Point Clouds With Color: A Simple Open Library for Matching RGB and Depth Pixels from an Uncalibrated Stereo Pair

Jordan Nowak, Philippe Fraisse, Andrea Cherubini, Jean-Pierre Daurès

Point Clouds With Color: A Simple Open Library for Matching RGB and Depth Pixels from an Uncalibrated Stereo Pair

Jordan Nowak[1,2], Philippe Fraisse[1], Andrea Cherubini[1], Jean-Pierre Daures[2]

***Abstract*— Current day robots often rely – for visual perception – on the coupling of two cameras: one for color and one for depth. While for custom RGB-D cameras, the manufacturer takes care of aligning the two images, this is not done when two commercial cameras are coupled (e.g., on the Pepper robot) without having been calibrated beforehand. In this article, we present a simple open library for reconstructing the 3D position of RGB pixels without knowing the parameters of the two cameras. The library requires a simple preliminary calibration step based on pixel-to-pixel matching, and then automatically reconstructs 3D colored point clouds from a given set of pixels in the RGB image. The source code is available at the following link `https://github.com/jordan-nowak/OpenHSML`.**

I. Introduction

A. RGB-D perception

For many technological applications (e.g., in the fields of robotics, navigation, cartography, augmented reality...), it is essential to perceive effectively the environment. Multimodal perception refers to processes which relate two or more senses (sight, hearing, touch...). This process may aid in building a representation of the environment. In this article, we focus on multimodal perception using an RGB and a depth camera. We propose a library that finds the point correspondences between an RGB and depth image without using a checkerboard and without knowledge of the camera parameters. This open-source library is called OpenHSML: Open-source Hybrid Stereovision Matching Library [9].

Depth cameras are designed to output a three-dimensional representation of a scene. These sensors use either active or passive methods. In the first case, a direct depth measurement is obtained via structured light or laser beams, such as TOF (Time Of Flight). The second relies on the principle of stereovision. It is considered passive because it does not use direct measurements. It estimates the depth from two images taken simultaneously from different points of view.

We have tested our library with depth cameras using structured light projection. Yet, the library can work with any pair of RGB and depth images of the same scene, including TOF cameras. A grayscale image can also be used in place of the RGB image. Figure 1 shows the output of OpenHSML as it estimates the 3D position (in depth camera frame) of points selected in the RGB image. In our case, the world frame coincides with the depth camera frame.

[1]LIRMM, Univ Montpellier, CNRS, Montpellier, France. Emails: `firstname.lastname@lirmm.fr`

[2]Clinique Beau Soleil, Montpellier, France. Emails: `jean-pierre.daures@umontpellier.fr`

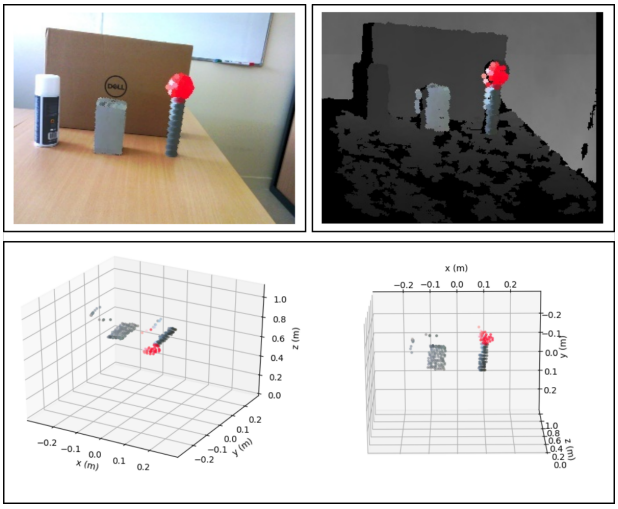

Fig. 1. Output generated by our library from the images obtained by the Pepper robot cameras. Top left: original RGB image with points manually selected from the scene's objects. Top right: the points are matched to the corresponding ones in the depth image. Bottom: two different views of the colored point cloud in 3D.

B. Motivation

A depth camera using structured light projection technology is often coupled with a camera providing a color image. This coupling is called an RGB-D camera. The color image is directly associated with the corresponding depth image, by an embedded library designed and provided by the RGB-D camera manufacturer. The most well-known camera of this type is the Kinect. It was created in 2010 by Microsoft to make users interact without a joystick with the video game console Xbox 360. Other RGB-D cameras include the RealSense[1] series from Intel and the Xtion series from Asus[2] . The manufacturers propose libraries, often open-source, to allow their customers to use the cameras and/or program their own applications. For the Asus cameras, the manufacturers propose to use OpenNI 2 SDK [10]. For the Kinect, Microsoft proposes a specific SDK [1]. Intel also developed an open-source library for RealSense [5].

These libraries use the camera's physical properties to align the RGB and depth images. These physical properties are the RGB-D intrinsic and extrinsic parameters. The intrinsic parameters define the internal properties of each camera (such as focal length, optical center, image distortion...). The extrinsic parameters define the position and orientation of one camera with respect to the other one. To obtain these parameters, the user can refer to the datasheet of the

[1]`https://www.intelrealsense.com/`

[2]`https://www.asus.com/me-en/3D-Sensor/`

manufacturer. However, in some cases, this datasheet may be non-existent, difficult to access, imprecise compared to the reality in the field, or even incomplete.

This is typically the case if one wants to couple a depth camera (regardless of the technology used) with a clearly dissociated RGB camera. An example of such coupling is shown in section IV-A, where we couple a structured light depth camera with the webcam of a computer. Another example is the humanoid robot Pepper. This robot – in its version 1.8a – features an Xtion camera from Asus along with an RGB camera. As with the Kinect, we aim at aligning the depth image with the RGB image. Pepper's SDK[3] does not allow this. One could use the intrinsic and extrinsic parameters of the system, which are available online in the Pepper technical documentation[4]. Yet, these are theoretic and can be quite different from reality. This type of problems has inspired us in developing the library OpenHSML. We contribute by proposing this open-source library allowing the user to match an RGB image and depth image observing a same scene, using the stereovision principle and without knowledge of the camera parameters.

II. Overview of our Approach

In this section, we present the methodology used to solve our problem. First, we present the previous works addressing this type of problem. Then, we recall the principles of stereovision. Finally, we present our method.

A. Related work

One of the simplest libraries for calibrating one or more RGB or gray-scale cameras is OpenCV: Open Computer Vision Library[5] [8], [15]. This library is widely used by the scientific community in the field of vision. Camera calibration is possible in OpenCV, via a test pattern whose properties are perfectly known [3]. The test pattern can be built in different ways like a classic black-white chessboard, symmetric or asymmetric pattern of circles. It can also be a set of easily detectable markers, as ArUco markers [2], [12], [22]. These markers allow an improvement over the other methods, since calibration is possible despite partial occlusions of the test pattern. In all cases, the algorithm will retrieve the intrinsic parameters of the camera. It is also possible to calibrate a stereovision system using two RGB cameras. Since the marker is known, it is possible to estimate their position and orientation to the camera. This makes it easy to have extrinsic parameters of the cameras. However, a depth camera cannot be calibrated directly with this method because the checkerboard is not detectable with the proposed patterns (ArUco markers or classic checkerboard). Indeed, the image provided by this camera does not allow the patterns to be viewed in the same way as with an RGB camera.

Structured light technologies include a projector and an infrared (IR) camera. To obtain the depth image, the embedded software estimates the disparity between the projected pattern and the pattern observed by the IR camera. Some researchers have used conventional calibration to find the camera parameters and to estimate the distortion of the depth image. This, to improve the alignment of the two images. For example, Khoshelham, Elberink [19], and Smisek et al. [23] propose to calibrate the depth sensor of the Kinect camera. Their calibration reduces lens and depth distortion. To find the depth camera's parameters with conventional camera calibration, the authors first obstruct the camera projector. Then, with the use of a halogen light source, they illuminate the test pattern. This simplifies the extraction of the test pattern from the infrared camera image. It is possible to use this method with ROS: Robot Operating System[6] with tutorial [6]. This idea works, but only for this type of technology. We aim at calibrating the camera pair directly from the depth image and not from the infrared camera.

Wu et al. [24] propose a different approach to calibrate the Kinect, via a semi-transparent test pattern. This can be easily visible in both images without using a halogen light source. Indeed, with a digital tool, the user shows where the test pattern is in both the RGB and in the depth image. Then, s/he can obtain the parameters of the cameras.

With depth sensors, it is possible to have an error that may be due to distortion or a systematic measurement error of the sensor. Various works have been carried out to date to correct this type of error by calibrating the system from a drawing [13], [18], [21], [25]. They all offer the use of a checkerboard attached to a surface such as a simple wall, a swivel table, or a table. The checkerboard provides a ground truth after calibration of the RGB camera. This way, authors attenuate the depth distortion by estimating an undistorted model. Indeed, they seek intrinsic and extrinsic RGB-D camera parameters using only one plane with a checkerboard.

Herrera et al. provide a Matlab toolbox [7] to calibrate a pair of camera RGB-D and to take a distortion correction by identifying parameters. Basso et al. provides a C++ toolbox implementable on ROS [11]. They calibrate also an RGB-D camera system to reduce distortion in depth image and to refine the constructor parameters.

Compared with these libraries, we do not calibrate the cameras and do not search to determine parameters. Indeed, we propose an alternative method to link data from the two cameras without knowing the system's nominal parameters. We use the principles of stereovision with the epipolar lines.

Liu et al. [20] is the more recent work. They use a sphere to calibrate their system RGB-D. Their framework is capable to detect the sphere in the two images. It estimates the intrinsic and extrinsic parameters simultaneously and also corrects the depth measurement error. However, compared with us, they do not provide a toolbox.

It is also interesting to mention the work of Bauer et al. on improving the vision of the Pepper robot [14]. In their

[3] https://www.softbankrobotics.com/emea/en/support/pepper-naoqi-2-9/downloads-softwares

[4] http://doc.aldebaran.com/2-5/family/pepper_technical/video_overview.html

[5] https://opencv.org/

[6] https://www.ros.org/

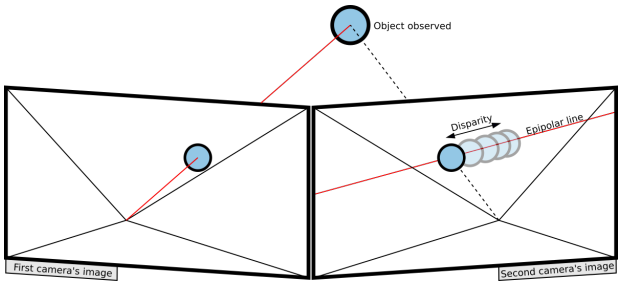

Fig. 2. Principle of stereovision: disparity observed between two color images looking at the same object but from different points of view.

work, they seek to improve 3D perception by fusion method between the depth image raw data with the monocular depth prediction from the RGB image.

All our developments are open-source and provided in a library which automatically matches relevant pixels from an RGB and a depth camera, to output the corresponding "colored point cloud".

B. Principles of stereovision

Before going into the general explanations of our algorithm, let us present the strategy used to tackle this problem. The configuration of our system is as follows: an RGB camera and a depth camera observe a common scene from two different points of view. This arrangement allows us to use the principles of stereovision[17]. Indeed, stereoscopy exploits two cameras to estimate the scene depth from the disparity between the two images (see Fig. 2).

An RGB-D camera provides a depth and a color image, but the two are not linked. We investigated a strategy to detect the same object in both images. Yet, the existing libraries require knowing both the intrinsic and extrinsic parameters. In our work, we try to go beyond this requirement. Nevertheless, we need two matrices well known in this field: the *Fundamental matrix* $\mathbf{F}$ and the *Projection matrix* $\mathbf{P}$. Both can be defined without the use of the nominal parameters, but if they are not known beforehand, a *calibration step* (also implemented in our library) is needed.

For each point in one of the two images, there exists a corresponding epipolar line in the other image (figure 2 show an example of this line). This gives us a first information: in the second image, the point is on this line. However, this information is not sufficient alone, since it gives infinite solutions along this line. The correct solution can be determined knowing the exact depth of this point.The equation of the epipolar line of a point can be determined using the *Fundamental matrix* $\mathbf{F}$. Let us name $\mathbf{m_L}$ (respectively, $\mathbf{m_R}$) the coordinates of the point respectively in the left camera (right) image. The coefficients a, b and c define the epipolar line equation as $au + bv + c = 0$. These can be obtained for each image by the following relation:

$$\begin{cases} [a_L, b_L, c_L]^\top = \mathbf{F} * \mathbf{m_R} \\ [a_R, b_R, c_R]^\top = \mathbf{F^T} * \mathbf{m_L} \end{cases} \quad (1)$$

Matrix $\mathbf{F}$ is defined from projection matrices of two cameras ($\mathbf{P_R}$ and $\mathbf{P_L}$) and epipolar point $\mathbf{e_L} = [e_{Lx}, e_{Ly}, e_{Lz}]^\top$ (projection of camera's origin left in image plane right) via

$$\mathbf{F} = \mathbf{e_L}^\times \mathbf{P_L} \mathbf{P_R}^+, \quad (2)$$

with $\mathbf{P_L}$ and $\mathbf{P_R}$ expressed as

$$\begin{cases} \mathbf{m_L} = \mathbf{P_L} [x, y, z]^\top \\ \mathbf{m_R} = \mathbf{P_R} [x, y, z]^\top \end{cases}$$

and

$$\mathbf{e_L}^\times = \begin{bmatrix} 0 & -e_{Lz} & e_{Ly} \\ e_{Lz} & 0 & -e_{Lx} \\ -e_{Ly} & e_{Lx} & O \end{bmatrix}.$$

Furthermore, we use the $\mathbf{P}$ matrix. For any 3D point of world frame coordinates $\mathbf{p} = [x, y, z]^T$, that allow to find its image projection $\mathbf{m} = [u, v]^T$, up to a scale factor λ:

$$\lambda \begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix} = \mathbf{P} \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix} = \begin{bmatrix} P_{11} & P_{12} & P_{13} & P_{14} \\ P_{21} & P_{22} & P_{23} & P_{24} \\ P_{31} & P_{32} & P_{33} & P_{34} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix} \quad (3)$$

Matrix $\mathbf{P}$ can be obtained from the intrinsic parameters (contained in the camera matrix $\mathbf{K}$) and extrinsic parameters (rotation matrix $\mathbf{R}$ and translation vector $\mathbf{t}$ between camera and world frame) via:

$$\mathbf{P} = \mathbf{K} * [\mathbf{R}|\mathbf{t}], \quad (4)$$

with

$$\mathbf{K} = \begin{bmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}. \quad (5)$$

In (5), f_x and f_y are the focal lengths along the x and y-axis, c_u, and c_v the coordinates in pixels of the optical center in the image plane, and γ is the skew between the axes (often set to 0).

In Sec. II-D, we will explain in more detail how we obtain these two matrices. In the next section, we explain how we use them to obtain "colored point clouds" from an pair of RGB and depth cameras.

C. Point matching algorithm

Here, we outline the method used in our library; the corresponding pseudocode is given in Algorithm 1.

The first step is the calibration if it has not been done before. A function asks the user to enter the parameters necessary for the calibration (w, h, H_{FOV}, V_{FOV}, c_u and c_v) to the line 1 in Algorithm 1. Then, it is asked to match points in the calibration images and a function estimates, at the end of this step, the two matrices $\mathbf{F}$ and $\mathbf{P}$ (line 2). Then, the algorithm saves all these parameters, and these two matrices in the *Parameter file* (lines 3 and 4).

The user must input n points of the RGB image in pixel coordinates ($\mathbf{m_i} = [u_i, v_i]^T$); these are represented as a vector (**rgb_px**). The library return the Cartesian coordinates of the corresponding 3D points in the depth camera frame.

To this end, the library needs the fundamental and projection matrices, $\mathbf{F}$ and $\mathbf{P}$. These are given in a *Parameter file*

▷ **Input:** rgb_px = { $\mathbf{m_1}$, $\mathbf{m_2}$, ..., $\mathbf{m_n}$ }
▷ **Output:** depth_pt = { $\mathbf{p_1}$, $\mathbf{p_2}$, ..., $\mathbf{p_n}$ }
if calibration_step **then**
1 $H_{FOV}, V_{FOV}, c_u, c_v, w, h$ = SetParameters();
2 $\mathbf{F}, \mathbf{P}$ = SetCalibration();
3 $ParameterFile \leftarrow H_{FOV}, V_{FOV}, c_u, c_v, w, h$;
4 $ParameterFile \leftarrow \mathbf{F}, \mathbf{P}$;
end
5 $\mathbf{F}, \mathbf{P} \leftarrow$ Parameter File;
for k = 1,2, ..., n **do**
6 $[a, b, c]^\top$ = computeEpipolarLine($\mathbf{m_k}$, $\mathbf{F}$);
7 $\mathbf{sp}$ = SampleEpipolarLine(a, b, c, w, h, s);
8 $\mathbf{pj}$ = ProjectPointInRGB($\mathbf{sp}$, $\mathbf{P}$);
9 index = GetNearestPoint($\mathbf{pj}$, $\mathbf{m_k}$);
10 $\mathbf{l_k}$ = $\mathbf{pj}[\mathbf{index}]$;
11 $\mathbf{p_k}$ = Get3DPointCoordinates($\mathbf{pj}[\mathbf{index}]$);
end

Algorithm 1: Pseudocode of the algorithm returning, for a given set of pixels in the RGB image, the Cartesian coordinates of the corresponding 3D points in the depth camera frame.

(in YAML format), which is loaded at initialization (line 5 in Algorithm 1). To generate this file, if s/he does not know the values of $\mathbf{F}$ and $\mathbf{P}$, the user should calibrate the stereo pair by manually matching as many points as possible in the two images. This step is explained in Sec. II-D.

We use the epipolar lines in the depth image corresponding to the RGB pixels. Each epipolar line is defined by its coefficients a, b and c such that the epipolar line equation is: $au + bv + c = 0$. To determine a, b and c, we use OpenCV function computeCorrespondEpilines()[7], which takes as input the fundamental matrix $\mathbf{F}$ (line 6).

Then, we sample the segment of the epipolar line within the depth image bounds (line 7). This function needs the a, b, and c coefficients of the epipolar line to be sampled, the size of the depth image given by w and h and the sampling step s. To determine which depth image pixel on this sampled epipolar line corresponds to the desired point, we project all these points in the RGB image (line 8). This is done via the $\mathbf{P}$ matrix and equation (3); since these pixels are in the depth image, we can determine their 3D coordinates in the depth camera frame. There exist two types of depth images:

- Some depth cameras directly give the point 3D coordinates in the depth camera frame, i.e. the coordinates x_i, y_i, and z_i of each pixel i.
- Other depth cameras return only the z coordinate of each pixel i, noted z_i. In such cases, the user must provide: the fields of view (V_{FOV} for the vertical axis and H_{FOV} for the horizontal axis), the optical centre (c_u, c_v), and the image resolution in pixels (w and h). Then, the x_i and y_i coordinates at pixel i are determined via:

$$\begin{cases} x_i = z_i * (u_i - c_u) * tan(H_{FOV})/(w/2) \\ y_i = z_i * (v_i - c_v) * tan(V_{FOV})/(h/2) \end{cases} \quad (6)$$

In fact, if the user only provides a depth image on the z-axis, the parameters H_{FOV}, V_{FOV}, c_u and c_v are required. Indeed, in this case, these parameters permit to estimate the position of the 3D points on the x and y-axis (relation given by the equation 6). However, it is possible not to make this step if we directly give to the algorithm the depth images with the 3D point coordinates following axes x, y, and z.

Finally, we calculate the distance in the RGB image between each of these projected points and the selected pixel $\mathbf{m_i}$. The projected point that is the closest to the original point in the RGB image is the best candidate (lines 9, 10 and 11 in Algorithm 1), and we consider it to be the one ($\mathbf{l_i}$) corresponding to $\mathbf{m_i}$. The algorithm returns the corresponding 3D $\mathbf{p_i}$.

D. Calibration method for estimating $\mathbf{F}$ *and* $\mathbf{P}$

The resolution can be different between the color image and the depth image. However, this resolution must be the same during calibration and experimentation. This is necessary for $\mathbf{F}$ and $\mathbf{P}$ to work properly otherwise these matrices are no longer valid and must be estimated with the correct image sizes.

In this section, we explain how to calibrate the RGB-D camera to obtain the $\mathbf{F}$ and $\mathbf{P}$ matrices saved in the *Parameter file* to be loaded by the library. This step is very important and must be carried out as precisely as possible. To obtain a good calibration:

- the user must select many points, with a good distribution both in the image and depth space;
- s/he should select points easily observable in both images (e.g., vertices, specific shapes, ...).
- it is advisable to use an uncluttered background, such as a wall.
- to make the correspondence manually, the user must take an object that is easily detectable in both images. For example, it is possible to use a sheet of paper, a box, or a semi-transparent checkerboard as [24], whose edges and vertices are easily visible in the depth image.

This set of points is used to determine $\mathbf{F}$ and $\mathbf{P}$. Indeed, to obtain the *Fundamental Matrix* $\mathbf{F}$, we simply apply the OpenCV function: cv::findFundamentalMat()[8] to the set of points. To estimate the *Projection Matrix* $\mathbf{P}$ we proceed as follows. Generally, this matrix is determined via the camera's intrinsic and extrinsic parameters. Yet, it can also be defined experimentally applying least squares to the points selected during calibration, i.e. the pixels of the RGB image associated to their 3D coordinates in the depth camera frame (derived from the depth image). By developing (3), for n points we obtain 2n independent linear equations with twelve unknowns (the elements of $\mathbf{P}$). Since (3) is defined up to a

[7] https://docs.opencv.org/3.4.12/d9/d0c/group__3d.html

[8] https://docs.opencv.org/3.4.12/d9/d0c/group__3d.html

scale factor, we can normalize all coefficients by P_{34}. We then obtain a new matrix equation with 11 unknowns.

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{7}$$

with:

$$\mathbf{x} = \begin{bmatrix} \frac{P_{11}}{P_{34}} \frac{P_{12}}{P_{34}} \frac{P_{13}}{P_{34}} \frac{P_{14}}{P_{34}} \frac{P_{21}}{P_{34}} \dots \frac{P_{33}}{P_{34}} \end{bmatrix}^\top,$$

$$\mathbf{A} = \begin{bmatrix} 0,0,0,0,x_1,y_1,z_1,1,-v_1x_1,-v_1y_1,-v_1z_1,-v_1 \\ x_1,y_1,z_1,1,0,0,0,0,-u_1x_1,-u_1y_1,-u_1z_1,-u_1 \\ \dots \\ 0,0,0,0,x_n,y_n,z_n,1,-v_nx_n,-v_ny_n,-v_nz_n,-v_n \\ x_n,y_n,z_n,1,0,0,0,0,-u_nx_n,-u_ny_n,-u_nz_n,-u_n \end{bmatrix},$$

$$\mathbf{b} = \begin{bmatrix} u_1 & v_1 & \dots & u_n & v_n \end{bmatrix}^T.$$

The solution of (7) can be obtained via pseudo-inversion:

$$\mathbf{x} = (\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top\mathbf{b} = \mathbf{A}^\dagger\mathbf{b} \tag{8}$$

To estimate all the elements of $\mathbf{x}$ we need a minimum of 6 points. From $\mathbf{x}$, we derive $\mathbf{P}$ via:

$$\mathbf{P} = \begin{bmatrix} X_{11} & X_{12} & X_{13} & X_{14} \\ X_{21} & X_{22} & X_{23} & X_{24} \\ X_{31} & X_{32} & X_{33} & 1 \end{bmatrix}. \tag{9}$$

III. The OpenHSML library

A. Overview

The library OpenHSML described in this article is available online [9] and is open-source under the GNU Lesser General Public License version 3 (LGPLv3) [4]. This license allows integration with open- or closed-source software as long as any modification is shared with the community.

OpenHSML is written in C++ and developed on the Unix system Ubuntu 18.04. It is packaged using PID[9], a build and deployment system based on CMake. It allows to automatically deploy all dependencies that the project needs to run. The user can use this environment for his developments or install the project in an autonomous way, that is to say without having to manually create a PID workspace. A ReadMe file explains the steps needed to install OpenHSML, its dependencies (OpenCV, Eigen and yaml-cpp) and launch the demonstration.

The OpenHSML project hierarchy is the following:

- apps: examples to help get started with OpenHSML.
- build: build directory.
- share: files/folders necessary for calibration.
- src: source files.
- include: header files, with the same structure as src.

A set of images is provided along with the library, in the following folders: *share/resources/calibration/2d* for the RGB images and *share/resources/calibration/depth* for the depth images.

If the user wants to save his/her own calibration images, s/he must save these images correctly as follows. The RGB images can be saved in any image format readable by OpenCV (for example .png or .jpg). The depth images are save in a YAML file with storage class[10] in OpenCV.

B. Tutorial

Within the library OpenHSML, we provide a tutorial for testing its functionalities. It includes both the calibration and matching steps. We present both steps hereby.

1) Calibration step: First, the program asks the user to modified the default values of the parameters ($H_{FOV}, V_{FOV}, c_u, c_v, w$ and h) if necessary. Then, the program asks the user if s/he wants to calibrate the model by doing point matching in the scrolling images. In this step, the algorithm displays the two images side by side. The image on the left will be the image with which the user can always interact to select points. Therefore, when selecting a point in the left image, the images swap places to allow selection of the point in the second image. To facilitate calibration, it is possible to activate several selection modes, via the keyboard:

- the p key activates the point mode, which is the default mode, and allows stitch by stitch selection;
- the l key activates the line mode, allowing a certain number of points to be sampled homogeneously between two selected points;
- the q key activates the quadrilater mode, allowing a multitude of points to be sampled homogeneously within four selected points.

For these digital tools to work properly, the user must select the points in both images in the same order. To move to the next image, s/he should press the ESC key.

Once the user ends the calibration (s/he considers the selected points to be sufficient), the algorithm saves the estimated $\mathbf{F}$ and $\mathbf{P}$ matrices in the *Parameter file*

2) Point Matching step: Once the calibration is completed, the point matching step can start.

To use the library, the user must be provided the RGB image, and depth image in cv::Mat, the basic image container in OpenCV. The program displays only the RGB image and asks to select the points that the user wishes to be found in the depth image. Note that it is again possible to use the digital selection modes presented above.

When the user is done selecting the points, a simple press on the *ESC* key will launch the estimation. To visualize the steps, as well as the result obtained, one can add the argument *−display* in the execution command. If one wants to save these images, s/he can add the argument *−save <path/to/backup_folder>*. This last argument will also record an overview of the distribution of the calibration points and the estimated epipolar lines for these points.

Figure 3 shows the results of the matching step, obtained with a tutorial test image (acquired by Pepper's cameras) after calibration. It shows two intermediate steps of the matching phase, which is qualitatively successful: the algorithm automatically finds, in the depth image, the skeleton detected (with OpenPose process [16]) in the RGB image.

[9] http://pid.lirmm.net

[10] https://docs.opencv.org/3.4.12/da/d56/classcv_1_1FileStorage.html

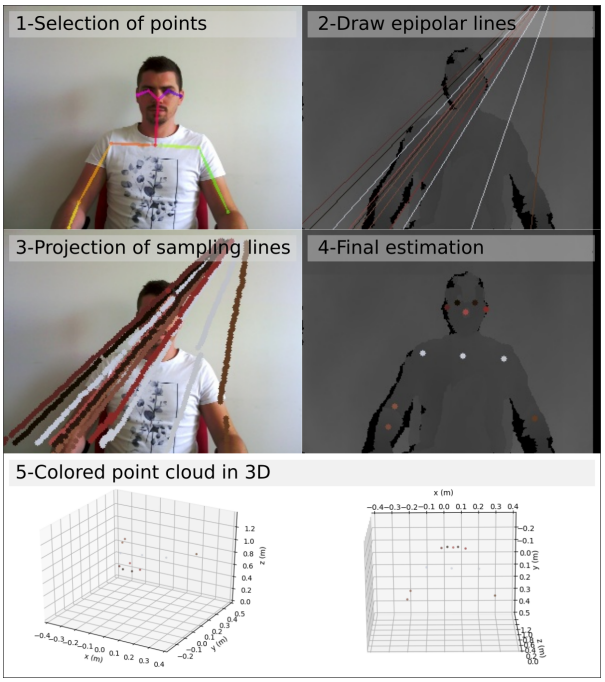

Fig. 3. Results obtained with Pepper's cameras (the front camera and its Asus Xtion). Image 1 presents the input point selected in the RGB image (with OpenPose). Image 2 represents all epipolar lines determined in the depth image. Image 3 shows the sample lines projected in the RGB image. Image 4 the points estimated in the depth image (i.e., output of OpenHSML) and Image 5 two different views of the colored point cloud in 3D.

IV. Results and Discussion

A. Experiments with three different RGB-D stereo pairs

In this part, we present experiments carried out with different pairs of cameras. We tested three different pairs. The first is the Intel RealSense D435, a compact RGB-D system with structured light depth camera. The second is composed of the Asus Xtion depth camera of the humanoid robot Pepper, coupled with it front camera. The last pair couples a computer webcam with the depth camera of the RealSense D435 to show that our system can work on many types of hybrid (RGB and depth) stereo pairs.

After calibrating our three pairs of cameras, we capture test images where an object is present in the scene. Figure 4 presents the results obtained with the three pairs of cameras.

B. Discussion

Despite these nice results, we observed two types of frequent errors. The first is caused by "holes" in the depth map. It mean that the depth information for these pixels is missing. They may be due to the estimation error of the structured light depth camera system. The second reason is that if the two cameras are too far apart, a 3D point may be located farther behind the object being observed (e.g. on a wall in the background). It can therefore be hidden in the RGB image, which can be problematic for projection. Indeed, this point can be projected onto the object in the RGB image and the algorithm can select it as the corresponding point in the depth image.

We assumed that the depth and RGB images are not distorted. In some cases, the user may need to calibrate these

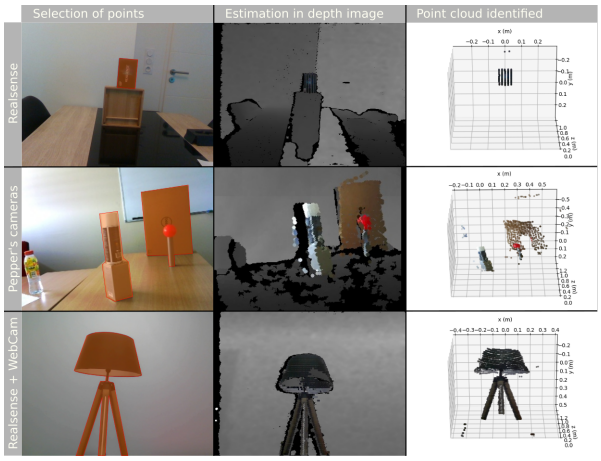

Fig. 4. Some results obtained from three different camera pairs. For each row, the first column shows the area in which the pixels were selected. The second column shows the final estimate in the depth image. Finally, the last column shows the point cloud obtained with this estimate.

cameras beforehand to improve their rendering. However, in our experiments with the RealSense D435 and with Pepper's camera, we did not perform standard intrinsic/extrinsic parameter calibration. Yet, we obtained consistent results at reasonable distances (less than 2 meters) and with objects centered in the image. Indeed, depth estimation is better in the centre of the image and at shallower depths. It can also depends on the calibration step and if the calibration points cover several depths of the area to be analyzed.

Also, as calibration is carried out at a certain distance from the camera, this can cause estimation errors for more distant areas. In our examples we do not exceed two metres during calibration for example. However, these cameras are known to be more accurate indoors and at short range.

V. Conclusion

A lot of work has been done in recent years to considerably improve the calibration of RGB-D cameras. This has enabled more and more applications to be realized with better accuracy and depth estimation. Indeed, one of the major concerns with this type of device is its high distortion and noise in the depth image data.

In our study, we propose a simple approach for finding the point correspondence between an RGB and a depth image. This approach uses the principles of stereovision to find the points of interest in the depth image without going through image alignment. Our method requires a quick and simple calibration, without the need for a checkerboard, to determine the stereo model. This model allows us to solve our problem without having to determine the camera parameters.

The advantage of our library is that it is very easy to use. It does not depend on any particular camera. It is quite possible to couple an RGB camera with a depth camera of any type.

However, our library does not currently allow for the correction of errors due to distortion in the depth image. It also does not estimate the systematic error in the depth measurement of the used system.

REFERENCES

[1] Azure kinect developer kit. https://azure.microsoft.com/en-gb/services/kinect-dk/.

[2] Calibration with aruco and charuco. https://docs.opencv.org/master/da/d13/tutorial_aruco_calibration.html.

[3] Camera calibration with opencv. https://docs.opencv.org/master/d4/d94/tutorial_camera_calibration.html.

[4] Free software foundation (2016, nov. 18). gnu lesser general public license. available:. https://www.gnu.org/licenses/lgpl-3.0.en.html.

[5] Intel realsense sdk 2.0: Start building your own depth applications. https://www.intelrealsense.com/developers/.

[6] Intrinsic calibration of the kinect cameras. http://wiki.ros.org/openni_launch/Tutorials/IntrinsicCalibration. last edited 2015-02-06 by AlexanderReimann.

[7] Kinect calibration toolbox: A matlab toolbox for calibrating the kinect sensor. https://sourceforge.net/projects/kinectcalib/. (2015, Aug. 9).

[8] Opencv: Open source computer vision library. https://github.com/opencv/opencv.

[9] Openhsml - open-source hybrid stereovision matching library. https://github.com/jordan-nowak/OpenHSML. J. Nowak (2020, Oct.).

[10] Openni 2 sdk binaries and docs. https://structure.io/openni.

[11] Rgb-d calibration: A human-friendly, reliable and accurate calibration (extrinsic and instrinsic parameters) framework for rgb-d cameras. http://iaslab-unipd.github.io/rgbd_calibration/. (2018, Nov. 14).

[12] Gwon Hwan An, Siyeong Lee, Min-Woo Seo, Kugjin Yun, Won-Sik Cheong, and Suk-Ju Kang. Charuco board-based omnidirectional camera calibration method. Electronics, 7(12):421, 2018.

[13] Filippo Basso, Emanuele Menegatti, and Alberto Pretto. Robust intrinsic and extrinsic calibration of rgb-d cameras. IEEE Trans. on Robotics, 34(5):1315–1332, 2018.

[14] Zuria Bauer, Felix Escalona, Edmanuel Cruz, Miguel Cazorla, and Francisco Gomez-Donoso. Refining the fusion of pepper robot and estimated depth maps method for improved 3d perception. IEEE Access, 7:185076–185085, 2019.

[15] Gary Bradski and Adrian Kaehler. Learning OpenCV: Computer vision with the OpenCV library. O'Reilly Media, Inc, 2008.

[16] Zhe Cao, Gines Hidalgo, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Openpose: realtime multi-person 2d pose estimation using part affinity fields. arXiv preprint arXiv:1812.08008, 2018.

[17] Richard Hartley and Andrew Zisserman. Multiple view geometry in computer vision. Cambridge University Press, 2003.

[18] Daniel Herrera, Juho Kannala, and Janne Heikkilä. Joint depth and color camera calibration with distortion correction. IEEE Trans. on Pattern Analysis and Machine Intelligence, 34(10):2058–2064, 2012.

[19] Kourosh Khoshelham and Sander Oude Elberink. Accuracy and resolution of kinect depth data for indoor mapping applications. Sensors, 12(2):1437–1454, 2012.

[20] Hongyan Liu, Daokui Qu, Fang Xu, Fengshan Zou, Jilai Song, and Kai Jia. Approach for accurate calibration of rgb-d cameras using spheres. Optics Express, 28(13):19058–19073, 2020.

[21] Ilya V Mikhelson, Philip G Lee, Alan V Sahakian, Ying Wu, and Aggelos K Katsaggelos. Automatic, fast, online calibration between depth and color cameras. Journal of Visual Communication and Image Representation, 25(1):218–226, 2014.

[22] Francisco J Romero-Ramirez, Rafael Muñoz-Salinas, and Rafael Medina-Carnicer. Speeded up detection of squared fiducial markers. Image and vision Computing, 76:38–47, 2018.

[23] Jan Smisek, Michal Jancosek, and Tomas Pajdla. 3d with kinect. In Consumer depth cameras for computer vision, pages 3–25. Springer, 2013.

[24] Zhengyang Wu, Wenzhe Zhu, and Qing Zhu. Semi-transparent checkerboard calibration method for kinect's color and depth camera. In 2018 Int. Conf. on Network, Communication, Computer Engineering (NCCE 2018). Atlantis Press, 2018.

[25] Cha Zhang and Zhengyou Zhang. Calibration between depth and color sensors for commodity depth cameras. In Computer vision and machine learning with RGB-D sensors, pages 47–64. Springer, 2014.