Calibrating and Creating Point Cloud from a Stereo Camera Setup Using OpenCV

Article · August 2016

CITATIONS
0

READS
2,209

1 author:

Mohan Chand Hanumara
Asian Institute of Technology
1 PUBLICATION **0** CITATIONS

SEE PROFILE

Calibrating and Creating Point Cloud from a Stereo Camera Setup Using OpenCV

H. Mohan Chand
Department of Industrial System Engineering
Asian Institute of Technology
Pathumthani, Thailand
tinnted@gmail.com

Abstract--**This paper explains how we can calibrate stereo camera setup and then create a disparity map from them and then convert it into point cloud which can be further used for several applications. All this is done using two open source computer vision libraries namely, OpenCV and PCL and other dependencies.**

Keywords--Calibration; Point cloud; OpenCV; PCL; Disparity Map; computer vision;

I. Introduction

At present there are many complex algorithms and methods have been proposed for generating a good disparity map from stereo or multiple camera setup with some being quality oriented and others are performance oriented and depending on the method, the implementation can be quite complex sometimes. But over the years many popular good algorithms for creating a good disparity map are being added to open source computer vision libraries like OpenCV and PCL and some of them use GPU to do the computing as CPU is not that good at doing thousands of parallel computing at once.

In this paper I will explain how we can use OpenCV and PCL to create a disparity map and then tune the disparity map by changing its parameters and then finally converting it to point cloud. And this is done on both CPU and GPU to see any performance differences and also check out different types of algorithms that are available in OpenCV to create a disparity map.

And at the moment of writing this paper there are 4 different algorithms/methods available in OpenCV to generate a disparity map, 2 uses CPU and 3 are on GPU from which 1 method is implemented on both CPU and GPU. Although there are 3 different functions on GPU, I was only able to make one method work, maybe it's because there are changes in the other two functions in the new version of OpenCV (3.0) and also the whole GPU implementation is changed to CUDA which is a feature of famous GPU brand Nvidia. So I will only use the other 3 functions and compare them.

II. Calibration and undistortion

First thing we need to do is to calibrate the cameras and get the intrinsic and extrinsic parameters which will be later used to rectify the camera input and also to get the relation between the two cameras in real world.

The intrinsic parameters are the parameters like Camera matrix and distortion coefficients and these need to be calculated for each camera. These parameters are called intrinsic because they pertain to the camera and hence if we calculate them once, we don't need to calculate them again because they don't change for the said camera.

And the extrinsic parameters are the parameters like Rotation matrix, Translational matrix, Fundamental Matrix, Projection matrix and essential matric which change with change in camera orientation or camera moment. These parameters give us the relations between the cameras in real world.

Fig 1: Stereo calibration using checkerboard technique

A. *Intrensic Parameters*

Today's cheap pinhole cameras introduces a lot of distortion to images. Two major distortions are radial distortion and tangential distortion. Due to radial distortion, straight lines will appear curved. Its effect is more as we move away from the centre of image.

The camera matrix will have information like focal length and correct camera centre offset. And the distortion coefficients related to tangential and radial distortion and other constants are in distortion coefficients matrix.

As shown in the Fig 1, we will use a chess board type image and find out the parameters by finding out the corners of the boxes using already available function, ***findChessboardCorners()*** from OpenCV and repeat the process for different orientations of the chess board at different distances. Then the corner points along with board parameters like number of boxes along the length and width and also the size of the square is passed to another function, ***calibrateCamera()*** to get the intrinsic parameters. And the Camera Matrix can be further calibrated using another function called ***getOptimalNewCameraMatrix()*** and this process is repeated for the other camera and all the data is stored in a file.

Note: Using bigger chess board or and also using circle pattern is said to yield better results

$$\text{camera matrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{Distortion coefficients} = (k_1 \quad k_2 \quad p_1 \quad p_2 \quad k_3)$$

B. *Extrinsic Parameters*

Similar to the previous calibration we will again consider the corner points of the chessboard from both cameras at the same time and then similar to previous method do it for different angles and distances and then using a function from OpenCV called ***stereoCalibrate()*** we can get the extrinsic parameters along with the camera matrix and distortion coefficients. This function can either take the intrinsic parameters as input or generate them as output. I would suggest to calculate them individually and then give them as inputs as this way it dramatically decreases RMS error, for me it decreased from 5 to 1.

C. *Undistortion*

And after getting all the required intrinsic and extrinsic parameters we need to use them to correct the input images and undistort them. With all the parameters we have calculated we can use ***stereoRectify()*** function to find rotation and projection matrices for each camera and also disparity to depth, Q matrix.

And then finally use the rotational matrix and projection matrix and also the intrinsic parameters we can calculate the undistorted images using ***initUndistortRectifyMap***(). Actually this function gives a remapping matrix which can be used to remap the input images using ***remap***() function. Example of the undistorted image can be seen in Fig 2.

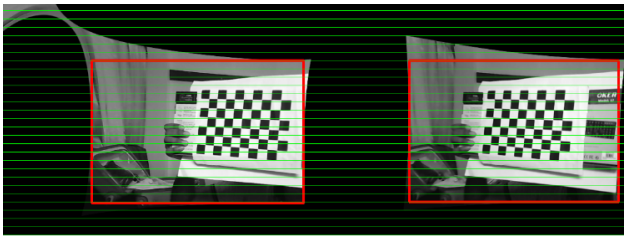

Fig 2: Rectified Stereo Images

D. *Hardware Calibration*

Before calibrating cameras for extrinsic parameters I suggest that camera setup is fixed with respect to each other as all the extrinsic parameters depend on this. I have used a bread board and glued the cameras to it and finally secured everything with glue gun as shown in Fig 3.

Fig 3. Hardware setup

III. Constructing Disparity

After rectifying the input images, we can now use those to find the disparity. As I said before we will do this on both CPU and GPU using 3 classes from OpenCV.

There are two functions which use different methods to create the disparity map, those two methods are

A. *Block Macthing*

As the name sounds, it considers block of pixels from both the images and tries to match them and as the images are rectified, the focal lengths are considered 1 and the camera planes are parallel, so it will then triangulate the real world points from the coordinates of the image and generates the disparity.

B. *Semi-Global Block Macthing*

It is similar to Block Matching but it uses pixel-wise, mutual information based matching cost for compensating radiometric differences of input images. It performs a fast approximation by path-wise optimization from all directions.

These algorithms are under their respective classes in OpenCV by their name and the GPU based are separated from CPU ones.

Calibration: Also with all the methods I suggest to tinker with different parameters of the methods available to get the best disparity.

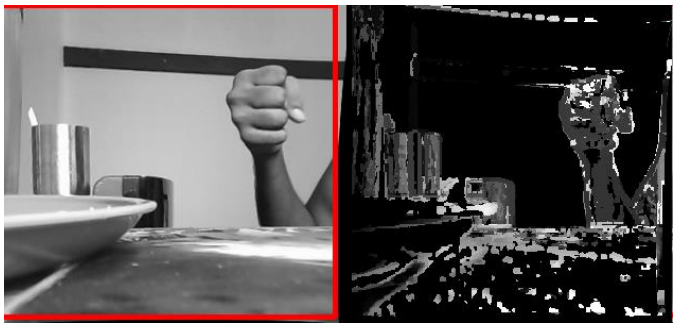

Fig.4 Left Input image and Disparity map generated using CUDA based Block-Matching

IV. Constructing and Visualizing Point Cloud

Although we can use OpenCV to create a point cloud and visualize and store it in one of many formats available for 3D data we are using PCL library as it is the most widely used Point Cloud library and also it is powerful with many of it point cloud handling functions. And also we can easily link or used this data for future use. With a good disparity map we can now proceed to converting it into point cloud so that it can be used in other applications. The point cloud houses dense or coarse points with each point having (X, Y, Z) values which represent the location in the 3D space.

And for our application we have to pass the (X, Y) values of the valid points in the disparity and then pass the disparity value or we can calculate the depth from focal length and baseline values or using disparity to depth matrix, Q we found out earlier.

And using the visualization module of the PCL we can easily view the point cloud data. PCL actually uses VTK library to visualize the data. But PCL has made it much more easy with its easy to use *cloudViewer*.

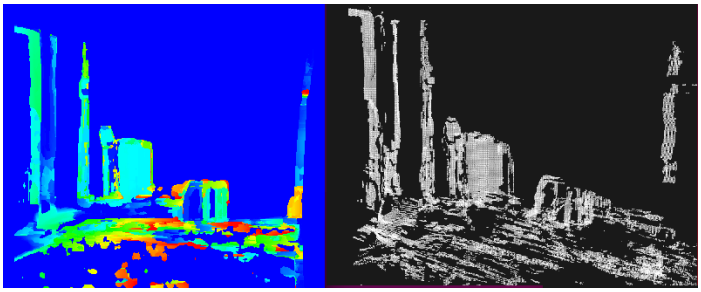

Fig5. Disparity left and Point cloud on right

V. RESULT

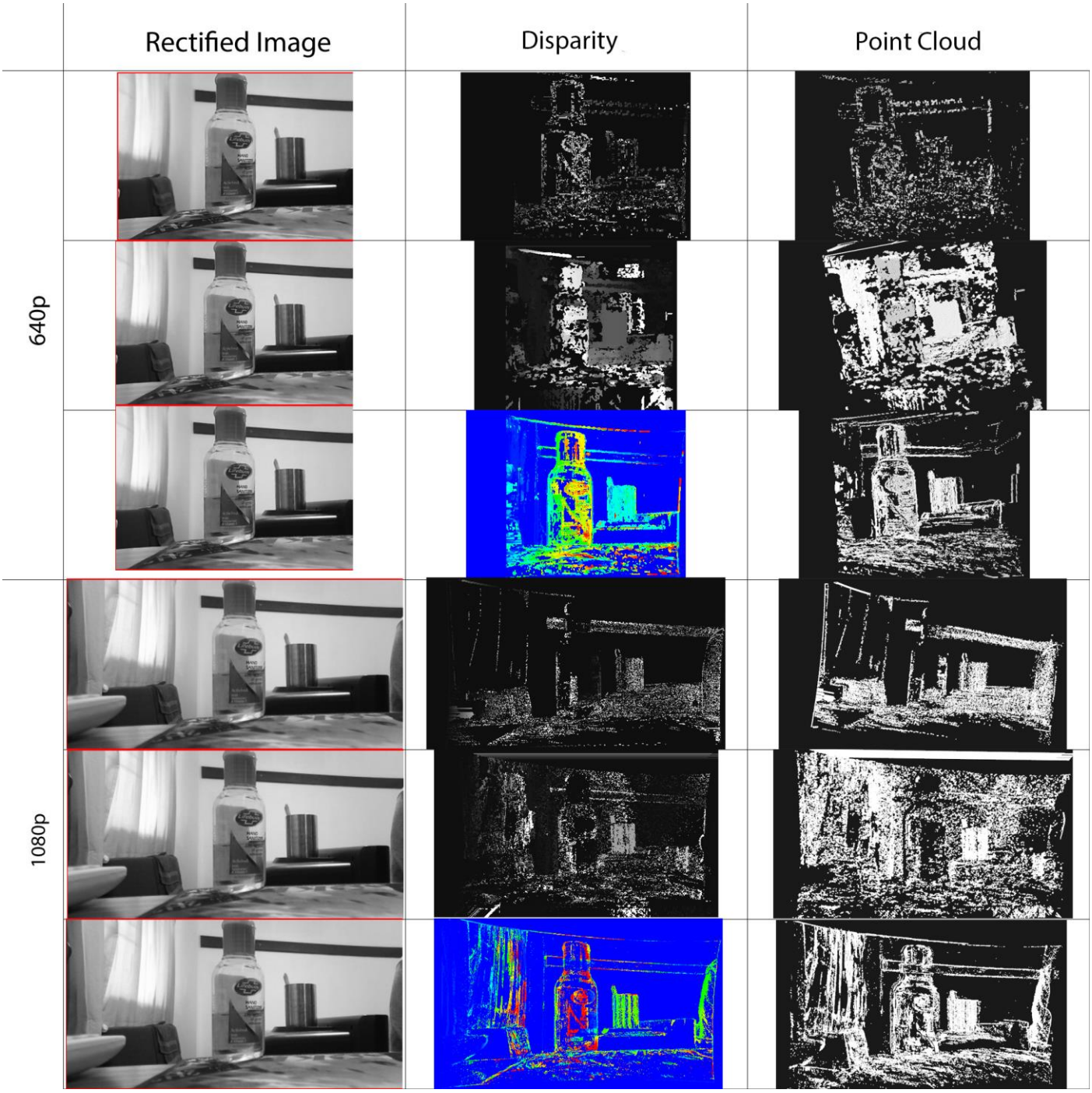

FIG 6- HAS INPUT IMAGE, DISPARITY, AND POINT CLOUD FROM STEREOBM, STEREOSGBM, GPU-STEREOBM FROM TOP TO BOTTOM RESP.

VI. PERFORMANCE

Time	Stereo BM	Stereo SGBM	GPU BM
640p	3.3 secs	7.9 secs	2.7 secs
1080p	17 secs	71 secs	11 secs

Table 1- Time taken by each methods on different resolutions

As I stated before I have run on both CPU and GPU and that too at two different resolutions that is at *640x480* and another *at 1920x1080*.

And I have benchmarked all three methods I used for every 50 first frames. And this is done on a computer with I7-4700HQ quad core processor and Nvidia 970M GPU, Although I am using a powerful processor the program is implemented such that it can only use single core/thread. And the GPU I am using is rated to have max Compute capability (5.2 score) and Nvidia claims to achieve 12x to 50x performance increase but since all programs use CPU combined with GPU the performance increase may differ.

And also this values can change depending on the scene or even bottleneck by the camera frame rate.

And the rms error after calibrating each camera and also rms error after calibrating stereo are given in the table below. We can clearly see that rms error for 640p is lower but not

better because 1080p has 6.75 times more pixels but the error only increased by 2 folds which is good.

RMS error	Left camera	Right camera	Stereo	Epipolar error
640p	0.388	0.399	1.01	1.65
1080p	0.759	0.739	2.80	5.28

Table 2 RMS error at different steps

VII. Conclusion

With the present open source computer vision libraries, we can easily generate a disparity map from stereo setup and convert that to point cloud. Although state of the art algorithms and complex methods are not available in the OpenCV library, but with the available functions for feature detections etc., we can implement new function.

And already available algorithms are just fine for most of the applications and also can be further made robust with use of filters or by better calibration. We can achieve denser and more accurate point cloud of a static scene if having the point cloud in real time is not a constraint.

And with CUDA support, computing can be accelerated a lot even in mobile devices as Nvidia mobile processors are available for low power applications as well. In fact, one of the recent computer vision breakthrough, self-driving car by Tesla uses Nvidia CUDA based ARM architecture processors.

Using high resolution images is also a profit and also with CUDA it seems we can get up to 5 fps real time point cloud generation.

VIII. References

[1] Christoph Vogel · Konrad Schindler · Stefan Roth "3D Scene Flow Estimation with a Piecewise Rigid Scene Model"

[2] Zhengyou Zhang, "Flexible Camera Calibration by Viewing a Plane from Unknown Orientations"

[3] Aroh Barjatya, "Block Matching Algorithms For Motion Estimation"

[4] Heiko Hirschm¨uller, "Semi-Global Matching – Motivation, Developments and Applications"

[5] Heiko Hirschm¨uller "Stereo Processing by Semi-Global Matching and Mutual Information"

[6] James Davis, Ravi Ramamoorthi, Szymon Rusinkiewicz, "Spacetime Stereo: A Unifying Framework for Depth from Triangulation"