7
Programming Vision Sensors Using Python and ROS

In the previous chapter, we have seen some of the robotic sensors used in our robot and its interfacing with the Launchpad board. In this chapter, we will mainly discuss vision sensors and its interface used in our robot.

The robot we are designing will have a 3D sensor and we can interface it with vision libraries such as OpenCV, OpenNI, and **Point Cloud Library** (**PCL**). Some of the applications of the 3D vision sensor in our robot are autonomous navigation, obstacle avoidance, object detection, people tracking, and so on.

We will also discuss the interfacing of vision sensors and image processing libraries with ROS. In the last section of the chapter, we will see a navigational algorithm for our robot called **SLAM** (**Simultaneous Localization and Mapping**) and its implementation using a 3D sensor, ROS, and image processing libraries.

In the first section, we will see some 2D and 3D vision sensors available on the market that we will use in our robot.

List of robotic vision sensors and image processing libraries

A 2D vision sensor or an ordinary camera delivers 2D image frames of the surroundings, whereas a 3D vision sensor delivers 2D image frames and an additional parameter called depth of each image point. We can find the x, y, and z distance of each point from the 3D sensor with respect to the sensor axis.

There are quite a few vision sensors available on the market. Some of the 2D and 3D vision sensors that can be used in our robot are mentioned in this chapter.

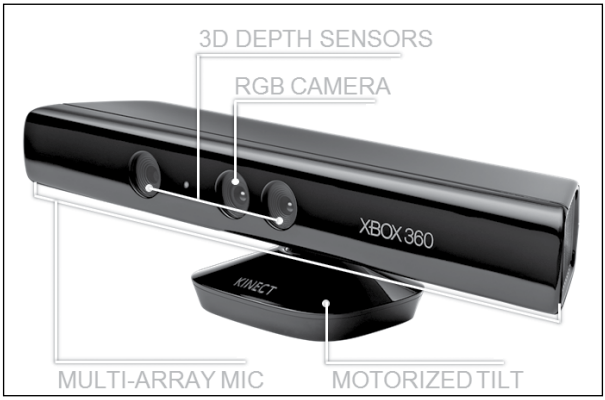
The following figure shows the latest 2D vision sensor called Pixy/CMU cam 5 (`http://www.cmucam.org/`), which is able to detect color objects with high speed and accuracy and can be interfaced to an Arduino board. Pixy can be used for fast object detection and the user can teach which object it needs to track. Pixy module has a CMOS sensor and NXP (`http://www.nxp.com/`) processor for image processing:

Pixy/CMU Cam 5

The commonly available 2D vision sensors are webcams. They contain a CMOS sensor and USB interface, but there is no inbuilt processing for the object detection. The following image shows a popular webcam from Logitech that can capture pictures of up to 5 megapixel resolution and HD videos:

Logitech HD Cam

We can take a look at some of the 3D vision sensors available on the market. Some of the popular sensors are Kinect, Asus Xtion Pro, and Carmine.



Kinect

Kinect is a 3D vision sensor used along with the Microsoft Xbox 360 game console. It mainly contains an RGB camera, an infrared projector, a depth sensor, a microphone array, and a motor for tilt. The RGB and depth camera capture images at a resolution of 640 x 480 at 30 Hz. The RGB camera captures 2D color images, whereas the depth camera captures monochrome depth images. Kinect has a depth sensing range from 0.8 m to 4 m.

Some of the applications of Kinect are 3D motion capture, skeleton tracking, face recognition, and voice recognition.

Kinect can be interfaced to PC using the USB 2.0 interface and programmed using Kinect SDK, OpenNI, and OpenCV. Kinect SDK is only available for Windows platforms and is developed and supplied by Microsoft. The other two libraries are open source and available for all platforms. The Kinect we are using here is the first version; the latest versions of Kinect only support Kinect SDK running on Windows.

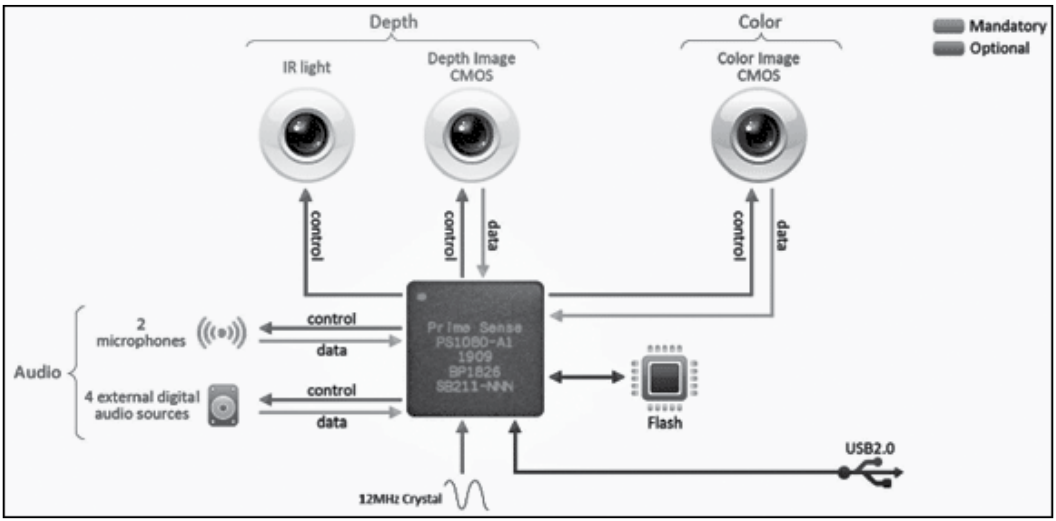
Asus Xtion Pro

Asus Xtion Pro is a 3D sensor designed for PC-based motion sensing applications. Xtion Pro is only for 3D sensing and it doesn't have any sound sensing facilities. It has an infrared projector and a monochrome CMOS sensor to capture the infrared data. Xtion Pro communicates to the PC via the USB 2.0 interface. Xtion can be powered from the USB itself and can calculate a sense depth from 0.8 m to 3.5 m from the sensor.

The applications of Kinect and Xtion Pro are the same except for voice recognition. It will work in Windows, Linux, and Mac. We can develop applications in Xtion Pro using OpenNI and OpenCV.

PrimeSense Carmine

The Prime Sense team developed the Microsoft Kinect 3D vision system. Later, they developed their own 3D vision sensor called Carmine. The technology behind Carmine is similar to Kinect. It works with an IR projector and a depth image CMOS sensor. The following figure shows the block diagram of Carmine:



Carmine block diagram

Similar to Kinect, Carmine has an RGB CMOS sensor, a depth image CMOS, and an IR light source. It also has an array of microphones for voice recognition. All sensors are interfaced in **System On Chip** (**SOC**). Interfacing and powering is performed through USB.

Carmine can capture RGB and depth frames in 640 x 480 resolution and can sense depth from 0.35 m to 3.5 m. Compared to Kinect, the advantages are small power consumption, small form factor, and good depth sensing range.

Carmine can be interfaced to a PC and it will support Windows, Linux, Mac, and Android platforms. Carmine is supported by OpenNI; developers can program the device using OpenNI and its wrapper libraries.

Apple Inc bought Prime Sense in November 2013. You can buy Carmine at the following link:

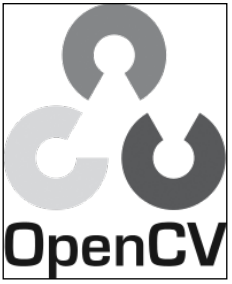
```
http://www.amazon.com/dp/B00KO908MM?psc=1
```

Introduction to OpenCV, OpenNI, and PCL

Let's discuss about the software frameworks and libraries that we are using in our robots. First, we can discuss OpenCV. This is one of the libraries that we are going to use in this robot for object detection and other image processing functionalities.

What is OpenCV?

OpenCV is an open source BSD-licensed computer vision based library that includes hundreds of computer vision algorithms. The library, mainly aimed for real-time computer vision, was developed by Intel Russia research, and is now actively supported by Itseez (`http://itseez.com/`).



OpenCV logo

OpenCV is written mainly in C and C++ and its primary interface is in C++. It also has good interfaces in Python, Java, Matlab/Octave and wrappers in other languages such as C# and Ruby.

In the new version of OpenCV, there is support for CUDA and OpenCL to get GPU acceleration (`http://www.nvidia.com/object/cuda_home_new.html`).

OpenCV will run on most of the OS platforms (such as Windows, Linux, Mac OS X, Android, FreeBSD, OpenBSD, iOS, and Blackberry).

In Ubuntu, OpenCV, and Python, wrappers are already installed when we install the `ros-indigo-desktop-full` package. If this package is not installed, then we can install the OpenCV library, ROS interface, and Python interface of OpenCV using the following command:

```
$ sudo apt-get install ros-indigo-vision-opencv
```

If you want to install only the OpenCV Python wrapper, then use the following command:

```
$ sudo apt-get install python-opencv
```

If you want to try OpenCV in Windows, you can try the following link:

`http://docs.opencv.org/doc/tutorials/introduction/windows_install/windows_install.html`

The following link will guide you through the installation process of OpenCV on Mac OS X:

`http://jjyap.wordpress.com/2014/05/24/installing-opencv-2-4-9-on-mac-osx-with-python-support/`

The main applications of OpenCV are in the field of:

- Object detection
- Gesture recognition
- Human-computer interaction
- Mobile robotics
- Motion tracking
- Facial recognition

Now we can see how to install OpenCV in Ubuntu 14.04.2 from source code.

Installation of OpenCV from source code in Ubuntu 14.04.2

We can install OpenCV from source code in Linux based on the following documentation of OpenCV:

`http://docs.opencv.org/doc/tutorials/introduction/linux_install/linux_install.html`

After the installation of OpenCV, we can try some examples using the Python wrappers of OpenCV.

Reading and displaying an image using the Python-OpenCV interface

The first example will load an image in grayscale and display it on screen.

In the following section of code, we will import the `numpy` module for image array manipulation and the `cv2` module is the OpenCV wrapper for Python in which we can access OpenCV Python APIs. NumPy is an extension to the Python programming language, adding support for large multidimensional arrays and matrices along with a large library of high-level mathematical functions to operate on these arrays (`https://pypi.python.org/pypi/numpy`):

```
#!/usr/bin/env python
import numpy as np
import cv2
```

The following function will read the `robot.jpg` image and load this image in grayscale. The first argument of the `cv2.imread()` function is the name of the image and the next argument is a flag that specifies the color type of the loaded image. If the flag is > 0, the image returns a three channel RGB color image, if the flag = 0, the loaded image will be a grayscale image, and if the flag is < 0, it will return the same image as loaded:

```
img = cv2.imread('robot.jpg',0)
```

The following section of code will show the read image using the `imshow()` function. The `cv2.waitKey(0)` function is a keyboard binding function. Its argument is time in milliseconds. If it's 0, it will wait indefinitely for a key stroke:

```
cv2.imshow('image',img)
cv2.waitKey(0)
```

The `cv2.destroyAllWindows()` function simply destroys all the windows we created:

```
cv2.destroyAllWindows()
```

Save the preceding code with a name called `image_read.py` and copy a JPG file with `robot.jpg` as its name. Execute the code using the following command:

```
$python image_read.py
```

The output will load an image in grayscale because we used `0` as the value in the `imread()` function:

The following example will try to open webcam. The program will quit when the user presses any button.

Capturing from web camera

The following code will capture the webcam having device name `/dev/video0` or `/dev/video1`.

We need to import the following modules if we are using OpenCV API's:

```
#!/usr/bin/env python
import numpy as np
import cv2
```

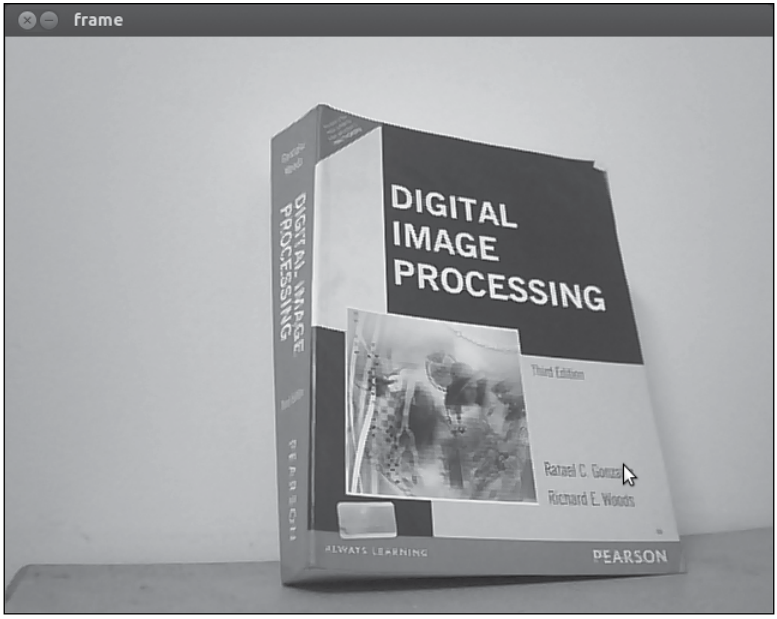
The following function will create a `VideoCapture` object. The `VideoCapture` class is used to capture videos from video files or cameras. The initialization arguments of the `VideoCapture` class is the index of a camera or a name of a video file. Device index is just a number to specify the camera. The first camera index is 0 having device name `/dev/video0`; that's why we use `0` here:

```
cap = cv2.VideoCapture(0)
```

The following section of code is looped to read image frames from the `VideoCapture` object and shows each frame. It will quit when any key is pressed:

```
while(True):
    # Capture frame-by-frame
    ret, frame = cap.read()
    # Display the resulting frame
    cv2.imshow('frame',frame)
    if cv2.waitKey(10):
        break
```

The following is a screenshot of the program output:



You can explore more OpenCV-Python tutorials from the following link:

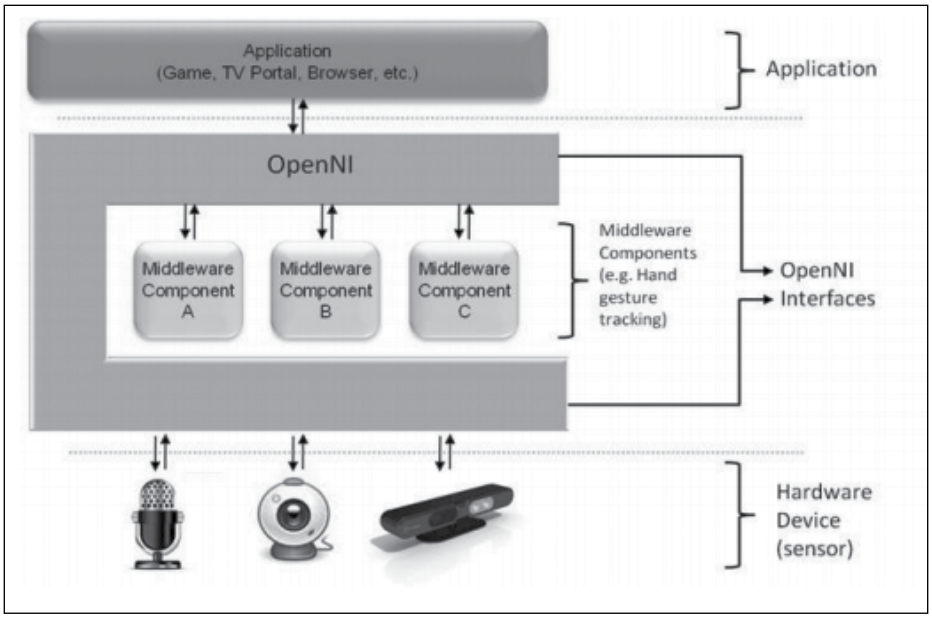
`http://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/py_tutorials.html`

In the next section, we will look at OpenNI library and its application.

What is OpenNI

OpenNI is a Multilanguage, cross-platform framework that defines API's to write applications using **Natural interaction** (**NI**). Natural interaction is defined in terms of experience. It means, people naturally communicate through gestures, expressions, movements, and discover the world by looking around and manipulating physical stuff.

OpenNI API's are composed of a set of interfaces to write NI applications. The following figure shows a three-layered view of the OpenNI library:



The top layer represents the application layer that implements natural interaction-based application. The middle layer is the OpenNI layer and it will provide communication interfaces that interact with sensors and middleware components that analyze the data from the sensor. Middleware can be used for full body analysis, hand point analysis, gesture detection, and so on. One of the examples of middle layer is NITE, which can detect gesture and skeleton.

The bottom layer shows the hardware devices that capture visuals and audio elements of the scene. It includes 3D sensors, RGB cameras, an IR camera, and a microphone.

OpenNI is cross-platform and has been successfully compiled and deployed on Linux, Mac OS X, and Windows.

In the next section, we will see how we to install OpenNI in Ubuntu 14.04.2.

Installing OpenNI in Ubuntu 14.04.2

We can install the OpenNI library along with ROS packages. ROS is already interfaced with OpenNI, but the complete installation of `ros-indigo-desktop-full` may not install OpenNI packages; we need to install it from the package manager.

The following is the installation command:

```
$ sudo apt-get install ros-indigo-openni-launch
```

The source code and latest build of OpenNI for Windows, Linux, and MacOS X is available at the following link:

`http://structure.io/openni`

In the next section, we will see how to install PCL.

What is PCL?

PCL is a large scale, open project for 2D/3D image, and Point Cloud processing. The PCL framework contains numerous algorithms included to perform filtering, feature estimation, surface reconstruction, registration, model fitting, and segmentation. Using these methods, we can process Point Cloud and extract key descriptors to recognize objects in the world based on their geometric appearance and create surfaces from the Point Clouds and visualize them.

PCL logo

PCL is released under the BSD license. It's open source, and free for commercial, or research use. PCL is cross-platform and has been successfully compiled and deployed on Linux, Mac OS X, Windows, and Android/iOS.

You can download PCL at the following link:

`http://pointclouds.org/downloads/`

PCL is already integrated into ROS. The PCL library and its ROS interface will install along with ROS full desktop installation. In the previous chapter, we discussed how to install ROS full desktop installation. PCL is the 3D processing backbone of ROS. Refer to the following link for details on the ROS-PCL package:

`http://wiki.ros.org/pcl`.

Programming Kinect with Python using ROS, OpenCV, and OpenNI

Let's look at how we can interface and work with the Kinect sensor in ROS. ROS is bundled with OpenNI driver, which can fetch RGB and the depth image of Kinect. This package can be used for Microsoft Kinect, PrimeSense Carmine, Asus Xtion Pro, and Pro Live.

This driver mainly publishes raw depth, RGB, and IR image streams. The `openni_launch` package will install packages such as `openni_camera` and `openni_launch`. The `openni_camera` package is the Kinect driver that publishes raw data and sensor information, whereas the `openni_launch` package contains ROS launch files. It's basically an XML file that launches multiple nodes at a time and publishes data such as point clouds.

How to launch OpenNI driver

The following command will open the OpenNI device and load all nodelets to convert raw depth/RGB/IR streams to depth images, disparity images, and point clouds. The ROS `nodelet` package is designed to provide a way to run multiple algorithms in the same process with zero copy transport between algorithms.

```
$ roslaunch openni_launch openni.launch
```

You can view the RGB image using a ROS tool called `image_view`

```
$ rosrun image_view image_view image:=/camera/rgb/image_color
```

In the next section, we will see how to interface these images to OpenCV for image processing.

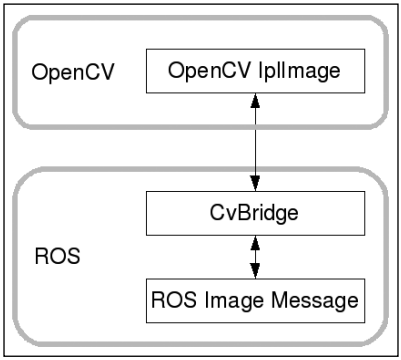
The ROS interface of OpenCV

ROS is integrated into many libraries. OpenCV is also integrated into ROS mainly for image processing. The `vision_opencv` ROS stack includes the complete OpenCV library and interface to ROS.

The `vision_opencv` provides several packages:

- `cv_bridge`: This contains the `CvBridge` class; this class converts from ROS image messages to OpenCV image data type and vice versa
- `image_geometry`: This contains a collection of methods to handle image and pixel geometry

The following diagram shows how OpenCV is interfaced to ROS:



OpenCV-ROS interfacing

The image data type of OpenCV are IplImage and Mat. If we want to work with OpenCV in ROS, we have to convert IplImage or Mat to ROS Image messages. The ROS package `vision_opencv` has the `CvBridge` class; this class can convert IplImage to ROS image and vice versa.

The following section shows how to create a ROS package; this package contains node to subscribe RGB, depth image, process the RGB image to detect edges, and display all images after converting to an image type equivalent to OpenCV.

Creating ROS package with OpenCV support

We can create a package called `sample_opencv_pkg` with the following dependencies, that is, `sensor_msgs`, `cv_bridge`, `rospy`, and `std_msgs`. The `sensor_msgs` dependency defines messages for commonly used sensors, including cameras and scanning laser rangefinders; `cv_bridge` is the OpenCV interface of ROS.

The following command will create the ROS package with the preceding dependencies:

```
$ catkin-create-pkg sample_opencv_pkg sensor_msgs cv_bridge
rospy std_msgs
```

After creating the package, create a `scripts` folder inside the package and save the code in the mentioned in the next section.

Displaying Kinect images using Python, ROS, and cv_bridge

The first section of the Python code is given below. It mainly includes importing of `rospy`, `sys`, `cv2`, `sensor_msgs`, `cv_bridge`, and the `numpy` module. The `sensor_msgs` dependency imports the ROS data type of `Image` and `CameraInfo`. The `cv_bridge` module imports the `CvBridge` class for converting ROS image data type to the OpenCV data type and vice versa:

```
import rospy
import sys
import cv2
import cv2.cv as cv
from sensor_msgs.msg import Image, CameraInfo
from cv_bridge import CvBridge, CvBridgeError
import numpy as np
```

The following section of code is a class definition in Python to demonstrate `CvBridge` functions. The class is named as `cvBridgeDemo`:

```
class cvBridgeDemo():
    def __init__(self):
        self.node_name = "cv_bridge_demo"
        #Initialize the ros node
        rospy.init_node(self.node_name)

        # What we do during shutdown
        rospy.on_shutdown(self.cleanup)

        # Create the OpenCV display window for the RGB image

        self.cv_window_name = self.node_name
        cv.NamedWindow(self.cv_window_name, cv.CV_WINDOW_NORMAL)
        cv.MoveWindow(self.cv_window_name, 25, 75)

        # And one for the depth image
```

```
        cv.NamedWindow("Depth Image", cv.CV_WINDOW_NORMAL)
        cv.MoveWindow("Depth Image", 25, 350)

        # Create the cv_bridge object
        self.bridge = CvBridge()

        # Subscribe to the camera image and depth topics and set
        # the appropriate callbacks
        self.image_sub =
rospy.Subscriber("/camera/rgb/image_color", Image,
self.image_callback)
        self.depth_sub =
rospy.Subscriber("/camera/depth/image_raw", Image,
self.depth_callback)

        rospy.loginfo("Waiting for image topics...")
```

The following code gives a callback function of the color image from Kinect. When a color image comes on the `/camera/rgb/image_color` topic, it will call this function. This function will process the color frame for edge detection and show the edge detected and raw color image:

```
    def image_callback(self, ros_image):
        # Use cv_bridge() to convert the ROS image to OpenCV format
        try:
            frame = self.bridge.imgmsg_to_cv(ros_image, "bgr8")
        except CvBridgeError, e:
            print e

        # Convert the image to a Numpy array since most cv2
functions

        # require Numpy arrays.
        frame = np.array(frame, dtype=np.uint8)

        # Process the frame using the process_image() function
        display_image = self.process_image(frame)

        # Display the image.
        cv2.imshow(self.node_name, display_image)

        # Process any keyboard commands
        self.keystroke = cv.WaitKey(5)
        if 32 <= self.keystroke and self.keystroke < 128:
            cc = chr(self.keystroke).lower()
```

```
            if cc == 'q':
                # The user has press the q key, so exit
                rospy.signal_shutdown("User hit q key to quit.")
```

The following code gives a callback function of the depth image from Kinect. When a depth image comes on the `/camera/depth/raw_image` topic, it will call this function. This function will show the raw depth image:

```
    def depth_callback(self, ros_image):
        # Use cv_bridge() to convert the ROS image to OpenCV
format
        try:
            # The depth image is a single-channel float32 image
            depth_image = self.bridge.imgmsg_to_cv(ros_image,
"32FC1")
        except CvBridgeError, e:
            print e

        # Convert the depth image to a Numpy array since most cv2
functions
        # require Numpy arrays.
        depth_array = np.array(depth_image, dtype=np.float32)

        # Normalize the depth image to fall between 0 (black) and
1 (white)
        cv2.normalize(depth_array,
depth_array, 0, 1, cv2.NORM_MINMAX)

        # Process the depth image
        depth_display_image =
self.process_depth_image(depth_array)

        # Display the result
        cv2.imshow("Depth Image", depth_display_image)
```

The following function is called `process_image()`, which will convert the color image to grayscale, then blur the image, and find the edges using the canny edge filter:

```
    def process_image(self, frame):
        # Convert to grayscale
        grey = cv2.cvtColor(frame, cv.CV_BGR2GRAY)

        # Blur the image
        grey = cv2.blur(grey, (7, 7))
```

```
            # Compute edges using the Canny edge filter
            edges = cv2.Canny(grey, 15.0, 30.0)

            return edges
```

The following function is called `process_depth_image()`. It simply returns the depth frame:

```
        def process_depth_image(self, frame):
            # Just return the raw image for this demo
            return frame
```

This function will close the image window when the node shuts down:

```
        def cleanup(self):
            print "Shutting down vision node."
            cv2.destroyAllWindows()
```

The following code is the `main()` function. It will initialize the `cvBridgeDemo()` class and call the ros `spin()` function:

```
    def main(args):
        try:
            cvBridgeDemo()
            rospy.spin()
        except KeyboardInterrupt:
            print "Shutting down vision node."
            cv.DestroyAllWindows()

    if __name__ == '__main__':
        main(sys.argv)
```

Save the preceding code to `cv_bridge_demo.py` and change the permission of the node using the following command. The node is only visible to the `rosrun` command if we give it executable permission.

```
$ chmod +X cv_bridge_demo.py
```

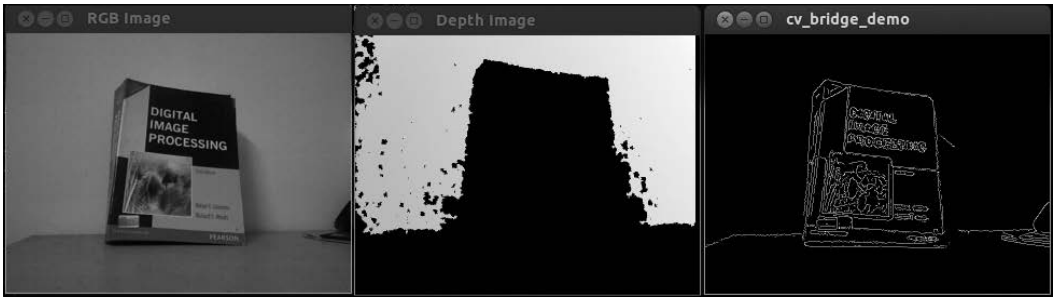
The following are the commands to start the driver and node. Start the Kinect driver using the following command:

```
$ roslaunch openni_launch openni.launch
```

Run the node using the following command:

```
$ rosrun sample_opencv_pkg cv_bridge_demo.py
```

The following is the screenshot of the output:



RGB, depth, and edge images

Working with Point Clouds using Kinect, ROS, OpenNI, and PCL

A Point Cloud is a data structure used to represent a collection of multidimensional points and is commonly used to represent 3D data. In a 3D Point Cloud, the points usually represent the *x*, *y*, and *z* geometric coordinates of an underlying sampled surface. When the color information is present, the Point Cloud becomes 4D.

Point Clouds can be acquired from hardware sensors (such as stereo cameras, 3D scanners, or time-of-flight cameras). It can be generated from a computer program synthetically. PCL supports the OpenNI 3D interfaces natively; thus it can acquire and process data from devices (such as Prime Sensor's 3D cameras, Microsoft Kinect, or Asus XTion PRO).

PCL will be installed along with the ROS indigo full desktop installation. Let's see how we can generate and visualize Point Cloud in RViz, a data visualization tool in ROS.

Opening device and Point Cloud generation

Open a new terminal and launch the ROS OpenNI driver along with the Point Cloud generator nodes using the following command:

```
roslaunch openni_launch openni.launch
```

This command will activate the Kinect driver and process the raw data into convenient outputs like Point Cloud.

We will use RViz 3D visualization tool to view Point Clouds.

The following command will start the RViz tool:

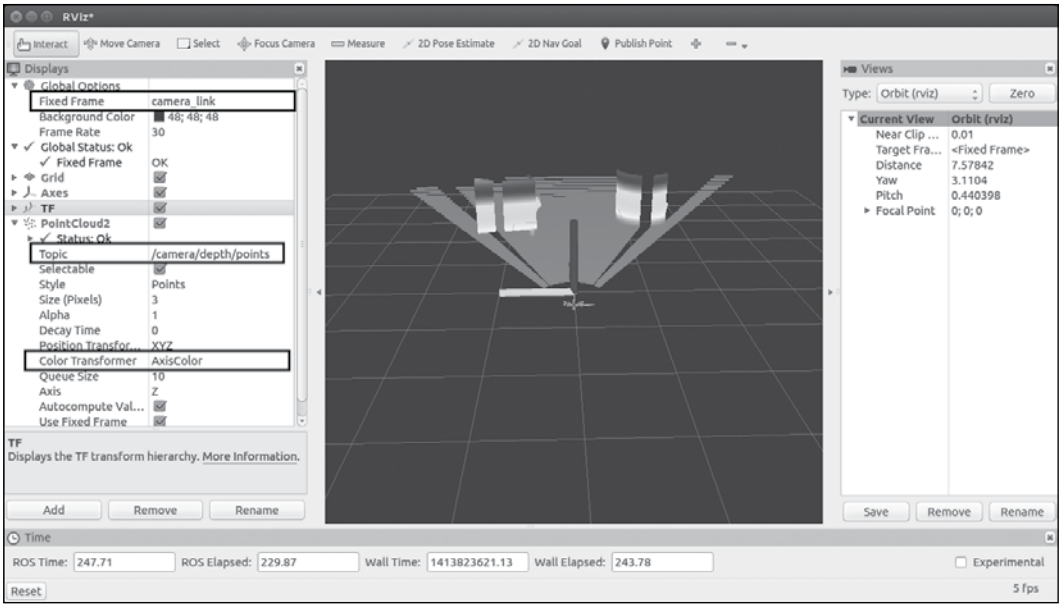
```
$ rosrun rviz rviz
```

Set the RViz options for **Fixed Frame** (at the top of the **Displays** panel under **Global Options**) to **camera_link**.

On the left-hand side of the **RViz** panel, click on the **Add** button and choose the **PointCloud2** display option. Set its topic to **/camera/depth/points.**

Change **Color Transformer** of **PointCloud2** to **AxisColor**.

The following figure shows a screenshot of RViz Point Cloud data. In this screenshot, the near object is marked in red and the far object is marked in violet and blue. The object in front of Kinect is represented as cylinder and cube:



Point Cloud of a robot

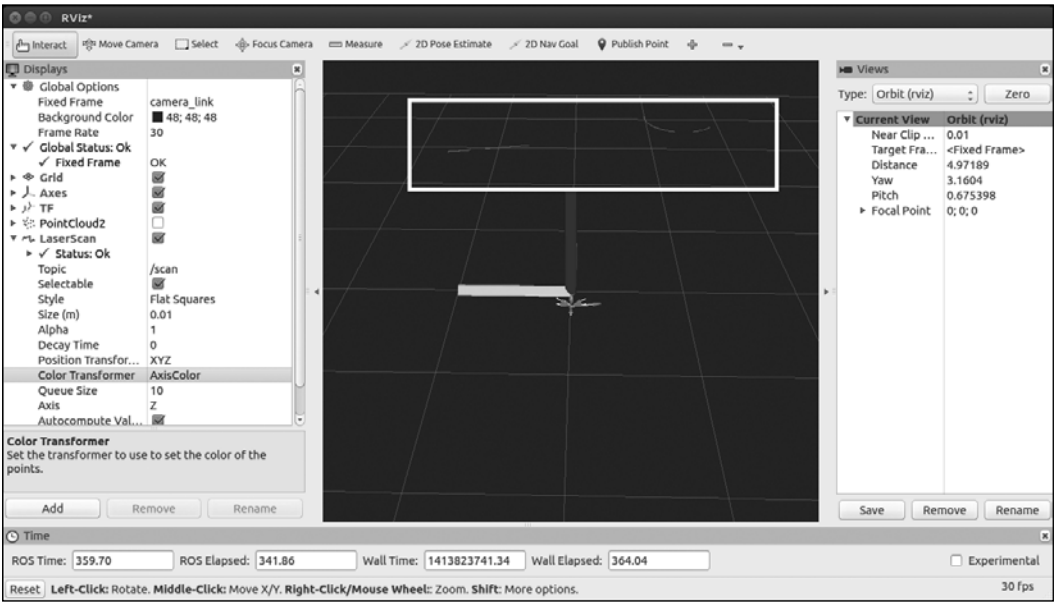
Conversion of Point Cloud to laser scan data

We are using Kinect in this robot for replicating the function of expensive laser range scanner. Kinect can deliver Point Cloud data which contains the depth of each point of surrounding. The Point Cloud data is processed and converted to data equivalent to a laser scanner using the ROS `depthimage_to_laserscan` package. The main function of this package is to slice a section of the Point Cloud data and convert it to a laser scan equivalent data type. The `Pointcloud2` data type is `sensor_msgs/PointCloud2` and for the laser scanner, the data type is `sensor_msgs/LaserScan`. This package will perform this processing and fake the laser scanner. The laser scanner output can be viewed using RViz. In order to run the conversion, we have to start the convertor nodelets that will perform this operation. We have to specify this in our launch file to start the conversion. The following is the required code in the launch file to start the `depthimage_to_laserscan` conversion:

```
  <!-- Fake laser -->
  <node pkg="nodelet" type="nodelet"
name="laserscan_nodelet_manager" args="manager"/>
  <node pkg="nodelet" type="nodelet"
name="depthimage_to_laserscan"
        args="load depthimage_to_laserscan/
DepthImageToLaserScanNodelet
laserscan_nodelet_manager">
    <param name="scan_height" value="10"/>
    <param name="output_frame_id" value="/camera_depth_frame"/>
    <param name="range_min" value="0.45"/>
    <remap from="image" to="/camera/depth/image_raw"/>
    <remap from="scan" to="/scan"/>
  </node>
```

Along with starting the nodelet, we need to set certain parameters of the nodelet for better conversion. Refer to `http://wiki.ros.org/depthimage_to_laserscan` for a detailed explanation of each parameter.

The laser scan of the preceding view is shown in the following screenshot. To view the laser scan, add the **LaserScan** option. This is similar to how we add the **PointCloud2** option and change the **Topic** value of **LaserScan** to **/scan**:



Working with SLAM using ROS and Kinect

The main aim of deploying vision sensors in our robot is to detect objects and perform robot navigation in an environment. SLAM is a technique used in mobile robots and vehicles to build up a map of an unknown environment or update a map within a known environment by tracking the current location of a robot.

Maps are used to plan the robot trajectory and to navigate through this path. Using maps, the robot will get an idea about the environment. The main two challenges in mobile robot navigation are mapping and localization.

Mapping involves generating a profile of obstacles around the robot. Through mapping, the robot will understand how the world looks. Localization is the process of estimating a pose of the robot relative to the map we build.

SLAM fetches data from different sensors and uses it to build maps. The 2D/3D vision sensor can be used as an input to SLAM. The 2D vision sensors such as laser range finders and 3D sensors such as Kinect are mainly used as an input for a SLAM algorithm.

ROS is integrated into a SLAM library using OpenSlam (`http://openslam.org/gmapping.html`). The `gmapping` package provides laser-based SLAM as a node called `slam_gmapping`. This can create a 2D map from the laser and pose data collected by a mobile robot.

The `gmapping` package is available at `http://wiki.ros.org/gmapping`.

To use `slam_gmapping`, you will need a mobile robot that provides odometry data and is equipped with a horizontally mounted, fixed, laser range finder. The `slam_gmapping` node will attempt to transform each incoming scan into the odom (odometry) tf frame.

The `slam_gmapping` node takes in `sensor_msgs/LaserScan` messages and builds a map (`nav_msgs/OccupancyGrid`). The map can be retrieved via a ROS topic or service.

The following code can be used to make a map from a robot with a laser publishing scans on the `base_scan` topic:

```
$ rosrun gmapping slam_gmapping scan:=base_scan
```

Questions

1. What are 3D sensors and how are they different from ordinary cams?
2. What are the main features of a robotic operating system?
3. What are the applications of OpenCV, OpenNI, and PCL?
4. What is SLAM?
5. What is RGB-D SLAM and how does it work?

Summary

In this chapter, we saw vision sensors to be used in our robot. We used Kinect in our robot and discussed OpenCV, OpenNI, PCL and their application. We also discussed the role of vision sensors in robot navigation and a popular SLAM technique and its application using ROS. In the next chapter, we will discuss speech processing and synthesis to be used in this robot.