3DCV Hw3

R11944014 戴靖婷

tags: `3DCV` `Python` `Report`

How to execute your codes, including the package used and the environment.

- Package and environment:
  - python: 3.9.13
  - open3d: 0.15.1
  - numpy: 1.23.3
  - opencv-ptyhon: 4.5.1
  - os, argparse, glob, multiprocessing
- Code execution:
  - Camera calibration:
    - `python camera_calibration.py [path/to/calibration/video] --output [path/to/output/npy_file]`
  - Visual odometry:
    - `python vo.py [path/to/frame/directory] --camera_parameters [path/to/camera_parameters.npy]`

Briefly explain your method in each step

Camera calibration

- I made use of the sample code, *camera_calibration.py*.
  - Execute the code.
  - Press *space* to add the picture at that moment for calibration. If adding at least 4 pictures, one can press *q* for early stop.
  - It ouputs the camera intrinsic matrix and distortion coefficients to *camera_calibration.npy*.

- The following is the upload result of camera_calibration.npy.

```
save images: 1
save images: 2
save images: 3
save images: 4
save images: 5
save images: 6
save images: 7
save images: 8
save images: 9
save images: 10
=> start corner detection and calibration
=> corners found in 10 images
=> Overall RMS re-projection error: 0.23172875496800185
Camera Intrinsic
[[512.46650699   0.         312.11733564]
 [  0.         513.73217895 185.2362081 ]
 [  0.           0.           1.        ]]
Distortion Coefficients
[[ 1.06077554e-01 -6.61732724e-01 -1.84496126e-03 -1.90539681e-04
   1.06386895e+00]]
```

Visual Odometry

- This part is executed in *vo.py* *(http://vo.py)*.
- Feature Matching
  - I used ORB as the feature extractor and computed Hamming distance for binary feature matching.

```python
        # Initiate ORB detector
        self.orb = cv.ORB_create()
        # Create BFMatcher object
        self.bf = cv.BFMatcher(cv.NORM_HAMMING, crossCheck=True)
```

- Pose from Epipolar Geometry (pseudo codes and comments)
  - First of all, I constructed a class, *Frame*, to store all the information each frame needs.

```python
class Frame:
    def __init__(self, R, t, scale, keypoints, descriptors) -> None:
        """
            Parameters:
                R: rotation matrix in WCS
                t: translation matrix in WCS
                scale: scale compared to the first two frames
                kp: keypoints of this frame
                des: descriptors of this frame
        """

        self.R = R
        self.t = t
        self.scale = scale
        self.kp = keypoints
        self.des = descriptors
```

  - Another class, *SimpleVO*, is for the complete visual odometry.
  - Before process all frames, I preprocessed the first and the second frames first. It is because the extrinsic parameters and the scale between them are the basis of the

other frames.

```python
    # Preprcess the first two frames as the basis
    def preprocess(self):
        # Initiate two relative frames and rotation / translation matrices in WCS
        self.pre_frame: Frame = None
        self.cur_frame: Frame = None
        self.R_WCS = None
        self.t_wcs = None

        # Initiate ORB detector
        self.orb = cv.ORB_create()
        # Create BFMatcher object
        self.bf = cv.BFMatcher(cv.NORM_HAMMING, crossCheck=True)

        # Process the first and the second frames
        img_0 = cv.imread(self.frame_paths[0])
        kp_0, des_0 = self.orb.detectAndCompute(img_0, None)
        self.pre_frame = Frame(np.eye(3, dtype=np.float64), np.zeros((3, 1), dtype=np.float64), 1, kp_0, des_0)

        img_1 = cv.imread(self.frame_paths[1])
        kp_1, des_1 = self.orb.detectAndCompute(img_1, None)
        R, t = self.get_rel_pose(kp_0, des_0, kp_1, des_1)
        t = -t  # for the correct direction
        self.cur_frame = Frame(R, t, 1, kp_1, des_1)

        self.R_WCS, self.t_WCS = R, t
```

- Then, process all the other frames one by one.
  - Detect and compute the features by ORB.

```python
    # Calculate the pose of all the frames
    def process_frames(self, queue):

        # Preprocess the first two frames
        self.preprocess()

        # Process the other following frames
        for frame_path in self.frame_paths[2:]:
            post_img = cv.imread(frame_path)
            post_kp, post_des = self.orb.detectAndCompute(post_img, None)

            rel_R, rel_t = self.get_rel_pose(self.cur_frame.kp, self.cur_frame.des, post_kp, post_des)
            rel_t = -rel_t
            post_frame = Frame(rel_R, rel_t, 1, post_kp, post_des)

            scale = self.get_rel_scale(self.pre_frame, self.cur_frame, post_frame)
            if scale > 2.5:
                scale = 2.5
            post_frame.scale = scale

            # Rotation and translation in the world cooridinate system
            self.R_WCS = rel_R @ self.R_WCS
            self.t_WCS = self.t_WCS + scale * self.R_WCS @ rel_t

            queue.put((self.R_WCS, self.t_WCS))

            # Update new frame
            self.pre_frame = self.cur_frame
            self.cur_frame = post_frame

            # show image
            img = cv.drawKeypoints(post_img, post_kp, None, color=(0, 255, 0))
            cv.imshow('frame', img)

            if cv.waitKey(30) == 27:
                break
```

  - Calculate the relative rotation and translation matrices between this frame and the previous one.
    - Match the descriptors of the frame1 and frame2 and undistort the matched points by distortion coefficients.
    - Find Essential matrix and recover the relative rotation and translation matrices from E.

```python
    # Calculate the relative pose between two frames
    def get_rel_pose(self, kp1: np.array, des1: np.array, kp2: np.array, des2: np.array):
        """
            Parameters:
                kp1, des1: the keypoints and descriptors of the formal frame
                kp2, des2: the keypoints and descriptors of the latter frame

            Return:
                rel_R: relative rotation matrix between the two frames
                rel_t: relative translation matrix between the two frames
        """

        # Match descriptors
        matches = self.bf.match(des1, des2)

        pnts1 = np.array([kp1[m.queryIdx].pt for m in matches])
        pnts2 = np.array([kp2[m.trainIdx].pt for m in matches])

        # Undistort points for Essential matrix calculation
        pnts1 = cv.undistortPoints(pnts1, self.K, self.dist, None, self.K)
        pnts2 = cv.undistortPoints(pnts2, self.K, self.dist, None, self.K)

        # Find Essential matrix and recover rotation and translation matrices from E
        E, _ = cv.findEssentialMat(pnts1, pnts2, self.K)
        _, rel_R, rel_t, _ = cv.recoverPose(E, pnts1, pnts2, self.K)

        return rel_R, rel_t
```

- Compute the relative scale between the three frames. I set a threshold (2.5) to deal with the impossible scale which is extremely large.
  - Match the three frames by using the intersection of the matched indices of cur_frame. And then find all the corresponding points of the matched indices.
  - Then, find the projection matrices of the three frames for triangulation.
  - Iterate the procedure of calculating scales for many times and choose the median of the scales as the output in order to get a stable result without being affected by the outliers.
  - In the loop, first randomly choose two indices to get two points for calculating the distance.
  - Reconstruct these points in 3D space by *cv2.triangulatePoints*.
  - Then calculate the distances between each two reconstructed points and compute the ratio of two distances as the scale in this round.

```python
    # Calculate the relative scale between three frames
    def get_rel_scale(self, pre_frame: Frame, cur_frame: Frame, post_frame: Frame):
        """
            Parameters:
                pre_frame: the previous frame
                cur_frame: the current frame
                post_frame: the next frame

            Return:
                scale: the median of the relative scales
        """

        # Match between the three frames by the current frame
        matches_1, matches_2= self.bf.match(cur_frame.des, pre_frame.des), self.bf.match(cur_frame.des, post_frame.des)
        queryIdx_1, queryIdx_2 = [m.queryIdx for m in matches_1], [m.queryIdx for m in matches_2]

        # Find the matched index of the current frame
        cur_matchIdx, pre_ind, post_ind = np.intersect1d(queryIdx_1, queryIdx_2, return_indices=True)

        # Correspond the index to the matched points of all three frames
        cur_pnts = np.array([cur_frame.kp[idx].pt for idx in cur_matchIdx])
        pre_pnts = np.array([pre_frame.kp[matches_1[pre_i].trainIdx].pt for pre_i in pre_ind])
        post_pnts = np.array([post_frame.kp[matches_2[post_i].trainIdx].pt for post_i in post_ind])

        # Triangulation
        # Projection matrices: cur_proj_mat and post_proj_mat are the relative projection matrices of pre_proj_mat
        pre_proj_mat = self.K @ np.hstack((np.eye(3), np.zeros((3, 1))))
        cur_proj_mat = self.K @ np.hstack((cur_frame.R, cur_frame.t))

        rel_post_R = post_frame.R @ cur_frame.R
        rel_post_t = cur_frame.t + cur_frame.R @ post_frame.t
        post_proj_mat = self.K @ np.hstack((rel_post_R, rel_post_t))

        # Iterate many times in order to get stable result
        num_pts = len(cur_pnts)
        scales = []
        for _ in range(num_pts // 2):

            # Randomly choose two indices for two points
            rand_idxs = np.random.choice(num_pts, size=2, replace=False)
            rand_pre_pnts = pre_pnts[rand_idxs].T
            rand_cur_pnts = cur_pnts[rand_idxs].T
            rand_post_pnts = post_pnts[rand_idxs].T

            # Reconstructed points in 3D homogeneous coordinates
            homo_pnts_1 = cv.triangulatePoints(pre_proj_mat, cur_proj_mat, rand_pre_pnts, rand_cur_pnts)
            homo_pnts_2 = cv.triangulatePoints(cur_proj_mat, post_proj_mat, rand_cur_pnts, rand_post_pnts)

            homo_pnts_1 /= homo_pnts_1[-1, :]
            homo_pnts_2 /= homo_pnts_2[-1, :]

            homo_pnts_1, homo_pnts_2 = homo_pnts_1.T, homo_pnts_2.T

            # Distances between each two reconstructed points
            dist1, dist2 = np.linalg.norm(homo_pnts_1[0] - homo_pnts_1[1]), np.linalg.norm(homo_pnts_2[0] - homo_pnts_2[1])

            # Calculate the scale, the ratio of two distances
            scale = dist2 / dist1
            scales.append(scale)

        return np.median(scales)
```

- Update the accumulated rotation and translation matrices in the world coordinate system. Then, put them into the queue for multiprocess calculation and plot the trajectory.
- After all, update the frames and display another window for real video with feature extraction circles.

- Last, in the run function, it plots the keypoints in a window while plotting the 3D point set in another window.

```python
    def run(self):
        vis = o3d.visualization.Visualizer()
        vis.create_window()

        queue = mp.Queue()
        p = mp.Process(target=self.process_frames, args=(queue, ))
        p.start()

        keep_running = True
        while keep_running:
            try:
                R, t = queue.get(block=False)
                if R is not None:
                    # insert new camera pose
                    vc = vis.get_view_control()
                    line_set = self.get_lineset(R, t)
                    vis.add_geometry(line_set)
            except:
                pass

            keep_running = keep_running and vis.poll_events()
        vis.destroy_window()
        p.join()
```

- For plotting the camera pose and trajectory, I set the apex and the four corners of the image plane first and constructed the line set by these points.

```
    # Get the line set of each camera pose for open3d plotting
    def get_lineset(self, R: np.array, t: np.array):
        """
            Parameters:
                R, t: rotation and translation matrices
            Return:
                line_set: line sets for plotting
        """

        # Point set: apex and four corners of the image plane
        pnts = np.array([[0, 0, 0], [1, 1, 3], [1, -1, 3], [-1, 1, 3], [-1, -1, 3]])
        pnts = R @ pnts.T + t
        pnts = pnts.T

        # Line set
        lines = [[0, 1], [0, 2], [0, 3], [0, 4], [1, 2], [1, 3], [2, 4], [3, 4]]

        line_set = o3d.geometry.LineSet()
        line_set.points = o3d.utility.Vector3dVector(pnts)
        line_set.lines = o3d.utility.Vector2iVector(lines)

        # blue
        colors = np.tile([0, 0, 1], (8, 1))
        line_set.colors = o3d.utility.Vector3dVector(colors)

        return line_set
```

Results Visualization

- Youtube link:
  - https://youtu.be/9THGw-TgwYQ (https://youtu.be/9THGw-TgwYQ)