**CALIBRATION**

The first step is to calibrate the camera. In other words, we need to find the intrinsic camera matrix and distortion coefficients for both lenses. To do that, we use the calibration images with a checkerboard. The size of each square pattern is 33.6 x 33.6mm.

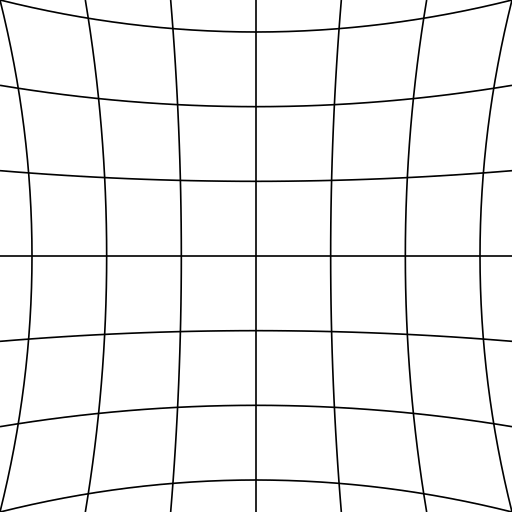
To find the corners on the checkerboard function cv2.findChessboardCorners() was used. The pixels at which the corner was found are multiplied by the size of the square. Therefore, we get our camera intrinsic matrix in mm.

In the next step, we use cv2.calibrateCamera(). The output from the function call is a camera matrix A picture containing table

Description automatically generated and distortion coefficients. The distortion coefficients are later used to undistort the camera image. Every lens has a small distortion.

Shape

Description automatically generated with low confidenceBarrel

 Pincushion

As the last step in the calibration, we get the excentric parameters R, T, E, and F for the stereo camera by calling stereoCalibrate()

* R = Rotation matrix between camera1 and camera2
* T = translation between camera1 and camera2
* E = Essential matrix
* F = Fundamental matrix

**UNDISTORTION**

While taking pictures with the camera, straight lines are changed to curves because of the lens distortion. Without distortion, the Kalman filters and tracking, in general, would be complicated and nonlinear, as the object would be moving on the curve in the picture, although they are following a straight line in the real world.

To remove this issue, we undistort the image. Firstly, we use the function getOptimalNewCameraMatrix to "prepare" the image using parameter alpha, which helps us adjust the black areas and crop undistorted images.

In the next step, we call the function undistort() to undistort the image, crop it using roi values, and save the picture, as it will be needed later for the Kalman filter.

A picture containing floor, indoor

Description automatically generated -> A blue ping pong table

Description automatically generated with medium confidence

Rectification

This step took us more time than expected. Since we need to track the object in 3D, the best way to compute the depth is to find a disparity map using opencv2 functions or the code we wrote ourselves during Week3. Generally speaking, there were three approaches to getting the missing depth coordinate.

* Disparity map
* Triangulation
* Increasing object size on the video

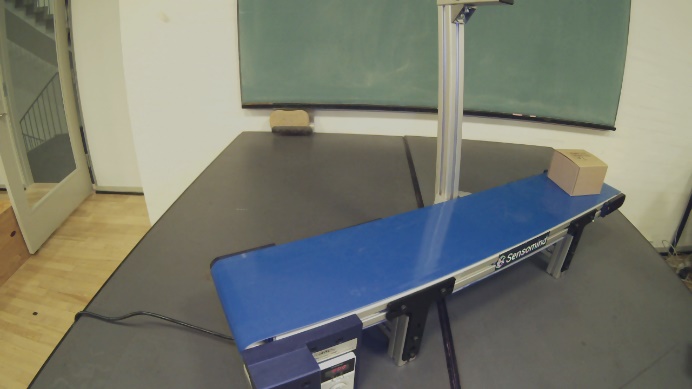
Triangulation was possible, but as we do not know the distance between lenses, we would be missing the unit. Also, to make it work, we would need to track the object in both left and right images to get the coordinates of the moving object in both pictures. Not having the object's position in mm means that, e.g. the gripper will not be able to track the object in the real world, as we need to know the perfect position in mm. The same issue is with the third option, as with the increasing object size, we will miss the unit. We can get the depth in mm using the disparity map since we know the camera's exact intrinsics and eccentrics parameters from calibration.

**Uncalibrated Rectification**

To rectify the images, we have two options using OpenCV. Either we can use stereoRectifyUncalibrated(), which needs matched points from both left and right images and a fundamental matrix. Undistorted images should be used as the StereoRectifyUncalibrated function heavily depends on epipolar geometry.

The matched point are matched using keypoints and SIFT Descriptors from sift.detectAndCompute(), which are matched using BFMatcher().match() and sorted with the best matches in the beginning. We decided to choose the 200 best matches. A fundamental matrix from calibration was used to get the fundamental matrix, and the homogeneous matrices were found using stereoRectifyUncalibrated(). In the last step, we rectified the images using warpPerspective().

Nevertheless, this approach is not working for our example, as for every image, the rectified image is moved, rotated, and scaled a bit differently. Therefore we decided to use Calibrated Rectification instead, which gives us more stable results.

 -> A picture containing text, table, worktable

Description automatically generated

**Calibrated Rectification**

The Calibrated rectification was more successful than the uncalibrated, providing us with a good starting point for computing the depth of the image. Firstly, we ran the function stereoRectify() to get the rotation R and projection matrix P, later used in the initUndistortRectifyMap function. This function gives us the mapping matrices for remapping. The remap() function finally gives us the rectified image.

A picture containing floor, indoor

Description automatically generated -> A picture containing floor, indoor, sport, blue

Description automatically generated

**Epipolar Lines**

To verify that the rectification was successful, we needed to draw epipolar lines to the image. For this purpose, we needed once more to find the key points using swift, match the points, and choose the 200 best matches. After that, we needed to get the fundamental matrix F, which was done by running the findFundamentalMat() function. The lines were received using the computeCorrespondEpilines() function and then drawing the lines to the image by iterating across the lines in the draw\_lines function, which we used from the week 3 exercise.

Chart

Description automatically generated

**Image Depth**

This part took us the most time, as we spent hours finding the mistake in rectification, since the map was resulting in pure noise. We used the stereoBM\_create() function to get the stereo image for the left camera. Then, the stereo for the right camera was done by running ximgproc.createRightMatcher(), which was necessary for later, as we decided to use wls\_filter to get the best results possible.

After setting up a couple of parameters for the result from StereoBM\_create(), we computed the disparity map for both left and right images by running compute() function.

In the last step, we used ximgproc.createDisparityWLSFilter() for the left camera, setLambda and setSigmaColor, and in the last step we filtered the data, using filter() function. The final results are pretty decent and can be seen below.

The final depth coordinate to track the object in 3D is then computed from the disparity map by looking at the object's centroid from the tracking task and finding the depth in the surrounding area. There are many parameters that can be tuned, which highly change the final results.

A picture containing background pattern

Description automatically generated

A picture containing dark, light, night sky

Description automatically generated