
          secs: 1510415864
          nsecs: 148304216
        frame_id: camera_rgb_optical_frame
      pose:
        position:
          x: 0.0201272971812
          y: -0.02393358631
          z: 0.383437954847
        orientation:
          x: 0.713140734773
          y: -0.681737860948
          z: 0.153311144456
          w: 0.0562092015923
---
```

Figure 22: Position and orientation of markers detected by ROS AprilTag

Let's now discuss how to use `apriltags` to get the position of three objects. After calibrating our vision system, as shown in previous sections, we could configure and launch the `apriltags_ros` node. However, you could run this demo using the files of the `apriltags_ros_demo` ROS packages provided with the code of this book, or directly downloaded from the following Git repository:

```
$ git clone https://github.com/jocacace/apriltags_ros_demo
```

This package contains useful files to launch `apriltags` properly configured. In particular, you can find two main files:

- `bags`: This contains the `object.bag` bagfile, in which a video data and camera calibration information are streamed via ROS topics
- `launch`: This contains the launch file `apriltags_ros_objects.launch` that plays the `objects.bag` bagfile, and launches the `apriltags_ros` node configured to recognize three different markers

To configure the `apriltags_ros` node, we have to modify the launch file in the following way:

```
<node pkg="apriltags_ros" type="apriltag_detector_node"
name="apriltag_detector" output="screen">

    <remap from="image_rect" to="/camera/rgb/image_raw" />
    <remap from="camera_info" to="/camera/rgb/camera_info" />

    <param name="image_transport" type="str" value="compressed" />

    <param name="tag_family" type="str" value="36h11" />

    <rosparam param="tag_descriptions">[
      {id: 6, size: 0.035, frame_id: mastering_ros},
      {id: 1, size: 0.035, frame_id: laptop},
      {id: 7, size: 0.035, frame_id: smartphone}
      ]
    </rosparam>
</node>
```

In this launch file, we could set the topic used by the camera to stream the video stream and its image, using the `remap` command:

```
<remap from="image_rect" to="/camera/rgb/image_raw" />
<remap from="camera_info" to="/camera/rgb/camera_info" />
```

In addition, we must inform `apriltags_ros` about which markers must be detected. First, we should specify the family name:

```
<param name="tag_family" type="str" value="36h11" />
```

Finally, the ID of the markers, their size expressed in meters, and the frame attached to each tag, should be specified in the following way:

```
    <rosparam param="tag_descriptions">[
      {id: 6, size: 0.035, frame_id: mastering_ros},
      {id: 1, size: 0.035, frame_id: laptop},
      {id: 7, size: 0.035, frame_id: smartphone}
      ]
    </rosparam>
```

Note that the size of the marker is represented by the length of one of the sides of its black border. This information allows AR marker detectors to estimate the distance of the tag from the camera. For this reason, you should pay attention when providing this measure to get a precise pose estimation.

In this case, we want to detect three different markers, with IDs 6, 1, and 7 respectively, and with a size of 3.5 centimetres. Each marker is linked to a different frame, which will be even displayed on Rviz using `tf`. The complete launch file can be found in the `apriltags_ros_demo/launch` directory.

We can launch this example directly using the provided launch file, which firstly reproduces a `bagfile` containing the video of a scene composed by three objects or markers and the calibration of the camera:

```
$ roslaunch apriltag_ros_demo apriltags_ros_objects.launch
```

Now you can read the poses of these objects via the `apriltags_ros` topic output, or visualize them using Rviz, as show in the next image:

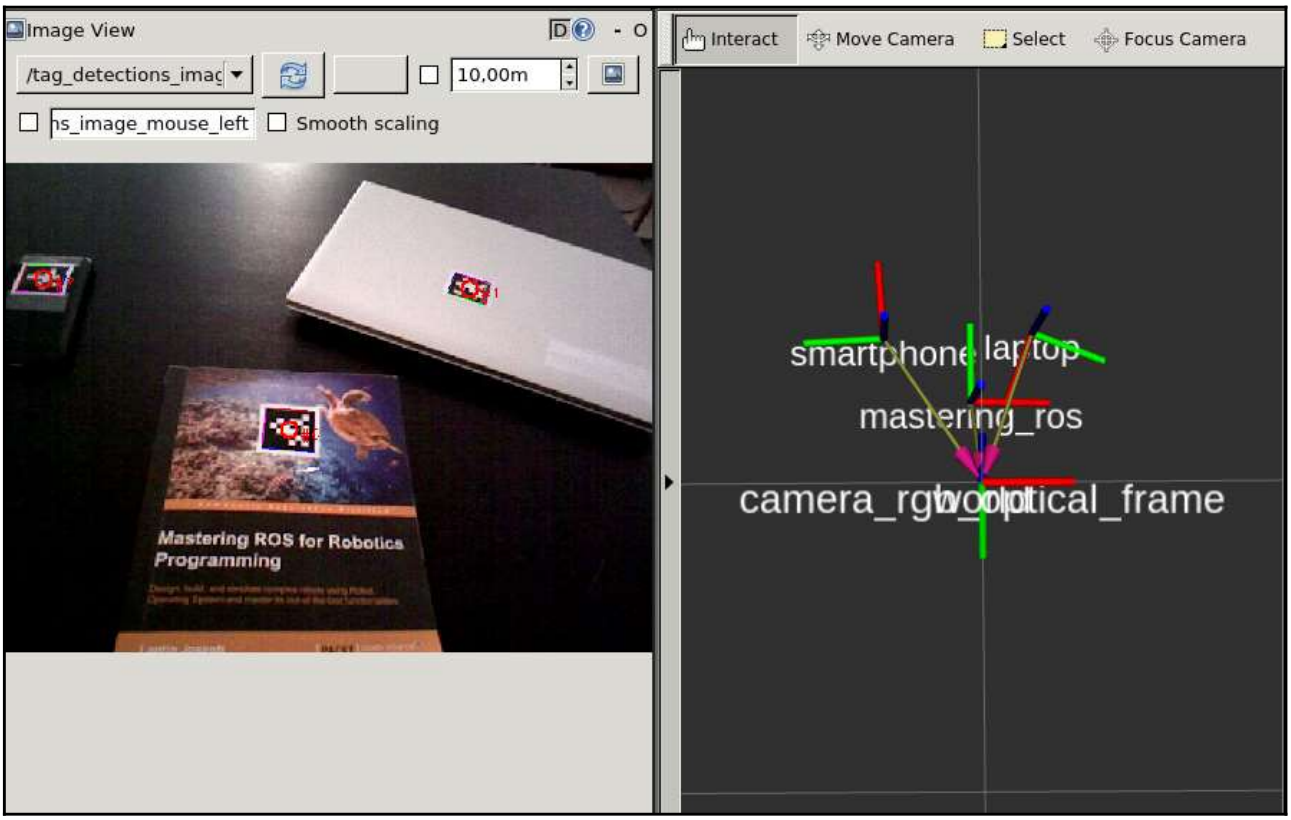

Figure 23: Tracking multiple objects using AR markers and ROS

Questions

- What are the packages in the `vision_opencv` stack?
- What are the packages in the `perception_pcl` stack?
- What are the functions of `cv_bridge`?
- How do we convert a PCL cloud to a ROS message?
- How do we do distributive computing using ROS?
- What is the main benefit of the AR markers?

Summary

This chapter was about vision sensors and their programming in ROS. We saw the interfacing packages used to interface the cameras and 3D vision sensors, such as `vision_opencv` and `perception_pcl`. We looked at each package and its functions on these stacks. We looked at the interfacing of a basic webcam and processing image, using ROS `cv_bridge`. After discussing `cv_bridge`, we looked at the interfacing of various 3D vision sensors and laser scanners with ROS. After interfacing, we learned how to process the data from these sensors, using the PCL library and ROS. Finally, in the last part of the chapter, we showed how to use fiducial markers to easily perform object pose estimation and localization. In the next chapter, we will look at the interfacing of robotic hardware in ROS.