Comp Photography
Final Project

Xing Gao

Summer 2016

xinggao@gatech.edu

Stereo Imaging

Compute the 3D location with points correspondence from stereo image pair

The Goal of Your Project

The Goal is to experience the pipeline of stereo imaging from taking images to creating disparity map.

The project is motivated by the stereo lecture and Szeliski’ s book

Scope Changes

Did you run into issues that required you to change project scope from your proposal?

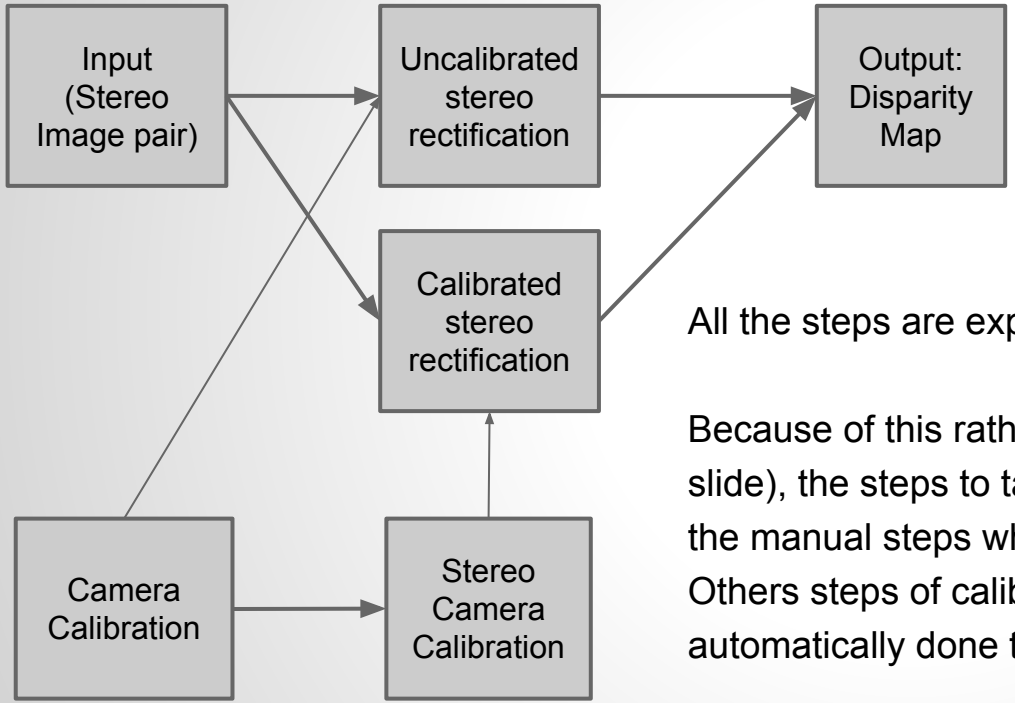
- My original proposal was to measure distance based on stereo camera by following "http://dsc.ijs.si/files/papers/S101%20Mrovlje.pdf".

  After more research done, I still like the idea of stereo camera distance measure, but by following Szeliski's book, it has more photography computation involved. It contains more computational technique, like Epipolar geometry, correspondence, and disparity.

Input

Output

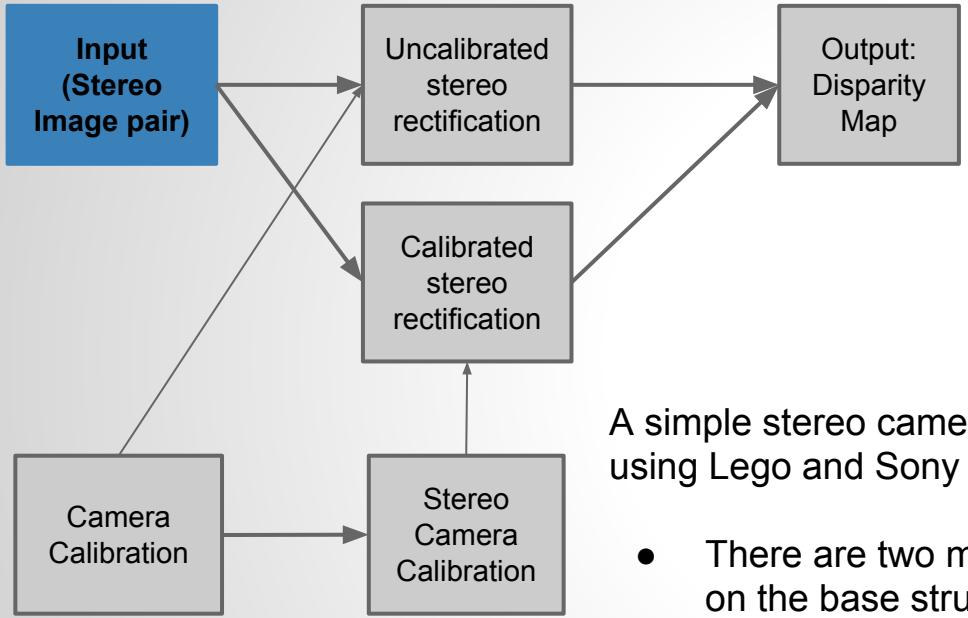
Your Pipeline

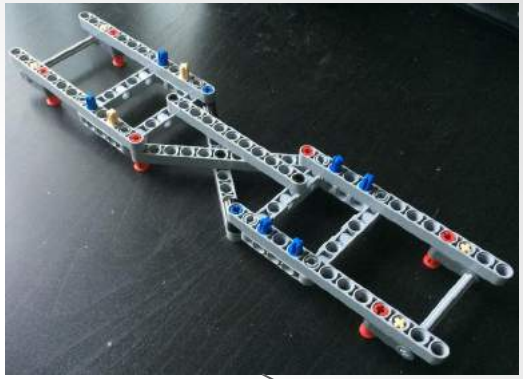


All the steps are explained in detail in following slides

Because of this rather primitive setup of stereo camera (see in next slide), the steps to take stereo images, and images for calibration are the manual steps which require careful planning.
Others steps of calibration, rectification and computing disparity are automatically done through photography computation.

Your Pipeline

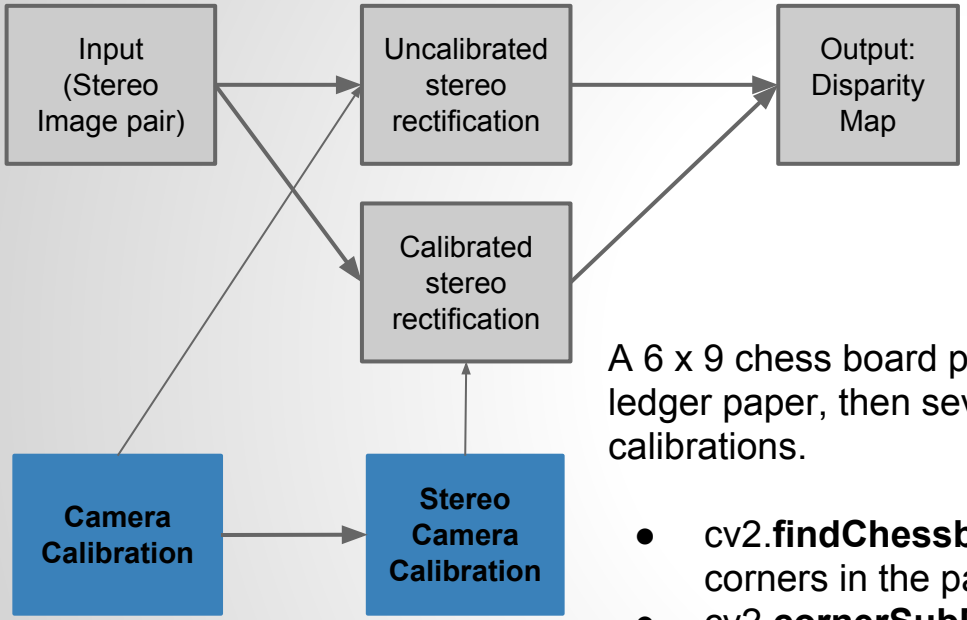




A simple stereo camera has been made using Lego and Sony NEX-5r camera

- There are two mounting locations on the base structure
- The camera is attached to the lego rigidly through the tripod mount
- Then camera can be mounted on the two locations with easy switching

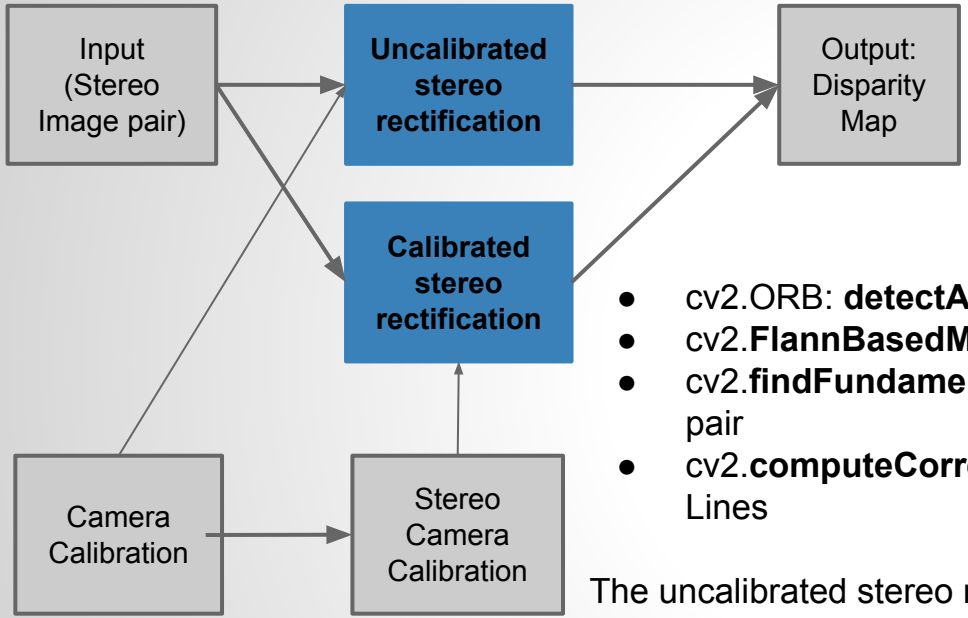
Your Pipeline



A 6 x 9 chess board pattern has been printed on a ledger paper, then several images are taken for the calibrations.

- cv2.**findChessboardCorners**() to find the corners in the pattern
- cv2.**cornerSubPix**() to refine the corner locations
- cv2.**calibrateCamera**() to calculate the intrinsic parameters of the camera
- cv2.**stereoCalibrate**() to estimates transformation between two cameras

Your Pipeline



- cv2.ORB: **detectAndCompute**() to find all the features
- cv2.**FlannBasedMatcher**() to match the features
- cv2.**findFundamentalMat**() to calculate the fundamental matrix from image pair
- cv2.**computeCorrespondEpilines**() to calculated the corresponding Epipolar Lines

The uncalibrated stereo rectification from Hartley's algorithm:

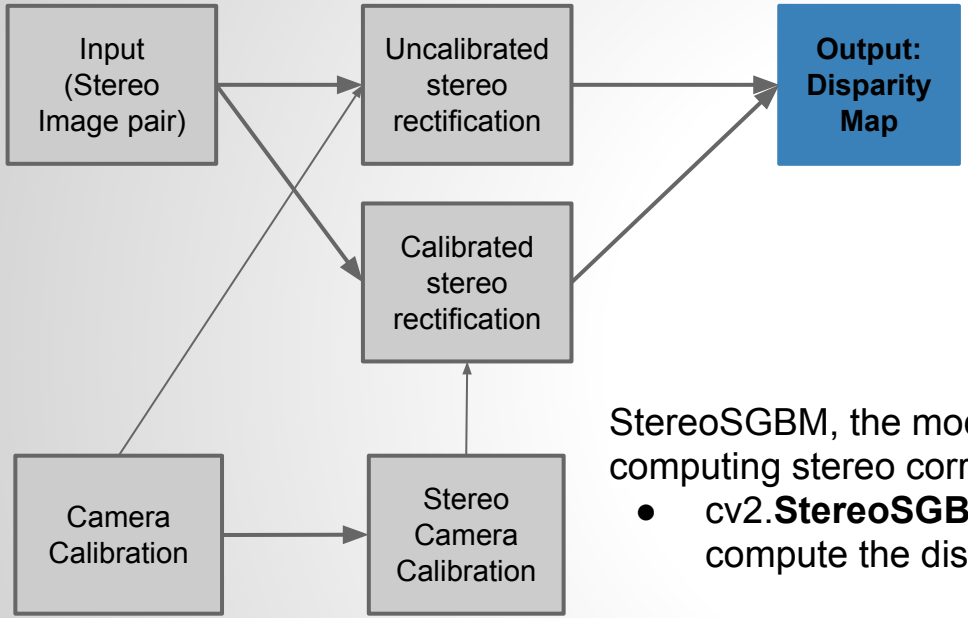
- cv2.**stereoRectifyUncalibrated**() to Computes rectification transform

The calibrated stereo rectification from Bouguet's algorithm

- cv2.**stereoRectify**() Computes rectification transforms

cv2.**initUndistortRectifyMap**() and cv2.**remap**() to undistort and rectify the images

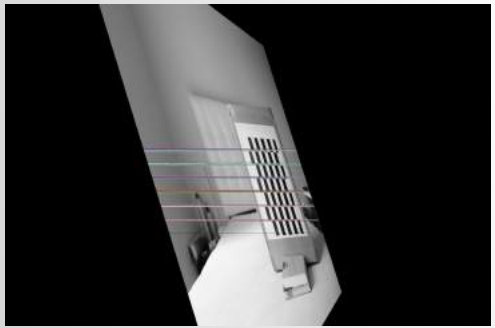
Your Pipeline

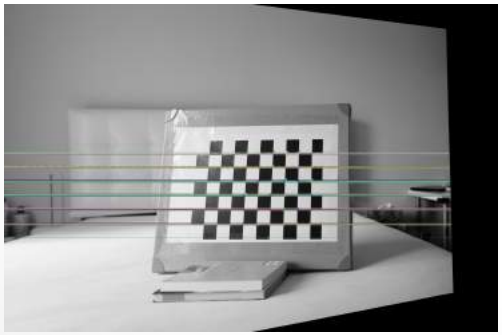


StereoSGBM, the modified H. Hirschmuller algorithm, to computing stereo correspondence

- cv2.**StereoSGBM_create**() and **compute**() to compute the disparity map

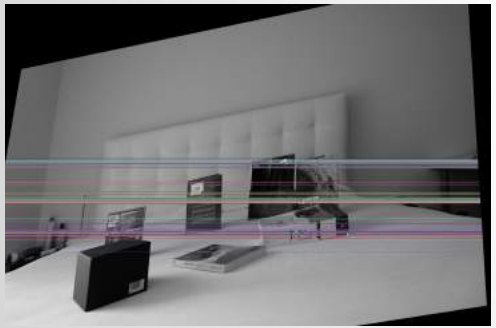
Demonstration 1: Show complete Input/Output results

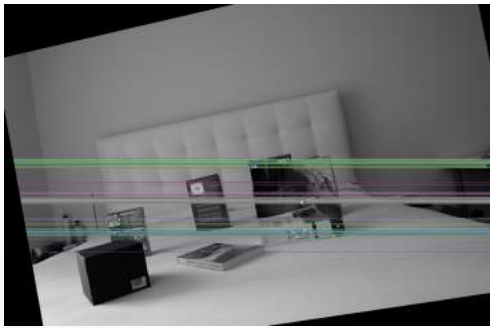




Disparity map using images from calibrated stereo rectification is shown above
Images from uncalibrated stereo rectification has Epipolar Lines aligned on the same level, but images are too distorted

Demonstration 2: Show complete Input/Output results





Disparity map using images from calibrated stereo rectification is shown above
Images from uncalibrated stereo rectification has Epipolar Lines aligned on the same level, but images are too distorted

Demonstration 3: Show complete Input/Output results

Disparity map using images from calibrated stereo rectification is shown above
Images from uncalibrated stereo rectification has Epipolar Lines aligned on the same level, but images are too distorted

Computation: Project Development

The project is mainly developed by following the two books below

- Computer Vision: Algorithms and Applications by Richard Szeliski
- Learning OpenCV by Gary Bradski and Adrian Kaehler

There are some of pitfalls I experienced in the project, with more details in the following slides.

Computation: Project Development

- Stereo Camera setup

I started with taking two images casually by hands. As expected, the pair images are too different in view angle for further computation. The lego structure improves the stability of the image pair.

This setup with lego structures is still very error-prone, since the single camera has to move between two locations to take the image pairs. Best effort has been given to keep the base structure static, fix the two locations, and minimize the difference of camera mountings.

- Calibration error

Because the stereo camera calibration depends on the fixed transformation of the two cameras. A tiny change in the mounting makes the stereo calibration invalid. As a result, after moving the stereo camera from one scene to another. Another stereo camera calibration was done.

Computation: Project Development

- Choice of Input

Due to this simple stereo setup, the object in great distance cannot be processed in this pipeline, because errors are magnified for them. The input has been chosen that most of objects are within 5 meters.

- Uncalibrated and calibrated stereo rectification

Hartley's algorithm (cv2.stereoRectifyUncalibrated()) seems result in a lot of shearing distortion in the rectified images, which makes almost impossible to calculated the disparity map. The multi-stage stereo rectification algorithm of Loop and Zhang (1999) can be studied in the future to reduce the distortion introduced by the projective component.

The calibrated stereo rectification generates better result, which aligns the points very well, for the disparity map computation.

Computation: Project Development

- Disparity map computation

Two method are tested:

StereoBM using the block matching algorithm, shown on left.

StereoSGBM using the semi-global block matching algorithm, shown on right.

They both need parameters to be properly tuned for the image. StereoSGBM seems show more details, although it requires more parameters tuning, since there are more of them.

Computation: Code Explanation

- **Calibration**.py to calibrate camera

findChessboardCorners finds the approximate locations of chessboard corners, then *cornerSubPix* determines their positions more accurately. With the corners data *calibrateCamera* computes the intrinsic parameters for this planar calibration patterns, including camera matrix, and distortion coefficients

```
for img in images:
    gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
    ret, corners = cv2.findChessboardCorners(gray, (9,6),None)
    if ret == True:
        objpoints.append(objp)
        corners = cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)
        imgpoints.append(corners)

ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray.shape[::-1],None,None)
```

Computation: Code Explanation

- **stereoCalibration**.py to calibrate stereo camera

Similar to Calibration.py in previous slide, *findChessboardCorners* and *cornerSubPix* compute the locations of chessboard corners for both cameras in stereo pair. With the corners data and the intrinsic parameters from *calibrateCamera, stereoCalibrate* computes the transformation between two cameras, including R -rotation matrix, T -translation vector, E -essential matrix, and F -Output fundamental matrix.

retval, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, R, T, E, F = cv2.stereoCalibrate(objpoints, imgpoints0, imgpoints1, M, dis, M, dis, gray.shape[::-1])

Computation: Code Explanation

- **rectify**.py to rectify images

The images are first undistorted using the intrinsic parameters. ORB is used to detect the features, find keypoints and computes their descriptors. Then FlannBasedMatcher is used to match the features with knnMatch. Later the matches that less ambiguous are selected by comparing the distance of first and second best matches.

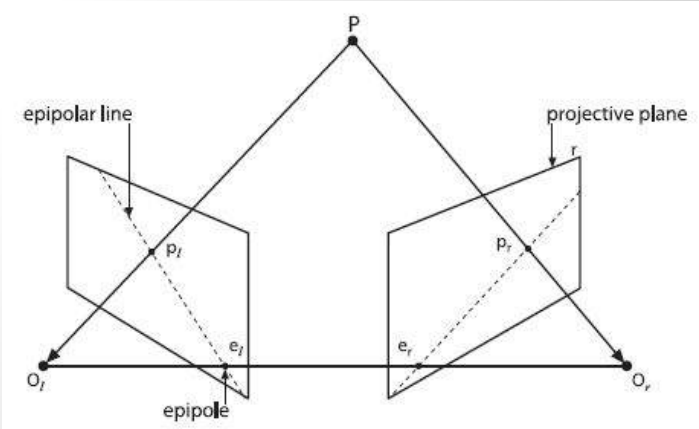
```
img1 = cv2.undistort(img1, M, dis)
img2 = cv2.undistort(img2, M, dis)
…
SIFT = cv2.ORB_create()
kp1, des1 = SIFT.detectAndCompute(img1,None)
kp2, des2 = SIFT.detectAndCompute(img2,None)
...
flann = cv2.FlannBasedMatcher(index_params,search_params)
matches = flann.knnMatch(des1,des2,k=2)
…
for i,(m,n) in enumerate(matches):
    if m.distance < 0.8*n.distance:
        ...
```

Computation: Code Explanation

- **rectify**.py

With the matched and selected keypoints, the fundamental matrix is computed using FM_LMEDS method that can recognize and remove outliers, since there are more points than minimum requirement. Fundamental matrix gives the relationship between a pixel in one image and the corresponding epipolar line in the other image. The points used for the fundamental matrix calculation is used to compute the epipolar lines

```
...
F, mask = cv2.findFundamentalMat(pts1,pts2,cv2.FM_LMEDS)
pts1 = pts1[mask.ravel()==1]
pts2 = pts2[mask.ravel()==1]
lines1 = cv2.computeCorrespondEpilines(pts2.reshape(-1,1,2), 2,F)
...
```



Computation: Code Explanation

- **rectify**.py

Using Hartley's algorithm Uncalibrated stereo rectification and Bouguet's algorithm Calibrated stereo rectification, rectification transformations can be calculated. *stereoRectifyUncalibrated* relies on the epipolar geometry. *stereoRectify* minimize the resulting reprojection distortions while maximizing common viewing area, with the rotation matrix and translation from calibration. *initUndistortRectifyMap* computes the undistortion and rectification transformation map, which is then used by the remap method to rectify the images.

```
...
suc, h1, h2  = cv2.stereoRectifyUncalibrated(pp1, pp2, F, size)
...
MinvH1M = np.dot(np.linalg.inv(M), np.dot(h1, M))
mapx1, mapy1 = cv2.initUndistortRectifyMap(M, dis, MinvH1M, M, size, cv2.CV_32FC1)
...
R1, R2, P1, P2, Q, validPixROI1, validPixROI2 = cv2.stereoRectify(M, dis, M, dis, size, R, T)
mapx1, mapy1 = cv2.initUndistortRectifyMap(M, dis, R1, P1, size, cv2.CV_32FC1)

img1 = cv2.remap(img1, mapx1, mapy1, cv2.INTER_LINEAR)
```

Computation: Code Explanation

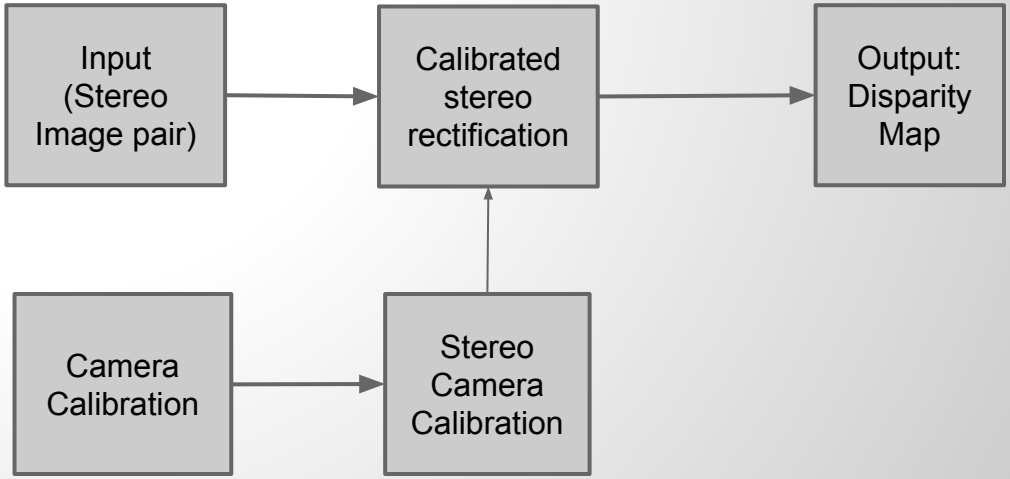
- **disparity**.py

StereoSGBM is used to compute the disparity map of two rectified images. Its parameters need to be tuned for the image for the good result, like the minDisparity and numDisparity should match the minimum and maximum disparity in the images, and blockSize should be balanced between noise and details.

```
window_size = 4
min_disp = 16
num_disp = 64-min_disp
stereo = cv2.StereoSGBM_create(minDisparity = min_disp,
    numDisparities = num_disp,
    blockSize = 3,
    P1 = 8*3*window_size**2,
    P2 = 32*3*window_size**2,
    disp12MaxDiff = 1,
    uniquenessRatio = 25,
    speckleWindowSize = 300,
    speckleRange = 10
)
...
disparity = stereo.compute(img1,img2)
```

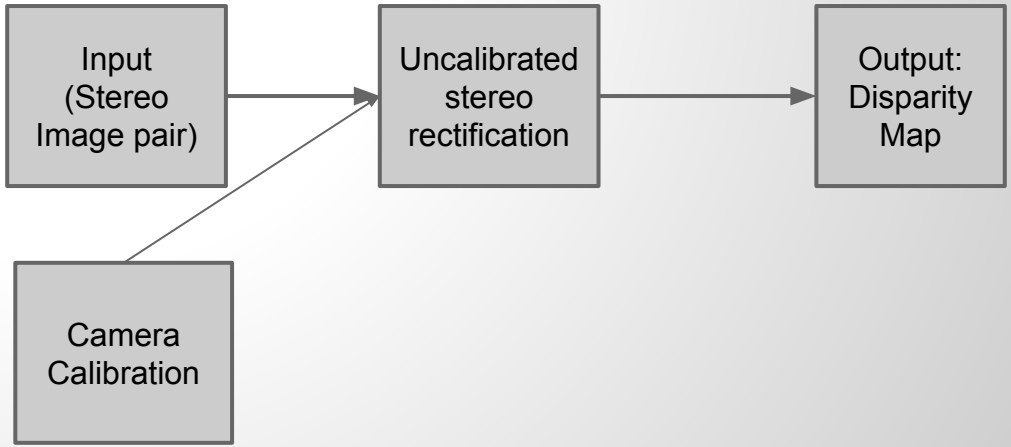
Details: What worked?

- The pipeline to calibrate camera, stereo calibration, calibrated stereo rectification, and compute disparity works well. The disparity map can be produced from multiple images.



Details: What did not work? Why?

- The pipeline to use uncalibrated stereo rectification and compute disparity map didn't work well. The epipolar line can be computed and rectified to be horizontal, but the rectified images has been distorted a lot. It seems there was to much shearing distortion. The reason is not clear to me. With more time, different stereo camera setup can be tested to understand the reason. Also The multi-stage stereo rectification algorithm of Loop and Zhang (1999) seems reduce the shearing distortion, which can be studied as well.



Details: What would you do differently?

- If I have a proper stereo camera, more experiment can be done, especially for the distant objects.
- The multi-stage stereo rectification algorithm of Loop and Zhang (1999) can be studied in the future to reduce the distortion introduced by the projective component.

Resources

- Learning OpenCV by Gary Bradski and Adrian Kaehler
- Computer Vision: Algorithms and Applications by Richard Szeliski
- Computing Rectifying Homographies for Stereo Vision by Charles Loop and Zhengyou Zhang
- http://docs.opencv.org/2.4/modules/imgproc/doc/imgproc.html
- http://docs.opencv.org/3.1.0/da/de9/tutorial_py_epipolar_geometry.html#gsc.tab=0

Your Code

https://github.com/xinggao1/6475final