Computer Vision
Assignment 2 - Stereo

Nguyễn Công Thành

May 12, 2023

This assignment was accomplished with the assistance of my friends, Kim Ngân, Nhật Tân, and Hà Huy, under the supervision of thầy Dương. I'd like to express my heartfelt gratitude to them.

1. Theory

Let $\mathbf{x}_l = (u_l, v_l, 1)$ and $\mathbf{x}_r = (u_r, v_r, 1)$, K_r and K_l, be the coordinates of the object in the images, and the intrinsic matrices of the left camera, the right camera, respectively. We have

$$\mathbf{x}_l^T {K_l^{-1}}^T E K_r^{-1} \mathbf{x}_r = 0. \tag{$\spadesuit$}$$

Note that for the simple stereo system, we have

$$K_l = K_r = \begin{bmatrix} a & 0 & x \\ 0 & b & y \\ 0 & 0 & 1 \end{bmatrix} = K, \quad abxy \neq 0,$$

and

$$E = T_\mathbf{x} R = T_\mathbf{x} I = T_\mathbf{x} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -k \\ 0 & k & 0 \end{bmatrix}, \quad k \neq 0.$$

Thus, ($\spadesuit$) can be written as

$$\mathbf{x}_l^T {K^{-1}}^T T_\mathbf{x} K^{-1} \mathbf{x}_r = 0,$$

where

$$K^{-1} = \begin{bmatrix} \frac{1}{a} & 0 & -\frac{x}{a} \\ 0 & \frac{1}{b} & -\frac{y}{b} \\ 0 & 0 & 1 \end{bmatrix},$$

$${K^{-1}}^T T_\mathbf{x} K^{-1} = \begin{bmatrix} \frac{1}{1} & 0 & -\frac{x}{a} \\ 0 & \frac{1}{b} & -\frac{y}{b} \\ 0 & 0 & 1 \end{bmatrix}^T \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -k \\ 0 & k & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{1} & 0 & -\frac{x}{a} \\ 0 & \frac{1}{b} & -\frac{y}{b} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -\frac{k}{b} \\ 0 & \frac{k}{b} & 0 \end{bmatrix}.$$

Therefore,

$$\mathbf{x}_l^T \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -\frac{k}{b} \\ 0 & \frac{k}{b} & 0 \end{bmatrix} \mathbf{x}_r = 0$$

or

$$\frac{k}{b}\left(v_l - v_r\right) = 0,$$

implies that $v_l = v_r$ since $k \neq 0$. So, a point on one image plane can be found on the other image by searching along a horizontal line of the same vertical coordinate.

2. Programming

The work in this part mostly referred to the tutorial and repo cited in the reference part. To be honest there are some places I still can not quite grasp so I will try to present what I have learned from them.

Problem 1. Since the stereo in this problem is the simple stereo, our task is not too complicated since we only need to compare the neighboring pixel values in the same row of the stereo image pair to find the corresponding points between the two images. Given this, we can find the depth by the formula

$$z = \frac{bf_x}{u_l - u_r}.$$

Now, since we have assumed our scene is extremely simple, consisting of only the object and a simple background, to track the object, we only need to track the object with the smallest depth and issue an alert when this distance is close to the camera.

In practice, to implement this stated idea, we use the StereoBM class in OpenCV. However, the parameters of this function can be affected by many factors so to quickly updated the suitable parameters in practice, we use the OpenCV GUI tools to create trackbars to change the parameters.

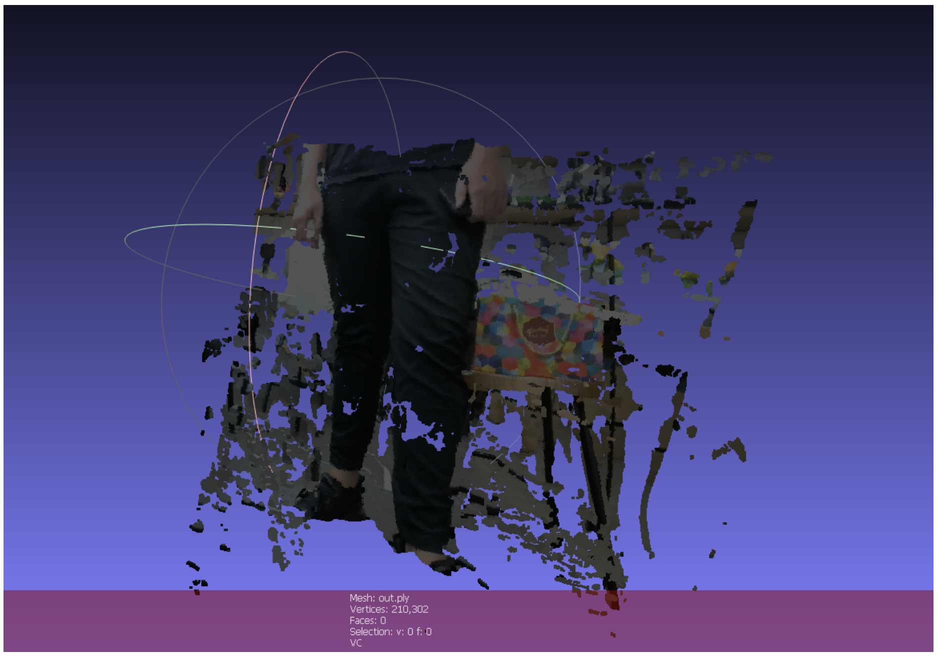
During the implementation, we realized that OpenCV was customizing (scaling or something, we don't know) the parameters so after calculating the disparity, taking bf_x (which we already know) divided by it won't give us the depth. Therefore, we run the code and compare it with reality to calculate the right value for bf_x. This means that for every time running the code, we need to modify bf_x to find the corresponding value of it with respect to the system.

For the part of the point cloud file, at first, we can only generate a good result with the parameters used in the repo sent by thầy Dương. We do not understand why with parameters that we tune ourselves, even though it gives very accurate results in terms of disparity, it still can not give a good result for point cloud files (it is more like a 2D picture instead of 3D). After many tries, we realize that increasing the value of the minDisparity parameter can give a better result as we want. This is because a larger minDisparity value can help to prevent the algorithm from incorrectly matching pixels with similar intensity values but different depths.

The code for this assignment can be found in Problem1.py file. I tried to give comments on every function/step so the reader can easily understand how it works.

One lesson that we feel very deeply when doing this problem is that there is always a trade-off, besides the convenience, OpenCV also brings many difficulties because we do not understand how it works, as well as customize it to our liking when encountering some unexpected issues.

Here is the 3D result on the demo day (I do not save the result for the warning part, but thầy Dương checked it, it is very nice right ♡)



```
Mesh: out.ply
Vertices: 210,302
Faces: 0
Selection: v: 0 f: 0
VC
```

Problem 2. This problem is more challenging than the previous one since we need to care about almost every point in the scene, not just a single object. Therefore, a convenient way like using OpenCV in problem 1, which runs very fast might not give a good result if we do not spend (a lot) time to tune. Thus, I decided to "do the best" in each step to hope for an automatically good result. The trade-off is that the running time for it is very long, about 3-20 minutes based on the size of the images.

Here are the ideas for my solution (again, the code mostly referred to the repos cited in the reference part. I am not a good coder, however, I understand (hope so) all the techniques of this solution so we can believe that I can code on my own as the repos)

STEP 1: CALIBRATION. Using SIFT to detect features in 2 images. Use these features to find the best Fundamental matrix by RANSAC. From this, and the intrinsic matrix of the camera, find the Essential matrix. Then, decompose it to find the Rotation and Translation matrix. The Projection matrix can then compute by these matrices. Note that for all combinations of R and T, we have 4 configurations for P so we use the Cheirality condition to find the best P among these options.

$\Longrightarrow$ The result is good if we have many features, meaning that the images are similar to each other. Otherwise, our result will be bad or we even can not find the result.

STEP 2: RECTIFICATION. We had F, E, P, R, T so the problem is quite simple. We just use OpenCV to find the epipolar lines and then transform them to get the images with coinciding epipolar lines.

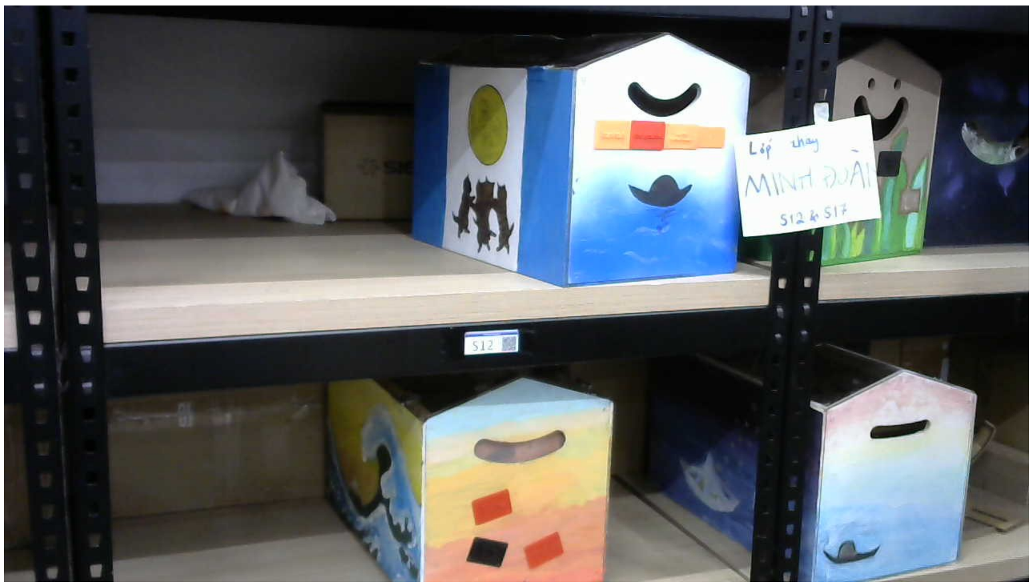
STEP 3: CORRESPONDENCE. Now we just need to search along the rows of the rectified images. The technique used is SSD.

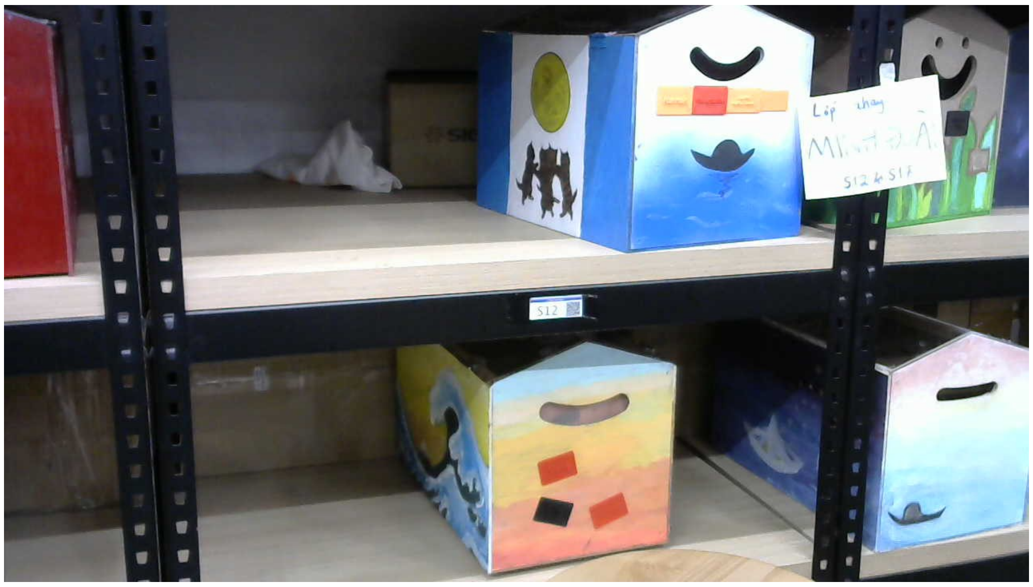
$\Longrightarrow$ need to choose the appropriate window size. The result is not good if the image contains repeating patterns.

STEP 4: COMPUTE DEPTH MAP. Compute disparity and use the same formula as problem 1 to get the depth map. Note that we are only interested in the depth relation of the points in the image, so we can assume that the baseline equal to 1.

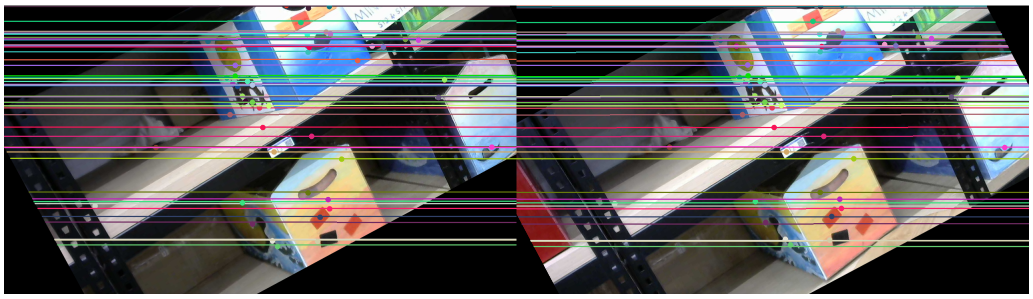
The result on the demo day shown below

Two original images:

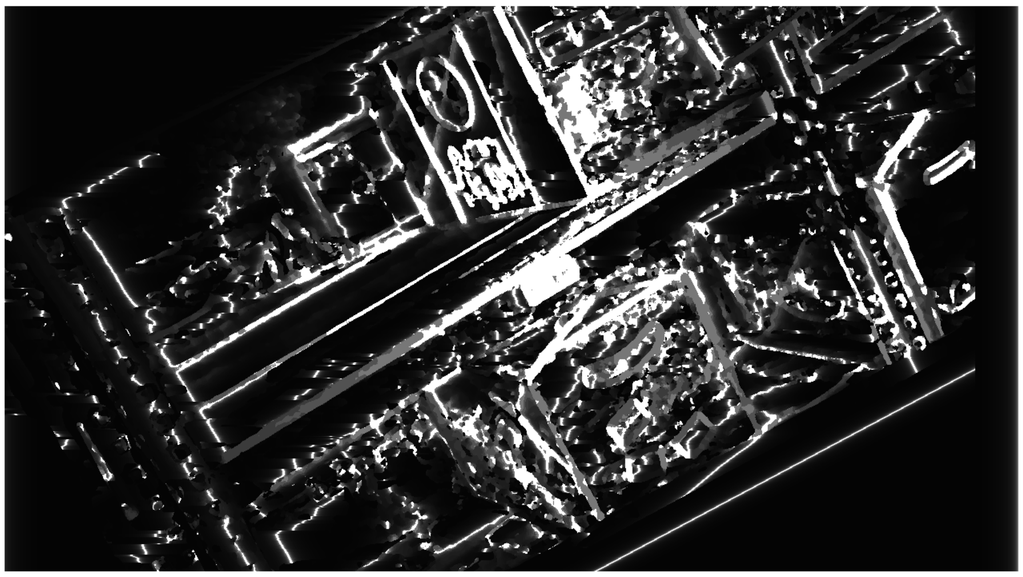


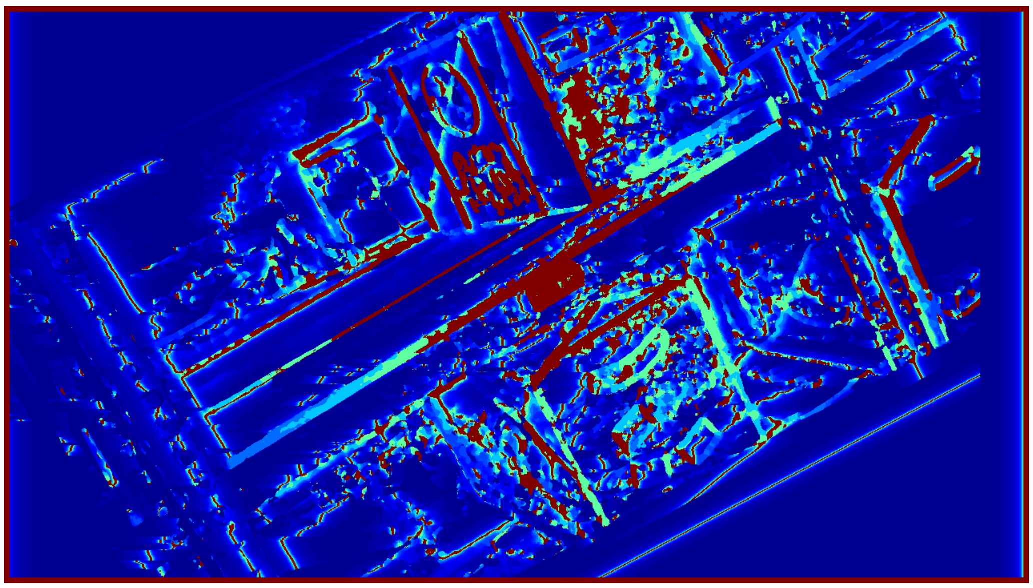


Rectified image:



Depth map and heat map:





Tài liệu

[1] Shree Nayar, T. C. Chang, First Principles of Computer Vision course https://www.youtube.com/@firstprinciplesofcomputerv3258

[2] Stereo Camera Depth Estimation With OpenCV (Python/C++)
https://learnopencv.com/depth-perception-using-stereo-camera-python-c/

[3] OpenCV: Open Source Computer Vision Library
https://github.com/opencv/opencv/blob/master/samples/python/stereo_match.py

[4] abhijitmahalle, Stereo-Vision
https://github.com/abhijitmahalle/stereo-vision