Hw3 Visual Odometry

- ***Department:*** Electrical Engineering
- ***Name:*** Ching-Hsiang Wu (武敬祥)
- ***ID:*** R11921080

Content

Visual Odometry

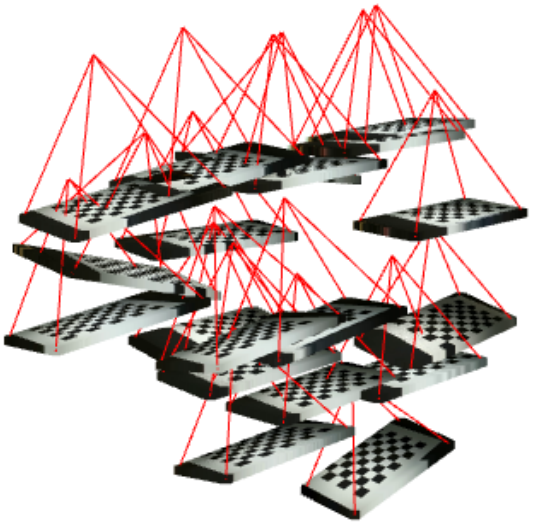
Camera calibration

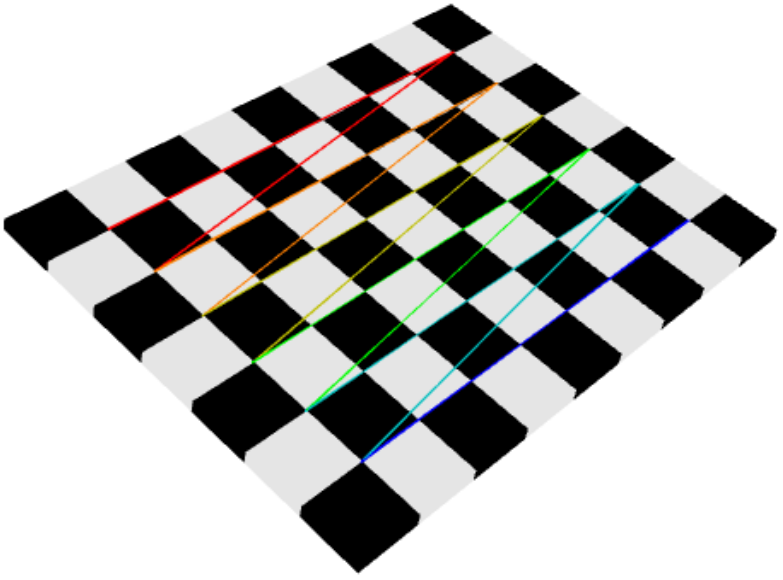
Camera calibration is done by executing the example code `camera_calibration.py` .I select 20 images and the final camera intrinsic parameters and the distortion coefficients are

$$K = \begin{bmatrix} f_x & s & o_x \\ 0 & f_y & o_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 519.78 & 0 & 314 \\ 0 & 519.85 & 177 \\ 0 & 0 & 1 \end{bmatrix}$$

$$D = \begin{bmatrix} k_1 & k_2 & p_1 & p_2 & k_3 \end{bmatrix} = \begin{bmatrix} 0.129 & -0.972 & -0.00314 & -0.000135 & 1.85 \end{bmatrix}$$

the 20 camera poses to the chessboard are shown in the 3D visualization below.





Feature matching

Using the Oriented FAST and Rotated BRIEF (ORB) as a feature extractor, and computing the Hamming distance because ORB provides binary descriptors. Next, sorting all the key point pairs after brutally matching all the features between two images.

Pose from epipolar geometry

Following the visual odometry procedures on page 25 of lecture handout 18. After capturing the new frame img_{k+1} and extracting and matching features between img_{k+1} and img_k, I estimate

the essential matrix $E_{k+1,k}$ by opencv `findEssentialMat` and obtain the $[{}^{k+1}_{k}R \quad {}^{k+1}t_{k_{org}}]$ from the opencv `recoverPose`. Be careful that the rotation and translation we got from `recoverPose` are reversed from the lecture slides, so we convert $[{}^{k+1}_{k}R \quad {}^{k+1}t_{k_{org}}]$ to $[{}^{k}_{k+1}R \quad {}^{k}t_{k+1_{org}}]$ by

$$
\begin{aligned}
{}^{k}_{k+1}R &= ({}^{k+1}_{k}R)^{-1} \\
{}^{k}t_{k+1_{org}} &= -({}^{k+1}_{k}R)^{-1} \cdot {}^{k+1}t_{k_{org}}
\end{aligned}
$$

Noticing that ${}^{k+1}t_{k_{org}}$ is a unit length vector. Therefore, we recover the proper length of ${}^{k}t_{k+1_{org}}$ by the following equation.

$$
\begin{aligned}
\|{}^{k}t_{(k+1)_{org}}\| &= \frac{\|{}^{k}X - {}^{k}X'\|}{\|{}^{k-1}X - {}^{k-1}X'\|} \cdot \|{}^{k-1}t_{k_{org}}\| \\
{}^{k}t^{*}_{(k+1)_{org}} &= {}^{k}t_{(k+1)_{org}} \cdot \|{}^{k}t_{(k+1)_{org}}\|
\end{aligned}
$$

where

${}^{k}t^{*}_{(k+1)_{org}}$: real length of translation.

${}^{k-1}X, {}^{k-1}X'$: camera coordinate of two scene triangulated points in frame $\{k-1\}$.

${}^{k}X, {}^{k}X'$: camera coordinate of **the same** two scene triangulated points in frame $\{k\}$.

How to select the two scene points is an “ART”. In brief, I select the two key points from the frame $\{k-1\}$, and if the same two key points appear in the image of frame $\{k\}$, I use the triangulated points of the key points(also obtained from the `recoverPose`) to calculate the scale of translation. However, the scale sometimes explodes and sometimes shrinks, I draw some reasons why this fails and provide some solutions to fix it.

Reasons to fail:

1. Key points are not precise.
2. Selected key points are from the moving object.
3. Selected key points are too close.

Solutions:

1. Setting the upper bound and lower bound of the translation scale
2. Averaging each translation scale

The detailed skills to conquer the bad key point selections are shown in the following pseudo-code. All in all, I think it is more like a “cheating” and trivial process.

- **Visual odometry** algorithm

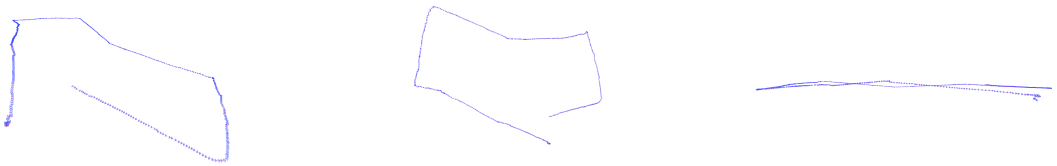
```
def process_frames():
  1. read img_k
  2. obtain keypoint_k, descriptors_k of img_k by orb
  3. initialize
     t_last_norm = 1
```

```
    last_good_matches = []
    last_mask = []
for _ in img_list:
  4. read img_k1
  5. obtain keypoint_k1, descriptors_k1 of img_k1 by orb
  6. find good_matches by brutal force matching of descriptors_k and descriptors_k1
  7. sorting the good_matches in order of hamming distance
  8. obtain the points_k and points_k1 from good_matches
  9. E, mask_e <- findEssentiaMat
  10. r,t,mask_r,triangulatedPts <- recoverPose
  11. mask = mask_e and mask_r
  12. inliers_points_pair_id <- mask, last_mask, good_matches, last_good_matches
  13. triangulatedPts, last_triangulatedPts <- inliers_points_pair_id
  14. translation_scale<- calculatedAvgRealScale(triangulatedPts,last_triangulatedPts,t_last_norm)
  15. real_t = t*translation_scale
  16. reassign
      keypoint_k = keypoint_k1
      descriptors_k = descriptors_k1
      t_last_norm = norm(real_t)
      last_good_matches = good_matches
      last_triPoints = triPoints
      last_mask = mask
```

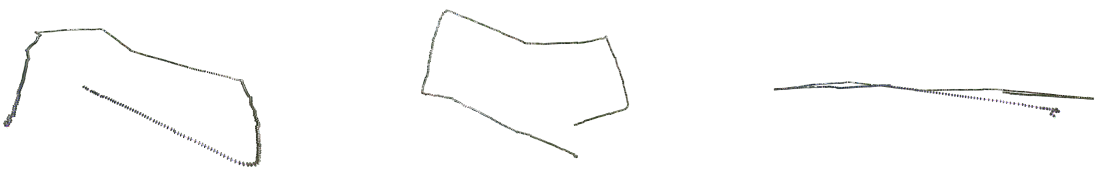
Visualization results

- The image after executing the ORB. The color means the hamming distance of each key point. If the color more red, the hamming distance is smaller, and if the color more blue the hamming distance is larger.

- The trajectory of the camera



- The trajectory of the camera with the image



Package Introduction

None

How to Execute?

- **Environment**
  - `Python == 3.8.12`
  - `Opencv == 4.5.1`
  - `Numpy == 1.24.3`
  - `Open3d == 0.11.2`
- **Camera calibration**

  Run the following command on the terminal: `python camera_calibration.py calib_video.avi`
- **Visual odometry**

  Run the following command on the terminal: `python vo.py`

Youtube Link

3DCV__Homework3

This is a demonstration of how to execute the code of 3DCV homework3. After executing the vo.py, two windows are showing up. One is for live processing of feature extraction and the other is for 3D visualization of visual odometry.

https://www.youtube.com/watch?v=X04MG9JxGBs