Exercise 2: Stereo Vision

Due: 15.11.2018

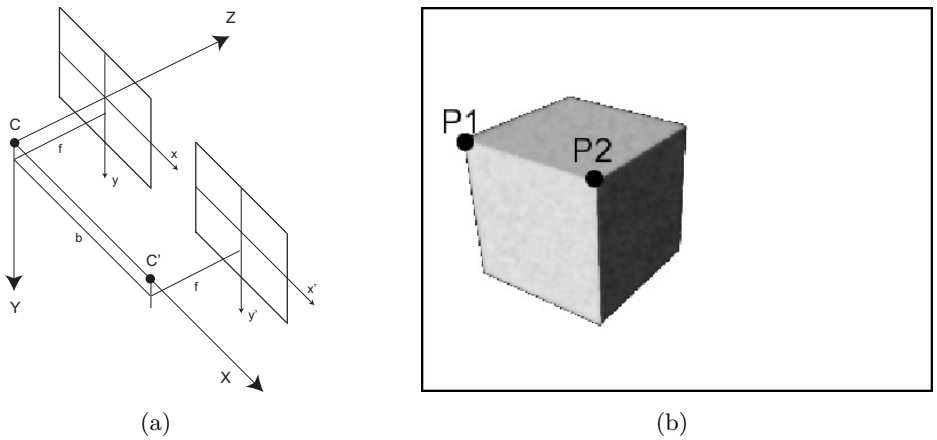
In the preceding exercises, several image processing methods have been introduced. The two processed images (showing a cube from different points of view) shall now be used in order to reconstruct a 3d wire frame version of that cube. In the theoretical part, the position of the right camera has to be derived. The practical part is about looking for correspondences between the two images and about triangulation.

1 Theoretical Exercises

For stereo reconstruction, both the internal and external parameters of each camera must be known. The internal parameters are given: The focal length is 35 mm (1.378 in). The width of the film back equals to 1.417 in and its height is 0.945 in. The external parameters are provided only partly. The missing information has to be derived.

1.1 External Camera Parameters

The camera setup is shown in Fig. 1(a). Determine the position C' of the right camera. In the general case, this task requires the coordinates of three points in space and the image coordinates of these points with respect to the right camera. However, since there is no rotation involved two points are sufficient. In the world coordinate system, point P_1 is located at $(-0.023, -0.261, 2.376)$ and point P_2 at $(0.659, -0.071, 2.082)$. In the image of the right camera (640x480), P_1 has coordinates $(52, 163)$ and P_2 is positioned at $(218, 216)$ - see Fig. 1(b).



(a) (b)

Figure 1: (a) The world coordinate system is defined to be the same as the coordinate system of the left camera (C lies in the origin). Both cameras look in the direction of the positive Z-axis. Neither is rotated. (b) The cube observed by the right camera C'.

As a starting point, look at the relevant equation in the script. You can obtain kx and ky by using the image resolution and the size of the film back. Do not forget that the center of each image at this scale is at $(320, 240)$. Round the calculated camera position with a precision of half an inch. You can not assume that C' is 0 along the Y and Z directions

2 Practical Exercises

The goal of the practical exercise is to compute a 3d reconstruction for a given image pair. In this exercise we will be dealing with 2 examples, ie the *tsukuba_left.pgm*, *tsukuba_right.pgm* image pair, and the *cube_left.pgm*, *cube_right.pgm* from the previous exercises. Copy the respective image files from `~cvcourse/pics/` or `http://people.ee.ethz.ch/~cvcourse/pics/` to a new directory. Also place the template `2_stereo_template.py` in this directory.

3d reconstruction from two views can be split into two sub task: (1) finding corresponding points in the two images and (2) recovering the 3d world coordinates (of the "physical" point) given such corresponding points. The latter is called triangulation and requires fully calibrated cameras, ie, the internal and external camera parameters must be known. Calibration of cameras has been addressed in the theoretical exercise (see Fig. 1(a)). Assume now that the two optical camera centers are 1.0 inch apart from each other. The optical axes of the cameras are parallel and their image planes are coplanar, with coincident x-axes.

Given our special camera setup, we know that corresponding points must have the same y coordinate which simplifies finding point correspondences. Hence, we can process each line of the image (ie, each *scan line*) separately. In order to identify corresponding points we need a measure of similarity. In this exercise you should use the normalised cross-correlation which will be defined later.

2.1 Triangulation

Implement the function `def triangulate(_xLeft, _xRight, _y, m_width, m_height)`. The first two parameters correspond to the x-coordinate of a projected 3d-points array in the left and the right gray scale image. `_y` is the common y-coordinates array. `m_width` and `m_height` are the width and height of the image. Note that this method uses the (globally defined) camera parameters. The function must return the world coordinates in inch.

Check the code with the test case provided in the jupyter notebook.

2.2 Correlation Coefficient

Implement `def computeCorrelation(_grayLeft, _grayRight, _xLeft, _xRight, _y, maskWidth, maskHeight)`. The method is supposed to compute the similarity between a patch from the left and and a patch from the right image during correspondence search. Check the code with the test case provided in the jupyter notebook.

The normalised cross-correlation of patches surrounding two points (in the two views), is defined as

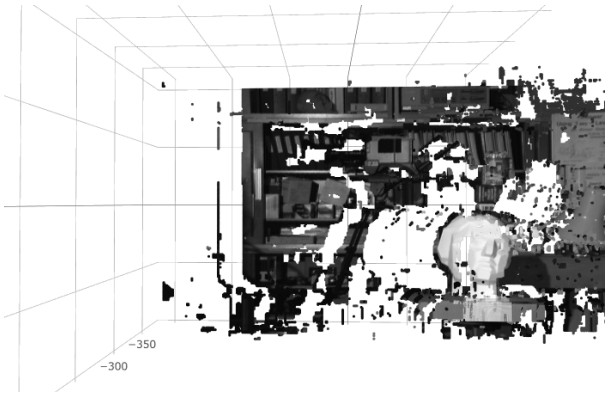
$$NCC(p_L, p_R) = \frac{1}{|\Delta|\sigma_L\sigma_R} \sum_{\Delta} (I(p_L + \Delta) - \mu_L) \cdot (I(p_R + \Delta) - \mu_R), \tag{1}$$

where Δ defines the (square) neighbourhood of a pixel, $\mu_R = \frac{1}{|\Delta|} \sum_{\Delta} I(p_R + \Delta)$ and $\sigma_R = \sqrt{\frac{1}{|\Delta|} \sum_{\Delta} (I(p_R + \Delta) - \mu_R)^2}$.

2.3 Stereo Reconstruction

The final step is to implement the function `def definePointsThreeD(_grayLeft, _grayRight, _cannyLeft, _cannyRight)` which does the actual 3d reconstruction. It makes use of the above function to identify corresponding points and to compute the 3d world coordinates. The output, ie one 3d point for each pair of corresponding points, should be stored in an array called `points`. Task: In each scan line, find one corresponding pixel in the right image for each pixel in the left image (note that the x-coordinate of the pixel in the right image always has to be smaller than the x-coordinate of the pixel in the left image - explain why). Use the correlation coefficient as a similarity measure for the correspondence search. Compute a 3d-point by triangulation for every pixel pair and add it to the `points`.

Test your code with the `tsukuba` image pairs. Your results should look like Fig. 2 for a correlation mask of 11×11.



(a)

Figure 2: 3d reconstruction of tsukuba.

2.4 Cubes

As you may have noticed, computing point correspondences is quite time consuming. The reason is that one has to compare each point in the left image to many points of the corresponding scan line in the right image. You could now implement a faster method (for the `cube_left.pgm`, and `cube_right.pgm` images) which makes use of the canny edge maps (`cube_left_canny.pgm`, and `cube_right_canny.pgm` images). Adapt the method `definePointsThreeD` to take advantage of the canny images. This allows to limit the correspondence search to only a few pixels per scan line. Do not forget to set `m_cubeMode` to 1 and provide the canny edge maps.

Can you suggest any modification of this pipeline, that may lead to better results?