Assignment 2 - part 2

Purpose This is the second part of the Assignment2. The purpose of this part of the assignment is to gain some insight into camera calibration, explore some of the benefits of having a calibrated camera and to know more about the mathematics behind the camera calibration and computer graphics. You will be working on camera calibration to obtain extrinsic parameters and then the camera matrix (P=K[R|t]) will be used to map points from the world coordinates system to the camera coordinates system and visa versa. In particular you will calibrate your web camera and use the camera matrix to augment video sequences with simple 3D graphics.

This part is divided into two sections, namely *camera calibration* and *augmentation*, but where the latter depends on the first.

This assignment is going to be used for the next mandatory assignment (Assignment3). In the next mandatory assignment you will do some shading and texture mapping on the cube that you draw in this assignment.

Remember that there is not only one solution to the assignment and we encourage you to be creative when solving it :)

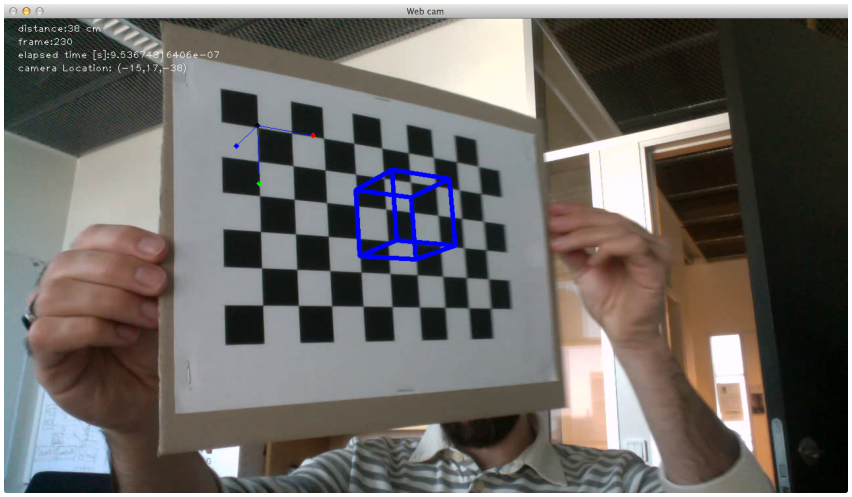

Figure 1: Augmented cube and the the world coordinate system

Start up!

1. The *SIGBTools.py* has been updated so replace the old version by the updated version of *SIGBTools.py* that comes with assignment2_part2.

2. Unless copies of the calibration pattern have been printed by the TA, print the './Images/CalibrationPattern.jpg' on a sheet of paper (A4) and glue it on a peace of carton.

There is a file *Assignment_Cube.py* in the main folder of this assignment that contains a general structure for this assignment and also for the next assignment. Inside the file there are several comment lines as below:

```
''' <comment code> SOME TEXT...'''
```

The <comment code> is the address of the comment used in this document and indicates the order of the steps you need to follow for this assignment. Please follow this document step by step and write your piece of code for each step below the comment line in the *Assignment_Cube.py*. We suggest you to make a function for each step and just call the function below the comment line.

In the main body of code first you see some variables defined, then a box (cube) and some faces are defined that you will use them later. At the end there is the code *run()* that runs the main loop of the program. Run the program as it is. It will show a video sequence on a window. Read the instruction in the debug console that shows

how to interact with the program. you see that you can control your program using your keyboard. For example the space key pause or plays the video, or pressing the "p" enable or disables the processing the frame image (no processing at this point). Some of the keys are not active at the moment and you can use them later.

3. You can switch between video or camera playback by changing the second input of the function `run()` to 0.

4. You can change the playback speed by the first input of the function `run()` (only works when playing video files not the camera capture).

5. You can use the function `RecordVideoFromCamera()` in the *SIGBTools* that records a video (Pattern.avi in the source folder) from the camera. This may be useful when you want to do the calibration or test the program on a same video over and over.

Camera Calibration

6. There is a function called `calibrateCamera()` in the *SIGBTools.py* that calibrate the camera and saves all the data such as the camera calibration matrix (K), rotation and translation vectors, and the image of the first calibration view. When the function calibrates the camera it actually returns the external parameters for each view that is used for the calibration. Find the function in the *SIGBTools.py* and see how it works. In your report explain what are the following variables that are used inside the function: `pattern_points`, `img_points`, and `img_points`

7. Go to the comment line <000>, call the function `SIGBTools.calibrateCamera()` with the right signature and calibrate your camera (use 0 as the last input of the function when you want to use your webcam instead of a video file).
   Remember that you need to press the space key to grab the samples during the calibration. By default, `calibrateCamera()` takes 5 sample views. When calibration is done `calibrateCamera()` uses the non-linear distortion coefficients and undistorts the rest of the video. You can verify your calibration by seeing the undistorted image. If the undistorted video does not seem correct redo the calibration again.

8. Change the `calibrateCamera()` so that it takes more or less sample views. How does increasing/decreasing the number of taken samples influence the calibration performance?

9. `calibrateCamera()` function saves the calibration data in a folder called numpyData in your source folder. These numpy files contain the calibration information (e.g. translation and rotatin vectors) for all 5 sample views. If done right, camera calibration for a fixed lens only needs to be done once. So comment out the step <000> so that it does not calibrate the camera again when you run the program.

10. Go to the comment line <001>, load all the calibration data that are stored in the numpyData folder. Choose proper names for each variable. The code snippet below shows an example how to load numpy data:

```
translationVectors = np.load('numpyData/translationVectors.npy')
print "translationVectors of the second view", array(translationVectors)[1]
```

11. (no coding in this step!) Look at the calss *camera* inside the *SIGBTools*. You can easily define a camera by knowing the camera matrix. Look at the other functions inside the *SIGBTools*, you might find them useful later.

12. `calibrateCamera()` function also saves the the image of the first sample frame of the calibration in the source folder. It is called *'01.png'*. We call this image the first camera view. Go to the comment line <002>, calculate the camera matrix ($P_1 = K[R_1|t_1]$) for the first camera view (first view of the calibration). (Hint: Use the function cv2.Rodrigues() that converts the rotation vector (default output of SIGBTools.calibrateCamera() for the rotations) to a 3x3 rotation matrix).
    (you can always use camera.factor() to decompose the camera matrix to K, R and t whenever you want to print or use these elements separately).

13. Go to the comment line <003>, here we want to check if P_1 is correct. Load the image of the first view saved during the calibration and project the chesspattern points (obj_points of the first view which are the coordinates of the pattern corners in the world coordinate system) to the first view. Draw the projected points in the first view image. All the points should be exactly projected onto the right positions.

14. Go to the comment line <004>, use the distortionCoefficients of the camera and undistort the image. (You can enable/disable it later by pressing the key "u")

Augmentation

In this section, you will calculate the full camera matrix in each view (each video frame) and you use that for projecting a 3D cube into that view. The idea is to find the camera matrix of each view with two different methods and compare their performance.

Finding the camera matrix using homography

1. You obtained the full camera matrix of the first view at step <002>. Find the homography of the first view (${}^{1}_{cs}\mathrm{H}$) from the camera matrix. (homography from the chess pattern plane in the world coordinate system to the first view camera image).

2. Go to the comment line <005>, and for each frame (current view) locate 4 outer corners of the calibration pattern while the video is playing. Display these 4 points with circles on the calibration pattern in the images.

3. Use the 4 corners of the pattern in the first (already loaded at step <001>) and the current view and estimate the homography, ${}^{2}_{1}\mathrm{H}$, from the "first view" to the "current view".

4. Go to the comment line <006>. Here you need to calculate the homography from the chess pattern plane in the world coordinate system to the second view camera image (${}^{2}_{cs}\mathrm{H}$). This can be done by having ${}^{2}_{1}\mathrm{H}$ and ${}^{1}_{cs}\mathrm{H}$.

5. Find the camera matrix ($P_2 = K[R_2|t_2]$) from ${}^{2}_{cs}\mathrm{H}$ and call this matrix **P2_Method1**. (hint: there is an example in the lecture slides. you just need to calculate the third column of the rotation matrix)

Finding the camera matrix directly using extrinsic parameters

6. There is a function called *GetObjectPos()* in the SIGBTools that finds the object pose (rotation and translation) from the 3D-2D point correspondences. Use this function and calculate the camera matrix of the current view directly without using homography, and call that **P2_Method2**
   Hint: Object points needed for this part can be defined as below:.

```
pattern_size=(9,6)
pattern_points = np.zeros( (np.prod(pattern_size), 3), np.float32 )
pattern_points[:,:2] = np.indices(pattern_size).T.reshape(-1, 2)
pattern_points *= chessSquare_size
obj_points = [pattern_points]
obj_points.append(pattern_points)
obj_points = np.array(obj_points,np.float64).T
obj_points=obj_points[:,:,0].T
```

7. Print and compare the camera matrices P2_Method1 and P2_Method2 obtained above. Explain the difference between these two methods and the values in each matrix.

8. Modify your code so that it can switch between these to methods by pressing a key like "x". We want to compare the results of these to methods in the next step.

9. Go to the comment line <007>, test the camera matrix P_2 by projecting the chesspattern points in the world coordinate system to the current camera view. Draw the projected points in the image with some circles. Are all points projected to the right place in the image? Compare the two methods you have used for finding the P_2 when you use them for projection.

10. Go to the comment line <008>, draw the world coordinate axes attached to the chesspattern plane (see Figure 1).

11. Go to the comment line <009>, here we want to project a cube attached to the pattern plane into the current view. The cube (`box`) has been already defined in the main body of the *Assignment_Cube.py* and you just need to project the box and draw the projection result using the function `DrawLines()`.

12. See whether the two camera matrices P2_Method1 and P2_Method2, have a same performance and accuracy when projecting the 3D cube (especially for the extreme viewing angles). If you see any difference explain that in your report.

13. Record some videos of augmenting the cube using two different camera matrices and hand in your result video with your reports. (you can use the function `RecordVideoFromCamera()`)

Ideas for extra credit

14. * Make the augmentation object rotate around its own axis e.g. independently of the actual object in the scene e.g. like the video shown in the lecture.

15. * Use a different object as the augmentation object e.g. a pyramid