Computer Vision 22/23 Course Assignment 2 Report

Voxel-based 3D Reconstruction

G. Picarella (2713810)

March 4, 2023

1 Camera calibrations

1.1 Intrinsic camera matrices

Our calibration approach is based on a fixed sampling interval $s = 20$ for each camera calibration video. A sampled frame with sampling s at index k is considered valid for the calibration only if `cv2.findChessboardCorners(...)` is able to detect the checkerboard corners. Because of that, even if all the calibration videos have the same number of frames, each camera uses a different number of training frames during calibration $(97, 125, 82, 176)$. Each C_{I_i} is the estimated intrinsic camera matrix for the i-th camera.

$$C_{I_1} = \begin{pmatrix} 489.49 & 0 & 330.11 \\ 0 & 492.52 & 223.05 \\ 0 & 0 & 1 \end{pmatrix}$$

$$C_{I_2} = \begin{pmatrix} 487.20 & 0 & 329.25 \\ 0 & 488.06 & 234.36 \\ 0 & 0 & 1 \end{pmatrix}$$

$$C_{I_3} = \begin{pmatrix} 486.70 & 0 & 320.67 \\ 0 & 485.13 & 234.03 \\ 0 & 0 & 1 \end{pmatrix}$$

$$C_{I_4} = \begin{pmatrix} 487.26 & 0 & 346.80 \\ 0 & 489.38 & 243.62 \\ 0 & 0 & 1 \end{pmatrix}$$

1.2 Extrinsic camera vectors

For the estimation of each camera extrinsic parameters we based our approach on the following assumptions: the checkerboard distance from the camera, its rotation and the video resolution makes the `cv2.findChessboardCornes` detection fail for every video frame; the checkerboard is fixed for the entire video length. Because of those reasons, for each camera, our method selects the first video frame in checkerboard.avi, detects the outer checkerboard corners with an automatised approach with minimal user interaction (more info in the extra-tasks section), computes the inner corners by linear interpolation in the warped 2D space and finally estimates the camera extrinsic vectors with `cv2.solvePnP(...)`. Each C_{T_i} and C_{R_i} are the estimated extrinsic translation and rotation vectors for the i-th camera.

$$C_{T_1} = \begin{pmatrix} 444.42 & 904.81 & 3847.76 \end{pmatrix}$$

$$C_{T_2} = \begin{pmatrix} 523.43 & 1418.19 & 3288.25 \end{pmatrix}$$

$$C_{T_3} = \begin{pmatrix} -359.73 & 1245.22 & 3646.10 \end{pmatrix}$$

$$C_{T_4} = \begin{pmatrix} -60.62 & 932.34 & 4578.30 \end{pmatrix}$$

$$C_{R_1} = \begin{pmatrix} 1.605 & -0.691 & 0.571 \end{pmatrix}$$

$$C_{R_2} = \begin{pmatrix} 1.199 & -1.169 & 1.187 \end{pmatrix}$$

$$C_{R_3} = \begin{pmatrix} 0.220 & 2.104 & -2.043 \end{pmatrix}$$

$$C_{R_4} = \begin{pmatrix} 0.713 & -1.944 & 1.752 \end{pmatrix}$$

2 Foreground extraction

Our foreground extraction method comprises two main stages: the background subtractor training and the foreground mask computation (in both cases, one for each camera view). In the first stage, a Mixture Of Gaussians (MOG) subtractor is trained using the background video in HSV color space. A MOG model estimates

a mix of gaussian distributions for each pixel and based on a series of automatically estimated thresholds, it determines if a pixel is foreground or not. All the MOG models are trained with an automatic learning rate, $history = backgroundFrames$ and $backgroundRatio = 0.95$. In the second stage, the foreground mask for each frame is computed with a cascade of computations: first, the frame is converted from BGR to HSV color space; then, the noisy foreground mask F_{c_i} for camera c_i is computed with the i-th trained subtractor; then, a list of contours and their hierarchies is computed for the mask F_{c_i}. We now set F_{c_i} equal to the white shape obtained by filling the contour C_{max} which maximises `cv2.contourArea(...)`; this has the double effect of isolating the horseman mask and removing most of the noise previously contained in F_{c_i}. Next, a refinement strategy is applied as follows: a `cv2.fillPoly(..., BLACK)` and `cv2.drawContours(..., WHITE)` operations are applied for each child contour contained in C_{max} with `cv2.arcLength(...)` $\geq T_i$. In this way, we are able to restore all the black islands contained in the horseman main silhouette that would've been otherwise lost. As last step, a final thresholding operation is applied on F_{c_i} in order to remove any gray shade that may originate during the contours and areas drawing. Optionally, we give the possibility to apply a denonising step on the initial BGR frame; it shows better results in terms of details around the horseman head but, on the other hand, we spotted a general degradation of the chair legs detection and consequently set it to `False` by default. Finding a contour perimeter threshold T_i for a specific video is straightforward and fast: it's the minimum perimeter length among all the contours we are interested in within the horseman silhouette. In our case, we used the thresholds $T = (80, 80, 80, 40)$.

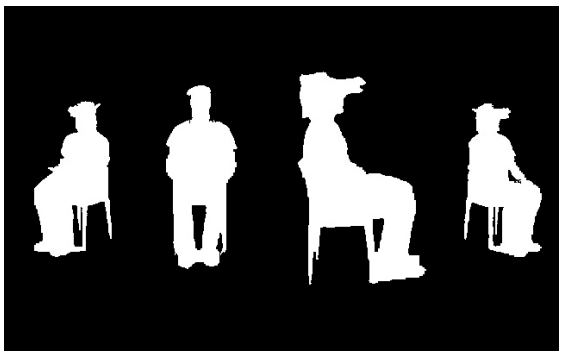

Figure 1: Foreground masks at frame 0

3 Voxels back-projection and 3D rendering

We build a look-up table encoded in a python dictionary with one entry per camera. Each entry contains a dictionary mapping a 2D pixel coordinate to a list of 3D voxel coordinates. The relationship between 2D pixels and 3D voxels is established by back-projecting every 3D voxel in the volume with each camera intrinsics and extrinsics using `cv2.projectPoints(...)`. Right before then, each 3D voxel coordinate is translated and scaled according to a specified volume center and size in world coordinates and the checkerboard square size in mm used during the camera calibration phase. The back-projection phase is very slow, that's why we compute the look-up table once and store it in a binary file for any future use. The computation has to be repeated only if the calibration data/volume center or size is changed. In our case, we implemented a parallel and memory-efficient algorithm which allows to speed-up the computation of the look-up table (more info in the extra-tasks section). In addition, for each 3D voxel we store 4 visibility booleans (one per camera). For each new call to `set_voxel_positions(...)`, we build a list containing the next frame for each camera. Next, the list of binary masks is computed by the foreground extraction routine. Finally, we render a voxel if its back-projected pixels are set to white for all foreground masks. A bottleneck can be easily spotted here, because every time we have to recompute the entire visibility array and check which voxel should be drawn. In our case, we implemented a more efficient logic for the voxel reconstruction routine which takes into account only the pixels that have changes from the previous frame (more info in the extra-tasks section). In our case, we used a volume of size 76x115x92.

3.1 Camera positions

For each camera i, the rotation matrix R_i is computed with `cv2.Rodrigues(...)`, then the camera position in openCV reference frame is found by $-R_i^T C_{T_i}$. The result has to be scaled by $\frac{1}{sm}$ where sm is the square size in mm specified in the calibration phase. Finally, we swap the Y and Z axis and negate the new Z axis.

3.2 Camera rotations

For each camera i, the rotation matrix R_i is computed with `cv2.Rodrigues(...)`, then, because of the row major memory layout used by GLM, a new GLM matrix is created from R_i^T. Finally,

we negate the Y axis and rotate the camera 90 degrees around the Y axis.

4 Our Choice Tasks

4.1 Automating the detection of the four corner points of the checkerboard

We based our corner detection approach on the assumption that the checkerboard doesn't move for the entire video. We train a KNN subtractor on the background video and perform a foreground extraction from the first video frame of the checkerboard video. We convert the picture to grayscale and apply `cv2.equalizeHist(...)` to emphasize the black squares. Then, we binarize and compute the convex hull of all the black pixels contained in the foreground polygon. In this way, we are able to compute the inverse binarized mask of the extracted foreground and isolate the squares mask by selecting only the pixels contained in the convex-hull. Finally, we approximate the squares polygon shape with `cv2.approxPolyDP(...)` and return the list of 4 corner points sorted clock-wise. The user is anyway asked to check for the quality of the approximations and move the corners by hand in case the approximation quality is not considered sufficiently good.

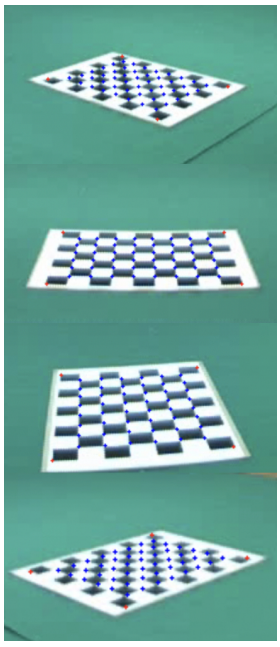

Figure 2: Estimated corner points for each checkerboard view

4.2 Optimizing the construction of the voxel model

We based our optimizations on one main assumption: the horseman doesn't move much across adjacent frames in the video. So based on that, at the k-th $= 0$ frame we compute and store the visibility table V^t, the set of voxels to be drawn V^d and the current foreground masks F_k in the global state. Then at the k-th > 0 frame, we perform a XOR between F_k, F_{k-1} to detect which frames have changed X_k. Next, we perform a AND between X_k, F_k to find all the pixels which are now black and then perform a AND between $X_k, \neg F_k$ to find all the pixels which are now white. We then update V^t by exploiting numpy vectorized operations and check only for the voxels with at least one new visibility value set to `True`. Finally, we update V^d by subtracting with an efficient set operation all the voxels to be removed and by adding all the voxels to be drawn.

4.3 Speeding-up the creation of the look-up table

As pointed out before, the main bottlenecks for the look-up table computation are the limited memory availability of a system and the `cv2.projectPoints(...)` function. We created a script which improves the performances of the look-up table computation by exploiting multiprocessing and volume segmentation. We solve the memory limitations by splitting the input volume into chunks with fixed size. Each chunk is translated and scaled by the proper parameters and saved in a shared memory region. Next, we run one process per camera and compute the back-projection for each chunk in parallel. By exploiting the shared memory region we can access the same data in different processes without having to copy it. Once the computation is complete, the look-up table is saved in a binary file.

4.4 Implementing the surface mesh using the Marching Cubes algorithm

We use the function provided by ScImage `measure.marching_cubes(...)` which computes a 3D surface mesh for a given 3D voxel volume. The volumetric array passed to the function is a binary array containing 1 only for position containing a voxel in the model, 0 otherwise. Finally, we construct a `Poly3DCollection` object to plot the resulting mesh surface using `matplotlib`.

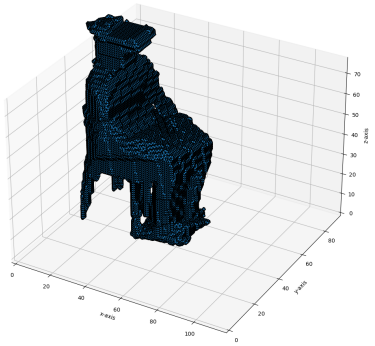

Figure 3: 3D voxel mesh surface obtained with the Marching Cubes algorithm

5 Video

A video showing the final result of our 3D Voxel-based reconstruction can be found on Youtube.