Final Report

Hua Liu

1. Introduction

There is an increasing need for geometric 3D models in the movie industry, the games industry, mapping (Street View) and others. Generating these models from a sequence of images is much cheaper than previous techniques (e.g. 3D scanners).

Develop platform:

1) C++ with opencv library
2) Matlab for showing 3D points cloud result

2. Method

2.1 Feature Matching

There are two ways of feature matching:

- Method one:

3) In all images, detect keypoints and compute descriptors using SURF
4) Matched point pairs between the two conjunctive images
5) Choose “good matches”: Calculate max and min distances between keypoints, and label matches with distance less than 5 times the minimum distance as "good matches"
6) Tracks features across all images and generise “pairwiseMatches”(a vector of “good matches”)

In opencv, step2)3)4) can be done with one class *FeaturesMatcher*:

- Method two:

1) In all images, detect keypoints using SURF
2) Convert keypoints to Point2f format and track them using OpticalFlow.

In opencv, step2) can be done with *calcOpticalFlowPyrLK(),goodFeaturesToTrack()*

2.2 Estimation of Camera Pose

1) Find fundamental matrix F between each image pair using RANSAC. The corresponding command in opencv library is findFundamentalMat().
2) Refine intrinsic matrix using class BundleAdjusterBase: BundleAdjusterBase minimizes the reprojection errors by optimizing the position of the 3D points and the camera parameters, using features across all images and corresponding pairwiseMatches.
3) Find essential matrix E using $E = K_r^{-T} F K_l^{-1}$, where K is the optimized intrinsic matrix.
4) Assume camera matrix P1 is at origin - no rotation or translation. And Decompose E using SVD to get the second camera matrix P2 . However, by decomposing E, you can only get the direction of the translation. So there are generally 4 possible solutions(HZ 9.19). of which we select the one that results in reconstructed 3D points in front of both cameras.

In opencv, Step 6) can be done by *cv::decomposeEssentialMat(E, R1, R2, t);*and Generally 4 possible seconde camera poses exists P2:[R1,t], [R1,−t], [R2,t], [R2,−t]. Then get the final one with positive depth after triangulation.

Alternatively, it can be done by *recoverPose(E, points[0], points[1], R, t, focalLength, principalPoint);*This command decomposes E and get the final R,t using Cheirality check, which means that the triangulated 3D points should have positive depth.

I chose the second one.

2.3 Triangulation between img[i] and img[i+1]

1) Restore each image from lens distortion using the distortion coefficients obtained from camera calibration.
2) For each pair of images, compute the 3D locations of keypoints in one of them using P1, P2 calculated from section 2.2.

In opencv, step1) can be done with *cv::undistortPoints(points[i], points[i], cameraMatrix, distortion);* Step2) can be done with *cv::triangulatePoints(P1, P2, points[i], points[i+1], points4D);*

2.4 (Sparse)3D Reconstruction from multiple views

1) construct extrinsic matrix for all cameras in reference to the first one using:

   1t3=1t2+2R1*2t3=1t2+inv(1R2)*2t3=1t2+1R2.transpose*2t3

   And 1R3=1R2*2R3
2) Project each 3D world points to the camera coordinate of the first camera, using the extrinsic matrix in step1)
3) Take the average location of projected 3D points as final result
4) This will generate a metric reconstruction up to a scale factor.

In opencv, One way to handle this is to use *solvePnP*.

2.5 (Dense)3D Reconstruction

Substitute keypoints with all points.

2.6 Augment Reality

1) find chess board corners in image
2) generate corresponding 3D corners with z=0
3) Estimate camera extrinsic matrix from 3D-2D points correspondences
4) generate 3D with same X,Y in step2) but Z=3(for example)
5) project 3D points generated in step4) using estimated camera extrinsic matrix in step3)

In opencv, step3) can be done with:

cv::solvePnP(obj, corners, cameraMatrix, distortion, rvec, tvec);

And step5) can be done with:

cv::projectPoints(obj2, rvec, tvec, cameraMatrix, distortion, corners2);

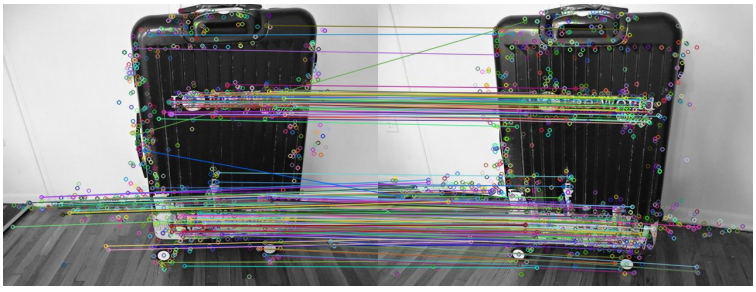
2.7 Other tips on refining results

1) use app to fix focal length
2) Resize images to make computation time reasonable

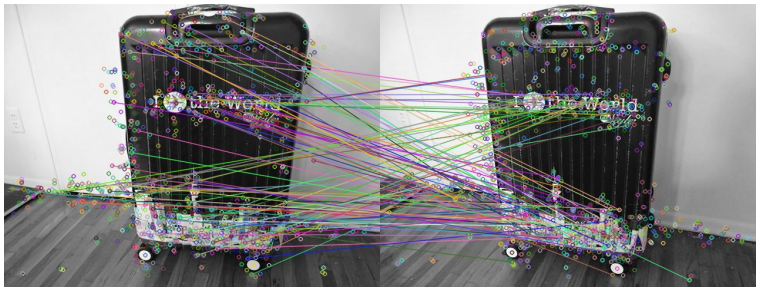
3. Result

3.1 Feature Matching tracking through all images

Using method one from 2.1 to track same features throw 3 images:



Img1. feature matching between img[0] and img[1]



Img2. feature matching between img[1]and img[2]

It is clear Img1 is a successful match. While Img2 failed. So this approach is not suitable.

Using method two from 2.1 to track same features throw 7 images:

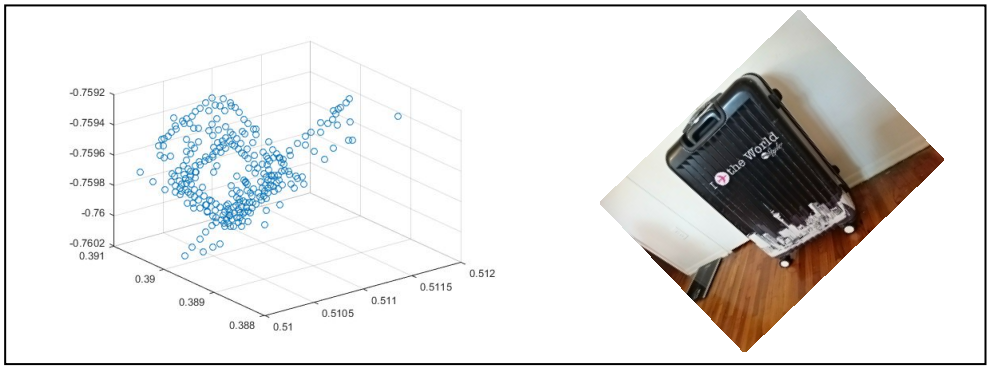
Img3. Optical flow of keypoints throw out 7 images

Img3 is a successful match. And after test, in the whole process tracked key points are all 307. This feature is very convenient for the later triangulation. And there is no need to manually manipulate matches to get matched point sets.

However, since in this way we can only get the location of points. So there is no way to construct class *Feature* and *MatchInfo*. Therefor, we can not use class *BundleAdjustmentBase* which takes class *Feature* and *MatchInfo* as Initial parameters.

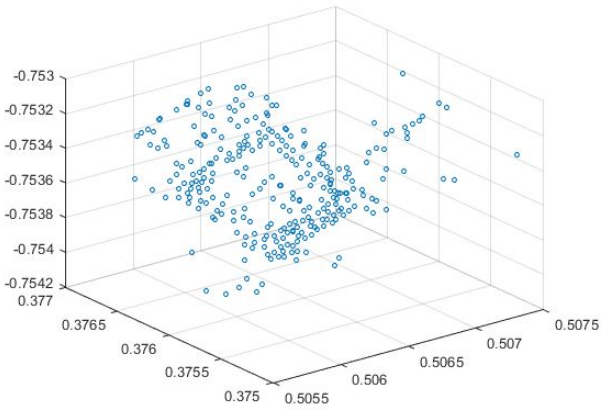
3.2 Triangulation between one image pair

Using Optical Flow to reconstruct 3D points from two images



Img4. reconstructed 3D points and one of the original image

3.3 3D Reconstruction (Sparse) with Estimation of Camera Pose

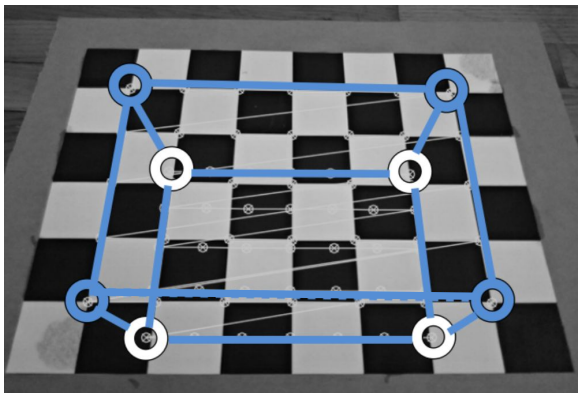


Img5. reconstructed 3D points

```
t: !!opencv-matrix
  rows: 3
  cols: 1
  dt: d
  data: [ 6.79537570634600 74e-001, 1.18093546536876 00e+000,
    1.85678881535986 71e+000 ]
RFst: !!opencv-matrix
  rows: 3
  cols: 3
  dt: d
  data: [ -2.49453165937173 93e-001, -6.25899063815959 72e-001,
    7.38934015943389 59e-001, -6.42250927693004 85e-001,
    6.78022273305624 48e-001, 3.57490619148178 37e-001,
    -7.24766765160624 33e-001, -3.85403890504205 70e-001,
    -5.71119057029991 33e-001 ]
```

Img6. Camera pose

3.4 Augment Reality



Img7. Camera pose: blue circles are on the board. White circles are inside of the board.

4. Conclusion

Multi View Stereo algorithms provide viable methods for building 3D models of objects, buildings and scenes. Using OpticalFlow requires small motion between frames. More work is required to determine how to reconstruct 3D models with fewer images.