3D Perception and Sensor Fusion – Lab 1: Stereo Reconstruction

Ryan Crowell & Enis Hidri

3D Perception and Sensor Fusion

Objective

The objective of this lab is to understand and implement the stereo reconstruction equations and usage with two cameras. A stereo vision system allows for the recovery of 3D information from 2D images by exploiting geometric constraints between a calibrated stereo pair. This lab walks through all stages of stereo vision:

1. Monocular calibration
2. Stereo calibration
3. Rectification
4. Sparse reconstruction
5. Dense reconstruction
6. Comparison between block matching and semi global matching.

1. Monocular Calibration

The first step involves calibrating each individual camera to recover intrinsic parameters and distortion coefficients. This is done by observing a known calibration pattern (chessboard) from multiple perspectives. With the known distances between points and the distance between the two cameras the calibration process can easily estimate the camera's intrinsic parameter matrix (K) and distortion coefficients using OpenCV's cv2.calibrateCamera() function.

```
# Compute the camera parameters given all the samples collected
print('- Found pattern in %d/%d images' % (len(images_used), len(files)))
print('- Calibrating...')
error, K, dist, rvecs, tvecs = cv2.calibrateCamera(obj_points, img_points, (w, h), None, None)
```

Figure 1: cv2 library calibration function

To run the code the following has to be typed into the terminal with the usage of the ArgumentParser class within the *argparse* library. These parameters are defined and set up within the desired executed function.

```
## Calibation monocular (intrinsic)
python 3dpsf_lab_1_stereo/monocular_calibration.py \
--row_corners 8 \
--col_corners 12 \
--squares_size 0.02 \
--images_dir ./sample_data/calibration/left/ \
--out_calib ./sample_data/left_calib.yaml
```

Figure 2: Terminal input using arparse library parameters

After running this line for each camera a .yaml file that contains all of the intrinsic parameters of the camera along with the distortion coefficients and the mean reprojection error. These different bits of data describe how the camera converts 3D points on a 2D image plane with the error associated along with how light gets distorted through the lens.

```yaml
%YAML:1.0
---
camera_matrix: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ 1007.4544869794573, 0., 942.05085125440689, 0.,
       1005.604205583193, 560.50371017508269, 0., 0., 1. ]
distortion_coefficients: !!opencv-matrix
   rows: 1
   cols: 5
   dt: d
   data: [ -0.12575445402717594, 0.084823757684138495,
       0.00432355646537462, -0.00054770853928695, -0.039309763357041907 ]
mean_error: 0.28506118965290589
```

Figure 3: .yaml file with cameras intrinsics, distortions, and error

The mean error we see here is within a good range of what we should expect from a well calibrated camera. When running the debug parameter the chessboard is detected properly, this is known because the detected chessboard images show points at each inside corner and sharing colors with all other corners in the same column with a diagonal line leading to the next column in sequence. The detected versus reprojected images correctly place two different crosses one for each at first glance they look as if they overlap correctly but on a few they are off by a few pixels (Figure 6). Observe the three figures below:

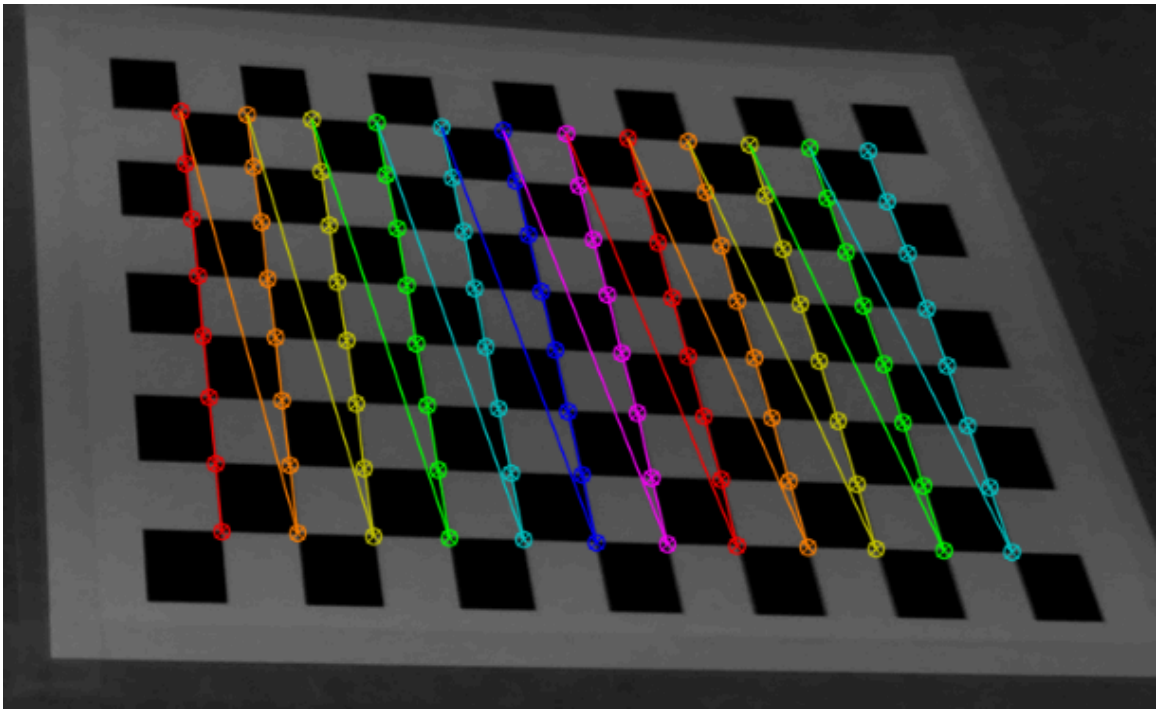

Figure 4: Chessboard detection

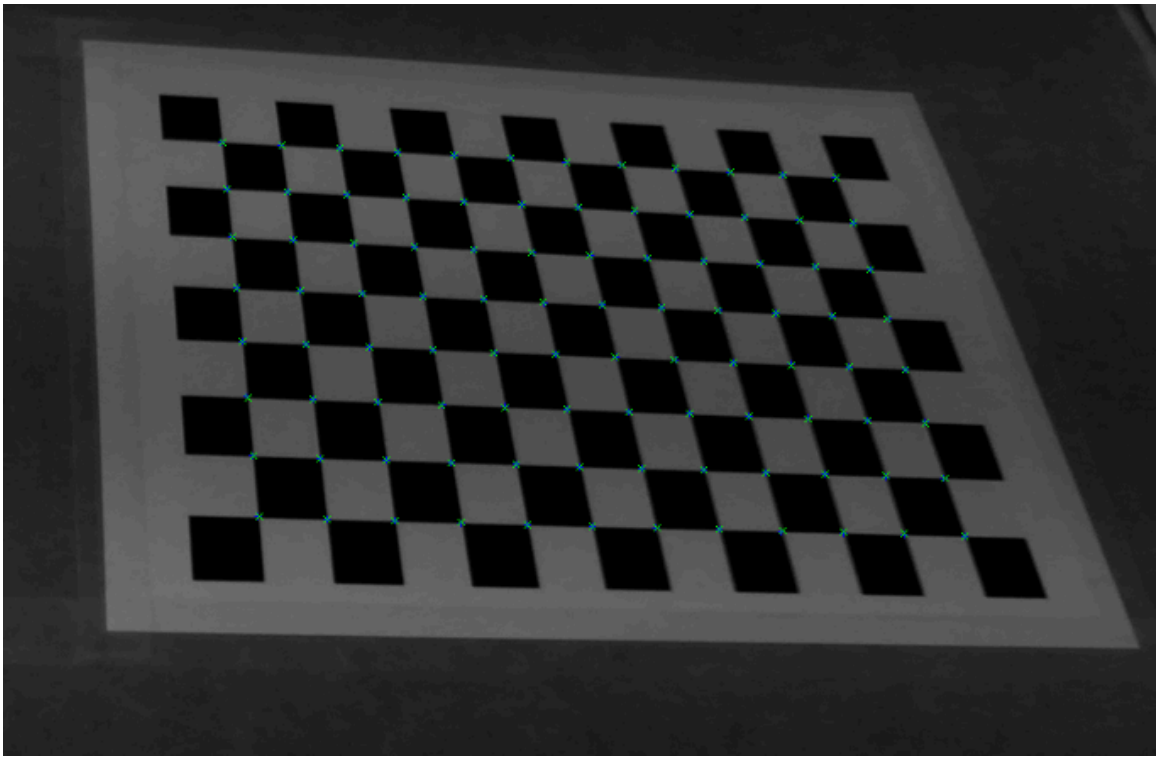

Figure 5: Chessboard detection vs. Reprojection

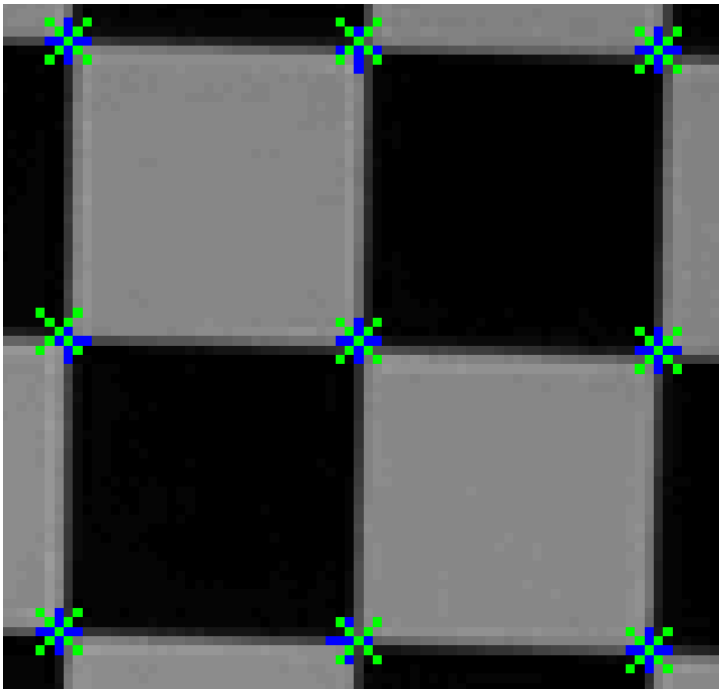

Figure 6: Zoomed in photo of Figure 5

Exercise 1

As observed in Figure 6 the blue and green crosses don't all exactly align which indicates an imperfect calibration. However for this lab it is within reasonable to perform reconstruction. The mean reprojection error is less than 0.3 which as described before is great calibration.

2. Stereo Calibration

After calibrating the individual cameras, the next step is to determine the extrinsic parameters – the rotation (R) and translation (T) between both cameras. This process is performed using the cv2.stereoCalibrate() function. These parameters describe the relative pose of one camera with respect to the other, allowing for 3D triangulation of matched points. The function outputs a .yaml file similar to the one seen in the monocular calibration with some added matrices that assist in describing a camera with respect to the other.

```yaml
%YAML:1.0
---
image_width: 1920
image_height: 1080
stereo_rotation: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ 0.99998228388551458, -0.0012024820848954461,
       -0.0058297471596570945, 0.001222567651660279, 0.99999332544996422,
       0.0034430195844675359, 0.0058255680793497158,
       -0.0034500858478331568, 0.99997707956942461 ]
stereo_translation: !!opencv-matrix
   rows: 3
   cols: 1
   dt: d
   data: [ -0.1187257705187383, 0.00021950638934707399,
       0.0067753814166330653 ]
left_camera_matrix: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ 1007.4544869794573, 0., 942.05085125440689, 0.,
       1005.604205583193, 560.50371017508269, 0., 0., 1. ]
left_distortion_coefficients: !!opencv-matrix
   rows: 1
   cols: 5
   dt: d
   data: [ -0.12575445402717594, 0.084823757684138495,
       0.00432355646537462, -0.00054770853928695, -0.039309763357041907 ]
right_camera_matrix: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ 1016.4425641769585, 0., 965.49882679615723, 0.,
       1013.5308405859081, 565.94900572322797, 0., 0., 1. ]
right_distortion_coefficients: !!opencv-matrix
   rows: 1
   cols: 5
   dt: d
   data: [ -0.1222450334800107, 0.072000474169391662,
       0.0051386544094590 39, -0.0018588440096023251,
       -0.031883663749881795 ]
essential_matrix: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ -7.0046127326421567e-06, -0.0067760935098981831,
       0.00019617358725641107, 0.0074669064421303676,
       -0.00041776137541162084, 0.11868355051238841,
       -0.0003646527870014 0115, -0.11872471412514173,
       -0.00040749548632719164 ]
fundamental_matrix: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ 3.9040746617403 98e-08, 3.7836567474777748e-05,
       -0.02234585401258786, -4.1736936625968104e-05,
       2.3394108452955999e-06, -0.63033042542204709, 0.02564912248418813,
       0.63598411853412373, 1. ]
```

Figure 7: Stereo .yaml file

Exercise 2

Even though this also outputs all the info described with the monocular calibration it is better to run the previous one first to provide a stable starting point for the stereo calibrate such that it has fewer unknowns leading to a more accurate and stable result of the extrinsic parameters.

4. Rectification

Rectification simplifies the matching process by aligning epipolar lines horizontally. This process creates new virtual camera parameters where corresponding points in both images lie on the same image row (same Y-coordinate). The rectification is computed using cv2.stereoRectify().

```
R1, R2, P1, P2, Q, roi1, roi2 = cv2.stereoRectify(stereo_calib["left_camera_matrix"],
                                                  stereo_calib["left_distortion_coefficients"],
                                                  stereo_calib["right_camera_matrix"],
                                                  stereo_calib["right_distortion_coefficients"], im_size,
                                                  stereo_calib["stereo_rotation"],
                                                  stereo_calib["stereo_translation"], alpha=param.alpha)
left_maps = cv2.initUndistortRectifyMap(stereo_calib["left_camera_matrix"],
                                        stereo_calib["left_distortion_coefficients"],
                                        R1, P1, im_size, cv2.CV_16SC2)
right_maps = cv2.initUndistortRectifyMap(stereo_calib["right_camera_matrix"],
                                         stereo_calib["right_distortion_coefficients"],
                                         R2, P2, im_size, cv2.CV_16SC2)
```

Figure 8: cv2 stereo rectification and undistort functions

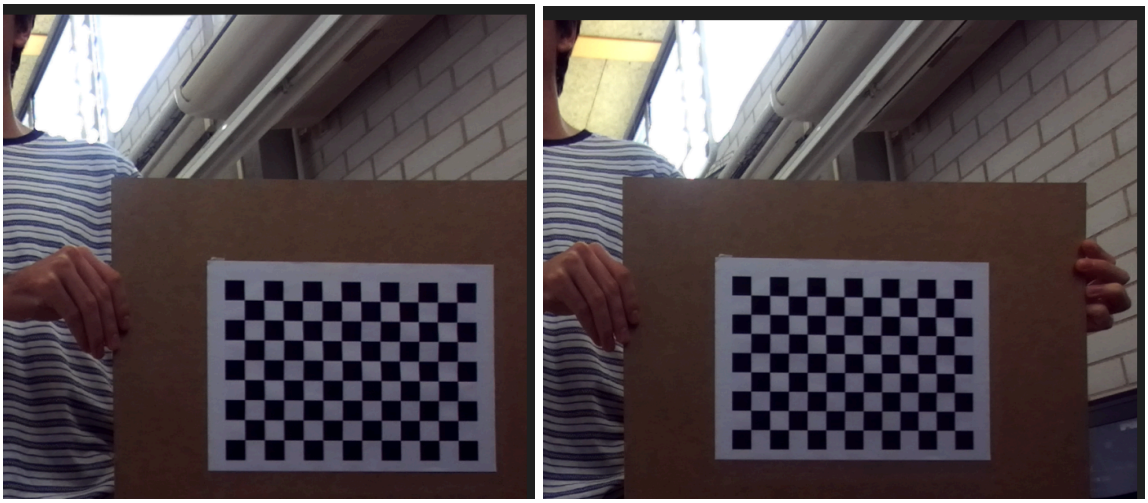

Figure 9: Alpha 0 rectified left and right images

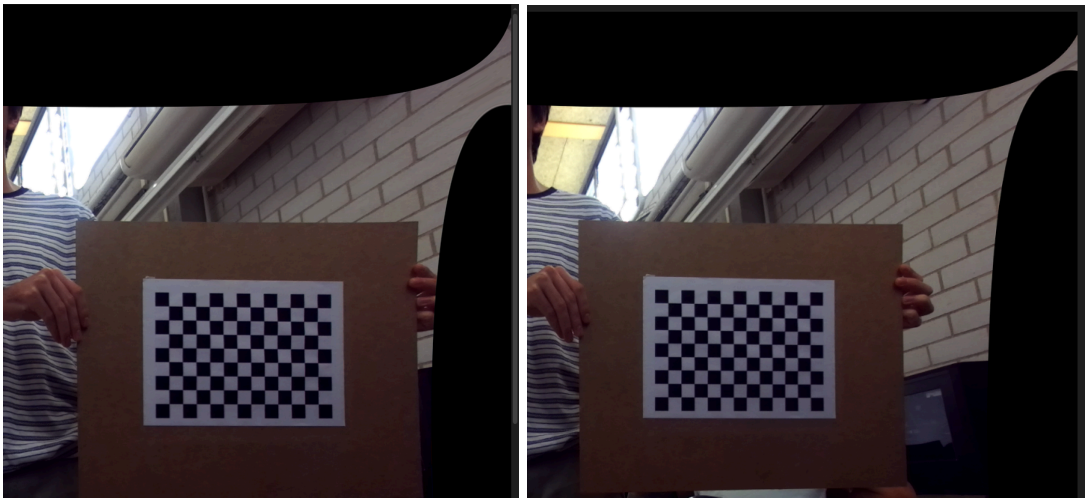

Figure 10: Alpha 1 rectified left and right images

Exercise 3

The alpha parameter controls how much of the original image you want preserved. When rectifying the images the function warps the images such that epipolar lines become horizontal which helps with matching features across the two different images. Because of the warping some areas of the image get compressed or extended causing some invalid pixels. Alpha helps with determine how much of the original image you want, alpha 1 with leave all of the original image in view and have no data for the other pixels needed to make it square while alpha 0 will zoom into the photo until only valid pixels remain but removing some valid pixels as well (such as the pixels in the top right of the images in Figure 10).

4. Sparse Reconstruction and Exercise 4

Using the rectified stereo images, sparse feature-based reconstruction can be performed. This step involves detecting feature points, matching them between images, and triangulating their 3D positions using cv2.triangulatePoints(). SIFT is used as the default feature detector and descriptor.

Below in Figure 11 the output for the debug of the sparse matching is displayed showing first the left and right images with the key features shown, second the matches between those key features in the reconstructed image and lastly the 3D location of all the matched features using a heatmap to display points in and out of the page.

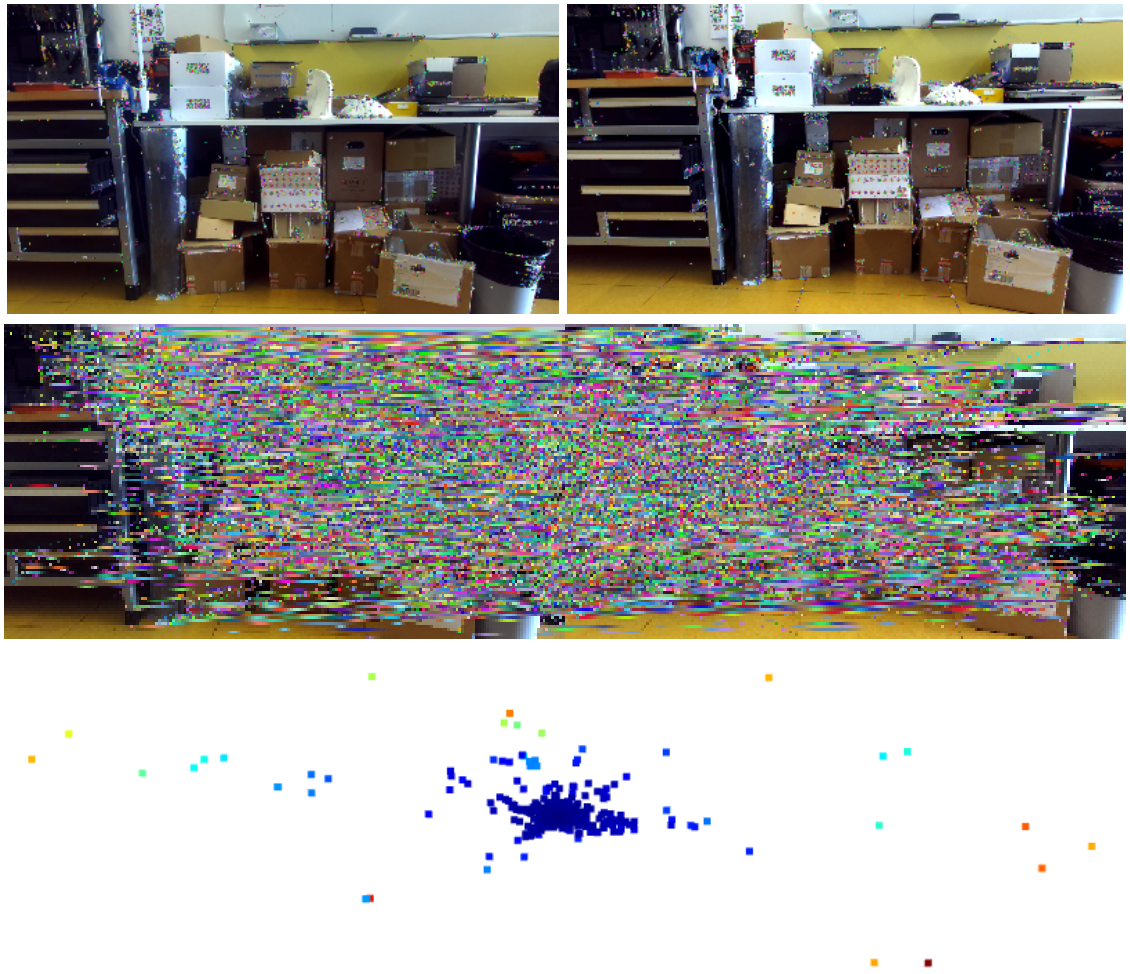

Figure 11: Sparse matching Debug

In this figure the important features are the ones in blue that represent matches found in the two images, the points that are further spread out and straying toward the color red are outliers in the matches as most pixels in the image are relatively close to one another in the 3D space. This could be due to unsatisfactory matches because the function is not taking advantage of the known camera geometry.

5. Epipolar Constraint Filtering

In the previous matching step, some matches may not satisfy the epipolar constraint. This exercise filters out matches that do not have approximately the same Y-coordinate in both images, ensuring they lie along the same epipolar line. The code added for exercises 6 below adds this constraint utilizing the camera's known geometry.

```python
# Reconstruct the 3D points if the camera calibration is available
if param.rect_stereo_calib_file:
    print('- Computing the sparse 3d point cloud...')
    # Collect the 2D projections on each image
    p2dL = np.array([kpL[m.queryIdx].pt for m in matches]).reshape(-1, 1, 2)
    p2dR = np.array([kpR[m.trainIdx].pt for m in matches]).reshape(-1, 1, 2)

    # Triangulate the points
    p3dh = cv2.triangulatePoints(left_P, right_P, p2dL, p2dR)
    p3d = cv2.convertPointsFromHomogeneous(p3dh.T) # De-homogenize
    # De-homogenize
    p3dh = p3dh / p3dh[3]
    p3d = p3dh[:3].T # And transpose, for convenience

    # Remove obvious outliers:
    #  - points behind the camera
    p3d = p3d[np.where(p3d[:,2] > 0)]
    #  - points too far from the camera (>10 meters, assuming the calibration was done in meters)
    p3d = p3d[np.where(p3d[:,2] < 10)]

    # Convert the points to Open3D for visualization/saving
    pcd = o3d.geometry.PointCloud()
    pcd.points = o3d.utility.Vector3dVector(p3d)
```

Figure 12: Code for removing obvious ooutliers

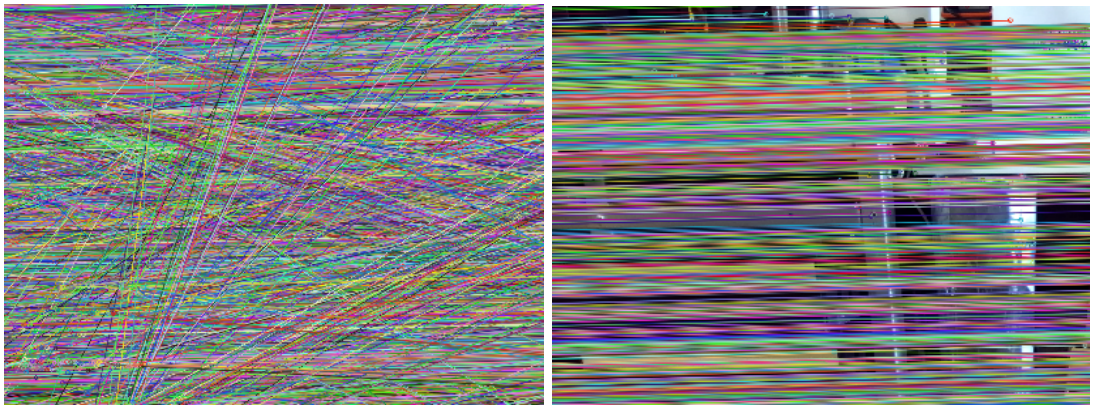

Figure 13: Results from removing the outliers

As it can be seen from the two images of the results the outlier rejection and restriction of points now shows only straight horizontal lines to matches between points which leads to more correct matches.

6. Exercise 6 (Feature Detector Comparison)

Alternative feature detectors/descriptors such as ORB, AKAZE, or BRISK can be tested to compare their performance against SIFT in terms of computation time and accuracy. To do this simply change the line “sift = cv2.SIFT_create()” to “sift = cv2.ORB_create()” this uses the detector ORB to find matches between the points and uses binary encoding!

Figure 14: Matches using ORB

It is obvious through visual inspection that the ORB descriptor finds fewer matches than SIFT. ORB seemed to run much faster however. Even still the lack of features is a major downside of ORB. There is no wonder why SIFT is the typical descriptor used and preferred in many applications.

7. Dense Reconstruction

Dense reconstruction aims to estimate depth for all image pixels using stereo matching algorithms. The Block Matching (BM) algorithm computes disparities for each pixel based on local intensity differences. Parameters such as numDisparities and blockSize influence accuracy and smoothness of the disparity map. Below find the block matching function used from the cv2 library giving the results found in Figure 16 and 17.

```
73      stereo = cv2.StereoBM_create(numDisparities=num_disparities,
74                                   blockSize=block_size)
75      stereo.setMinDisparity(min_disparity)
76      stereo.setPreFilterType(prefilter_type)
77      stereo.setPreFilterSize(prefilter_size)
78      stereo.setPreFilterCap(prefilter_cap)
79      stereo.setDisp12MaxDiff(disp12_max_diff)
80      stereo.setTextureThreshold(texture_threshold)
81      stereo.setUniquenessRatio(uniqueness_ratio)
82      stereo.setSpeckleWindowSize(speckle_size)
83      stereo.setSpeckleRange(speckle_range)
```

Figure 15: Function for Block Matching Algorithm

Figure 16: Original Statue photo

Figure 17: Disparity Map

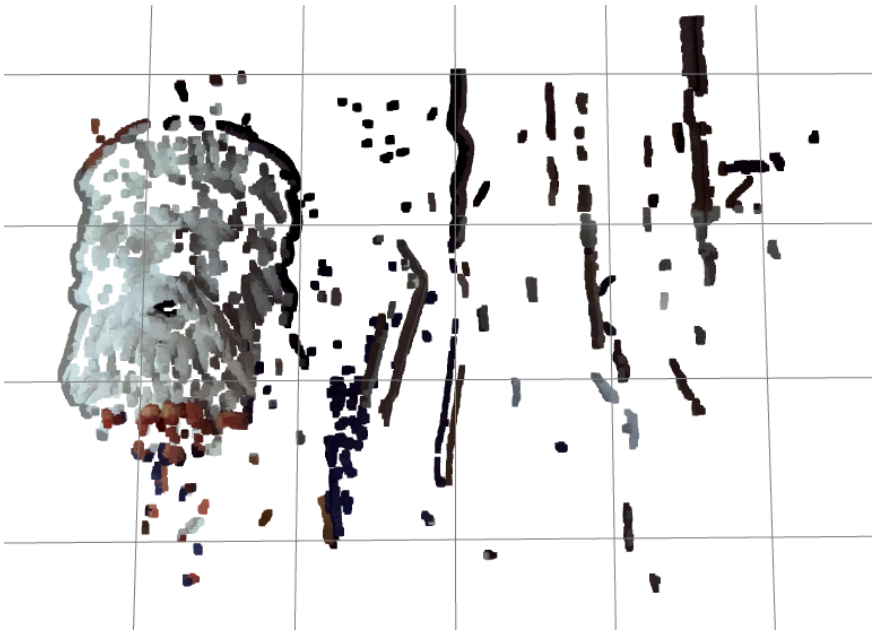

Figure 18: 3D Point cloud of matches

The above results are using the best parameters found through tuning with the GUI. The most ideal parameters are below.

```
# Current parameters: STATUE ALPHA 0
block_size= 2 * 10 + 5
num_disparities=16 * 25
min_disparity=134
prefilter_type=0
prefilter_size=15
prefilter_cap=62
texture_threshold=100
uniqueness_ratio=15
speckle_size=60
speckle_range=25
disp12_max_diff =25
```

Figure 19: Ideal Parameters

It is worth to mention the most important parameters that have been tuned:

- **Block_size**: size of the comparison window; larger values smooth the disparity map but reduce detail.
- **numDisparities**: range of disparity search; defines how far features are shifted between images and affects depth range.
- **minDisparity**: starting disparity value; aligns the search range to the actual image offset.
- **uniquenessRatio**: filters ambiguous matches; higher values reduce mismatches but can leave more empty areas.
- **textureThreshold**: removes regions with low texture that produce unreliable matches.
- **speckleWindowSize / speckleRange**: eliminate small noise regions in the disparity map for smoother results.

8. Semi-Global Block Matching (SGBM)

The Semi-Global Block Matching algorithm refines the dense reconstruction by enforcing consistency between neighboring pixels while maintaining efficiency. This section involves replacing the BM algorithm with cv2.StereoSGBM_create() and comparing results.

```
# Current parameters: STATUE ALPHA 0 semi global matching
block_size= 11
num_disparities=16 * 20
min_disparity=153
uniqueness_ratio=9
speckle_size=100
speckle_range=31
disp12_max_diff =8
prefilter_cap=63
```

```
86      block_size= 11
87      num_disparities=16 * 20
88      min_disparity=153
89      uniqueness_ratio=9
90      speckle_size=100
91      speckle_range=31
92      disp12_max_diff =8
93      prefilter_cap=63
94
95      stereo = cv2.StereoSGBM_create(minDisparity=min_disparity,
96                                     numDisparities=num_disparities,
97                                     blockSize=block_size,
98                                     P1=8 * 3 * block_size ** 2,
99                                     P2=32 * 3 * block_size ** 2,
100                                    disp12MaxDiff=disp12_max_diff,
101                                    preFilterCap=prefilter_cap,
102                                    uniquenessRatio=uniqueness_ratio,
103                                    speckleWindowSize=speckle_size,
104                                    speckleRange=speckle_range,
105                                    mode=cv2.STEREO_SGBM_MODE_SGBM_3WAY
106                                    )
```

Figure 20: Semiglobal cv2 function and ideal parameters

Figure 21: Original and Disparity map of semiglobal

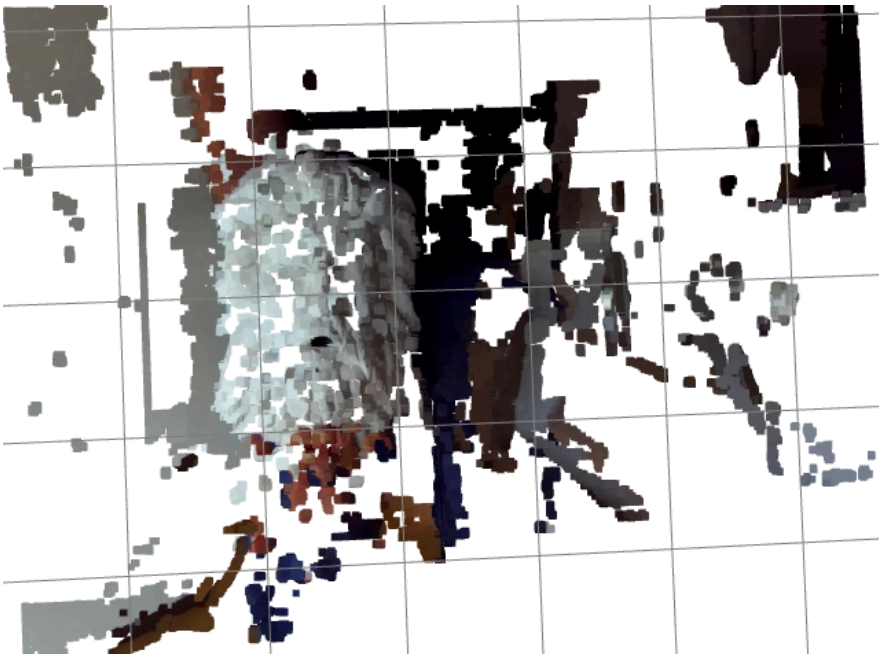

Figure 22: 3D point cloud using semiglobal

With semiglobal the results yielded a worse disparity map but a much better defined 3D point cloud when comparing it to the typical block matching. This is because it is not as strict with what it considers a feature match with less accuracy of the z axis.

9. Capturing New Data

Finally, new data was captured using the ZED2 stereo camera available in the lab. The data collection process included capturing calibration and reconstruction images. The same pipeline was applied to this new dataset to evaluate robustness and generalization.

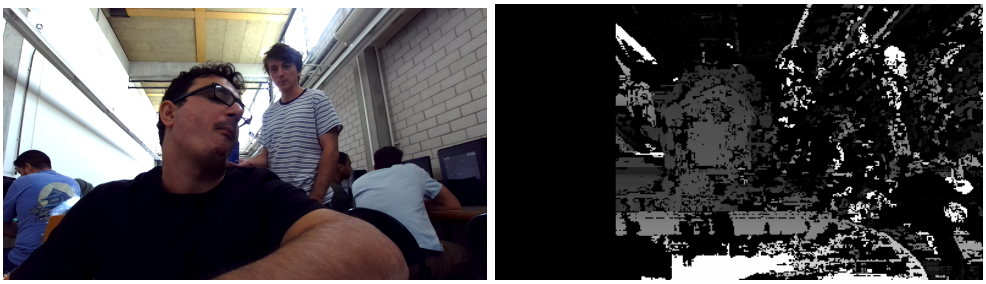

Figure 23: Personally Acquired Photo

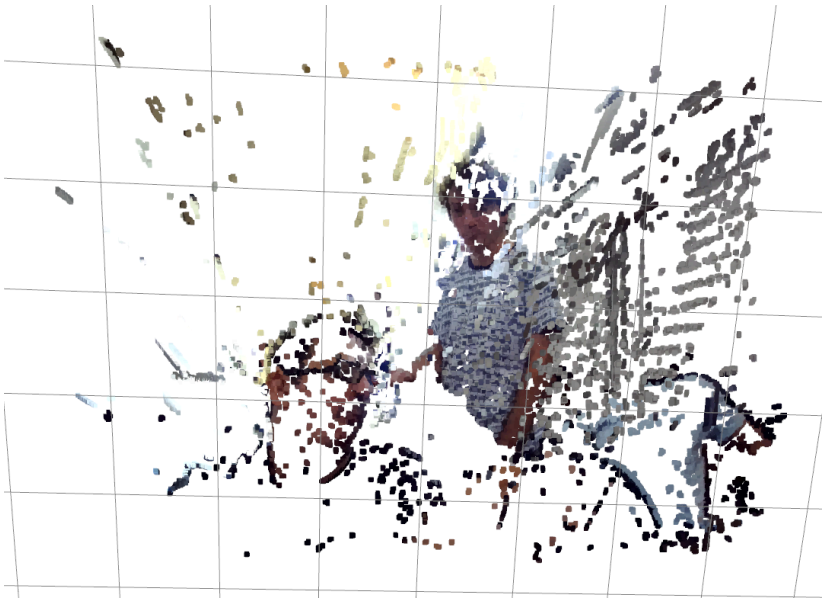

Figure 24: Returned 3D Point Cloud

As observed in the 3D reconstruction, the process successfully recovered most of the main scene features. The first person in the foreground is not well defined, likely due to a lack of distinctive texture or recognizable features. In contrast, the person in the background and the wall were reconstructed almost completely. However, in this implementation it was difficult to achieve an optimal trade-off between accurate calibration for nearby and distant objects simultaneously.

Conclusion

In this lab, we implemented a complete stereo vision pipeline. Through successive steps of calibration, rectification, and reconstruction, we demonstrated the geometric principles behind stereo vision. Sparse reconstruction provided feature-level depth, while dense reconstruction extended this to full-image depth estimation. The results highlight the importance of accurate calibration and rectification for reliable 3D perception along with comparison of different detectors and descriptors as well as different matching methods.