Kinect-ASUS-Xtion-Pro-Live-Calibration-Tutorials

Tao Chen

August 2016

Preparation

Installation

- Enter the following codes in your terminal

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

If you are using Asus Xtion Pro Live:

Modify `GlobalDefaults.ini`

```
sudo gedit /etc/openni/GlobalDefaults.ini
```

then uncomment the line: `;UsbInterface=2` (just delete the `;` symbol)

If you are using kinect v1.0:

```
mkdir ~/kinectdriver
cd ~/kinectdriver
git clone https://github.com/avin2/SensorKinect
cd SensorKinect/Bin/
tar xvjf SensorKinect093-Bin-Linux-x64-v5.1.2.1.tar.bz2
cd Sensor-Bin-Linux-x64-v5.1.2.1/
sudo ./install.sh
```

Test your Kinect(Asus Xtion Pro Live) with ROS

- To view the rgb image:

```
roslaunch openni_launch openni.launch
rosrun image_view image_view image:=/camera/rgb/image_raw
```

- To visualize the depth_registered point clouds:

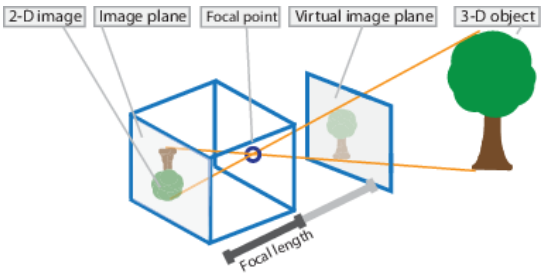
```
roslaunch openni_launch openni.launch depth_registration:=true
rosrun rviz rviz
```

A Brief Review of Camera Model

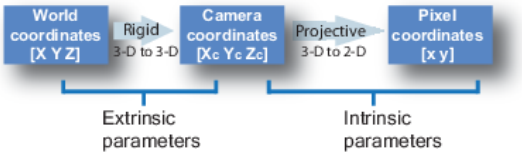
Camera can be modeled with a pinhole camera model and len distortion.

Pinhole Model

A pinhole camera is a simple camera without a lens and with a single small aperture. Light rays pass through the aperture and project an inverted image on the opposite side of the camera. Think of the virtual image plane as being in front of the camera and containing the upright image of the scene.



The pinhole camera parameters are represented in a 3-by-4 matrix called the camera matrix (or projection matrix). This matrix maps the 3-D world scene into the image plane. The calibration algorithm calculates the camera matrix using the extrinsic and intrinsic parameters. The extrinsic parameters represent the location of the camera in the 3-D scene. They represent a rigid transformation from 3-D world coordinate system to the 3-D camera's coordinate system. The intrinsic parameters represent the optical center and focal length of the camera. They represent a projective transformation from the 3-D camera's coordinates into the 2-D image coordinates.

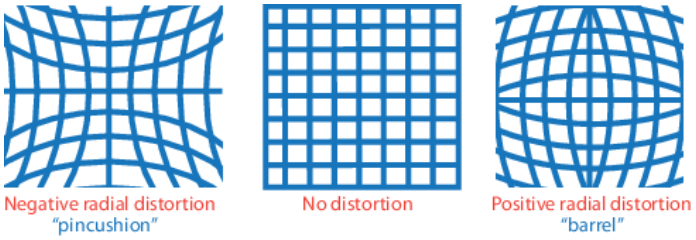


$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R & T \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = K \begin{bmatrix} R & T \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = P \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

where $\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$ is the point coordinates in the world coordinate system, $\begin{bmatrix} u \\ v \end{bmatrix}$ is the point coordinates (pixel) in the image coordinate system, λ is a scale factor(depth), f_x, f_y are the focal length in pixels, c_x, c_y are the optical center in pixels, s is the skew coefficient, K contains the intrinsic parameters, $\begin{bmatrix} R & T \end{bmatrix}$ contains the extrinsic parameters, P is the projection matrix.

Distortion in Camera

The camera matrix does not account for lens distortion because an ideal pinhole camera does not have a lens. To accurately represent a real camera, the camera model includes the radial and tangential lens distortion.



We need to takes into account the radial and tangential factors. For the radial factor one uses the following formula:

$$x_{corrected} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \; y_{corrected} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

For the tangential distortion one uses the following formula:

$$x_{corrected} = x + [2p_1 xy + p_2(r^2 + 2x^2)] \; y_{corrected} = y + [p_1(r^2 + 2y^2) + 2p_2 xy]$$

So for an old pixel point at (x, y) coordinates in the input image, its position on the corrected output image will be $(x_{corrected}, y_{corrected})$.

Calibrate Intrinsics

ROS provides an easy-to-use package that allows us to calibrate monocular camera. To calibrate the kinect intrinsics, we need to calibrate both the RGB camera as well as the IR camera. The Kinect detects depth by using an IR camera and IR speckle projector as a pseudo-stereo pair. We will calibrate the "depth" camera by detecting checkerboards in the IR image, just as we calibrated the RGB camera.

RGB Camera

Bring up the OpenNI driver:

```
roslaunch openni_launch openni.launch
```

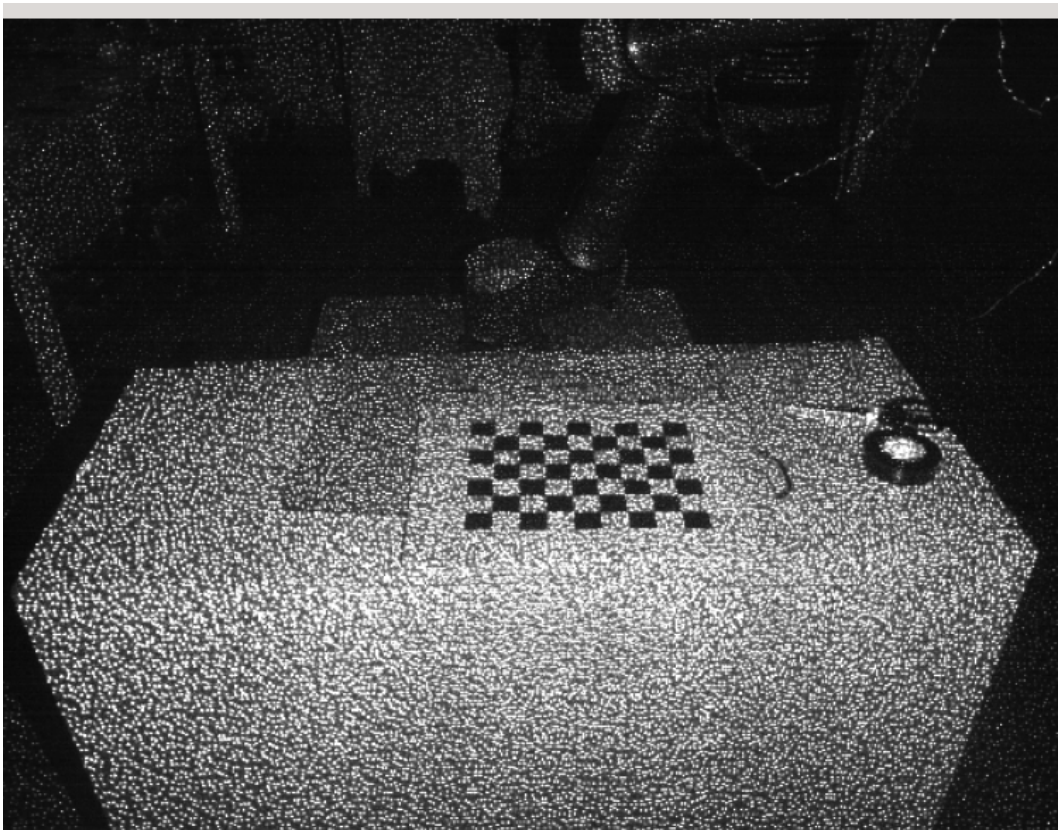
Now follow the standard monocular camera calibration instructions. Use the following command (**substituting the correct dimensions of your checkerboard**):

```
rosrun camera_calibration cameracalibrator.py image:=/camera/rgb/image_raw camera:=/camera/rgb --size 8x6
--square 0.0245
```

Don't forget to Commit your successful calibration.

IR(depth) Camera

The speckle pattern makes it impossible to detect the checkerboard corners accurately in the IR image.



The simplest solution is to cover the projector (lone opening on the far left) with one or two Post-it notes, mostly diffusing the speckles. An ideal solution is to block the projector completely and provide a separate IR light source.

```
rosrun camera_calibration cameracalibrator.py image:=/camera/ir/image_raw camera:=/camera/ir --size 8x6
--square 0.0245
```

The Kinect camera driver cannot stream both IR and RGB images. It will decide which of the two to stream based on the amount of subscribers, so kill nodes that subscribe to RGB images before doing the IR calibration.Don't forget to Commit your successful calibration.

When you click Commit, cameracalibrator.py sends the new calibration to the camera driver as a service call. The driver immediately begins publishing the updated calibration on its camera_info topic. **openni_camera** uses camera_info_manager to manage calibration parameters. By default, it saves intrinsics to `$HOME/.ros/camera_info/NAME.yaml` and identifies them by the device serial number:

```
$ ls $HOME/.ros/camera_info
depth_1504270110.yaml  rgb_1504270110.yaml
```

Calibrate Extrinsics

This is essentially a Perspective-n-Point (PnP) problem. We know that:

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = P \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

$$P = K \begin{bmatrix} R & T \end{bmatrix}$$

Therefore, if we know enough points with known world coordinates and pixel coordinates, we can solve P. Since we've just calibrated the intrinsics, we can now get $\begin{bmatrix} R & T \end{bmatrix}$.

OpenCV provides a handy function named `solvePnPRansac` which can solve this problem. To use that function, we need to provide enough points with their world coordinates as well as their correspondent pixel coordinates, intrinsic matrix, and distortion coefficients. You can see detailed implementation in **get_extrinsics.py**. To run this file, you need to change some parameters including the file directory to read in the intrinsic calibration result and output extrinsic calibration result, chessboard square width, the size of the chessboard. Notice that we need to get both the RGB and IR camera extrinsic parameters. However, we cannot get rgb image and ir image simultaneously if we are using openni_launch as the camera driver. Therefore, we need to calibrate one by one. Comment the other subscription to get the extrinsic parameters for each camera.

pose_estimation.py is just a tiny program that will show you your current pose relative to the chessboard. It will show the world coordinate system with its three axises (x-axis, y-axis, and z-axis). As you move the chessboard, the axises move together and remain on the same position on the chessboard.

One note here: The extrinsic matrix $\begin{bmatrix} R & T \end{bmatrix}$ describes how to transform points in world coordinates to camera coordinates. The vector T can be interpreted as the position of the world origin in camera coordinates, and the columns of R represent the directions of the world-axes in camera coordinates. It describes how the world is transformed relative to the camera. This is often counter-intuitive, because we usually want to specify how the camera is transformed relative to the world. It's often more natural to specify the camera's pose directly rather than specifying how world points should transform to camera coordinates. Luckily, building an extrinsic camera matrix this way is easy: just build a rigid transformation matrix that describes the camera's pose and then take it's inverse.

Let C be a column vector describing the location of the camera-center in world coordinates, and let $R_c$be the rotation matrix describing the camera's orientation with respect to the world coordinate axes. The transformation matrix that describes the camera's pose is then $\begin{bmatrix} R_c & C \end{bmatrix}$. Like before, we make the matrix square by adding an extra row of $\begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$. Then the extrinsic matrix is obtained by inverting the camera's pose matrix:

$$\begin{aligned} \begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix} &= \begin{bmatrix} R_c & C \\ 0 & 1 \end{bmatrix}^{-1} \\ &= \left(\begin{bmatrix} I & C \\ 0 & 1 \end{bmatrix} \begin{bmatrix} R_c & 0 \\ 0 & 1 \end{bmatrix} \right)^{-1} \\ &= \begin{bmatrix} R_c & 0 \\ 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} I & C \\ 0 & 1 \end{bmatrix}^{-1} \\ &= \begin{bmatrix} R_c^T & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} I & -C \\ 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} R_c^T & -R_c^T C \\ 0 & 1 \end{bmatrix} \end{aligned}$$

Register Depth to RGB

To get aligned depth image and rgb image, we need to register depth image to rgb image. The first thing we need to do is to get the transformation matrix between the ir(depth) camera and rgb camera.

$$Q_{rgb} = \begin{bmatrix} R_{rgb} & T_{rgb} \end{bmatrix} \times \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} R_{rgb} & T_{rgb} \end{bmatrix} \times \begin{bmatrix} Q \\ 1 \end{bmatrix} = R_{rgb} Q + T_{rgb}$$

$$Q_{ir} = [\, R_{ir} \quad T_{ir} \,] \times \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = [\, R_{ir} \quad T_{ir} \,] \times \begin{bmatrix} Q \\ 1 \end{bmatrix} = R_{ir}Q + T_{ir}$$

We can get from the above equations that:

$$\begin{aligned} Q_{rgb} &= R_{rgb}R_{ir}^{-1}(Q_{ir} - T_{ir}) + T_{rgb} \\ &= R_{rgb}R_{ir}^{-1}Q_{ir} + T_{rgb} - R_{rgb}R_{ir}^{-1}T_{ir} \end{aligned}$$

As we know that the transformation between the ir camera and rgb camera can be expressed as

$$Q_{rgb} = R_{irtorgb}Q_{ir} + T_{irtorgb}$$

Comparing these equations, we can get:

$$R_{irtorgb} = R_{rgb}R_{ir}^{-1}$$
$$T_{irtorgb} = T_{rgb} - R_{rgb}R_{ir}^{-1}T_{ir} = T_{rgb} - RT_{ir}$$

We've got the $R_{ir}, T_{ir}, R_{rgb}, T_{rgb}$ in the previous section. Hence, we can get $R_{irtorgb}, T_{irtorgb}$ which are just the rotation matrix and translation matrix between ir camera and rgb camera. Now, we can register the depth image to rgb image. For each pixel point $\begin{bmatrix} u \\ v \end{bmatrix}$ on the depth image, we do the following transformation:

- Transform the point's pixel coordinates in the depth image to the coordinates in the ir(depth) camera-centered coordinate system

$$\begin{bmatrix} x_{ir} \\ y_{ir} \\ z_{ir} \end{bmatrix} = K^{-1}\lambda \begin{bmatrix} u_{ir} \\ v_{ir} \\ 1 \end{bmatrix}$$

λ here is actually just the z coordinate(depth) of this point.

- Transform the point's coordinates in the ir(depth) camera-centered coordinate system to coordinates in the rgb camera-centered coordinate system

$$\begin{bmatrix} x_{rgb} \\ y_{rgb} \\ z_{rgb} \end{bmatrix} = R_{irtorgb} \begin{bmatrix} x_{ir} \\ y_{ir} \\ z_{ir} \end{bmatrix} + T_{irtorgb}$$

- Transform the point's coordinates in the rgb camera-centered coordinate system to coordinates in rgb image

$$\begin{bmatrix} u_{rgb} \\ v_{rgb} \\ 1 \end{bmatrix} = \frac{1}{z_{rgb}} K \begin{bmatrix} x_{rgb} \\ y_{rgb} \\ z_{rgb} \end{bmatrix}$$

Finally, we get the correspondent point of depth image in rgb image. You need to carefully implement the aforementioned procedures. It's better to implement these in a **matrix-operation** way instead of multiple for loops. Too many for loops will dramatically slow down the running speed. The detailed implementation is in **registration.py**. Again, you need to change some parameters including the file directory to read in the intrinsic calibration result and extrinsic calibration result, chessboard square width, the size of the chessboard.

The easy way:

Openni_launch has already taken care of all the abovementioned problems. You just need to subscribe `/camera/depth_registered/hw_registered/image_rect` to get the registered and rectified depth image and `/camera/rgb/image_rect_color` to get the rectified rgb images. They have good alignment to each other. The **image_to_world.py** allows you to select a region(a rectangle region) on the rgb image, and it will show the center point in red on the rgb image, output this point's world coordinates in the terminal. You can then verify how the accuracy of kinect.

References:

- Kinect Calibration
- Kinect Calibration_ros

- Kinect Calibration_github
- kinect intrinsic calibration
- kinect calibration technical
- depth_rgb_alignment
- camera calibration
- camera calibration
- camera coordinate system
- what does the projection matrix mean?
- CameraInfo Message
- Kinect Registration
- Extrinsic Matrix
- github ros depth_image_proc
- ros depth_image_proc
- ros rgbd_launch
- ros openni_launch