A Distributed Calibration Algorithm for Color and Range Camera Networks

Filippo Basso, Riccardo Levorato, Matteo Munaro, and Emanuele Menegatti

University of Padova, Department of Information Engineering (DEI), IAS-Lab
Via Ognissanti 72, I-35131 Padova, Italy.
{filippo.basso,riccardo.levorato,matteo.munaro,emg}@dei.unipd.it
http://robotics.dei.unipd.it

Abstract. In this tutorial chapter we present a package to calibrate multi-device vision systems such as camera networks or robots. The proposed approach is able to estimate – in a unique and consistent reference frame – the rigid displacements of all the sensors in a network of standard cameras, Kinect-like depth sensors and Time-of-Flight range sensors. The sensor poses can be estimated in a few minutes with a user-friendly procedure: the user is only asked to move a checkerboard around while the ROS nodes acquire the data and perform the calibration. To make the system scalable, the data analysis is distributed in the network. This results in a low bandwidth usage as well as a really fast calibration procedure. The ROS package is available on GitHub within the repository `iaslab-unipd/calibration_toolkit`[1]. The package has been developed for ROS Indigo in C++11 and Python, and tested on PCs equipped with Ubuntu 14.04 64bit.

Keywords: ROS, calibration, camera, depth, camera network, distributed system, Kinect, RGB-D.

1 Introduction

Robotic systems and camera networks consist of many heterogeneous vision sensors. The estimation of the poses of all such sensors with respect to a unique, consistent world frame, is a challenging and well-known problem. As a matter of fact, a good calibration of these sensors can be a useful starting point for several applications in the computer vision field (e.g. 3D mapping, people recognition and tracking [4, 15], microphone calibration for audio localization [14]) as well as in many robotics applications (e.g. simultaneous localization and mapping (SLAM) applications, grasping and manipulation).

However, even most of the time a good calibration is mandatory for the success of the application, there are still no tools that permit to easily calibrate multiple vision sensors together in a uniform way. In fact, most of the existing tools are for specific applications or specific sensors (e.g. stereo cameras); there

[1] `https://github.com/iaslab-unipd/calibration_toolkit`

Fig. 1. Example of sensors in a PC network that the proposed package aims to calibrate.

are only few methods developed to simultaneously calibrate an heterogeneous sensor network. As stated by Le et al. [13], the most followed approach is to divide the sensors into pairs and calibrate each pair independently, even using different algorithms for each one.

Our idea, i.e., the one behind the development of this package, is to go beyond this calibration technique, and develop an easy-to-use and easy-to-extend calibration package for ROS, such that users can add their own sensor types, their own error functions and perform the calibration. Moreover, since calibration is a time-consuming task, a fast procedure would be a very useful tool, especially when the involved sensors need often to be moved – and therefore re-calibrated.

We are still far from having a sensor-independent calibration toolbox like the aforementioned one, however, as we have already demonstrated [3, 14, 16], the very same calibration procedure can be used to calibrate standard cameras, Kinect-like and Time-of-Flight depth sensors as well as omnidirectional cameras and actuated laser scanners. So, we took what we had learned during the development of our previous works and packed it in a completely new package, heavily based on Eigen [11] and Ceres Solver [1] libraries.

The package addresses the problem of calibrating networks composed by cameras and depth sensors like the one in Fig. 1. The approach we followed is an extension of the classical single camera calibration procedure: users are asked to move a checkerboard pattern in front of the each camera and depth sensor and, as soon as any of the sensors see the checkerboard, the calibration starts. Then, whenever the pattern is visible by at least two sensors simultaneously, a constraint is added to the calibration problem. This process goes on until all the sensors are connected to the others. At the end, all the data are processed inside an optimization framework that improves the quality of the initial estimation. This procedure has already been tested on camera-only networks as part of

the OpenPTrack[2] project [16]. Here we explain how to configure and use the package as well as the theoretical background behind the developed algorithm. The remainder of the chapter is organized as follows:

- First, we review some of the existing works on camera and sensor network calibration.
- Second, we give a short overview of the calibration problem for camera networks.
- Third, we explain in detail how the calibration is performed from a theoretical point of view.
- Fourth, we introduce the readers to the package with a real-world example.
- Fifth, we describe how to configure and use the nodes provided by the package to calibrate a user-defined sensor network.
- Finally, we draw some conclusions.

2 Related Work

In the robot vision field, RGB cameras have been a key technology in the development of visual perception. In the last few years, the introduction of RGB-D sensors contributed deeply to the advancement of sensor fusion in practical applications. Auvinet et al. [2] proposed a new method for calibrating multiple depth cameras for body reconstruction using only depth information. Their algorithm is based on plane intersections and the NTP protocol for data synchronization. The calibration achieves good results: even if the depth error of the sensor is 10 mm, the reconstruction error with 3 depth cameras is, in the best case, less than 6 mm. A drawback of their implementation is that they have to manually select the plane corners and, above all, they only deal with depth sensors.

Another approach to solve the calibration problem is the one proposed by Le and Ng [13]: they jointly calibrate groups of sensors. More specifically, each group is composed by a set of sensors that can provide a 3D representation of the world (e.g. a stereo camera, an RGB camera and a depth camera, etc.). First of all they calibrate the intrinsics of each sensor, secondly they calibrate the extrinsic parameters of each group, then they calibrate the extrinsic parameters of each group with respect to all the others. Finally the calibration parameters are refined in one optimization step. Their experiments show that this method not only reduces the calibration error, but also requires a little human intervention. An advantage of having groups that output 3D data is that the same calibration objective can be used to calibrate a group with respect to all the others, regardless of the sensor type. Also, a joint calibration does not accumulate errors like a calibration based on sensor pairs do. However, they state that they should combine this two steps and jointly calibrate all parameters at once, as we proposed in our works [3] and propose here. In fact, the main drawback of this approach is that they always need to group the sensors beforehand in order to have 3D data outputs.

[2] `http://openptrack.org/`

Finally, Furgale et al. [10], recently developed a similar ROS package, *Kalibr*, that tackles the spatio-temporal calibration of multi-sensor systems composed of cameras and an IMU. To the best of our knowledge, this is the work most similar to ours.

3 Background

3.1 The Calibration Problem

Definition 1. *Let* $\mathfrak{S} = \{\mathsf{S}_1, \mathsf{S}_2 \dots \mathsf{S}_K\}$ *be a set of sensors. For each sensor* $\mathsf{S}_i \in \mathfrak{S}, i = 1 \dots K$*, the goal is to find its pose* ${}^{\mathcal{W}}\mathcal{S}_i$ *with respect to a common reference frame* $\mathcal{W}$*, namely the* world*.*

To solve such problem, it is important to know the concept of reference frame and how affine transformation (in a 3D space) works. Moreover, since we are dealing mostly with cameras, it is mandatory to know how 3D points are converted to pixels and vice-versa.

3.2 Affine Transformations

Let $\mathbf{x} = (x, y, z)^\mathsf{T}$ be a point in 3D space. For any non-zero real number w, $(xw, yw, zw, w)^\mathsf{T}$ is the set of homogeneous coordinates associated to the point $\mathbf{x}$. In particular, when $w = 1$, the resulting 4D vector $\widetilde{\mathbf{x}} = (x, y, z, 1)^\mathsf{T}$ is called the *normalized homogeneous form*.

A rigid transformation in 3D space is represented by a linear transformation $\mathbf{T} \in \mathbb{SE}(3)$ on normalized homogeneous vectors

$$\mathbf{T} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^\mathsf{T} & 1 \end{bmatrix}$$

where $\mathbf{R} \in \mathbb{SO}(3)$ is the *rotation matrix* and $\mathbf{t} \in \mathbb{R}^3$ the *translation* vector.

To transform a 3D point $\mathbf{x}$ it is sufficient to left-multiply its normalized homogeneous form $\widetilde{\mathbf{x}}$ by the transformation matrix $\mathbf{T}$

$$\widetilde{\mathbf{y}} = \mathbf{T} \cdot \widetilde{\mathbf{x}} \ .$$

The resulting vector $\widetilde{\mathbf{y}}$ is the normalized homogeneous form of the desired 3D point $\mathbf{y}$. This operation is equivalent to perform an affine transformation in $\mathbb{R}^3$

$$\mathbf{y} = \mathbf{R} \cdot \mathbf{x} + \mathbf{t} \ .$$

3.3 Reference Frames

A *reference frame* $\mathcal{F}$ is a coordinate system used to represent and measure position and orientation of objects. A transformation matrix ${}^{\mathcal{T}}_{\mathcal{S}}\mathbf{T}$ from a source reference frame $\mathcal{S}$ to a target frame $\mathcal{T}$ is a transformation matrix that allows to convert the coordinates of a point from $\mathcal{S}$ to $\mathcal{T}$.
Let, for example, $\mathbf{x}$ be a point and let ${}^{\mathcal{S}}\mathbf{x}$ be its coordinates in $\mathcal{S}$, $\mathbf{x}$'s coordinates in $\mathcal{T}$, namely ${}^{\mathcal{T}}\mathbf{x}$, can be computed as

$${}^{\mathcal{T}}\widetilde{\mathbf{x}} = {}^{\mathcal{T}}_{\mathcal{S}}\mathbf{T} \cdot {}^{\mathcal{S}}\widetilde{\mathbf{x}} \ .$$

3.4 Pinhole Camera Model

Let C be a camera and $\mathcal{C}$ its reference frame. The pinhole model describes the mathematical relationship between the coordinates of a 3D point and its projection onto the image plane. The model assumes that the camera has 4 parameters, namely *intrinsic parameters*:

- $(c_x, c_y)^\mathsf{T}$ is the *principal point* that is usually at the image center;
- f_x, f_y are the *focal lengths* expressed in pixel units;

usually arranged in a 3×4 matrix $\mathbf{K}_\mathsf{C}$

$$\mathbf{K}_\mathsf{C} = \begin{bmatrix} f_x & 0 & c_x & 0 \\ 0 & f_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \ .$$

The relationship between the coordinates of a 3D point ${}^{\mathcal{C}}\mathbf{x}$ and its projection onto the image plane $\mathbf{x}'$ (in pixels) is

$$s \cdot \widetilde{\mathbf{x}}' = \mathbf{K}_\mathsf{C} \cdot {}^{\mathcal{C}}\widetilde{\mathbf{x}} \ , \tag{1}$$

at least theoretically. Unfortunately real lenses usually have some distortion. So, the pinhole model is extended with a vector of *distortion coefficients* $\mathbf{d}_\mathsf{C}$ and a distortion function $\mathrm{d}_\mathsf{C}(\cdot)$. Equation (1) thus becomes

$$s \cdot \widetilde{\mathbf{x}}' = \mathbf{K}_\mathsf{C} \cdot \mathrm{d}_\mathsf{C}({}^{\mathcal{C}}\widetilde{\mathbf{x}}) \ . \tag{2}$$

For more details on the distortion function please refer to [6].
In the following, given a camera C, the *reprojection* of a 3D point onto C's image plane, i.e. (2), will be denoted by the function $\mathrm{r}_\mathsf{C}(\cdot)$, that is

$$\mathbf{x}' = \mathrm{r}_\mathsf{C}\left({}^{\mathcal{C}}\mathbf{x}\right) \ . \tag{3}$$

3.5 Notations

We use non-bold characters x to represent scalars, bold lower case letters $\mathbf{x}$ to represent vectors with no distinction between cartesian coordinates and homogeneous coordinates. Bold upper case letters $\mathbf{M}$ represent matrices. Note that matrices can be seen as ordered lists of vectors, one for each column.

The reference frame of an object B is in calligraphic style $\mathcal{B}$. The coordinates of an entity e with respect to the reference frame $\mathcal{F}$ are denoted by ${}^{\mathcal{F}}e$. According to this notation, the pose of a body A in B's coordinate system $\mathcal{B}$ is denoted as ${}^{\mathcal{B}}\mathcal{A}$ and the relative homogeneous transformation matrix is ${}^{\mathcal{B}}_{\mathcal{A}}\mathbf{T}$.

4 Camera-only Network Calibration

4.1 Pose Estimation

To solve the calibration problem (Def. 1) for a camera-only network, we use a *checkerboard pattern*. So, let B be an $R \times C$ checkerboard and let $\mathcal{B}$ be its reference

frame. Let also C be a camera with reference frame $\mathcal{C}$, intrinsic parameters $\mathbf{K}_\mathsf{C}$ and distortion coefficients $\mathbf{d}_\mathsf{C}$. We can estimate the checkerboard pose ${}^{\mathcal{C}}_{\mathcal{B}}\mathbf{T}$ in the camera reference frame by finding its corners in the image and solving the correspondent *Perspective-n-Point* (PnP) problem [9].
That is, let $\mathbb{I} = \{1 \dots R\} \times \{1 \dots C\}$ be the corner indices, let also

$$ {}^{\mathcal{B}}\mathbf{B} = \left\{ {}^{\mathcal{B}}\mathbf{b}_{r,c} \right\}_{(r,c)\in\mathbb{I}} $$

be the checkerboard corners and

$$ \mathbf{B}' = \left\{ \mathbf{b}'_{r,c} \right\}_{(r,c)\in\mathbb{I}} $$

the correspondent locations in the image. The checkerboard pose ${}^{\mathcal{C}}_{\mathcal{B}}\mathbf{T}$ can be estimated by means of a single function called *solvePnP*, i.e.

$$ {}^{\mathcal{C}}_{\mathcal{B}}\mathbf{T} = \text{solvePnP}\left(\mathbf{K}_\mathsf{C}, \mathbf{d}_\mathsf{C}, {}^{\mathcal{B}}\mathbf{B}, \mathbf{B}'\right) \quad , \tag{4} $$

and is the one that minimizes the *reprojection error*

$$ \mathrm{e}_{\mathrm{r}_\mathsf{C}}\left({}^{\mathcal{C}}_{\mathcal{B}}\mathbf{T}, {}^{\mathcal{B}}\mathbf{B}, \mathbf{B}'\right) = \sum_{(r,c)\in\mathbb{I}} \left\| \mathbf{b}'_{r,c} - \mathrm{r}_\mathsf{C}\left({}^{\mathcal{C}}_{\mathcal{B}}\mathbf{T} \cdot {}^{\mathcal{B}}\mathbf{b}_{r,c}\right) \right\|^2 \quad . \tag{5} $$

Now, let C_1 and C_2 be two different cameras, with reference frame $\mathcal{C}_1$ and $\mathcal{C}_2$ respectively and suppose they both can see the checkerboard at the same time. To be more precise, let's define $k \in \mathbb{N}$ as an *acquisition step*, that is, a progressive number that is incremented each time the checkerboard is moved to a different location and an acquisition for every camera is triggered at the same instant. Let also ${}^{\mathcal{W}}_{\mathcal{B}}\mathbf{T}^{(k)}$ be the checkerboard pose at step k. Using (4) we can estimate both ${}^{\mathcal{C}_1}_{\mathcal{B}}\mathbf{T}^{(k)}$ and ${}^{\mathcal{C}_2}_{\mathcal{B}}\mathbf{T}^{(k)}$. Starting from these two poses (and the fact the the checkerboard is in the same location), we can estimate the pose of one sensor with respect to the other ${}^{\mathcal{C}_1}_{\mathcal{C}_2}\mathbf{T}$ with a closed formula

$$ {}^{\mathcal{C}_1}_{\mathcal{C}_2}\mathbf{T} = {}^{\mathcal{C}_1}_{\mathcal{C}_2}\mathbf{T}^{(k)} = {}^{\mathcal{C}_1}_{\mathcal{B}}\mathbf{T}^{(k)} \cdot {}^{\mathcal{C}_2}_{\mathcal{B}}\mathbf{T}^{(k)^{-1}} \quad . \tag{6} $$

Then, remembering that affine transforms can be chained

$$ {}^{\mathcal{B}}_{\mathcal{A}}\mathbf{T} = {}^{\mathcal{B}}_{\mathcal{X}}\mathbf{T} \cdot {}^{\mathcal{X}}_{\mathcal{A}}\mathbf{T} \quad , $$

and that

$$ {}^{\mathcal{B}}_{\mathcal{A}}\mathbf{T} = {}^{\mathcal{A}}_{\mathcal{B}}\mathbf{T}^{-1} \quad , $$

we can find a solution to our calibration problem for a network composed by N cameras. We just need to estimate (or set) the pose ${}^{\mathcal{W}}_{\mathcal{C}_i}\mathbf{T}$ of one camera C_i with respect to the world reference frame $\mathcal{W}$ and move the checkerboard around until each camera pose is computed, using any of the aforementioned equations.

To estimate the pose of a camera C_i with respect to the world reference frame $\mathcal{W}$, first of all we must know whether we need to define a world reference frame $\mathcal{W}$ in the environment or not. In fact, if it really does not matter where such

reference frame is, any camera reference frame $\mathcal{C}_i$ can be set as the world, that is $\mathcal{W} = \mathcal{C}_i$ (or equally ${}^{\mathcal{W}}_{\mathcal{C}_i}\mathbf{T} = \mathbf{I}$) for one $i \in \{1 \dots N\}$. Otherwise the pose can be set manually: ${}^{\mathcal{W}}_{\mathcal{C}_i}\mathbf{T} = \mathbf{W}$ for some transformation matrix $\mathbf{W}$ and $i \in \{1 \dots N\}$; or estimated by moving the checkerboard to the desired position and setting ${}^{\mathcal{W}}_{\mathcal{B}}\mathbf{T}^{(k)} = \mathbf{I}$, for some $k \in \mathbb{N}$.

At this stage we have good estimations of the sensor poses, however, due to errors in the measurements, usually

$$ {}^{\mathcal{C}_1}_{\mathcal{C}_2}\mathbf{T}^{(k)} \neq {}^{\mathcal{C}_1}_{\mathcal{C}_2}\mathbf{T}^{(l)} $$

for two different steps k and l. Therefore we must perform an optimization step to refine the estimated camera poses, such that the error on the estimated poses is reduced as much as possible.

4.2 Optimization

Taking a step back to the acquisition part, we can organize the calibration data in a matrix, like the one in Fig. 2. Here, what we need to refine are the checkerboard poses but, following the *bundle adjustment* approach [12], we refine both the camera poses ${}^{\mathcal{W}}_{\mathcal{C}_i}\mathbf{T}$, $i = 1 \dots N$ and the checkerboard poses ${}^{\mathcal{W}}_{\mathcal{B}}\mathbf{T}^{(k)}$, $k = 1 \dots K$. Indeed, even if we perfectly know the pose of every camera with respect to the world, the pose of a checkerboard B estimated at step k using two different cameras, say C_i and C_j, is likely to be different, that is

$$ {}^{\mathcal{W}}_{\mathcal{C}_i}\mathbf{T}^{(k)} \cdot {}^{\mathcal{C}_i}_{\mathcal{B}}\mathbf{T}^{(k)} \neq {}^{\mathcal{W}}_{\mathcal{C}_j}\mathbf{T}^{(k)} \cdot {}^{\mathcal{C}_j}_{\mathcal{B}}\mathbf{T}^{(k)} \quad . $$

To achieve a satisfying solution, a good candidate to be minimized is the reprojection error defined in (5). The complete error function E_{C} that we minimize is therefore

$$ \begin{aligned} E_{\mathsf{C}} &= \sum_{k=1}^{K}\sum_{i=1}^{N} u_{ik} \cdot \frac{1}{\sigma^2_{\mathsf{C}_i}} \cdot \mathrm{e}_{\mathrm{r}_{\mathsf{C}_i}} \left({}^{\mathcal{W}}_{\mathcal{C}_i}\mathbf{T}^{-1} \cdot {}^{\mathcal{W}}_{\mathcal{B}}\mathbf{T}^{(k)}, {}^{\mathcal{B}}\mathbf{B}, \mathbf{B}'^{(k)}_i \right) \\ &= \sum_{k=1}^{K}\sum_{i=1}^{N} u_{ik} \cdot \frac{1}{\sigma^2_{\mathsf{C}_i}} \cdot \sum_{(r,c)\in\mathbb{I}} \left\| \mathbf{b}'^{(k)}_{i,r,c} - \mathrm{r}_{\mathsf{C}_i} \left({}^{\mathcal{W}}_{\mathcal{C}_i}\mathbf{T}^{-1} \cdot {}^{\mathcal{W}}_{\mathcal{B}}\mathbf{T}^{(k)} \cdot {}^{\mathcal{B}}\mathbf{b}_{r,c} \right) \right\|^2 \quad , \end{aligned} \tag{7} $$

where u_{ik} is an indication function equal to 1 if camera C_i sees the checkerboard at step k (otherwise it is 0) and the fraction $\frac{1}{\sigma^2_{\mathsf{C}_i}}$ is instead a normalization factor. The term σ_{C_i} is usually set to 1 or 0.5 and indicates the error on the corner's estimated position (in pixels) in the images provided by camera C_i.

4.3 Additional Constraints

In one of our first applications of this calibration algorithm, OpenPTrack [16], we needed to calibrate a camera network in a big room against the floor, i.e. extract the floor equation and set the world frame somewhere on it. We decided

Cameras		Steps							
		${}^{\mathcal{W}}_{\mathcal{B}}\mathbf{T}^{(1)}$	${}^{\mathcal{W}}_{\mathcal{B}}\mathbf{T}^{(2)}$	${}^{\mathcal{W}}_{\mathcal{B}}\mathbf{T}^{(3)}$	${}^{\mathcal{W}}_{\mathcal{B}}\mathbf{T}^{(4)}$	${}^{\mathcal{W}}_{\mathcal{B}}\mathbf{T}^{(5)}$	${}^{\mathcal{W}}_{\mathcal{B}}\mathbf{T}^{(6)}$	${}^{\mathcal{W}}_{\mathcal{B}}\mathbf{T}^{(7)}$	${}^{\mathcal{W}}_{\mathcal{B}}\mathbf{T}^{(8)}$
$\mathbf{K}_{\mathsf{C}_1}, \mathbf{d}_{\mathsf{C}_1}$	${}^{\mathcal{W}}_{\mathcal{C}_1}\mathbf{T}$				$\mathbf{B}_1'^{(4)}$				$\mathbf{B}_1'^{(8)}$
$\mathbf{K}_{\mathsf{C}_2}, \mathbf{d}_{\mathsf{C}_2}$	${}^{\mathcal{W}}_{\mathcal{C}_2}\mathbf{T}$	$\mathbf{B}_2'^{(1)}$	$\mathbf{B}_2'^{(2)}$			$\mathbf{B}_2'^{(5)}$		$\mathbf{B}_2'^{(7)}$	
$\mathbf{K}_{\mathsf{C}_3}, \mathbf{d}_{\mathsf{C}_3}$	${}^{\mathcal{W}}_{\mathcal{C}_3}\mathbf{T}$	$\mathbf{B}_3'^{(1)}$					$\mathbf{B}_3'^{(6)}$		$\mathbf{B}_3'^{(8)}$
$\mathbf{K}_{\mathsf{C}_4}, \mathbf{d}_{\mathsf{C}_4}$	${}^{\mathcal{W}}_{\mathcal{C}_4}\mathbf{T}$				$\mathbf{B}_4'^{(4)}$	$\mathbf{B}_4'^{(5)}$	$\mathbf{B}_4'^{(6)}$		
$\mathbf{K}_{\mathsf{C}_5}, \mathbf{d}_{\mathsf{C}_5}$	${}^{\mathcal{W}}_{\mathcal{C}_5}\mathbf{T}$		$\mathbf{B}_5'^{(2)}$	$\mathbf{B}_5'^{(3)}$				$\mathbf{B}_5'^{(7)}$	

Fig. 2. Matrix view of the calibration data. Each row is associated to a camera C_i and contains both the camera parameters $\mathbf{K}_{\mathsf{C}_i}$ and $\mathbf{d}_{\mathsf{C}_i}$, and the camera estimated pose ${}^{\mathcal{W}}_{\mathcal{C}_i}\mathbf{T}$. The columns are instead associated to the steps and contain the poses of the checkerboard at every step k, namely ${}^{\mathcal{W}}_{\mathcal{B}}\mathbf{T}^{(k)}$. A cell (i, k) contains the corners locations (in pixels) $\mathbf{B}_i'^{(k)}$ of the checkerboard at step k in the image provided by camera C_i, if the checkerboard is visible.

to estimate the floor coefficients during the calibration procedure, exploiting the fact that positioning a checkerboard on the floor would have allowed us to define the plane equation as well as the world reference frame. However, since the room was quite big and the checkerboard far from every camera, the results were not satisfactory: the plane had often a non-negligible rotation with respect to the real one. To overcome this issue, printing a bigger checkerboard was not a viable solution. Instead, we imposed that two or more checkerboards were lying on the same plane and added the geometrical constraints to the error model. In fact, if we fix the plane π on which a checkerboard can move, the checkerboard pose can be defined by a 2D transform ${}^{\mathcal{P}}_{\mathcal{B}}\mathbf{T}_2$ with respect to the reference frame of plane π, namely $\mathcal{P}$.

So, let define a plane by means of its reference frame ${}^{\mathcal{W}}_{\mathcal{P}}\mathbf{T}$, such that the x- and y-axes are on the plane and the z-axis is its normal, as depicted in Fig. 3. Then the pose of a checkerboard lying on π at step k is

$$ {}^{\mathcal{W}}_{\mathcal{B}}\mathbf{T}^{(k)} = {}^{\mathcal{W}}_{\mathcal{P}}\mathbf{T} \cdot {}^{\mathcal{P}}_{\mathcal{B}}\mathbf{T}_2^{(k)}, \tag{8} $$

where the 2D transform ${}^{\mathcal{P}}_{\mathcal{B}}\mathbf{T}_2^{(k)}$ is wrapped into a 3D one to perform the matrix multiplication

$$ {}^{\mathcal{P}}_{\mathcal{B}}\mathbf{T}_2^{(k)} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & t_x \\ \sin(\theta) & \cos(\theta) & 0 & t_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. $$

We can now substitute (8) into (7) to refine both the plane and the checkerboard pose.

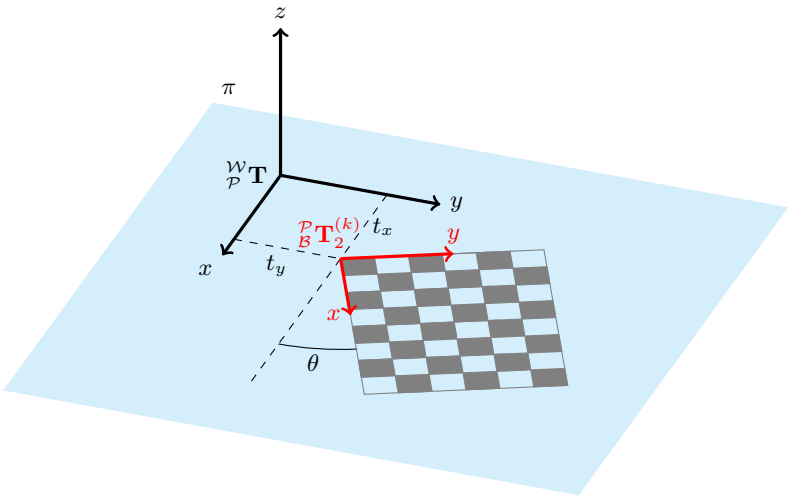

Fig. 3. Reference frames of the plane π and a checkerboard B lying on it.

5 Extension to a Depth Sensor-Camera Network

5.1 Pose Estimation

In Sect. 4 we have presented a calibration procedure for camera-only networks. Such procedure works well with cameras, but how can we calibrate a network composed by both cameras and depth sensors?

A depth sensor D provides an $R \times C$ point cloud view of the scene, i.e. an indexed set of 3D points

$$^{\mathcal{D}}\mathbf{P} = \left\{ {}^{\mathcal{D}}\mathbf{p}_{r,c} \right\}_{(r,c)\in\mathbb{I}},$$

where $\mathbb{I} = \{1 \dots R\} \times \{1 \dots C\}$, reflecting the shape of the scene in the sensor's field of view. Obviously, we cannot directly estimate the checkerboard pose using the corners, as we do for the camera-only network calibration, since they are not visible. However, we can exploit the 3D data to extract the pattern plane and perform a *plane-to-plane calibration* [20]. That is, supposing we have already estimated the checkerboard pose ${}^{\mathcal{W}}_{\mathcal{B}}\mathbf{T}^{(k)}$ at step k, we can define the checkerboard plane ${}^{\mathcal{W}}\pi^{(k)}_{\mathsf{B}}$ using three non-collinear corners. We can also estimate the pattern plane ${}^{\mathcal{D}}\pi^{(k)}_{\mathsf{B}}$ from the point cloud, using a for example a RANSAC-based [9] plane fitting algorithm.

So, let $\{k_1 \dots k_n\}$, with $n \geq 3$, be the intersection between the steps in which depth sensor D sees the checkerboard and those in which the checkerboard pose has been estimated, and let the plane equations be of the form $\mathbf{n}^{\mathsf{T}} \cdot \mathbf{x} - d = 0$.

Following [20], we define

$$
{}^{\mathcal{W}}\mathbf{N} = \left[{}^{\mathcal{W}}\mathbf{n}_{\mathsf{B}}^{(k_1)} \dots {}^{\mathcal{W}}\mathbf{n}_{\mathsf{B}}^{(k_n)} \right]^{\mathsf{T}} \qquad {}^{\mathcal{W}}\mathbf{d} = \left[{}^{\mathcal{W}}d_{\mathsf{B}}^{(k_1)} \dots {}^{\mathcal{W}}d_{\mathsf{B}}^{(k_n)} \right]^{\mathsf{T}}
$$
$$
{}^{\mathcal{D}}\mathbf{N} = \left[{}^{\mathcal{D}}\mathbf{n}_{\mathsf{B}}^{(k_1)} \dots {}^{\mathcal{D}}\mathbf{n}_{\mathsf{B}}^{(k_n)} \right]^{\mathsf{T}} \qquad {}^{\mathcal{D}}\mathbf{d} = \left[{}^{\mathcal{D}}d_{\mathsf{B}}^{(k_1)} \dots {}^{\mathcal{D}}d_{\mathsf{B}}^{(k_n)} \right]^{\mathsf{T}}
$$

and compute

$$
{}^{\mathcal{W}}_{\mathcal{D}}\mathbf{T} = \begin{pmatrix} {}^{\mathcal{W}}_{\mathcal{D}}\mathbf{R} & {}^{\mathcal{W}}_{\mathcal{D}}\mathbf{t} \\ \mathbf{0}^{\mathsf{T}} & 1 \end{pmatrix}
$$

as

$$
\begin{aligned}
{}^{\mathcal{W}}_{\mathcal{D}}\mathbf{t} &= \left({}^{\mathcal{W}}\mathbf{N}^{\mathsf{T}} \cdot {}^{\mathcal{W}}\mathbf{N} \right)^{-1} \cdot {}^{\mathcal{W}}\mathbf{N}^{\mathsf{T}} \cdot \left({}^{\mathcal{W}}\mathbf{d} - {}^{\mathcal{D}}\mathbf{d} \right) \ , \\
{}^{\mathcal{W}}_{\mathcal{D}}\mathbf{R} &= \mathbf{V} \cdot \mathbf{U}^{\mathsf{T}} \ ,
\end{aligned}
$$

where $\mathbf{U} \cdot \mathbf{S} \cdot \mathbf{V}^{\mathsf{T}}$ is the SVD decomposition of ${}^{\mathcal{D}}\mathbf{N}^{\mathsf{T}} \cdot {}^{\mathcal{W}}\mathbf{N}$.

5.2 Optimization

For what concerns the error function, we calculate it as a sort of distance between the plane defined by the checkerboard, say ${}^{\mathcal{W}}\pi_{\mathsf{B}}^{(k)}$, at step k, and the plane fitted to the checkerboard depth data of sensors D_j, $j = 1 \dots M$, namely ${}^{\mathcal{D}_j}\pi_{\mathsf{B}}^{(k)}$. So, let $\mathrm{p}_\pi(\mathbf{x})$ be the *line-of-sight* projection of a point $\mathbf{x}$ onto plane π as described in [18], we can define the error function E_{D} as

$$
\begin{aligned}
E_{\mathsf{D}} &= \sum_{k=1}^{K} \sum_{j=1}^{M} u_{jk} \cdot \frac{1}{\sigma_{\mathsf{D}_j}^2} \cdot \mathrm{e}_{\mathrm{p}_{\mathcal{D}_j \pi_{\mathsf{B}}^{(k)}}} \left({}^{\mathcal{W}}_{\mathcal{D}_j}\mathbf{T}^{-1} \cdot {}^{\mathcal{W}}_{\mathcal{B}}\mathbf{T}^{(k)}, {}^{\mathcal{B}}\mathbf{B} \right) \\
&= \sum_{k=1}^{K} \sum_{j=1}^{M} u_{jk} \cdot \frac{1}{\sigma_{\mathsf{D}_j}^2} \cdot \sum_{(r,c) \in \mathbb{I}} \left\| {}^{\mathcal{D}_j}\mathbf{b}_{r,c}^{(k)} - \mathrm{p}_{\mathcal{D}_j \pi_{\mathsf{B}}^{(k)}} \left({}^{\mathcal{D}_j}\mathbf{b}_{r,c}^{(k)} \right) \right\|^2 \ ,
\end{aligned}
$$

where

$$
{}^{\mathcal{D}_j}\mathbf{b}_{r,c}^{(k)} = {}^{\mathcal{W}}_{\mathcal{D}_j}\mathbf{T}^{-1} \cdot {}^{\mathcal{W}}_{\mathcal{B}}\mathbf{T}^{(k)} \cdot {}^{\mathcal{B}}\mathbf{b}_{r,c} \ ,
$$

and u_{jk} is an indication function equal to 1 if sensor D_j sees the checkerboard at step k (otherwise it is 0) and the fraction $\frac{1}{\sigma_{\mathsf{D}_j}^2}$, as for cameras, is a normalization factor. But, while for cameras, σ_{D_j} is usually set to corner estimation error in pixels, for depth sensor it describes the error on the depth estimation. This means that each depth sensor may have a different normalization factor. Moreover, there are sensors, like Kinects v1, for which this error depends on the depth value, that is

$$
\sigma_{\mathsf{D}_j}(z) = a + b \cdot z + c \cdot z^2 \ , \tag{9}
$$

for some coefficients (a, b, c). Typical values[3] for a Kinect v1 are: $a = 0, b = 0, c = 0.0035$. Taking into account this fact, the error function E_{D} becomes

$$
E_{\mathsf{D}} = \sum_{k=1}^{K} \sum_{j=1}^{M} u_{jk} \cdot \sum_{(r,c) \in \mathbb{I}} \frac{1}{\sigma_{\mathsf{D}_j}^2 \left({}^{\mathcal{D}_j}z_{r,c}^{(k)} \right)} \cdot \left\| {}^{\mathcal{D}_j}\mathbf{b}_{r,c}^{(k)} - \mathrm{p}_{\mathcal{D}_j \pi_{\mathsf{B}}^{(k)}} \left({}^{\mathcal{D}_j}\mathbf{b}_{r,c}^{(k)} \right) \right\|^2 \tag{10}
$$

[3] `http://wiki.ros.org/openni_kinect/kinect_accuracy`

where ${}^{\mathcal{D}_j} z_{r,c}^{(k)}$ is the z component of ${}^{\mathcal{D}_j}\mathbf{b}_{r,c}^{(k)}$.

6 ROS Environment Configuration

6.1 Dependencies

The package depends on some external libraries, well integrated in ROS, such as:

- Boost
- OpenCV
- Eigen 3.2
- PCL 1.7

They are all available in the Ubuntu 14.04 repository and easily installable. For what concerns the optimization algorithm, instead, the package relies upon Ceres Solver [1] a library to solve non-linear least squares problems. In Ubuntu 14.04, it is possible to install version 1.8 of such library by selecting the apt package `libceres-dev`, however, due to some bugs on that library version, our package does not compile. To overcome this issue, we have prepared a script to download the latest tested version and install it. Just type

```
roscd calibration_toolkit/../scripts
./install_ceres.sh
```

on a terminal, and Ceres Solver as well as its dependencies will be installed on your system.

The algorithm assumes that all the sensors' intrinsic parameters are already estimated and expects that each camera publishes its own calibration parameters as a `sensor_msgs/CameraInfo` message. This is a common assumption for cameras, for which a lot of specialized tools (e.g. the `camera_calibration` ROS package) exist, while it is a bad assumption when dealing with depth sensors. In fact, for depth sensors, even if it has been demonstrated [5, 7, 8, 17, 19, 22] that depth measurements are not reliable, this operation is not really common. Unfortunately, up to this time, there is not an established way of calibrating such sensors, not even a ROS way to deal with their distortion. Users must therefore rely on external packages, like the ones presented in [5, 8] or [19].

Finally, it's worth noticing that the package is developed in C++11 and needs a compatible compiler to work. In particular we are currently compiling with gcc 4.8.

6.2 Basic Configuration

In order to allow communication between nodes in different computers, the environment variable `ROS_MASTER_URI` on every client PC must be set to the IP address of the PC where the master node is launched, namely the *master PC*. Additionally, the `ROS_IP` and `ROS_PC_NAME` environment variables must be set. This can be done temporarily by typing

```
export ROS_MASTER_URI=http://<MASTER_IP>:11311/
export ROS_IP=<MACHINE_IP>
export ROS_PC_NAME=<MACHINE_NAME>
```

on a terminal. Note that the PC names assigned can be whatever the user wants, not necessarily related to the real names of the PCs. To set them definitively, they can be added directly to the end of the `.bashrc` file in the home folder. As an example:

```
echo "export ROS_MASTER_URI=http://192.168.1.1:11311/
export ROS_IP=192.168.1.5
export ROS_PC_NAME=Phoenix" >> ~/.bashrc
```

7 Real-World Example

Suppose all the PCs are configured as explained in Sect. 6, and that we want to calibrate a network of two cameras connected to two different PCs that are called, respectively, `Phoenix` and `Gemini`. The multi-sensor calibration is performed by running a *master node* in a so called master PC, in our case `Lyra`, and a device driver in every PC attached to a device (one driver node for each device). First of all, we have to run the ROS drivers for all our devices. As an example, for a PointGrey camera, type on a terminal:

```
roslaunch pointgrey_camera_driver camera.launch
```

Then, we have to wrap these drivers in our calibration environment so that they can communicate with the master node in `Lyra`. To this aim, we run on both `Phoenix` and `Gemini`:

```
roslaunch multisensor_calibration camera_node.launch \
camera_name:=camera image_topic:=/camera/image \
camera_info_topic:=/camera/camera_info
```

where `image_topic` and `camera_info_topic` are the real topics on which the camera drivers publish their data. If everything is fine, we will see in our terminal:

```
[/Gemini/camera_node] All messages received.
[/Gemini/camera_node/get_device_info] Service started.
[/Gemini/camera_node/extract_checkerboard] Action server started.
```

At this point, `Phoenix` and `Gemini` are working as expected, we can therefore move to `Lyra`. Here we must let the master node know the network. We create a file called `network.yaml` in the `multisensor_calibration/conf` directory, open our favourite text editor, and fill `network.yaml` with the two PCs and the two cameras connected to them:

```
# Network configuration
network:
  - pc: "Phoenix"
    devices: ["camera"]
  - pc: "Gemini"
    devices: ["camera"]
```

Note that the strings in the `devices` arrays exactly match the value of the argument `camera_name` we set when launching `camera_node.launch`. We must then define the calibration pattern, i.e. the checkeboard, that we will use during the calibration procedure. So, let's create a file named `checkerboard.yaml` in the `multisensor_calibration/conf` directory and fill it with our checkerboard parameters:

```
# Checkerboard configuration
checkerboard:
  cols: 6
  rows: 5
  cell_width: 0.12
  cell_height: 0.12
```

We can now start the calibration procedure:

```
roslaunch multisensor_calibration master_node.launch
```

If all the nodes are launched correctly we'll have an output similar to the one below:

```
[/Lyra/master_node] Connected to [/Phoenix/camera_node/
    get_device_info] service.
[/Lyra/master_node] Connected to [/Gemini/camera_node/
    get_device_info] service.
[/Lyra/master_node] Connected to [/Phoenix/camera_node/
    extract_checkerboard] action server.
[/Lyra/master_node] Connected to [/Gemini/camera_node/
    extract_checkerboard] action server.
[/Lyra/master_node] Getting device infos...
[/Lyra/calibration] Sensor [/Phoenix/camera] added.
[/Lyra/calibration] Sensor [/Gemini/camera] added.
[/Lyra/master_node] Initialization complete.
```

We can then start the data acquisition phase. So, we take our checkerboard and move it around letting all the sensors see it. Every time we are in a good position we can publish an empty message on the topic `/Lyra/master_node/acquisition` to get one instance of the checkerboard (if visible) from each sensor:

```
rostopic pub /Lyra/master_node/acquisition std_msgs/Empty -1
```

Note that the, instead of publishing a message each time we need, we can let the publisher run at a fixed rate, substituting `-1` with the desired rate `-r <rate>`. In this case, we must pay attention not to move the checkerboard too quickly, to avoid blur calibration errors due to the non perfect synchronization of the sensors.

We can monitor the whole calibration procedure via Rviz (Fig. 4). It is sufficient to set the `world` frame as fixed frame, add the *tf* view and a *marker* view on topic `/Lyra/master_node/markers` and every time the checkerboard is detected or a sensor pose is estimated we will see it on the screen. In Fig. 4 some screenshots acquired during the calibration are shown. Before finishing the calibration, we lay the checkerboard on the floor and publish:

```
rostopic pub /Lyra/master_node/action std_msgs/String "begin
    plane" -1
```

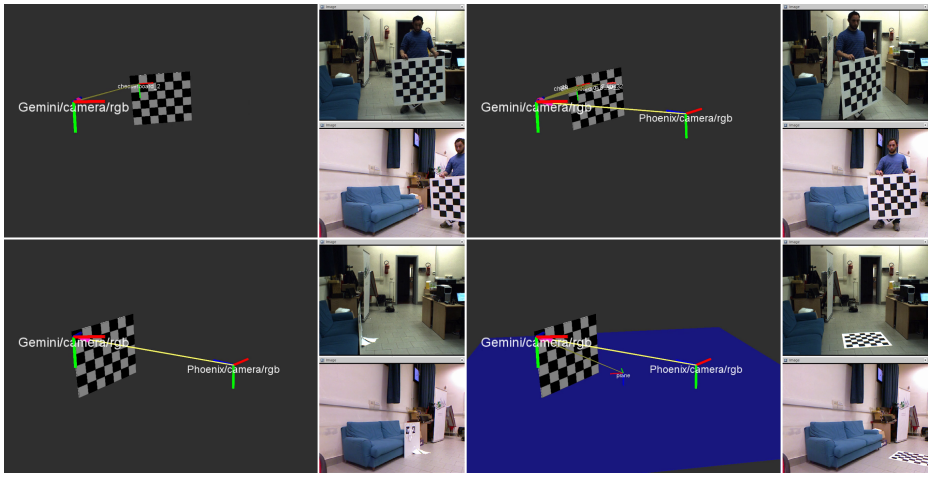

Fig. 4. Screenshots acquired during the calibration procedure. **Top-left:** as soon as one sensor detects the checkerboard pattern, it becomes part of the `tf` tree and its pose is published. **Top-right:** then, every sensor that sees the checkerboard is added to the tree. **Bottom-left:** as new detections arrive, the pose of the sensors are refined with the optimization algorithm, the checkerboard visualized is the last one. **Bottom-right:** when the program is asked to estimate a plane, its pose is published and can be visualized using Rviz.

From now on, the calibration algorithm assumes that all the checkerboards are lying on the same plane (see Sect. 4.3). We perform some acquisitions with the checkerboard lying on the floor and then publish an `end plane` instruction:

```
rostopic pub /Lyra/master_node/action std_msgs/String "end plane"
    -1
```

Finally, to get the results of the calibration, we use the service `get_results` offered by the master node. On a terminal we run:

```
rosservice call /Lyra/master_node/get_results
```

and get the estimated poses, similar to the ones below:

```
poses:
  - frame_id: '/world'
    child_frame_id: '/Gemini/camera'
    pose:
      position: {x: 0.0, y: 0.0, z: 0.0}
      orientation: {x: 0.0, y: 0.0, z: 0.0, w: 1.0}
  - frame_id: '/world'
    child_frame_id: '/Phoenix/camera'
    pose:
      position: {x: 1.12064, y: 0.321081, z: 0.565662}
      orientation: {x: 0.0471913, y: -0.377825, z: -0.0340694, w:
          0.924046}
```

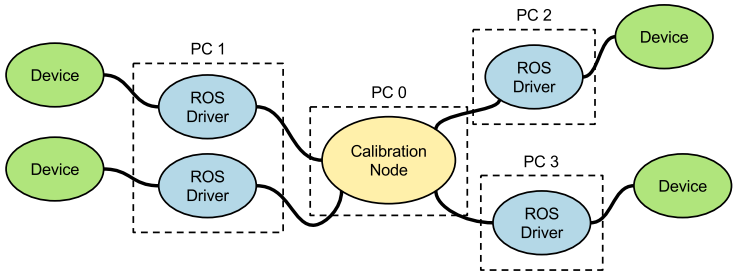

Fig. 5. Typical approach of network calibration algorithms. All the driver nodes are directly connected to a central node that performs the computation. The bandwidth usage is high.

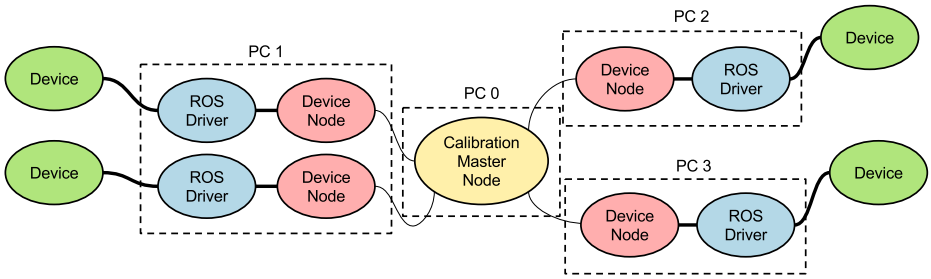

Fig. 6. Our approach to the network calibration procedure. The data provided by a device are elaborated by a node on the same PC and only the necessary calibration features are sent to the calibration node. Here, the thin lines that connect the master node with the device nodes mean that the quantity of data on the network is limited, with respect to the quantity of the typical approach (Fig. 5).

8 ROS Package

8.1 Architecture

The main purpose of the here-presented package is to allow the calibration of all the sensors (for now cameras and Kinect-like depth sensors) within a ROS network, no matter where they are. That is, suppose to have a set of sensors *distributed* in a PC network as in Fig. 1, the typical approach for the calibration procedure is to develop a *calibration node* that grabs the data generated by all the sensors, elaborate them and then estimate the sensors' rigid displacement in the scene (Fig. 5). This approach is clearly not scalable: more sensors means more bandwidth yet more computational power needed. To overcome such problem, we propose a different, distributed, architecture that allows us to both drastically reduce the bandwidth usage and distribute the computational cost over the network. In fact, we separate the data analysis from the calibration procedure. As depicted in Fig. 6, there are two sorts of calibration nodes: *device nodes* and the *master node*. A device node is responsible for elaborating the data provided

by its device, it extracts the corners from every image, creates a message and sends it to the master node that executes the calibration procedure.

Remembering that ROS device drivers typically stream data at a defined frame-rate, with the proposed architecture we are able to work in a request-reply way: it is the master node that asks for new data when needed. In particular, since ROS service calls are blocking, the communication relies on the `actionlib` stack.

8.2 Device Node

Node Parameters. A device node is responsible for getting the data from a device and, upon request, elaborate and send them to the master node. We first need to distinguish between the word *device* and *sensor*. We define a device as an item that can be connected to a PC, while a sensor, rather physical or virtual, is the item whose pose will be estimated in the calibration procedure. For example, a Kinect is a device composed by three different sensors: the RGB camera, the IR camera and a virtual depth sensor [7].

So, from a ROS perspective, a device node subscribes to the image and/or depth topics of the device sensors and keeps listening to an action topic until a request from the master node arrives. Then, the last images received are processed and the extracted calibration features are packed into a message and sent back to the master.

To explain how to configure and run a device node, we suppose we have to create a launch file for a Kinect v1. We first need to set the name of the device and its serial in case two or more Kinects are launched on the same PC:

```
<?xml version="1.0"?>
<launch>
  <arg name="device_name"   default="kinect1" />
  <arg name="device_serial" default="#1" />
```

Then, to avoid conflicts between nodes launched from different PCs, we group everything inside the namespace `$ROS_PC_NAME`:

```
  <group ns="$(env ROS_PC_NAME)">
```

Now we include the launcher for the Kinect driver. The argument `camera` let us define the namespace for all the topics published by the driver as well as the reference frames in the Kinect messages: here we add the `_driver` suffix just to avoid confusion. Note also that we have set the argument `depth_registration` to `false`: it avoids the driver to perform any factory-defined modifications to the point cloud.

```
    <include file="$(find openni_launch)/launch/openni.launch">
      <arg name="camera" value="$(arg device_name)_driver" />
      <arg name="device_id" value="$(arg device_serial)" />
      <arg name="depth_registration" value="false" />
      <arg name="publish_tf" value="false" />
    </include>
```

The last node we need to add is our device node. The package `multisensor_calibration` provides the node as a binary called `device_node`. We rename it to match our current device, paying attention that, for communication reasons, the node name needs the suffix `_node`.

```
<node pkg="multisensor_calibration" type="device_node"
      name="$(arg device_name)_node" output="screen">
```

We then define the Kinect sensors that we want to calibrate. They have to be defined within the `~device` namespace, in particular:

name – sets the device name, only for logging purposes;
sensors – defines the list of sensors to calibrate, it is divided into:
  intensity – the sensors that will be treated as pinhole cameras;
  depth – the sensors that will be treated as depth sensors;
<sensor> – sets, for each sensor defined in **sensors**:
  frame_id – its unique frame id;
  error – the error polynomial defined in (9), where [*depth sensors only*]:
    min_degree/max_degree – the minimum and maximum degree of the polynomial (in the example below $\sigma(z) = c_0 \cdot z^0 + c_1 \cdot z^1 + c_2 \cdot z^2$);
    coefficients – the polynomial coefficients c_i;
transforms – defines the known transforms between the sensors:
  <sensor> – the child sensor;
    parent – the parent sensor, that is, the frame to which the transform is defined;
    translation – the translation between the two sensors;
    rotation – the quaternion defining the rotation between the two sensors.

```
<rosparam param="device" subst_value="true">
  name: "$(arg device_name)"
  sensors:
    intensity: ["rgb"]
    depth: ["depth"]
  rgb:
    frame_id: "/$(env ROS_PC_NAME)/$(arg device_name)/rgb"
  depth:
    frame_id: "/$(env ROS_PC_NAME)/$(arg device_name)/depth"
    error:
      min_degree: 0
      max_degree: 2
      coefficients: [0.0, 0.0, 0.0035]
  transforms:
    depth:
      parent: "rgb"
      translation: {x: -0.025, y: 0.0, z: 0.0}
      rotation: {x: 0.0, y: 0.0, z: 0.0, w: 1.0}
</rosparam>
```

Finally, we have to connect the device node to the topics published by the driver. We use the *remapping* feature of ROS to set the sensor nodes listen to the right topics. Both for depth sensors and cameras, the default topics they listen

to are of the form `~/device/<sensor>/<topic type>`, where `<topic type>` is `image` for either images or depth images, `camera_info` for the camera calibration parameters and `cloud` for the point clouds:

```
      <remap from="~device/rgb/image"
             to="$(arg device_name)_driver/rgb/image_color"/>
      <remap from="~device/rgb/camera_info"
             to="$(arg device_name)_driver/rgb/camera_info"/>
      <remap from="~device/depth/cloud"
             to="$(arg device_name)_driver/depth/points" />
      <remap from="~device/depth/camera_info"
             to="$(arg device_name)_driver/depth/camera_info" />
    </node>
  </group>
</launch>
```

8.3 Master Node

Node Parameters. Let's take a look to the launch file for the master node to see the parameters it needs to run. The launch file for the master node is simple:

```
<?xml version="1.0"?>
<launch>
  <arg name="network_file" default="$(find
      multisensor_calibration)/conf/network.yaml" />
  <arg name="checkerboard_file" default="$(find
      multisensor_calibration)/conf/checkerboard.yaml" />
  <group ns="$(env ROS_PC_NAME)">
    <node pkg="multisensor_calibration" type="master_node"
          name="master_node" output="screen">
      <param name="network_file" value="$(arg network_file)" />
      <rosparam command="load" file="$(arg checkerboard_file)" />
    </node>
  </group>
</launch>
```

We first need to set the file containing the network description, `network_file`, to let the master node know the sensors to be calibrated. The network configuration is expected to be in a yaml file of the form

```
# Network configuration
network:
  - pc: "<ROS_PC_NAME_1>"
    devices: ["<DEVICE_NAME_1>", "<DEVICE_NAME_2>"]
  - ...
  - pc: "<ROS_PC_NAME_N>"
    devices: ["<DEVICE_NAME_1>", ..., "<DEVICE_NAME_N>"]
```

and typically stored in the `multisensor_calibration/conf` directory of the master PC. Here, each `pc` must match any of the names previously given to the PCs via the `export` command, while the strings in the `devices` array are the device names. They have to match the first part of a device node, that is, all but the suffix `_node`. Note that each device node is expected to be reachable in the network using the PC name as a namespace. As an example, the device

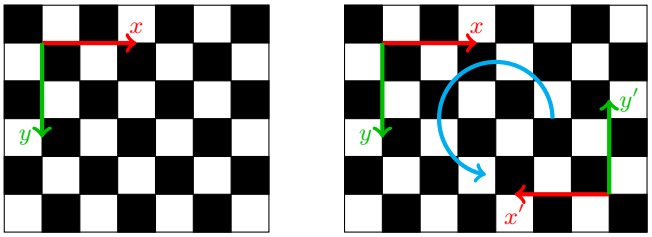

Fig. 7. The reference frame of a checkerboard is located internally with respect to one of the black corner-cells (if present), according to the checkerboard size, that is, with the x-axis along the columns and the y-axis along the rows. **Left:** the reference frame of a 5×6 checkerboard has only one possible location. **Right:** here, due to a rotational symmetry, the reference frame of the 5×7 checkerboard can be positioned in two different locations, leading to pose estimation problems.

node `camera` that runs in the PC named `Gemini`, is expected to be a node called `/Gemini/camera_node`.

The second parameter is the `checkerboard_file`. It must be a yaml file and contain the checkerboard pattern specifications:

```
# Checkerboard configuration
checkerboard:
  cols: <internal corners along the x dimension>
  rows: <internal corners along the y dimension>
  cell_width: <cell size along the x dimension in meters>
  cell_height: <cell size along the y dimension in meters>
```

Because of symmetry (see Fig. 7), it is mandatory that one of `cols` and `rows` is odd and the other even, no matter which. Otherwise two different sensors can assign to the same checkerboard two different reference frames, invalidating the calibration procedure. In fact, according to our experience, the OpenCV corner detector starts enumerating the corners from one of the black corner-cells (if present), in row-major order. We rely on this order to set the reference frame of the checkerboard: the x-axis along the columns and the y-axis along the rows.

Services and Messages. As already shown in Sect. 7, it is possible to interact with the master node with messages and services. The most important topic the node is listening on is `~acquisition`. Publishing a string on such topic would result in a data acquisition. Note that, the message queue on the node is set to 1, so even if the node receives multiple requests while still elaborating previous data, only the last one will be taken into account. Note also that `rostopic pub` permits to publish messages at a defined rate with the flag `-r <rate>`. Users must pay attention to use such feature since the sensors may be not perfectly synchronized. In fact, if the data are acquired while the pattern is still moving, the resulting calibration could be wrong.

The node is also listening to an action topic named `~action`. It is used to send special calibration commands in form of strings. For now the only two

commands accepted are `begin plane` and `end plane`. The former command tells the calibration algorithm that, from that instant on, the checkerboards are all lying on the same plane (see Sect. 4.3). The latter, instead, makes the algorithm go back to its standard behavior.

Finally, to get the results of the calibration, the master node offers a service called `~get_results`. It can be invoked with an empty request and returns a vector of messages of type `calibration_msgs/ObjectPose`. In the future we envision to improve this service with some options like:

- ask for the poses of checkerboards and/or planes;
- set a fixed transform between the world and a sensor in the request, the response will be filled with the poses transformed according to it.

9 Conclusions

We have presented a ROS package that lets users calibrate networks of sensors composed by cameras and Kinect-like depth sensors. A checkerboard pattern is used to estimate the relative poses between sensors and everything is optimized with a state-of-the-art non-linear least squares solver [1]. The choice of using ROS as the developing framework gives our implementation lots of advantages with respect to, for example, Matlab-based toolboxes [21]. The main advantage is the way the sensors and their synchronization are managed. On the other side, one of the main drawbacks of the presented approach is that it highly depends on the intrinsic parameters provided. If they are not well estimated, the calibration results will not be accurate. In the future we can think of adding the possibility to refine also the intrinsic parameters of cameras while estimating their pose.

We also think that it will be really useful to provide a tool to assess the quality of the obtained calibration, so that users can verify if the obtained results are reliable or not. Another important improvement will be at code level. In our idea, the package will be part of a *calibration ecosystem* for ROS, where contributors can implement their own calibration algorithms or contribute to the existing ones by adding, for example, new sensors. Hence, it will be necessary to define some standard interfaces between nodes and classes to lower the entry barrier for new developers and let the ecosystem grow.

Acknowledgments. The authors would like to thank Prof. Mohamed Chetouani, Salvatore Maria Anzalone and Stéphane Michelet from Université Pierre-et-Marie-Curie (UPMC) and the Institut des Systèmes Intelligents et de Robotique (ISIR) for their support and help.
The authors would also like to thank Jeff Burke, Alexander Horn and Randy Illum from University of California, Los Angeles (UCLA) for the extensive collaboration in designing and testing the calibration methods during the development of OpenPTrack [16]. OpenPTrack has been sponsored by UCLA REMAP and Open Perception. Key collaborators include the University of Padova and Electroland. Portions of the work have been supported by the National Science Foundation (IIS-1323767).

References

[1] Agarwal, S., Mierle, K., et al.: Ceres solver. http://ceres-solver.org

[2] Auvinet, E., Meunier, J., Multon, F.: Multiple depth cameras calibration and body volume reconstruction for gait analysis. In: Information Science, Signal Processing and their Applications (ISSPA), 2012 11th International Conference on. pp. 478–483 (July 2012)

[3] Basso, F., Levorato, R., Menegatti, E.: Online calibration for networks of cameras and depth sensors. In: Proceedings of the 12th Workshop on Non-classical Cameras, Camera Networks and Omnidirectional Vision (OMNIVIS). Hong Kong, China (June 2014)

[4] Basso, F., Munaro, M., Michieletto, S., Menegatti, E.: Fast and robust multi-people tracking from RGB-D data for a mobile robot. In: Proceedings of the 12th Intelligent Autonomous Systems (IAS) Conference. vol. 193, pp. 265–276. Jeju Island, Korea (June 2012)

[5] Basso, F., Pretto, A., Menegatti, E.: Unsupervised intrinsic and extrinsic calibration of a camera-depth sensor couple. In: Robotics and Automation (ICRA), 2014 IEEE International Conference on. pp. 6244–6249. Hong Kong, China (June 2014)

[6] Bradski, G.: The opencv library. Dr. Dobb's Journal of Software Tools (2000)

[7] Canessa, A., Chessa, M., Gibaldi, A., Sabatini, S.P., Solari, F.: Calibrated depth and color cameras for accurate 3D interaction in a stereoscopic augmented reality environment. Journal of Visual Communication and Image Representation 25(1), 227 – 237 (2014)

[8] Di Cicco, M., Iocchi, L., Grisetti, G.: Non-parametric calibration for depth sensors. In: Proceedings of the 13th International Conference on Intelligent Autonomous Systems. (IAS-13). Padova, Italy (2014)

[9] Fischler, M.A., Bolles, R.C.: Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. Communications of the ACM 24(6), 381–395 (June 1981)

[10] Furgale, P., Rehder, J., Siegwart, R.: Unified temporal and spatial calibration for multi-sensor systems. In: Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on. pp. 1280–1286 (Nov 2013)

[11] Guennebaud, G., Jacob, B., et al.: Eigen. http://eigen.tuxfamily.org (2010)

[12] Hartley, R.I., Zisserman, A.: Multiple View Geometry in Computer Vision. Cambridge University Press, second edn. (2004)

[13] Le, Q., Ng, A.: Joint calibration of multiple sensors. In: Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on. pp. 3651–3658 (October 2009)

[14] Levorato, R., Pagello, E.: Probabilistic 2D acoustic source localization using direction of arrivals in robot sensor networks. In: Brugali, D., Broenink,

J.F., Kroeger, T., MacDonald, B.A. (eds.) Simulation, Modeling, and Programming for Autonomous Robots. Lecture Notes in Computer Science, vol. 8810, pp. 474–485. Springer International Publishing (2014)
[15] Munaro, M., Basso, F., Menegatti, E.: Tracking people within groups with RGB-D data. In: Proceedings of the International Conference on Intelligent Robots and Systems (IROS). pp. 2101–2107. Vilamoura, Portugal (October 2012)
[16] Munaro, M., Horn, A., Illum, R., Burke, J., Rusu, R.B.: OpenPTrack: People tracking for heterogeneous networks of color-depth cameras. In: IAS-13 Workshop Proceedings: 1st Intl. Workshop on 3D Robot Perception with Point Cloud Library. pp. 235–247. Padova, Italy (July 2014)
[17] Smisek, J., Jancosek, M., Pajdla, T.: 3D with kinect. In: Computer Vision Workshops (ICCV Workshops), 2011. ICCVW 2011. IEEE International Conference on. pp. 1154–1160 (2011)
[18] So, E., Basso, F., Menegatti, E.: Calibration of a rotating 2D laser range finder using point-plane constraints. Journal of Automation, Mobile Robotics & Intelligent Systems 7(2), 30–38 (2013)
[19] Teichman, A., Miller, S., Thrun, S.: Unsupervised intrinsic calibration of depth sensors via SLAM. In: Proceedings of Robotics: Science and Systems. Berlin, Germany (June 2013)
[20] Unnikrishnan, R., Hebert, M.: Fast extrinsic calibration of a laser rangefinder to a camera. Tech. rep., Carnegie Mellon University (2005)
[21] Warren, M., McKinnon, D., Upcroft, B.: Online calibration of stereo rigs for long-term autonomy. In: Robotics and Automation (ICRA), 2013 IEEE International Conference on. Karlsruhe, Germany (2013)
[22] Zhang, C., Zhang, Z.: Calibration between Depth and Color Sensors for Commodity Depth Cameras. In: Multimedia and Expo (ICME), 2011 IEEE International Conference on. pp. 1–6 (2011)