Chapter 12

3D Vision

We have built our humanoid robot arms in the last chapter, and the next topic will be what practical tasks can be realized based on them. As humans, we naturally think that almost all basic dual-arm coordination tasks are impossible to accomplish without sensors such as eyes, especially if the task involves complex manipulation or object handling. It requires careful planning and control strategies to ensure smooth, stable, and collision-free movements.

One type of sensor that mimics human eyes is 3D cameras. These cameras emulate human eyes primarily through stereoscopic vision, using two sensors placed like human eyes to perceive depth. The `D435(i)` shown in Figure 12.1 is part of the Intel® `RealSense` Depth Camera series, desigend for easy integration of 3D sensing into devices and machines. The DS435(i) contains one Intel RealSense Module D430 (depth perception with stereo sensors) and an RGB camera. The D430 captures stereo images as well as depth (by using a structured-light 3D scanner) with a wide field of view and a maximum range of up to 10 meters, making it suitable for a variety of applications that require 3D perception of the environment. Models ending with 'i', such as the DS435i, include an additional `IMU` (inertial measurement unit) module, providing extra data that enables better dense reconstruction.

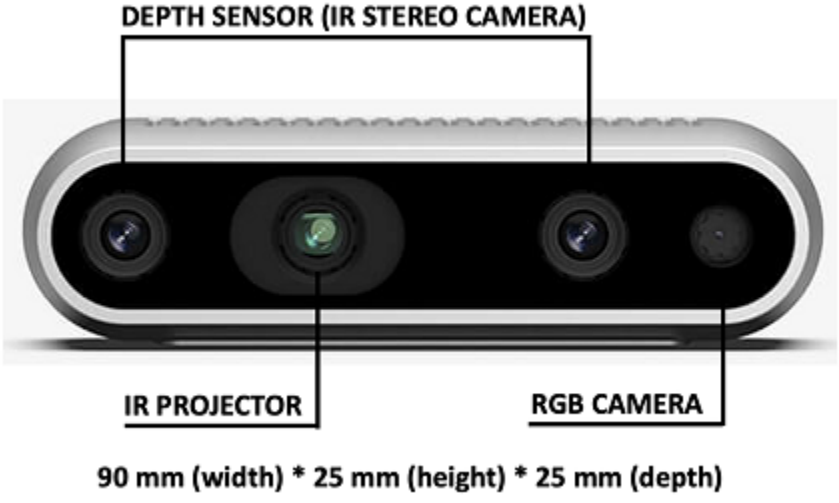

Figure 12.1: Intel RealSense D435 Depth Camera

Since our dual-arms platform is stationary , `D435` without `IMU` as our ROS reception model just meets

our requirement.

12.1 Install Depth Camera SDK and Drivers

Intel provides RealSense™ Software Development Kit 2.0 (SDK 2.0) also known as librealsense, a cross-platform library for `Realsense` depth camera (*https://github.com/IntelRealSense/librealsense*). The SDK 2.0 is supported on on Ubuntu (16.04 ~ 22.04). Besides that, the Intel RealSense also supports the ROS environment with the `realsense-ros` driver and ROS packages.

12.1.1 Installing RealSense SDK 2.0

To use your Intel `RealSense` D435 camera with ROS, you need to install both the `librealsense` SDK and the `realsense-ros` driver.

- Using ROS Distribution: You can install `librealsense2` over ROS distribution: An advantage of this method is that it will install `realsense2_camera` and its dependencies, including libraries: `librealsense2` and `librealsense2-udev-rules` . The disadvantage of this method is that you have to resolve any command errors during installation, such as "register public key error", "mismatched python library", `lsb_release` command does not exist, et al.

First update your ubuntu 18.04 to the latest packages

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get dist-upgrade
```

Installing `librealsense` Packages:

Register the server's public key:

```
$ sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-key
     F6E65AC044F831AC80A06380C8B3A55A6F3EFCDE ||
     sudo apt-key adv --keyserver  hkp://keyserver.ubuntu.com:80
     --recv-key  F6E65AC044F831AC80A06380C8B3A55A6F3EFCDE
```

Then add this server to the list of repositories:

```
$ sudo add-apt-repository "deb https://librealsense.intel.com/Debian/apt-repo
                 $(lsb_release -cs) main"
$ sudo apt update
```

Install `librealsense2-udev-rules` and executable demos and tools:

```
$ sudo apt-get install librealsense2-dkms librealsense2-utils
                librealsense2-dev librealsense2-dbg
```

- Install `librealsense` libraries from source code:

Alternatively, you can download source code, compile and build on your local machine:

```
$ git clone -b r/255 https://github.com/IntelRealSense/librealsense.git
$  cd librealsense
```

Install dependencies:

```
sudo apt-get install libudev-dev pkg-config libgtk-3-dev
sudo apt-get install libusb-1.0-0-dev pkg-config
sudo apt-get install libglfw3-dev libgl1-mesa-dev libglu1-mesa-dev
sudo apt-get install libssl-dev
```

Make non-root user access:

```
sudo cp config/99-realsense-libusb.rules /etc/udev/rules.d/
sudo udevadm control --reload-rules & udevadm trigger
```

Compile and build:

```
$ mkdir build
$ cd build
$ cmake ../ -DFORCE_RSUSB_BACKEND=true -DCMAKE_BUILD_TYPE=release
        -DBUILD_EXAMPLES=true -DBUILD_GRAPHICAL_EXAMPLES=true
$ make
$ sudo make install
```

You can check librealsense2 installation by running the below code and connecting the camera with the system (Figure 12.1).

```
cd Release
./rs-capture
```

or

```
$ LIBGL_ALWAYS_SOFTWARE=1 realsense-viewer
```

12.2 Install realsense_ros

- Install `Realsense-ros` packages over ROS distribution:

You can install realsense2_camera and the `realsense2_description` package using standard `sudo` command-line tools as shown below.

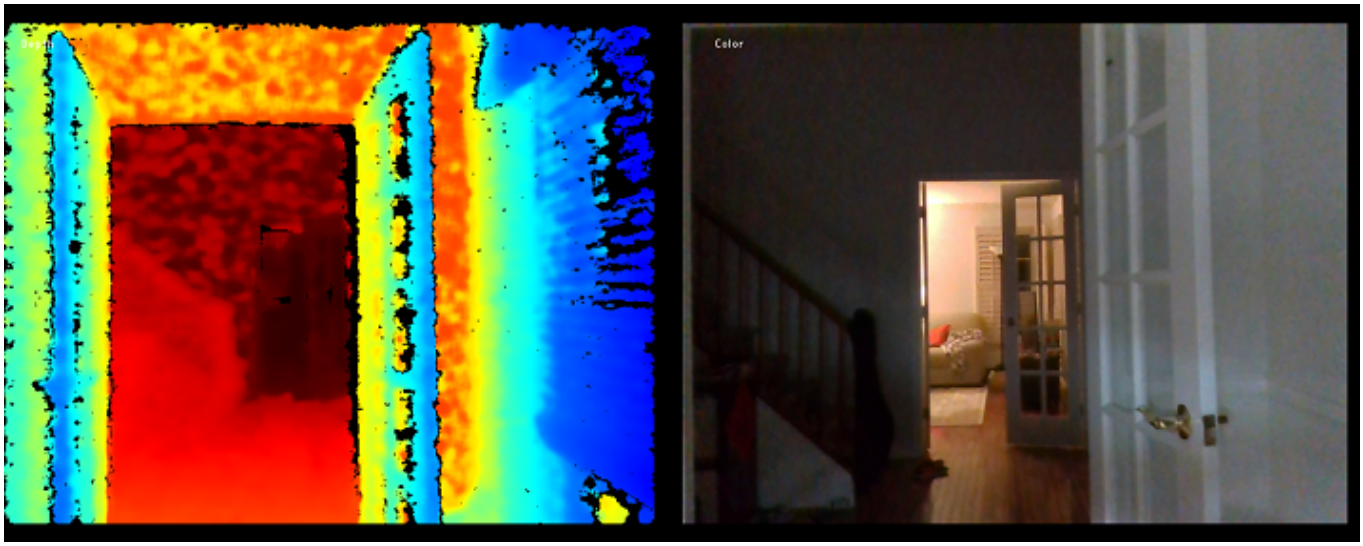

Figure 12.2: RealSense Viewer snapshot

```
# Example of ROS "melodic" version
$ sudo apt-get install ros-melodic-realsense2-camera
$ sudo apt-get install ros-melodic-realsense2-description
```

Remove the previous installed packages.

```
sudo apt-get --purge remove
sudo apt-get --purge remove ros-melodic-realsense2-camera
sudo apt-get --purge remove ros-melodic-librealsense2
sudo apt-get --purge remove ros-melodic-realsense2-description
```

- Build realsense-ros packages from Source Code:

Alternatively, you can install and build these ROS packages from their source code using `catkin_make` . The latest ROS(1) wrapper is version 2.3.2. To follow this method, it is necessary to have the librealsense2 SDK libraries already installed.

We'll clone `Realsense` 2.3.2 and build it from the source code including installing all the dependencies.

Navigate to src directory of your workspace and clone `RealSense` package and build:

```
$ cd ~/projects/ros_ws/src
$ git clone -b ros1-legacy https://github.com/IntelRealSense/realsense-ros.git
$ git checkout `git tag | sort -V | grep -P "^2.\ d+\.\ d+" | tail -1`
$ git clone https://github.com/pal-robotics/ddynamic_reconfigure.git
```

Compile and build

```
$ catkin_make clean
$ catkin_make
```

- Verify Installation:

To verify our packages, after installation, you can run `rospack list`, and you would find `realsense2_camera` and `realsense2_description` among the other full list.

```
$ rospack list
-realsense2_camera ~/catkin_ws/src/realsense-ros/realsense2_camera
-realsense2_description ~/catkin_ws/src/realsense-ros/realsense2_description
```

You can verify installation by launching *rs_camera.launch* as below after connecting your device to the system and checking all the topics it is publishing.

```
$ roslaunch realsense2_camera rs_camera.launch
$ rostopic list
```

You would find below and many more published topics according to your device.

```
peitao@peitao-Latitude-E5500:~$ rostopic list
/camera/color/camera_info
/camera/color/image_raw
/camera/color/image_raw/compressed
/camera/color/image_raw/compressed/parameter_descriptions
/camera/color/image_raw/compressed/parameter_updates
/camera/color/image_raw/compressedDepth
/camera/color/image_raw/compressedDepth/parameter_descriptions
/camera/color/image_raw/compressedDepth/parameter_updates
/camera/color/image_raw/theora
/camera/color/image_raw/theora/parameter_descriptions
/camera/color/image_raw/theora/parameter_updates
/camera/color/metadata
/camera/depth/camera_info
/camera/depth/image_rect_raw
/camera/depth/image_rect_raw/compressed
/camera/depth/image_rect_raw/compressed/parameter_descriptions
/camera/depth/image_rect_raw/compressed/parameter_updates
/camera/depth/image_rect_raw/compressedDepth
/camera/depth/image_rect_raw/compressedDepth/parameter_descriptions
/camera/depth/image_rect_raw/compressedDepth/parameter_updates
/camera/depth/image_rect_raw/theora
/camera/depth/image_rect_raw/theora/parameter_descriptions
/camera/depth/image_rect_raw/theora/parameter_updates
/camera/depth/metadata
/camera/extrinsics/depth_to_color
/camera/realsense2_camera_manager/bond
/camera/rgb_camera/auto_exposure_roi/parameter_descriptions
/camera/rgb_camera/auto_exposure_roi/parameter_updates
/camera/rgb_camera/parameter_descriptions
/camera/rgb_camera/parameter_updates
/camera/stereo_module/auto_exposure_roi/parameter_descriptions
/camera/stereo_module/auto_exposure_roi/parameter_updates
/camera/stereo_module/parameter_descriptions
/camera/stereo_module/parameter_updates
/diagnostics
/rosout
/rosout_agg
/tf
/tf_static
```

With the RViz tool, you can see 3D point cloud data virtualization by deep camera (Figure 12.3):

To view image only as Figureref{chap12_rqt_image_view}:

```
$ roslaunch realsense2_camera demo_pointcloud.launch
$ rqt_image_view
```

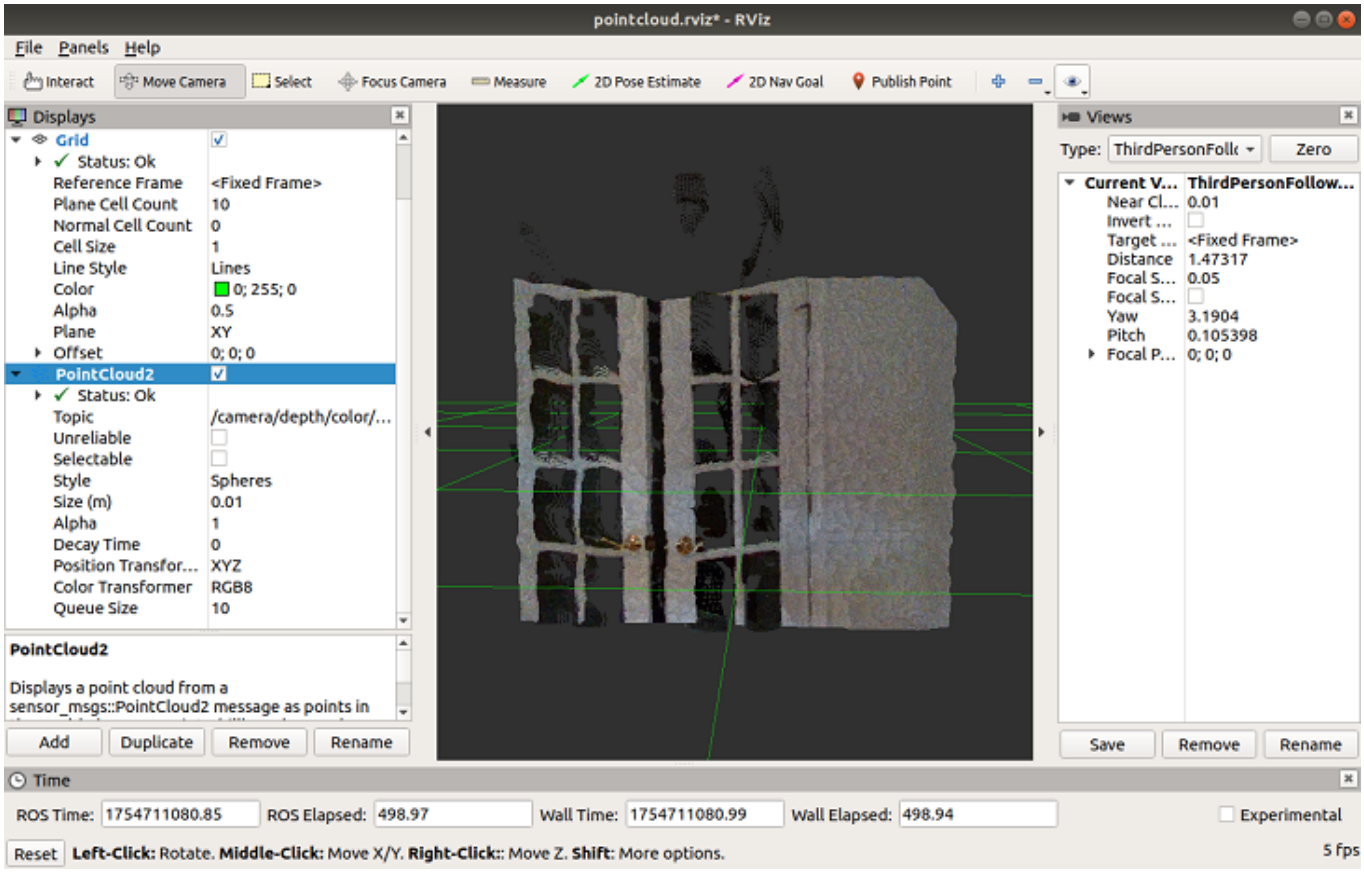

Figure 12.3: Point Cloud data captured in RViz

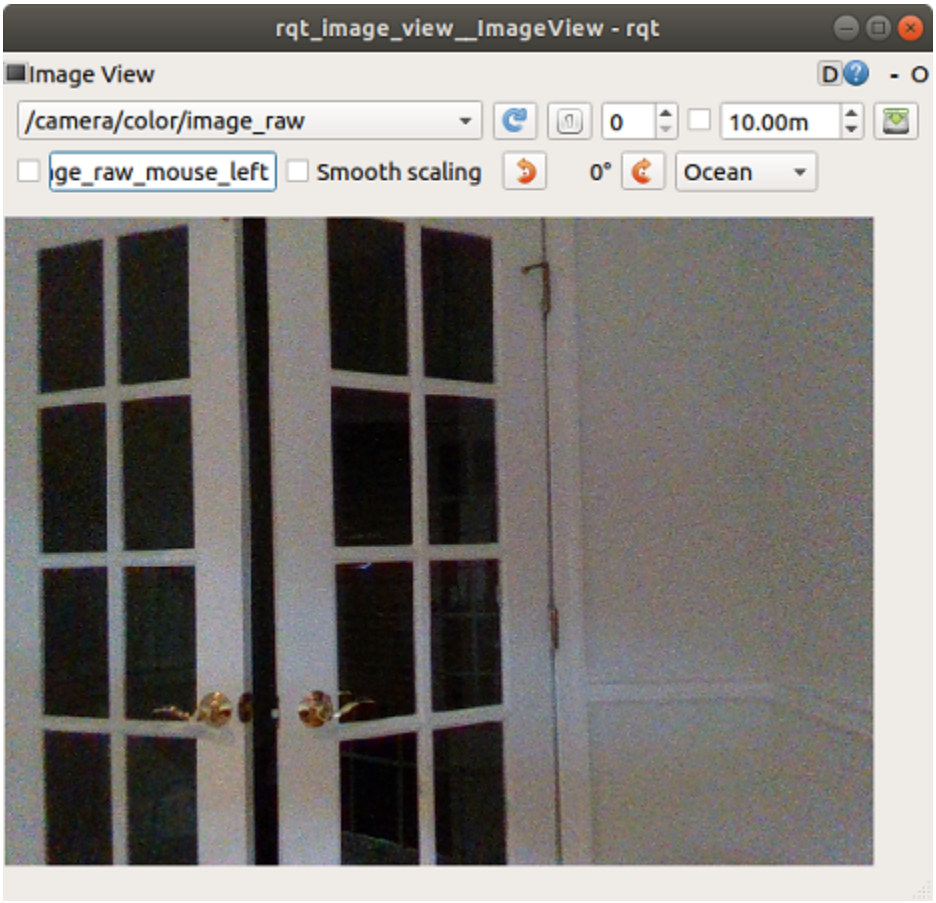

Figure 12.4: Rqt image view

If you've followed up till now, congratulations, you're good to start with your journey with the Intel RealSense Device with ROS. You can learn more by following these ROS examples. If you're facing any issue, please get in touch to find any possible solutions.

12.3 Using realsense2_ros Node

If we open *demo_pointcloud.launch*, we can find all realsens2_ros launch files include a common launch file: *nodelet.launch.xml*

```
<launch>
  <arg name="external_manager"    default="false"/>
  <arg name="manager"             default="realsense2_camera_manager"/>
  <arg name="serial_no"           default=""/>
  <arg name="usb_port_id"         default=""/>
  <arg name="device_type"         default=""/>
  ...
  <node unless="$(arg external_manager)" pkg="nodelet" type="nodelet"
       name="$(arg manager)" args="manager" output="$(arg output)"
       required="$(arg required)" respawn="$(arg respawn)"/>
  <node pkg="nodelet" type="nodelet" name="realsense2_camera"
       args="load realsense2_camera/RealSenseNodeFactory $(arg manager)"
       required="$(arg required)" respawn="$(arg respawn)">
    ...
    <param name="enable_pointcloud"
         type="bool" value="$(arg enable_pointcloud)"/>
    <param name="pointcloud_texture_stream"
         type="str" value="$(arg pointcloud_texture_stream)"/>
    <param name="pointcloud_texture_index"
          type="int" value="$(arg pointcloud_texture_index)"/>
    <param name="allow_no_texture_points"  type="bool"
           value="$(arg allow_no_texture_points)"/>
    <param name="ordered_pc" type="bool" value="$(arg ordered_pc)"/>
  </node>
</launch>
```

You will see that two nodes having the same `nodelet` type are loaded with two different arguments. What is `nodelet` ?

A nodelet is a package for fast communication based on a shared memory heap. It is designed to provide a way to run multiple algorithms in the same process with zero-copy transport between algorithms. This package provides both the `nodelet` base class needed for implementing a `nodelet` , as well as the `NodeletLoader` class used for instantiating `nodelets` .

The first `nodelet` in the *nodelet.launch.xml*, whose args="manager" indicates that the `nodelet` executable from the `nodelet` package is run as `nodelet` manager. A `nodelet` manager is a special type of node that provides a runtime environment for other `nodelets` to be loaded into. It handles the loading and unloading of `nodelets` and facilitates efficient, zero-copy communication between `nodelets` running within the same manager process.

The second `nodelet` with args="load realsense2_camera/RealSenseNodeFactory $(arg manager)" instructs the `nodelet` manager to load `<nodelet_package>/<NodeletClassName>` into the running nodelet_manager. Where `nodelet_package` is `realsens2_camera` and `NodeletClassName` is `RealSenseNodeFactory` class (abstract class), the real class instances are created during loading (RTTI) based on camera id (serial number and product Id). If we open the *realsense_node_factory.cpp* file

```
void RealSenseNodeFactory::StartDevice() {
 if (_realSenseNode) _realSenseNode.reset();
 try {
    ros::NodeHandle nh = getNodeHandle();
    ros::NodeHandle privateNh = getPrivateNodeHandle();
    // TODO
    std::string pid_str(_device.get_info(RS2_CAMERA_INFO_PRODUCT_ID));
    uint16_t pid = std::stoi(pid_str, 0, 16);
    switch(pid) {
      case RS430_PID:
      case RS430_MM_PID:
      case RS430_MM_RGB_PID:
      case RS435_RGB_PID:
      case RS435i_RGB_PID:
       _realSenseNode = std::shared_ptr<BaseRealSenseNode>
            (new BaseRealSenseNode(nh, privateNh, _device, _serial_no));
       break;
    }
  }
```

We can see that an `BaseRealSenseNode` object is created from the heap and based on the camera id (serial number and product ID) and the smart pointer `_realSenseNode` points to the new allocated object.

12.3.1 How To Write Nodelet Plugins

The `nodelet` plugin has a similar coding style to the robot `controller_manager` plugin that we have discussed before. Both are implemented as plugins using the `pluginlib` library. This means you write your `nodelet` as a C++ class that inherits from nodelet::Nodelet, and then use `PLUGINLIB_EXPORT_CLASS` to expose this class as a plugin. This allows the `nodelet` manager to dynamically load and instantiate your `nodelet` at runtime. 3 basic guidelines when implementing your nodelet.

1. Nodelet class must be derived from `nodelet::Nodelet` .
2. In your subclass nodelet, `onInit` has to be overloaded.
3. At the end of file `PLUGINLIB_EXPORT_CLASS` \(`yourpackage::YourNodeleteClass` , `nodelete::Nodelete`)

A `nodelet` example from *http://github.com/ros/common_tutorials*:

```
 #include <pluginlib/class_list_macros.h>
#include <nodelet/nodelet.h>
#include <ros/ros.h>
#include <std_msgs/Float64.h>
#include <stdio.h>

#include <math.h> //fabs

namespace nodelet_tutorial_math
{
class Plus : public nodelet::Nodelet
{
public:
  Plus()
  : value_(0)
  {}

private:
  virtual void onInit() {
    ros::NodeHandle& private_nh = getPrivateNodeHandle();
    private_nh.getParam("value", value_);
    pub = private_nh.advertise<std_msgs::Float64>("out", 10);
    sub = private_nh.subscribe("in", 10, &Plus::callback, this);
  }

  void callback(const std_msgs::Float64::ConstPtr& input) {
    std_msgs::Float64Ptr output(new std_msgs::Float64());
    output->data = input->data + value_;
    NODELET_DEBUG("Adding %f to get %f", value_, output->data);
    pub.publish(output);
  }

  ros::Publisher pub;
  ros::Subscriber sub;
  double value_;
};

PLUGINLIB_EXPORT_CLASS(nodelet_tutorial_math::Plus, nodelet::Nodelet)
}
```

- Example of `Nodelet` Development Workflow

1. Define your `Nodelet` Class: Create a C++ class that inherits from nodelet::Nodelet and implement its `onInit()` method for initialization logic.
2. Export the Class: Use `PLUGINLIB_EXPORT_CLASS` in the *.cpp* file to declare your class as a `nodelet` plugin.
3. Define the plugin in an XML file: Create an XML file (e.g., *nodelets.xml*) that defines the library containing your `nodelet` and provides metadata for `Pluginlib` to find it.
4. Configure *package.xml*: Add dependencies on `nodelet` and `Pluginlib` , and export the plugin

XML file using the `<nodelet>` tag.
5. Build with CMake: Configure *CMakeLists.txt* to build your `nodelet` as a shared library and link against necessary libraries.
6. Launch the `Nodelet` : Use a launch file to start a `nodelet` manager and load your `nodelet` into it.

By leveraging `nodelets` and `Pluginlib` , you can develop modular and efficient ROS applications, particularly beneficial for data-intensive robotic tasks.

12.4 Integrate RealSense D435 camera

12.4.1 D435 frames definition

The `D435` camera has two different Coordination System (CS) definitions (Figure 12.5). One is for ROS (X: Forward, Y: Left, Z: Up) and another is for the camera optical coordinate system (X: Right, Y: Down, Z: Forward).

In RealSense cameras, the origin point (0,0,0) is taken from the left IR (infra1) position and named as `camera_link_frame` , the left IR and `camera_link` coordinates provide static TFs between each sensor coordinate to the camera base (camera_link).

- Camera link Frame: As for the camera itself, it has its base frame named `camera_link` frame. `Camera_link` is defined according to point of view:
  - Imagine we are standing behind the camera, and looking forward.
  - Always use this point of view when talking about coordinates, left vs right IRs, position of sensor, etc.
  - Camera_link coordinator origin is located at the center of the left IR sensor.

- Depth sensor Frames: Although there are two IR sensors, only one camera depth frame is defined on left IRs and has the same origin as the base link. An extra optical sub-frame is defined for the optical system, i.e.,

$$RPY(-90, 0, -90) = R_z(-90)R_y(0)R_x(-90)$$

- TF from camera_link to camera color frames: In the context of sensor fusion, it often refers to the extrinsic calibration process of determining the spatial relation between two sensors and is called "Extrinsic from sensor A to Sensor B". For our particular case, it means TF from IR 1 depth sensor to the color sensor. If we look from behind the D435i extrinsic from depth to color, it means where is the depth in relative to the color. If we just look at the X coordinates, in the optical coordinates (again, from behind) and assume that **color**(RGB) sensor is (0,0,0), we can say that **depth** sensor is on the right of RGB by 0.0148m (+1.48cm) (Figure 12.6).

- Color sensor frames

Similarly to depth sensors, there are two frames: `camera_color_frame` and `camera_color_optical_frame` . Please note that the `Realsense` cameras are usually well calibrated (intrinsic parameters) when purchased. When you start Rviz and load TF frames, you can see the camera color frame and camera optical parameters are calibrated values close to the specified pose.

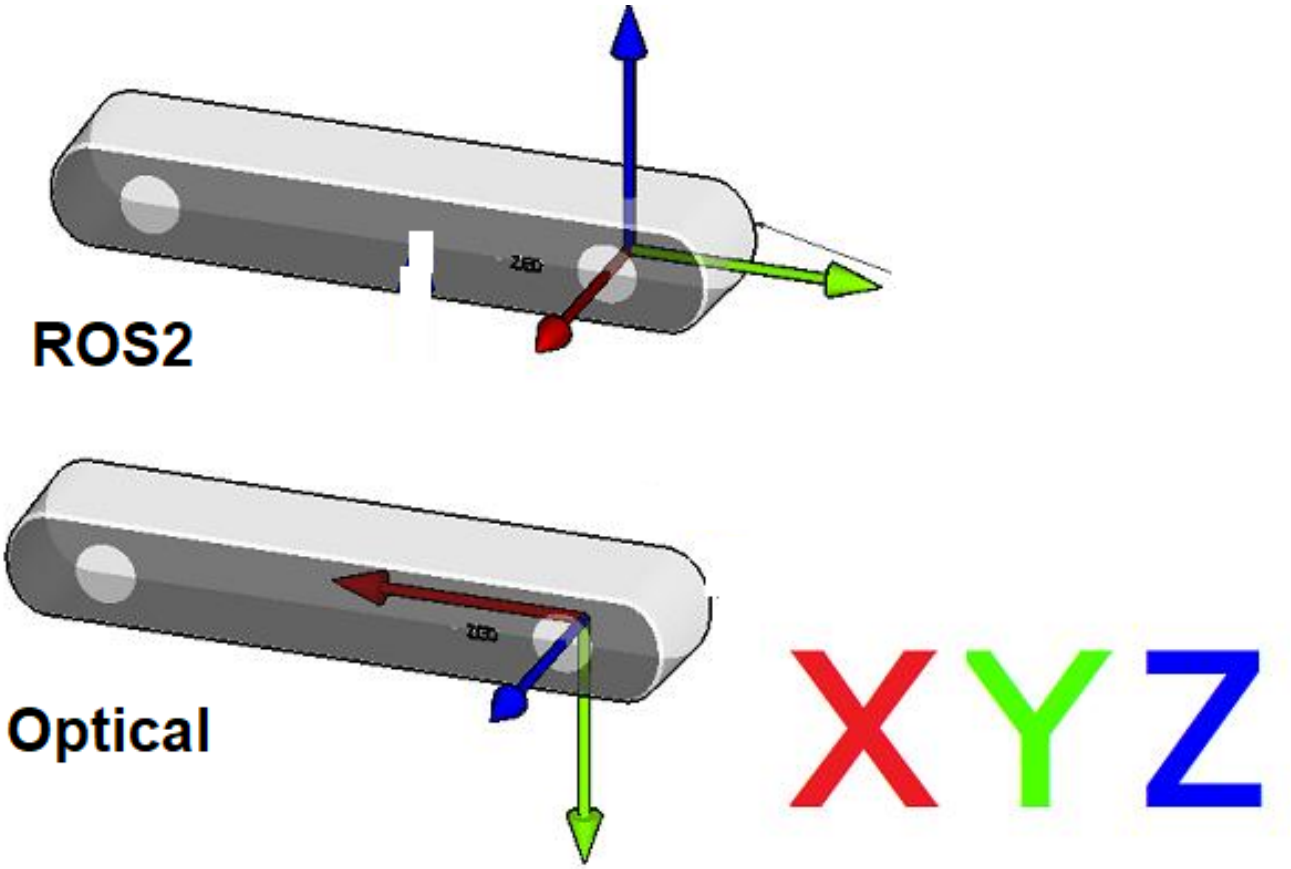

Figure 12.5: D435 ROS vs. Optical Frames

Figure 12.6: Distance between Depth and RGB sensor

All data published in our wrapper topics is optical data taken directly from our camera sensors. Static and dynamic TF can be used to convert between the camera optical CS or ROS CS.

In the last example, when running the last launch file *demo_pointcloud.launch*, static TFs of RGB sensor and Infra2 (right infra) sensor of D435i module can been shown in RViz. (by adding TF frames); it provides TFs from each sensor's ROS coordinates to its corresponding optical coordinates.

```
Frames:
    camera_link:
        position: 0;0;0
        orientation: 0; 0; 0; 1
    camera_color_frame:
        Parent: camea_link
        Position: -000064296; 0.014578; 0.0000140557
        Orientation: 0.00407381; 0.00151235; 0.00488893; 0.9999979
    Camera_color_optical_frame:
        Parent: camera_color_frame
        Position: -0.0064296; 0.0145778; 0.000140557
        Orientation: -0.501153; 0.500338; -0.49452; 0.50315
    Camera_depth_frame:
        Parent: camera_link
        Position: 0; 0; 0;
        Orientation: 0; 0; 0; 1
    Camera_depth_optical_frame:
        Parent: Camera_depth_frame
        Position: 0.0.0
        Orientation: -0.5; 0.5; -0.5; 0.5
```

If we open a new ROS terminal and input the following command:

```
$ rosrun tf view_frames
```

The resultant frame is depicted in Figure 12.7:

12.4.2 Mount D435 Camera

Next, we need to mount the `DS435` camera on the top of the torso (on the horizontal extruded bar). Frames assignments are shown in Figure 12.8. The position where the 3D depth sensor is mounted is at a location where it can cover almost the entire work area. A fixed, elevated, or overhead mount is often ideal to minimize obstruction by the robot arms or horizontal bar.

For the dual-arm system, we combine two arm base frames into one and move it to the bottom of the torso.

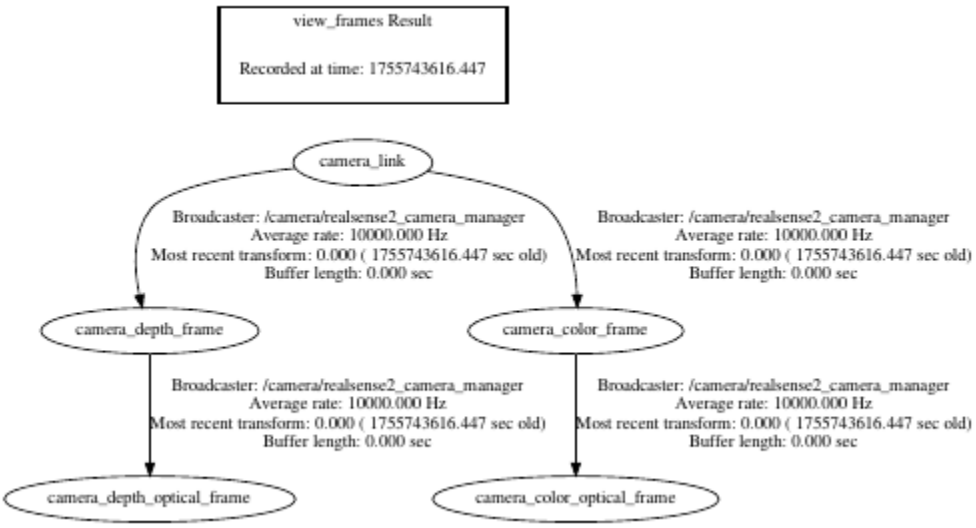

Figure 12.7: D435 frames tree

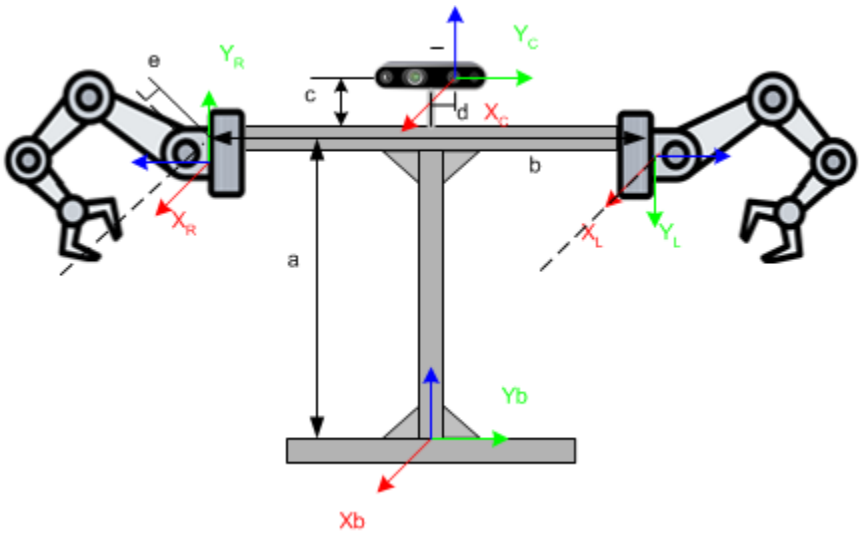

Figure 12.8: Dual arm and D435 camera frames assignment

```
Frames:
    Robot base Frame:
        position: [0;0;0]
        orientation: 0; 0; 0; 1
    Left arm link frame (coordination coincide with left arm Joint 1)
        parent frame: robot base frame
        position: [e; -b/2; a];
        orientation: rpy="1.57/2, 0, 0"
    Right arm link frame(coordination coincide with left arm Joint 1)
        parent frame: robot base frame
        position:[ e; b/2; a]
        orientation: rpy="-1.57/2, 0,0"
    camera link frame
        Parent frame: robot base frame
        position: [0; d; a+c];
        orientation: rpy="0, 0, 0"
```

For the left arm and right arm frames, the new robot base frame information can be updated in their URDF files, such as:

- Left arm URDF file : *hex7bot_left_arm.urdf.xacro*

```
<!-- World frame definition -->
 <link name="world"/>
  <joint name="joint_base" type="fixed">
       <parent link="world" />
       <child link="base_link" />
       <origin xyz="${dh_a0_offset} ${T_bar_horz_b/2} ${base_height}"
          rpy="-1.57075 0.3 0.2" />
       <axis xyz="0 0 1" />
  </joint>
```

- Right arm URDF file: *hex7bot_right_arm.urdf.xacro*

```
  <joint name="joint_base" type="fixed">
       <parent link="base" />
       <child link="base_link" />
       <origin xyz="${dh_a0_offset} ${-T_bar_horz_b/2}
           ${base_height}"/> rpy="1.57075 0 0" />
       <axis xyz="0 0 1" />
 </joint>
```

- Robot main URDF file including the RealSense camera's description.

```
<!-- In your robot's main URDF file (e.g., robot.urdf.xacro) -->
<xacro:include filename="$(find realsense2_description)
            /urdf/realsense_d435.urdf.xacro" />

  <!-- The world frame is a static reference point  -->
  <link name="world" />
  <!-- Define the transformation from the 'world' frame
         to the camera's parent frame -->
  <xacro::sensor_d435 name="camera" parent="world"
      use_nominal_extrinsics="true">
     <origin xyz="0.0 0.018 0.435" rpy= "0 0 0" />
  </xacro:sensor_d435>
```

Where `use_nominal_extrinsics="true"` : This parameter is crucial as it ensures all the necessary camera frames (like `camera_depth_optical_frame` , `camera_color_optical_frame` , etc.) and their relative transforms are included in the URDF file, which is often required for ROS functionality like point clouds.

We also need to ensure that the left and right arm ROS launch files are run on their host machine and the master launch file is run on the master ROS machine that connects to the RealSense camera and includes the robot_state_publisher node, which reads your URDF and publishes the `tf` (Transform) frames, including the relationship between base_link and camera_link.

```
<node name="robot_state_publisher" pkg="robot_state_publisher"
               type="robot_state_publisher" />
```

12.5 Target 3D Localization

Stereo cameras, such as the RealSense D435, work similarly to human eyes for depth perception. Our brain calculates the disparity between the images from each eye: objects closer to us appear to shift more, while distant objects appear to shift very little. The Realsense D435 captures two images from its two sensors and compares them. Because the distance between the sensors(the baseline) is know, these comparisons yield depth information. The measurable depth range is directly related to the baseline width - the wider the baseline, the farther the camera can see.

Unlike color or structured light cameras, RealSense D435 can measure depth using infrared light and work well in most lighting conditions, including outdoor environments, as they do not rely on any specific visual feature for depth perception.

With aligned depth and color information, the RealSense D435 camera can be used to detect or recognize objects: first detect an object/point in a 2D RGB image, then determine its 3D position, size, and distance (especially depth) information. It is a very useful feature in robot arm end-effector pose calibration. If we can attach one light source (e.g., tiny LED) on the robot's TCP, control the robot arm in different poses, and calculate and compare TCP poses in the robot base frame and camera frames for more precise localization. Particularly, we can train our dual arm with the RealSense camera by sampling possible end-effector positions and generating the dataset for deep learning, which will be discussed in the next chapter, and control robot arms in real time without calculating inverse kinematics.

To determine the position of a point in 3D space on the camera sensor, a projection matrix mapping from a 2D image position to 3D space needs to be calculated. The visual below shows the projection of a 3D point

P on the sensor using a `pinhole camera model` (Figure 12.9).

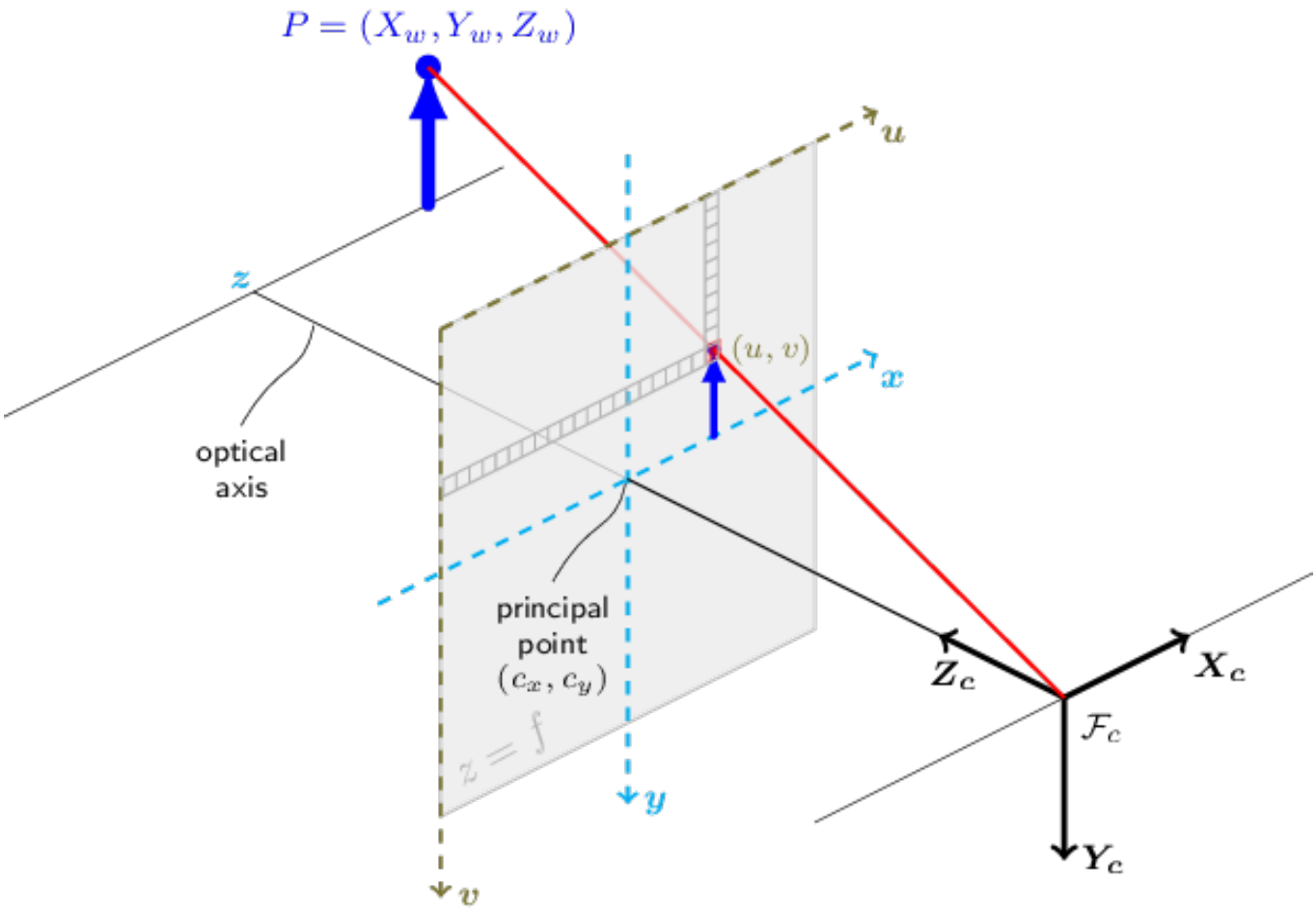

Figure 12.9: Pinhole Camera Model

This process can be mathematically modeled as shown in following equation.

$$\underbrace{\begin{bmatrix} u \\ v \\ 1 \end{bmatrix}}_{\text{2D pixel coordinate}} = \underbrace{\mathbf{P}}_{\text{transformation}} \underbrace{\begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}}_{\text{3D world coordinate}} \tag{12.5.1}$$

Where $P(X_w, Y_w, Z_w)$ is a 3D point expressed with respected to the world coordinator system, $p(u, v)$ is a 2D pixel in the image plane as shown in the figure.

P is a transform matrix (also called `projection matrix`) from the world coordinator to the image plane and can be divided into two parts: first, transform the same 3D position from the world coordinate to the

camera coordinate through translation and rotation:

$$\underbrace{\begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix}}_{\text{camera coordinate}} = \underbrace{\begin{bmatrix} R|t \end{bmatrix}}_{\text{extrinsic matrix}} \underbrace{\begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}}_{\text{3D world coordinator}} \tag{12.5.2}$$

This `extrinsic matrix` represents the position of the camera in the world coordinates. In other words, it describes the position of the camera with respect to the world coordinates through translation and rotation.

The second part is intrinsic transformation from the camera frame to the image plane.

$$\underbrace{\begin{bmatrix} u \\ v \\ 1 \end{bmatrix}}_{\text{2D pixel coordinate}} = \underbrace{\begin{bmatrix} 1 & 0 & c_x \\ 0 & 1 & c_y \\ 0 & 0 & 1 \end{bmatrix}}_{\text{2D Translation}} \times \underbrace{\begin{bmatrix} f_x & 0 & 0 \\ 0 & f_y & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{2D Scaling}} \times \underbrace{\begin{bmatrix} 1 & s/f_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{2D Shear}} \underbrace{\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix}}_{\text{camera coordinator}} \tag{12.5.3}$$

Where (c_x, c_y) is `principal point` offset, it is usually close to the image center. f_x and f_y are `focal lengths` on x and y axes which are expressed in pixel units.

$$\underbrace{\mathbf{K}}_{\text{Intrinsic Matrix}} = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{12.5.4}$$

The `intrinsic matrix` contains the properties of the camera sensor (focal length, principal point offset, and lens distortion) and can be found through camera calibration.

We can conclude that project matrix $\mathbf{P}$:

$$\mathbf{P} = \underbrace{\mathbf{K}}_{\text{Intrinsic Matrix}} \times \underbrace{\begin{bmatrix} R|t \end{bmatrix}}_{\text{Extrinsic Matrix}} \tag{12.5.5}$$

In a scenario where the camera origin and the world origin are perfectly aligned, and the camera sensor is centered, we no longer need the R and t matrices. Thus, simplifying this equation, we can write it as the perspective projection model shown in the homogeneous transformation matrix:

Here, u, v represent pixel coordinates, and x, y, z represent a point in the world coordinators.

$$\begin{bmatrix} u \\ v \\ 1 \\ 1/z \end{bmatrix} = \frac{1}{z} \begin{bmatrix} f_x & 0 & c_x & 0 \\ 0 & f_y & c_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{12.5.6}$$

And we can get mapping from 3D position to 2D image pixel with:

$$u = \frac{f_x x}{z} + c_x \tag{12.5.7}$$

$$v = \frac{f_y y}{z} + c_y \tag{12.5.8}$$

To find a point's 3D position from a 2D RGB coordinate (`deprojection`):

$$x = \frac{z(u - c_x)}{f_x} \tag{12.5.9}$$

$$y = \frac{z(v - c_y)}{f_y} \tag{12.5.10}$$

$$z = z \text{ :measured depth} \tag{12.5.11}$$

To find a point's 3D position from a 2D RGB coordinate on a Realsense D435, we need to make sure depth and color streams are aligned. That is also to say: first get the corresponding synchronized 2D (x, y) pixel and its depth value for that pixel, then use the depth and the RGB camera's intrinsic parameters (focal length and principal point) to project the 2D pixel into 3D space.

Let's first find one tiny light source source, such as LilyPad LED pad() which can be attached to TCP of one robot arm.

- First, install ros image geometry package

```
$ sudo apt-get install ros-melodic-image-geometry
```

- Launch the RealSense ROS node with a launch file *rs_aligned_depth.launch* to enable aligned streams with depth aligned to color images. The *rs_aligned_depth.launch* file is part of the `realsense2_camera` package. Prior to running this launch file, make sure the following boolean parameters are true to enable RGBD messages:
- `enable_rgbd` : new parameter, to enable/disable RGBD topic, changeable during runtime.
- `align_depth.enable` : align depth images to RGB images.
- `enable_sync` : let librealsense sync between frames, and get the frame set with color and depth images combined.
- `enable_color + enable_depth` : enable both color and depth sensors.

```
$ roslaunch realsense2_camera rs_algined_depth.launch
```

Running this launch with the above settings will create color and depth (RGBD) images/messages that are synchronized by frame time tag.

```
/camera/color/image_raw
/camera/aligned_depth_to_color/image_raw
```

If you could not see the aligned topic over "rostopic echo" or RViz tool, you may need to tweak other parameters, such as reducing 'depth_fps" or "color_fps" from "30" to "15" so that your computer can process them.

```
  <arg name="depth_fps"           default="15"/>
  <arg name="infra_fps"           default="15"/>
  <arg name="color_fps"           default="15"/>
```

- Develop a standard ROS node that subscribes to two aligned streams : one callback function is to locate the interested light source (e.g., red LED pixel location in the color image), and second callback function calculates its 3D position based on its depth image information. Here is the sample code:

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <sensor_msgs/Image.h>
#include <sensor_msgs/CameraInfo.h>
//need install sudo apt-get install
// ros-<your_ros_version>-image-geometry
#include <image_geometry/pinhole_camera_model.h>
#include <opencv2/highgui/highgui.hpp>

const std::string OPENCV_WINDOW = "LED Image Window";

class rs2PointDepthLocator {
public:
   rs2PointDepthLocator() : nh_("~"), it_(nh_) , best_pixel_(-1,-1){
       // Subscribe to aligned depth and camera info topics
       image_sub_ = it_.subscribe("/camera/aligned_depth_to_color/image_raw", 1,
               &rs2PointDepthLocator::depthImageCallback, this);
       info_sub_ = nh_.subscribe("/camera/aligned_depth_to_color/camera_info", 1,
               &rs2PointDepthLocator::infoCallback, this);

       color_sub_ = nh_.subscribe("/camera/color/image_raw", 1,
               &rs2PointDepthLocator::colorImageCallback, this);
   }

   void infoCallback(const sensor_msgs::CameraInfoConstPtr& info_msg) {
       // Update the camera model when info is received
       cam_model_.fromCameraInfo(info_msg);
       ROS_INFO_ONCE("Camera model updated.");
   }
```

- Detect Red LED on 2D RGB images

```
void colorImageCallback(const sensor_msgs::ImageConstPtr& msg) {
 try{
    // Convert the ROS image message to an OpenCV Mat.
    // `cv_bridge` will perform a color conversion if the image is not in BGR8.
    cv_bridge::CvImagePtr cv_ptr =
           cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
    cv::Mat image = cv_ptr->image;

    // Ensure the image was successfully converted.
    if (image.empty()){
        ROS_ERROR("Failed to convert image message to OpenCV format.");
        return;
    }

    // Define a threshold for what is considered "red."
    // Red pixels have a high Red value and low Green and Blue values.
    // Adjust these values as needed for your specific image.
    const int RED_THRESHOLD = 200;
    const int GREEN_THRESHOLD = 50;
    const int BLUE_THRESHOLD = 50;

    // Loop through each pixel in the image.
    for (int y = 0; y < image.rows; ++y){
      // Access the pointer to the current row for faster iteration.
      const cv::Vec3b* pixel_row = image.ptr<cv::Vec3b>(y);

      for (int x = 0; x < image.cols; ++x){
         // `Vec3b` stores BGR, so index 0 is Blue, 1 is Green, 2 is Red.
         int blue = pixel_row[x][0];
         int green = pixel_row[x][1];
         int red = pixel_row[x][2];

         // Check if the pixel meets the "red" criteria.
         if (red > RED_THRESHOLD && green < GREEN_THRESHOLD
              && blue < BLUE_THRESHOLD){
             best_pixel_.x = x;
             best_pixel_.y = y;
             ROS_INFO("Found red pixel at coordinates (%d, %d)", x, y);
             return;
         }
      }
    }
   ROS_INFO("No red pixel found in this image.");
  }
 catch (const cv_bridge::Exception& e){
     ROS_ERROR("cv_bridge exception: %s", e.what());
 }
}
```

```
void depthImageCallback(const sensor_msgs::ImageConstPtr& image_msg) {
  if (!cam_model_.initialized()) {
      ROS_WARN_ONCE("Waiting for camera info...");
      return;
  }
  if (best_pixel_.x < 0 || best_pixel_.y < 0){
      ROS_WARN_ONCE("waiting for valid pixel...");
      return;
  }

  try {
     cv_bridge::CvImagePtr cv_ptr;
     // 16UC1 for Z16 encoding
     cv_ptr = cv_bridge::toCvCopy(image_msg,
         sensor_msgs::image_encodings::TYPE_16UC1);
     // Get depth value for the pixel
     uint16_t depth_raw = cv_ptr->image.at<uint16_t>
             (best_pixel_.y, best_pixel_.x);
     if (depth_raw == 0) {
         ROS_WARN("Invalid depth value (0) at pixel (%d, %d).",
                 best_pixel_.x, best_pixel_.y);
         return;
     }

     // The depth is usually in millimeters (Z16 format)
     double depth_meters = static_cast<double>(depth_raw) / 1000.0;

     // Project the 2D pixel to a 3D ray and then scale by depth
     cv::Point2d pixel_point(best_pixel_.x,  best_pixel_.y);
     cv::Point3d ray = cam_model_.projectPixelTo3dRay(pixel_point);
     cv::Point3d point_3d;
     point_3d.x = ray.x * depth_meters;
     point_3d.y = ray.y * depth_meters;
     point_3d.z = ray.z * depth_meters;

     ROS_INFO("3D position of pixel (%d, %d) is: (%.4f, %.4f, %.4f) meters",
              best_pixel_.x, best_pixel_.y, point_3d.x, point_3d.y, point_3d.z);
  } catch (cv_bridge::Exception& e) {
      ROS_ERROR("cv_bridge exception: %s", e.what());
      return;
  }
}
```

```
private:
    ros::NodeHandle nh_;
    image_transport::ImageTransport it_;
    image_transport::Subscriber image_sub_;
    ros::Subscriber info_sub_;
    ros::Subscriber color_sub_;
    image_geometry::PinholeCameraModel cam_model_;

    cv::Point best_pixel_;
};

int main(int argc, char** argv) {
    ros::init(argc, argv, "realsense_depth_locator");
    rs2PointDepthLocator locator;
    ros::spin();
    return 0;
}
```

- Update CMakeList.txt with following libraries:

```
find_package(catkin REQUIRED COMPONENTS
  cv_bridge
  image_geometry
  image_transport
  pcl_msgs
  pcl_ros
  roscpp
  sensor_msgs
)
catkin_package(
  CATKIN_DEPENDS cv_bridge pcl_msgs pcl_ros pcl_visualize ros_cpp sensor_msgs
)

add_executable(rs2Point3DLocator   src/rs2Point3DLocator.cpp)
target_link_libraries(rs2Point3DLocator ${catkin_LIBRARIES} ${PCL_LIBRARIES})
```

- Update manifest file package.xml:

```
 <build_depend>image_transport</build_depend>
  <build_depend>cv_bridge</build_depend>
  <build_depend>pcl_msgs</build_depend>
  <build_depend>pcl_ros</build_depend>
  <build_depend>pcl_visualize</build_depend>

<exec_depend>cv_bridge</exec_depend>
  <exec_depend>image_transport</exec_depend>
  <exec_depend>pcl_msgs</exec_depend>
  <exec_depend>pcl_ros</exec_depend>
```

Build and run your custom node: Compile the C++ file and run it. You will see the 3D coordinates printed periodically as shown in Figure 12.10.

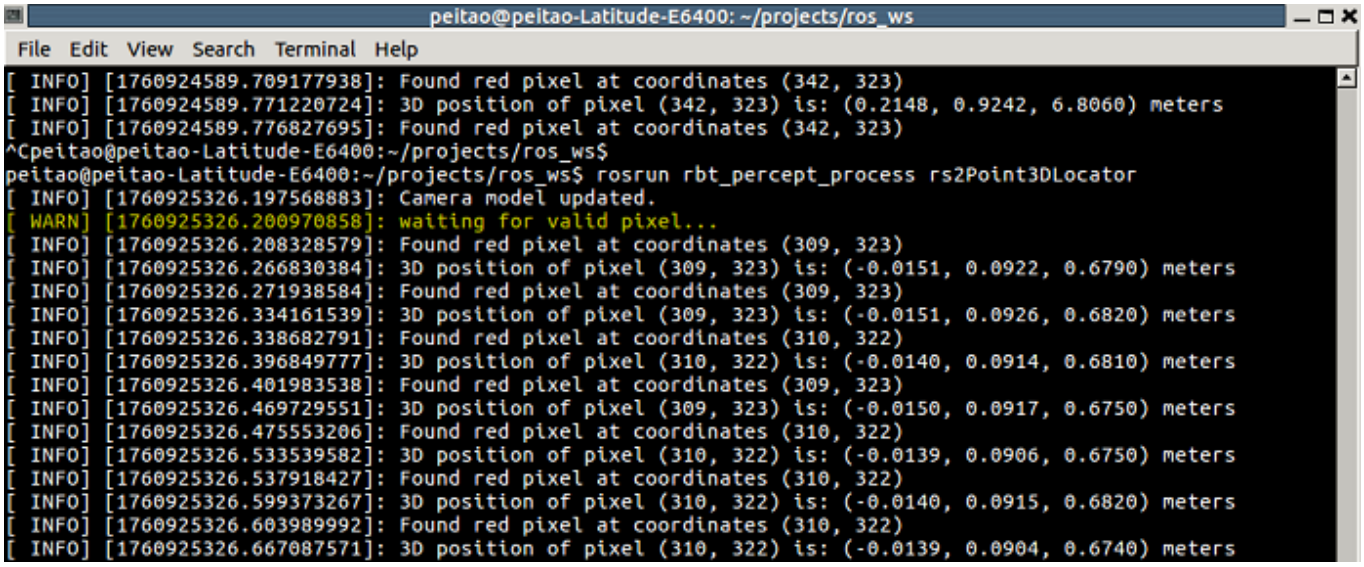

Figure 12.10: 3D localization output

Please note that `projectPixelTo3dRay` converts a 2D pixel in the camera's rectified image frame to a 3D ray that passes through that pixel and the camera's center. After multiplying (scaling) by the depth value at that pixel, the output 3D position (x, y, z) is in the camera's optical coordinate frame. To be used by other ROS nodes or packages, these positions have to be expressed in `camera_link_frame` (ROS) or even the `world` frame. This requires ROS `tf` or `tf2` (frame transformations) nodes load all camera frames including world frames from URDF files, perform the geometric (static for our case) transformation, and transform ray data (represented as a 3D vector or geometry_msgs/PointStamped) by publishing new `PointStamped` messages with new target frame id. We will discuss the detailed implementation in the next chapter.

Considering the red LED is not always displayed as one pixel on the 2D image, the picked red pixel for deprojection may be the edge of the LED source and cause large errors. The red LED detection can be improved with a $3x3$ pixel median filter or by converting the RGB image to HSV for better color detection, such as the following improvement:

```
void colorImageCallback2(const sensor_msgs::ImageConstPtr& msg) {
     cv_bridge::CvImagePtr cv_ptr;
     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;
     }
     // Convert the RBG image to HSV for better color detection
     cv::Mat hsv_image;
     cv::cvtColor(cv_ptr->image, hsv_image, cv::COLOR_BGR2HSV);
     // Define the target red color in HSV (example values)
     // Hue: ~0 or ~180 (wraps around), Saturation: high, Value: high
     // Note: The "best" red depends on your lighting and application.
     cv::Scalar target_red_hsv(0, 255, 255); // A very saturated bright red

     double min_distance = std::numeric_limits<double>::max();
     cv::Point best_pixel_location(-1, -1);

     // Iterate through the image to find the best red pixel
     for (int row = 0; row < hsv_image.rows; ++row) {
         for (int col = 0; col < hsv_image.cols; ++col) {
             // Get the HSV value of the current pixel
             cv::Vec3b current_hsv = hsv_image.at<cv::Vec3b>(row, col);

             // Calculate the Euclidean distance to the target red
             double hue_diff = std::min(std::abs(current_hsv[0] - target_red_hsv[0]),
                       180.0 - std::abs(current_hsv[0] - target_red_hsv[0]));
             double distance = std::sqrt(std::pow(hue_diff, 2) +
                       std::pow(current_hsv[1] - target_red_hsv[1], 2) +
                       std::pow(current_hsv[2] - target_red_hsv[2], 2));

             if (distance < min_distance) {
                 min_distance = distance;
                 best_pixel_.x = col;
                 best_pixel_.y = row;
             }
         }
     }
     // Draw a circle around the best red pixel
     if (best_pixel_location.x != -1) {
         cv::circle(cv_ptr->image, best_pixel_location, 10, CV_RGB(255, 0, 0), 2);
         ROS_INFO("Best red pixel found at: (%d, %d)",
                best_pixel_location.x, best_pixel_location.y);
     }
  }
```

If you need a more direct approach without the full ROS image_geometry abstraction, you can use the librealsense SDK's built-in functions.

```cpp
#include <ros/ros.h>
#include <sensor_msgs/Image.h>
#include <sensor_msgs/CameraInfo.h>
#include <cv_bridge/cv_bridge.h>
#include <librealsense2/rs.hpp>
#include <librealsense2/rs_advanced_mode.hpp>

// Store camera intrinsics and frames
rs2_intrinsics intrinsics;
std::shared_ptr<sensor_msgs::Image const> depth_frame_msg;
bool intrinsics_received = false;

void depth_image_callback(const sensor_msgs::ImageConstPtr& msg) {
    depth_frame_msg = msg;
}

void camera_info_callback(const sensor_msgs::CameraInfoConstPtr& msg) {
   if (intrinsics_received) return;

   // Copy intrinsics from ROS message to RealSense structure
   intrinsics.width = msg->width;
   intrinsics.height = msg->height;
   intrinsics.fx = msg->K[0];
   intrinsics.fy = msg->K[4];
   intrinsics.ppx = msg->K[2];
   intrinsics.ppy = msg->K[5];
   intrinsics.model = RS2_DISTORTION_BROWN_CONRADY; // D435 uses this distortion model

   intrinsics_received = true;
   ROS_INFO_ONCE("Camera intrinsics received.");
}
```

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

   ros::Subscriber depth_sub =
        nh.subscribe("/camera/aligned_depth_to_color/image_raw", 1,
                depth_image_callback);
   ros::Subscriber info_sub =
        nh.subscribe("/camera/aligned_depth_to_color/camera_info", 1,
                camera_info_callback);

   ros::Rate rate(10);
   // Check for new frames at 10 Hz
   while (ros::ok()) {
      ros::spinOnce();

      if (depth_frame_msg && intrinsics_received) {
        int u = 320; // x-coordinate
        int v = 240; // y-coordinate

        try {
           cv_bridge::CvImagePtr cv_ptr;
           cv_ptr = cv_bridge::toCvCopy(depth_frame_msg,
                  sensor_msgs::image_encodings::TYPE_16UC1);
           uint16_t depth_raw = cv_ptr->image.at<uint16_t>(v, u);
           if (depth_raw == 0) {
               ROS_WARN("Invalid depth at pixel (%d, %d).", u, v);
               continue;
           }
           // Get depth scale. In this case, we use 1mm,
           // but it's safer to get it from the camera.
           float depth_scale = 0.001f;
           float depth_in_meters = static_cast<float>(depth_raw) * depth_scale;
           // Use rs2_deproject_pixel_to_point to find 3D position
           float point_3d[3];
           float pixel_2d[2] = {(float)u, (float)v};
           rs2_deproject_pixel_to_point(point_3d, &intrinsics,
                pixel_2d, depth_in_meters);
           ROS_INFO("3D position of pixel (%d, %d) is: (%.4f, %.4f, %.4f) meters",
                    u, v, point_3d[0], point_3d[1], point_3d[2]);
        } catch (cv_bridge::Exception& e) {
            ROS_ERROR("cv_bridge exception: %s", e.what());
        }
      }
      rate.sleep();
   }
   return 0;
}
```

The `rs2_deproject_pixel_to_point` function from the librealsense SDK is designed for this task, often after aligning the depth stream to the color stream.

12.6 Perception Processing

In the last section, `Realsense D435` (not D435i) ROS nodes provides two kinds of perception info: depth information with point cloud data (`sensor_msgs/PointCloud2`) and color images (`sensor_msgs/Image`)

```
$ rostopic type /camera/depth/color/points
   sensor_msgs/PointCloud2
$ rostopic type /camera/color/image_raw
   sensor_msgs/Image
```

Compared with 2D images, which need `OpenCV` and advanced algorithms for 3D reconstruction, point clouds directly provide 3D environments with sampled space points. The Point Cloud library (`PCL`) on the host machine (PCL 1.8 for Ubuntu 18.04/melodic) provides a number of data types and algorithms to perform data processing on sampled data such as filtering, model estimation, segmentation, surface reconstruction, etc.

ROS provides a message-based interface through which PCL point clouds can be efficiently communicated and a set of conversion functions from native PCL types to ROS messages.

We have learned that `realsense_ros` node provides ROS-based point cloud type (`sensor_msgs/PointCloud2`). Next, we need to convert these messages back to the native PCL data type, call PCL algorithms to get processed data information, and then convert back to a ROS message for our ROS control.

In this section, we learn how to process these data from `realsense2_ros` by extracting important information and then control our arms (actions).

12.6.1 Point Cloud Processing

If you do a full installation on ROS Melodic{ROS!Melodic}, the PCL library 1.8 is by default installed on the system (`libpcl-dev`) and library header path (`/usr/include/pcl-1.8`)

To write a PCL ROS node with the PCL library, using the following commands to create a ROS package first

```
cd ~/projects/ros_ws/src
catkin_create_pkg rbt_percept_process rc2CloudViewer plc_ros roscpp sensor_msgs
```

Then add PCL libraries to new created *CMakefile.txt* so that the executable ROS node can be properly linked against the system's PCL library:

```
find_package(PCL 1.8 REQUIRED COMPONENTS
    common
    visualization
    io
    filters)


## PCL include dir
include_directories(
    include
    ${PCL_INCLUDE_DIRS})

link_directories(${PCL_LIBRARY_DIRS})
```

First we can create `realsense-ros` cloud view which subscribes to point clouds from `realsense-ros` node by the name `/camera/depth/color/points`

The source code: *rs2CloudViewer.cpp* is under the *~/projects/ros_ws/src/rbt_percept_process/src*

```
#include <iostream>
#include <ros/ros.h>
#include <pcl/visualization/cloud_viewer.h>
#include <sensor_msgs/PointCloud2.h>
#include <pcl_conversions/pcl_conversions.h>

class rs2CloudViewer
{
public:
  rs2CloudViewer():
  cloud_viewer("RealSense Point Cloud Viewer") {
     pcl_sub = nh.subscribe("/camera/depth/color/points",
         10, &rs2CloudViewer::cloudCB, this);
     viewer_timer = nh.createTimer(ros::Duration(0.1),
         &rs2CloudViewer::timerCB, this);
  }

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

     cloud_viewer.showCloud(cloud.makeShared());
  }

  void timerCB(const ros::TimerEvent&) {
     if (cloud_viewer.wasStopped()){
         ros::shutdown();
     }
  }
protected:
   ros::NodeHandle nh;
   ros::Subscriber pcl_sub;
   pcl::visualization::CloudViewer cloud_viewer;
   ros::Timer viewer_timer;
};

main(int argc, char **argv) {
   ros::init (argc, argv, "rs2_pcl_visualize");
   rs2CloudViewer rs2_cloudviewer;
   ros::spin();
   return 0;
}
```

and build this file as an execute file by adding the following lines in *CMakefile.txt*

```
add_executable(rs2CloudViewer src/rs2CloudViewer.cpp)
target_link_libraries(rs2CloudViewer ${catkin_LIBRARIES}
            ${PCL_LIBRARIES})
```

Next, we can open the realsense-ros cloud viewer: Open the second ROS terminal when leaving the first

ROS terminal running *demo_pointcloud.launch file*, run the following command:

```
rosrun rbt_percept_process rs2CloudViwer
```

The following window as shown in Figure 12.11 will pop up, zoom out to show all point cloud from Realsense camera.

Figure 12.11: RealSense Cloud viewer

- Filtering and downsampling

The two main issues that we may face when attempting to process point clouds are excessive noise and excessive density. The former causes our algorithms to misinterpret the data and produce incorrect or inaccurate results, while the latter makes our algorithms take a long time to complete their operation. Depending on raw cloud data quality and your specific goal, you can apply `outlier removal` before the voxel grid filter if you need to eliminate noisy and stray points first, or apply the voxel grid filter first if you want fast processing time. We take the voxel grid filter first as our point cloud processing pipeline order.

(1) Downsampling: Apply a `pcl::VoxelGrid` filter to reduce the number of points and create a uniform point density.

```
class rs2CloudDownSampler {
public:
   rs2CloudDownSampler() {
      pcl_sub = nh.subscribe("/camera/depth/color/points", 10,
            &rs2CloudDownSampler::cloudCB, this);
      pcl_pub = nh.advertise<sensor_msgs::PointCloud2>("voxel_grid/output", 1);
   }

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

      pcl::fromROSMsg(input, cloud);

      pcl::VoxelGrid<pcl::PointXYZ> voxelSampler;
      //Sets the input point cloud to be downsampled.
      voxelSampler.setInputCloud(cloud.makeShared());
      //Sets the size of the voxel grid, where x, y,
      // and z are the dimensions of each voxel.
      //A smaller leaf size results in less downsampling.
      voxelSampler.setLeafSize(0.01f, 0.01f, 0.01f);
      voxelSampler.filter(cloud_downsampled);

      pcl::toROSMsg(cloud_downsampled, output);
      pcl_pub.publish(output);
   }

 protected:
    ros::NodeHandle nh;
    ros::Subscriber pcl_sub;
    ros::Publisher pcl_pub;
};
```

(2) `Outlier Removal` : Apply a filter like `pcl::StatisticalOutlierRemoval` on your raw point cloud data to eliminate noisy and stray points.

```
class rs2CloudFilter {
  public:
   rs2CloudFilter() {
      pcl_sub = nh.subscribe("voxel_grid/output", 10,
        &rs2CloudFilter::cloudCB, this);
      pcl_pub = nh.advertise<sensor_msgs::PointCloud2>
        ("statistical_outlier_removal/output", 1);
   }

   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);

      pcl::StatisticalOutlierRemoval<pcl::PointXYZ> statFilter;
      statFilter.setInputCloud(cloud.makeShared());
      statFilter.setMeanK(10);
      statFilter.setStddevMulThresh(0.2);
      statFilter.filter(cloud_filtered);

      pcl::toROSMsg(cloud_filtered, output);
      pcl_pub.publish(output);
   }
  protected:
    ros::NodeHandle nh;
    ros::Subscriber pcl_sub;
    ros::Publisher pcl_pub;
};
```

Further Processing: Proceed with other operations, such as segmentation, feature extraction, or registration, on the now-clean and downsampled data.

(3) Segmentation and feature extraction: `Segmentation` is used to identify and separate individual objects within a scene, such as separating a table from the objects on it. Feature Extraction, or isolating regions of interest to extract features, like finding the surface of a plane for surface analysis. For the robot manipulation, it can be used to determine the precise location and shape of objects for pick-and-place operations. Two typical segmentation types in pcl_ros:

- `Euclidean Clustering` : Groups points that are close to each other in Euclidean space. Sample Consensus Segmentation (`SCS`): Identifies models of geometric primitives (like planes or spheres) within the point cloud, such as `SACSegmentation` . `Cylindrical Segmentation` : Fits cylinders to objects, useful for identifying cylindrical objects like cans or cubes.

```
class rs2CloudSegmentation
{
public:
  rs2CloudSegmentation() {
    pcl_sub = nh.subscribe("statistical_outlier_removal/output", 10,
        &rs2CloudSegmentation::cloudCB, this);
    ind_pub = nh.advertise<pcl_msgs::PointIndices>
        ("/rs2_cloud_process/pcl_segmented_indices", 1);
    coef_pub = nh.advertise<pcl_msgs::ModelCoefficients>
        ("/rs2_cloud_process/pcl_segmented_modecoeffient",1);
  }

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

    pcl::fromROSMsg(input, cloud);

    pcl::ModelCoefficients coefficients;
    pcl::PointIndices::Ptr inliers(new pcl::PointIndices());

    pcl::SACSegmentation<pcl::PointXYZ> segmentation;
    segmentation.setModelType(pcl::SACMODEL_PLANE);
    segmentation.setMethodType(pcl::SAC_RANSAC);
    segmentation.setMaxIterations(1000);
    segmentation.setDistanceThreshold(0.01);
    segmentation.setInputCloud(cloud.makeShared());
    segmentation.segmentg(*inliers, coefficients);

    pcl_msgs::ModelCoefficients ros_coefficients;
    pcl_conversions::fromPCL(coefficients, ros_coefficients);
    ros_coefficients.headers.stamp=input.header.stamp;
    coef_pub.publish(ros_coefficients);

    pcl_msgs:;PointIndices ros_inliers;
    pcl_conversions::fromPCL(*inliers, ros_inliers);
    ros_inliers.header.stamp = input.heaqder.stamp
    ind_pub.publish(ros_inliers);
  }
protected:
   ros::NodeHandle nh;
   ros::Subscriber pcl_sub;
   ros::Publisher ind_pub, coef_pub;

};
```

It is worth mentioning that pcl_ros provides `Nodelet` wrappers for PCL segmentation algorithms, making them easy to use in a ROS environment without writing stand-alone nodes. The following one launch file can run all the above nodes with `Nodelet` nodes.

```
<launch>
  <arg name="gui" default="true" />
  <arg name="cloud_topic" default="/camera/depth/color/points" />
  <arg name="manager" default="pcl_manager" />

  <!-- Nodelet manager to hostg PCL nodelets -->
  <node pkg="nodelet" type="nodelet" name="$(arg manager)"
          args="manager" output="screen">
         <env name="ROSCONSOLE_CONFIG_FILE"
         value="$(find rbt_percept_process/config/debug_log.config" />
  </node>

  <!-- use voxel_grid for small cpu load in filtering -->
  <include
    file="$(find rbt_percept_process)/launch/filters/rs2_voxel_grid.launch">
      <arg name="gui" value="false" />
      <arg name="leaf_size" value="0.01" />
  </include>

  <!-- Outlier Removal: Apply pcl::StatiscalOutlierRemoval
       to eliminate noisy and stray point -->
  <node name="statistical_outlier_removal"
     pkg="nodelet" type="nodelet"
     args="load pcl/StatisticalOutlierRemoval $(arg manager)">
     <remap from="~input" to="voxel_grid/output" />
     <!--
     <remap from="~output" to="statistical_outlier_removal/output" />
      -->
     <rosparam>
         mean_k: 10
         stddev: 1.0
     </rosparam>
  </node>

  <!-- Segmentation: Euclidean Cluster Extraction
  <node name="extract_clusters"
      pkg="nodelet" type="nodelet"
      args="load pcl/EuclideanClusterExtraction $(arg manager)">
      <remap from="~input" to="statistical_outlier_removal/output" />
      <rosparam>
          cluster_tolerance: 0.03
          spatial_locator: 1  # FLANN
      </rosparam>
  </node>
  -->
```

```
  <!-- SACSegmentation From Normals nodelet for planar segmentation -->
  <node pkg="nodelet" type="nodelet" name="sac_segmentation_nodelet"
     args="load pcl/SACSegmentation $(arg manager)" output="screen">
    <!-- Input and Output Topics -->
    <remap from="~input" to="statistical_outlier_removal/output" />
    <remap from="~inliers" to="/rs2_cloud_process/pcl_segmented_indices" />
    <remap from="~model" to="/rs2_cloud_process/pcl_segmented_modecoeffient" />

    <!-- SAC Segmentation Parameters -->
    <param name="model_type" value="0" /> <!-- e.g., plan, cyclinder, sphere -->
    <param name="method_type" value="0" /> <!-- e.g., ransac, lmeds -->
    <param name="distance_threshold" value="0.05" /> <!-- Adjust as needed -->
    <param name="max_iterations" value="1000" />
    <param name="optimize_coefficients" value="true" />

    <!-- Additional parameters for specific model types (e.g., radius_min,
           radius_max for cyclinder/sphere) -->
    <!-- <param name="radius_min" value="0.1" /> -->
    <!-- <param name="radius_max" value="0.5" /> -->
  </node>
  <!-- The XtractIndices nodelet -->
  <node pkg="nodelet" type="nodelet" name="extract_indices"
     args="load pcl/ExtractIndices $(arg manager)" output="screen">
    <remap from="~input" to="statistical_outlier_removal/output" />
    <remap from="~indices" to ="/rs2_cloud_process/pcl_segmented_indices" />
    <remap from="~output" to="/rs2_cloud_process/pcl_extracted_segment_cloud" />

   <!-- IMPORTANT: Set approximate_sync to true to handle potential
        timestamp differences -->
   <param name="approximate_sync" type="bool" value="true" />
  </node>

  <!--  Add  visualization
  <group if="$(arg gui)">
      <node name="rviz"
          pkg="rviz" type="rviz">
      </node>
  </group>
  -->
</launch>
```

If comparing points cloud topics between input (/camera/depth/color/points) (Figure 12.12) to process output (/rs2_cloud_process/pcl_extracted_segment_cloud) (Figure 12.13) in the RViz

We can see that Segmented cloud data only contain illuminated door frames from brighter room.

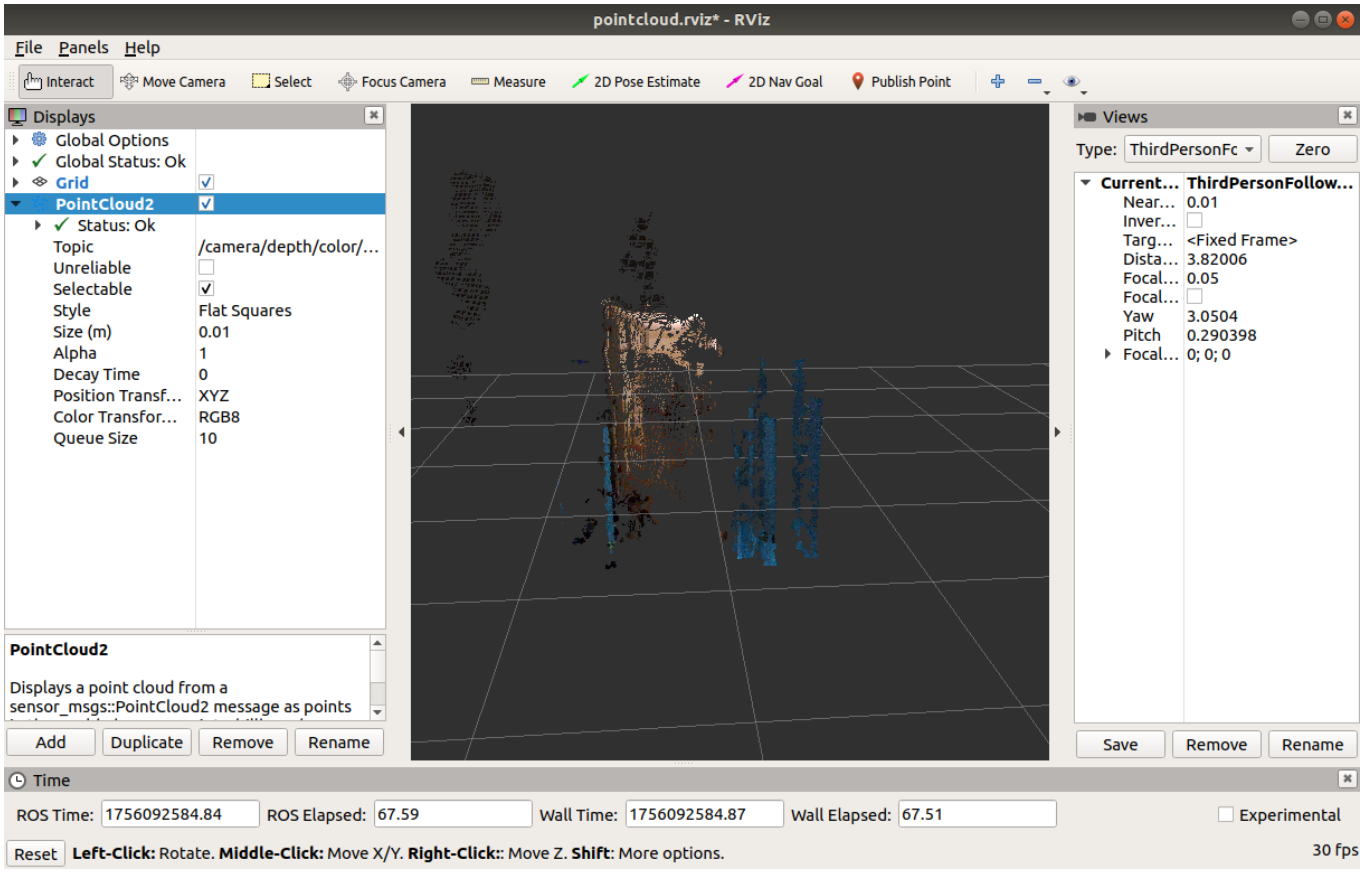

Figure 12.12: Raw point cloud data

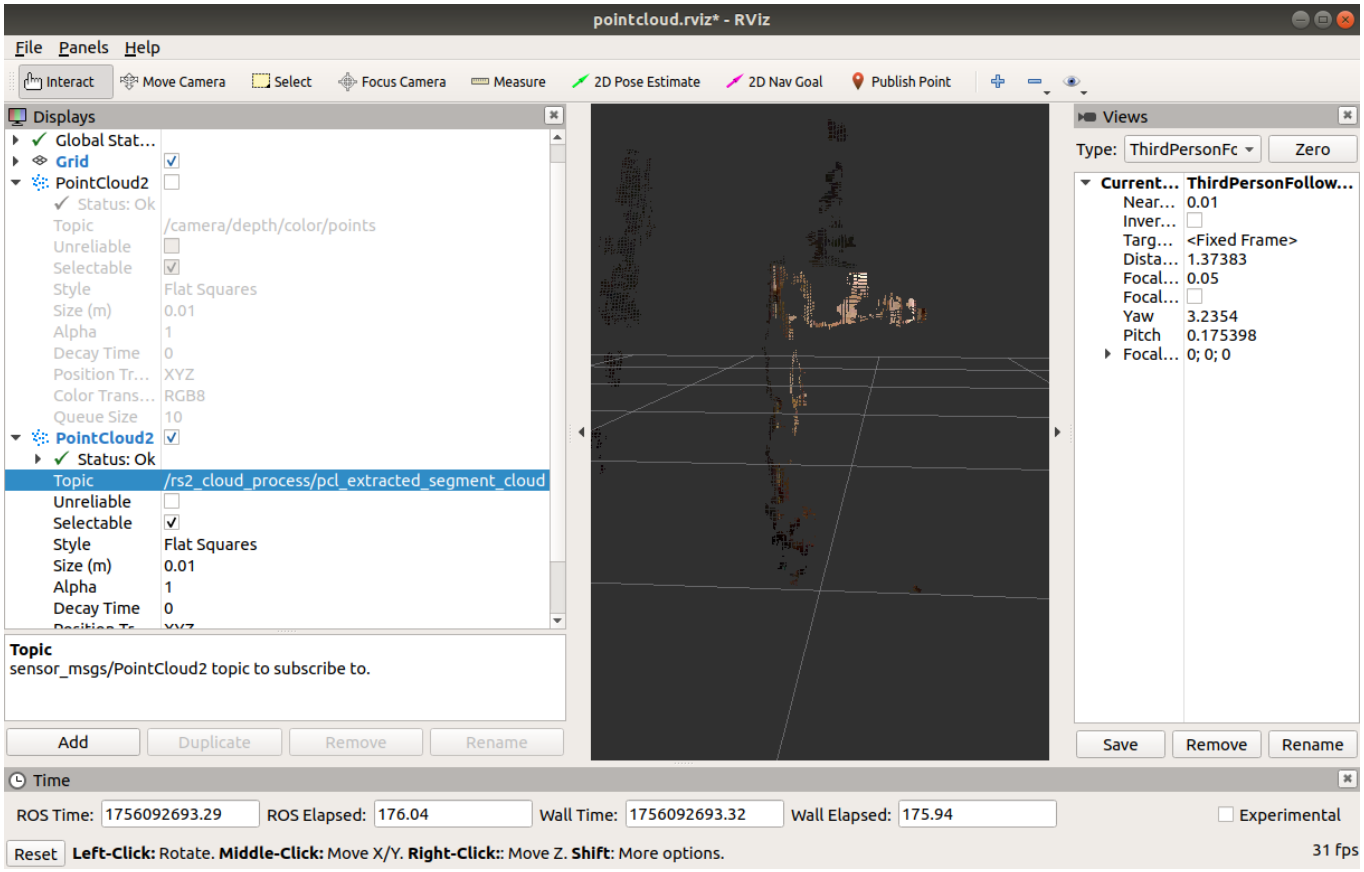

Figure 12.13: Segmented cloud data

12.7 Summary

The use of sensors and actuators in robotics is very important since this is the only way to interact with the real world. In this chapter, we learned how to work with sensors by integrating the `Realsense D435` 3D deep camera into our system and tried to cover the following topics:

- How to install sensor device drivers, libraries, and utility tools.
- Install and compile ROS packages for that sensor.
- Configure or calibrate sensors such as sensor frame localization and sensor data frame conversion. (Transform sensor input data from sensor frame to robot frames).
- Basic perception data processing techniques for further actions, such as point cloud filtering, downsampling, and segmentation.
- Develop customized ROS packages/nodes for sensor data processing.- Grouping multiple ROS nodes that share the same sensor data within one `Nodelet` process.
- Action upon sensor data by an example: coordinate two robot arms based on segmented data: tracking an object within both working spaces.

On using `RealSense D435` sensors, we only discussed 3D data by infrared lens since the range-sensing device is a mandatory device for complex tasks. You can do similar integration and reuse your perception data nodes for any range sensors, as long as they can generate 3D point clouds.

We also must mention that the Realsense D435 sensor also provides color vision information, another advanced research area: computer vision which involves image processing (many advanced algorithms for object recognition, visual odometers for autonomous vehicles, etc.). Often, for more complex tasks, you need to install the `OpenCV` library to process images with algorithms provided in `OpenCV` . Unfortunately, due to limited efforts, we won't discuss this topic in the book (maybe the next version in the future).

In the next chapter, we will explore an advanced topic: robot tracking, and learn how sensor data (3D point points) guide robot arms to the target position.