DEPARTMENT OF COMPUTER SCIENECE

# Programming Project Computer Vision – Report

Kunis Daniel, Maier Florian, Rüba Simon
June 26, 2020

## PREFACE

This report presents our approach for solving typical problems in Visual Geometry and gained insights and observations. The code can be accessed here.

## OBJECTIVE

Insert an artificially rendered object into a movie such that everything is geometrically consistent (we will not worry about correct shading, occluders in front of the inserted object etc.).

## CAMERA INTERNAL CALIBRATION

The code for camera calibration is located in the *src/camera_calibration.py*. Below we will thoroughly describe the chosen algorithm.

In order to recover the camera matrix $P$:

- We used the OpenCV's sampled chessboard image as a sufficiently richly structured scene and took several photos of it that are located in the 'resources/chessboard' folder of our project. The advantage of using a chessboard is that it establishes a new coordinate system within itself: we can address the internal corners of the chessboard like points where '(0, 0)' is the upper left internal corner. This significantly simplifies the problem of finding correspondences of points that we will use for finding $P$.

- Now we consider each image one by one and find the coordinates of the corners of the chessboard using the cv2.findChessboardCorners function. After the image coordinates of the corners are retrieved we improve their accuracy using cv2.cornerSubPix. After iterating over all images we have two arrays: 'obj_points' – with object points in the calibration pattern coordinate space; and 'img_points' – projections of calibration pattern points for each image.

- Solve the correspondences for $P$ using the cv2.calibrateCamera method. Under the hood it consists of 3 steps:
  - *Initialization.* Compute an initial value for the camera matrix using DLT. Assume that distortion coefficients are zero.
  - *Initial estimation.* Estimate the initial camera pose as if the intrinsic parameters have been already known using *cv2.solvePnP()*.
  - *Iterative optimization.* Minimize the reprojection error using the Levenberg-Marquardt optimization algorithm.

  The reprojection error is a geometric error and it is the mean value of squared distances between the observed projections img_points (that were retrieved by the corner finding algorithm) and the projected (using *cv2.projectPoints*) obj_points.

The described method is alike the *Gold Standard* algorithm described in HZ in 4.5 (p114).

### Observation 1

During our first iteration we used the chessboard images provided by OpenCV which are of size *640x480*. After switching to images made by our camera without changing any of the parameters we saw an excessive rise in the error (from *0.024* up to *0.5*) even though the quality and number of input images was increased from *13* up to *262* and we were expecting an error decrease. The explanation of this rise was that our new images were much bigger – *3024x4032* – and the cv2.cornerSubPix function could not find the absolute minimum within the given boundaries (we specify *winSize* which is half of the side length of the search window) since the search window became relatively smaller to overall image size. We solved it by resampling large input images that are bigger than *1920x1080* using the cv2.resize using the *INTER_AREA* interpolation method which is considered best for downsampling. This greatly improved the result – reprojection error became *0.141*.

## Inserting static shape into 3D scene using calibration rig

In the beginning, as a first iteration, we decided to insert a cube into a 3D scene using a calibration rig which greatly simplifies the objective since we can use the chessboard as the coordinate system and we can easily find the point correspondences.

The implementation of this step can be found in *src/drawing.py* and can be decomposed into these steps:

- Convert input video to image sequence.
- Iterate over all images and for each find the corners of the chessboard using cv2.findChessboardCorners, calculate the pose of the chessboard using RANSAC algorithm (implmetned in the cv2.solvePnPRansac function), project the 3D points of the cube (in chessboards' coordinate system) to the 2D image plane, draw the cube and save the image.
- Convert the image sequence to a video.

We chose RANSAC because of it's robustness and resistance to outliers.

### OBSERVATION 2

Do not use a square chessboard pattern. It might sound obvious, but we did not think of it in the beginning. Using a symmetrical rig confuses the algorithm and corrupts camera calibration and can be revealed only during the drawing of the cube.

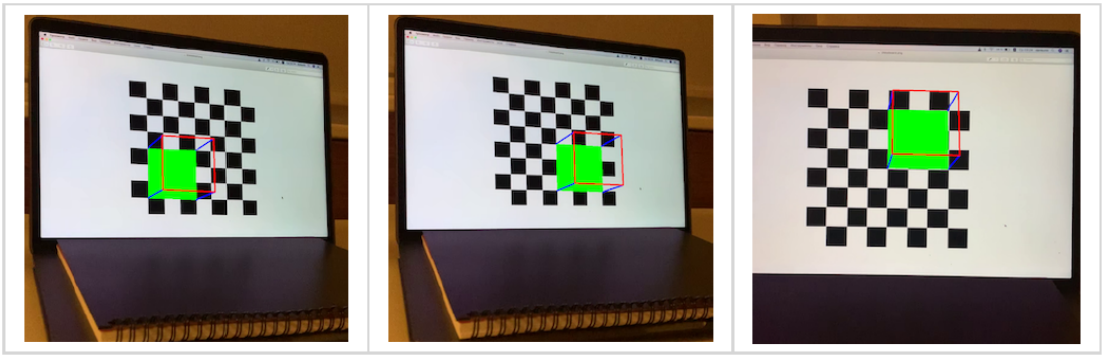

Figure 0.1: Consider this image sequence. All of them are from the same video.

## GETTING THE FRAMES

Our function *video_to_frames* extracts frames from a video and saves them to a directory. We give the possiblity to skip frames (e.g. only take every 6th frame) in order to reduce the number of frames and to speed up the process. We ran the program twice: once with every frame and once with every 6th frame. The first option has a really long computation time. The second option is fast. The difference between the videos is hardly visible. We provide both videos.

## FEATURE MATCHING AND FINDING KEYFRAMES

The feature matching is done in our *feature_matching* function in *point_correspondences.py*.

We tried keypoint detection with both SIFT and Harris Corner Detector. Since we got better results with SIFT (we got more points and they showed better results for scaling and rotating),

we are using SIFT. However, we left our function that uses the Harris Corner Detector in the code (see: *point_correspondences.py - get_key_points*).

Figure 0.2: Difference between Harris Corner Detector (Left) and SIFT (Right).

**Algorithm:**

The desired output: We want to find *keypoints* that we can trace through all of the *keyframes*. Moreover, we want all *intermediate_frame_matches* (i.e. of non-keyframes), containing the matches between their frame points to the keypoints of the previous and following keyframe.

- Using the *sift.detectAndCompute* function we find the keypoints of the first frame.
- We save all sift points for the first keyframe by using a data structure from *A fast 3D scene reconstructing method using continuous video*.
- In order to find keyframes, we iterate through all frames:
  - detect sift points for the current frame
  - using the *FLANN* algorithm we try to find 2 points in the current frame which are possible matches for a keypoint of the previous keyframe. Then we filter the results with Lowe's ratio test. The best result (i.e. the point which match the keypoint) must be less than 70% than the distance of the second best.
  - Optional: draw the matches and save the image

– Check if the current frame fulfills the requirements to be a keyframe. We decided to choose the requirements presented in *A fast 3D scene reconstructing method using continuous video*, chapter 3.1, adjusted for our objective:
  * The average feature distance in pixel must be more than ImageWidth/17.
  * Feature point matcches did not drop dramatically with respect to previous match.

The acutal output: In our last run of our program we got 56 frames. 11 of them were keyframes (the following frames were keyframes: 0, 3, 9, 13, 16, 19, 27, 35, 41, 49, 55). The distribution of the keyframes looks very good.

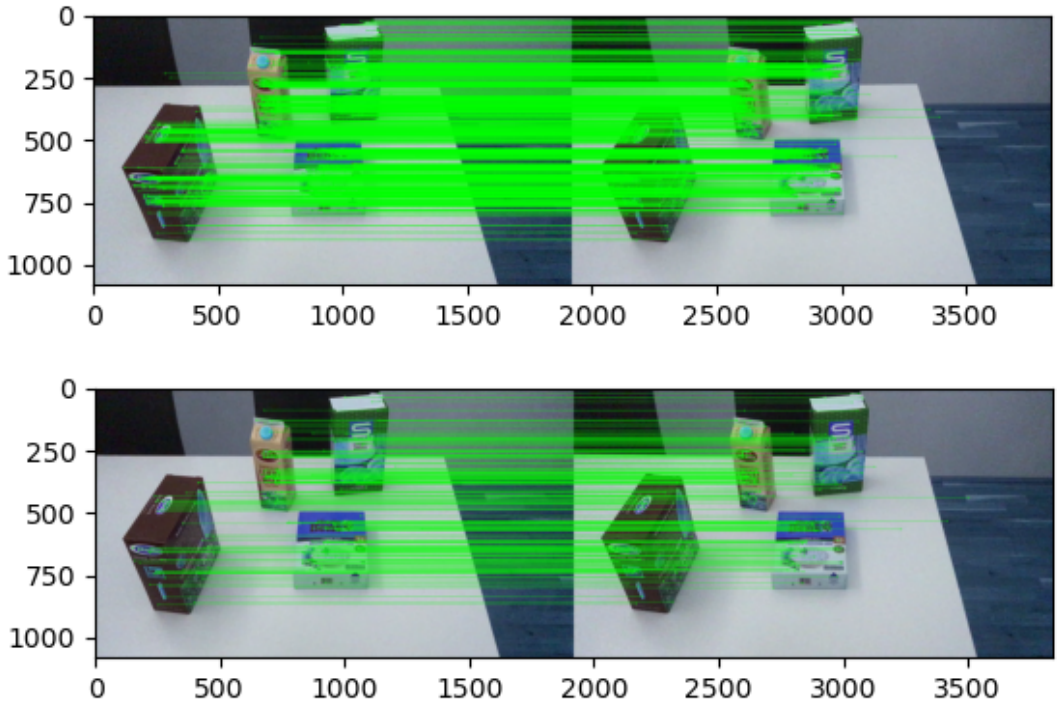


Figure 0.3: Matching Keypoints. First Picture shows matching between frame 0 and frame 1. Second Picture shows matching between frame 9 and frame 10

## STEREO RECONSTRUCTION

After finding the keyframes and matching the keypoints, we want to compute the camera matrices for every frame. This is done in our *stereo_reconstruction* function.

**Preparation:** We created a *Camera* class where we can save the rotation $R$, translation $t$

and projection matrix *P*.

For the initial reconstruction we used the algorithm presented in HZ (*chapter 18.6, algorithm 18.3,* "Extending the baseline"). We set our first keyframe as our origin (i.e. no rotation and translation). Using the previously calculated matches we calculate the camera of the last keyframe.
Then we calculate the 3d points by triangulation. Having them, we calculate the cameras for all other keyframes by resectioning using the *cv2.solvePnPRansac* function. At the end, we recover all intermediate cameras.

## DRAWING THE CUBE

At this time we have the cameras for all frames. Using a photo-editor (e.g. GIMP), we select 4 points in our first keyframe and the same 4 points in the last keyframe. By triangulating we then get 4 world points. We then hardcode the other 4 worldpoints, so it forms a cube.
After that we iterate over all frames and by using the *cv2.projectPoints* function, we get the 2d image points of the cube for every frame. By connecting all corners, we draw the cube. Every frame gets saved and at the end they get converted to a video.

**Additional Information:** Since our final result was pretty good in our opinion, we decided that bundle adjustment is not needed.

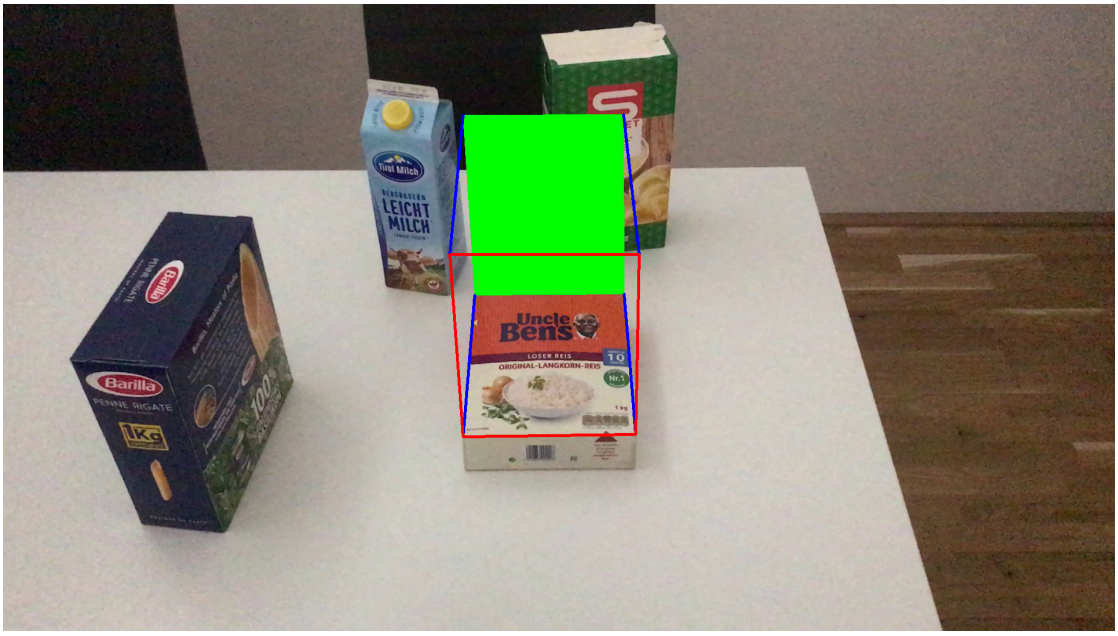


Figure 0.4: Drawing of the cube. First frame (out of 56)


Figure 0.5: Drawing of the cube. Frame 28

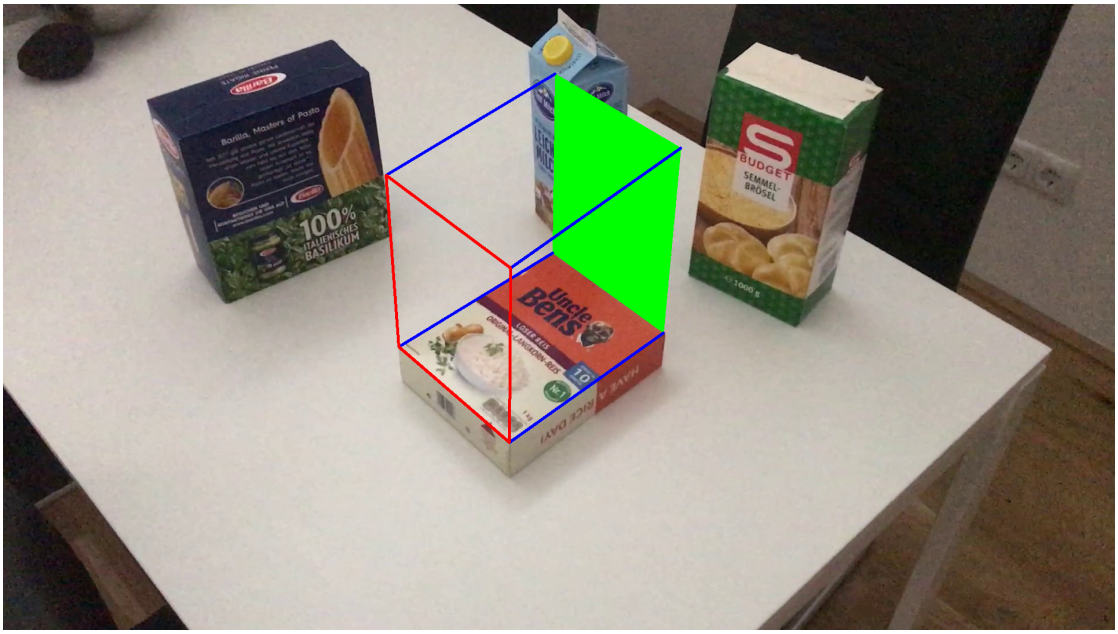


Figure 0.6: Drawing of the cube. Last frame (out of 56)

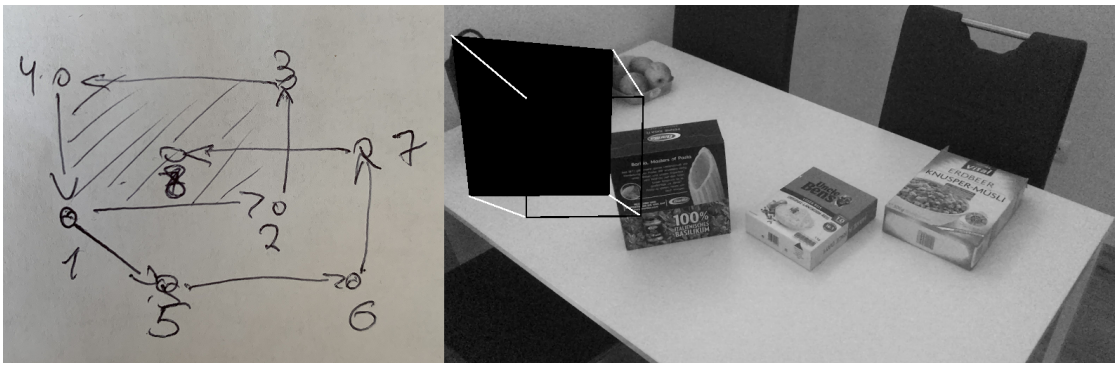


Figure 0.7: How we tried to fix our cube. Point 1 to 8 represent the order of our hardcoded cube

# CONTRIBUTIONS

| | Kunis Daniel | Maier Florian | Rüba Simon |
|---|---|---|---|
| Camera Calibration | | 50% (1) | 50% (2) |
| Utils | 30% | 30% | 40% |
| Feature Matching | Main part (~75%) | SIFT and HCD (3), drawing matches | |
| Stereo Reconstruction | 100% | | |
| Drawing Functions | | | 100% |
| Drawing the actual cube (4) | Pair programming (with screen share) | | |
| Refactoring | at half-time | | at the end |
| Report | (5) | everything else | |

(1) with sample chessboard images from the internet to test the algorithm
(2) with our photos of the chessboard (producing the actual results)
(3) The Code of SIFT and Harris Corner Detector wasn't used, but it is still in the .py file.
(4) Including hardcoding 4 worldpoints, triangulation of 4 points and orientate the cube.
(5) Camera Calibration, drawing on rig