3D Reconstructions from a Collection of Images

Stefanie Dao*
ctdao@ucsd.edu
University of California, San Diego
San Diego, CA, USA

Abstract

Keywords: computer vision, neural networks, 3D construction, multi view stereo

ACM Reference Format:
Stefanie Dao. 2021. 3D Reconstructions from a Collection of Images. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

One of the most exciting aspect of computer vision is teach machines what human takes for granted: constructing 3D visualization of object from 2D views. In addition, generating these models from a sequence of images is much cheaper than other direst techniques (e.g 3D scanners,...). In this tutorial, I will take a traditional approach to generate sparse 3D construction using Multi View Stereo algorithm and libraries such as Numpy and OpenCV.

2 Dataset and Tools

Our dataset consists of 49 jpg images in colors, capturing a stuff animal placed on a flat surface from different angel. Intrinsic and Extrinsic camera matrices of these images are also given in the form of a Numpy array.

Some of the libraries and tools that we are going to use in this experiment includes:

- Numpy: a library with support for large, multi-dimensional arrays and matrices
- OpenCV (version 4.5.4): a library of programming functions mainly aimed at real-time computer vision
- Pypotree: a package to help visualize 3D constructions
- Open3d: a package that supports outlier removal for further optimization

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

3 Approach

3.1 Identify points of interests

I first load the dataset into Drive and import in to Colab Notebook.

Images are converted from RGB to grayscale. Additionally, I also experiment with using image-enhancement techniques such as CLAHE (Contrast Limited Adaptive Histogram Equalization) and discuss its effect in the later part of the paper.

I will then use **SIFT algorithm** to detect features in each images and compute descriptors for finding 2D corresponding points later.

3.2 Find correspondence points

For a specific pair of images, I will derive correspondences between their keypoints. This can be achieved by using functions a two-sided brute force approach. After creating a $cv.BFMatcher()$ from OpenCV library, I apply $knnMatch$ to get the best k features matches and set parameter $k = 2$ to allow implementing another filter called ratio test, which first suggested by D.Lowe. The basic ideas is to check if the two matches found earlier for each keypoint is substantially different. If their distance are not within a threshold, that keypoint will be eliminated.

3.3 Validate Correspondence

Validating the correspondences we have is necessary since if the dataset we look at gets bigger, the error will accumulate and generate many false correspondences.

Since intrinsic and extrinsic matrices have already been given, I can build the Fundamental matrix for each pair of camera using the following formula:

$$F = K_1^{-T}([t]_\times R)K_2^{-1}$$

where

$$[t]_\times = \begin{bmatrix} 0 & -t_3 & t_2 \\ t_3 & 0 & -t_1 \\ -t_2 & t_1 & 0 \end{bmatrix}$$

and $R = Ext_1^T Ext_2$

(K_1, Ext_1 are intrinsic, extrinsic matrix of camera 1 and K_2, Ext_2 are intrinsics, extrinsic matrix of camera 2)

After obtaining fundamental matrix, I will use it to compute the score between the two matching keypoints in the two images. If the score is within a certain threshold, it will be added to our final list of correspondences.

3.4 Triangulate

Yielding a set of feature matches from the previous step, I proceed to reconstruct 3D points through triangulation.

In the context of two of images with intrinsic and extrinsic matrices given, we will first convert each intrinsic camera to form $K[I\ 0]$ and then take product between the new intrinsic matrices and the corresponding extrinsic matrices to compute camera matrix M for each camera.

Then I pass these camera matrices along with filtered keypoints that we have computed earlier for the two images as parameters for the function $triangulatePoints$. I convert the result back to Euclidean coordinates from Homogeneous coordinates and use pypotree to visualize the 3D reconstruction of the resulting $3xn$ Numpy array.

3.5 Reconstruct with multiple cameras

As every two neighboring images are photoed by neighboring cameras, I will do a sequence of matching-validating-triangulating on every pair of neighboring images (image 0 and 1, 1 and 2, 2 and 3, etc.) and put all of the points I calculated together. Since there are 49 images, which yields 50 neighboring pairs, it is fair to expect getting over 10x reconstructed points than only using two images.

3.6 Optimizing

The resulting 3D construction tends to contain noise and artifacts that one would like to remove. I will use *statistical_outlier_removal* (which removes points that are further away from their neighbors compared to the average for the point cloud) and *radius_outlier_removal* (which removes points that have few neighbors in a given sphere around them) functions from *Open3D* package to remove these outliers.

To implement this, I will convert my Numpy array that stores the 3D points into point cloud, then optimize using these functions and convert back to Numpy array for visualization.

4 Result

While processing images to find corresponding points of interest, I experiment with some techniques to enhance contrast of images and observe if these changes will make a difference in final constructions. In general, each images has around $10,000 - 30,000$ keypoints if SIFT algorithm with default parameters are used.

The threshold choice has a significant impact on the final 3D construction of the investigated object. When finding the correspondence points, I observe that the best result produced when the threshold for D.Lowe test is set to around 0.55. For the validating correspondences step, the threshold score should be relative to the size of the image. In this case, I found it works best around 10-15.

Figure 1. First image after processing without and with CLAHE enhancement

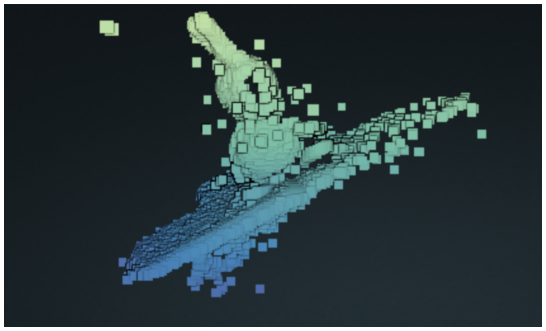

Figure 2. 3D reconstruction with $thresh_{valid} = 5$

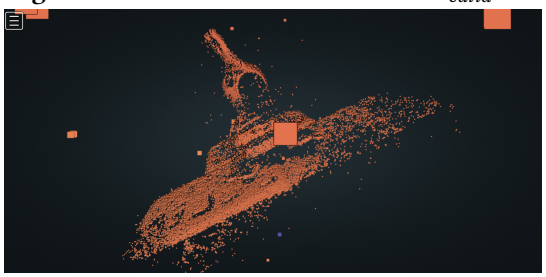

Figure 3. 3D reconstruction with $thresh_{valid} = 14$

For optimizing, I found using both the *statistical_outlier_removal* and *radius_outlier_removal* produces the same result if tuning the right parameters. However, I found using *radius_outlier_removal* with $nb\_points = 5, radius = 6$ gives slightly a better result

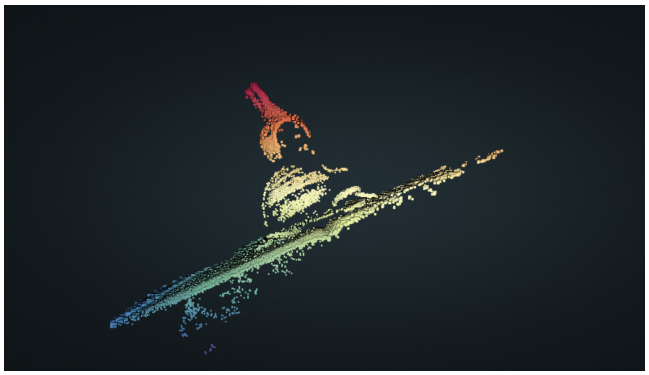

Figure 4. Optimization using $statistical\_outlier\_removal$

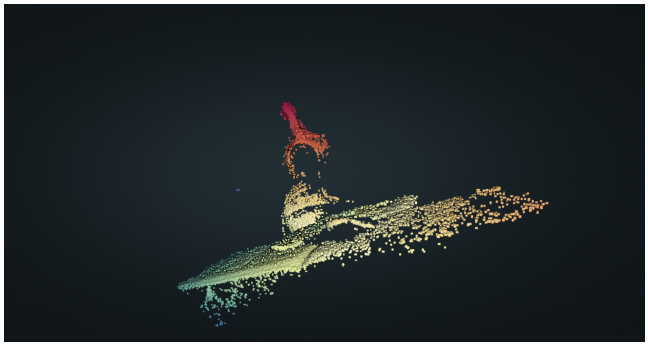

Figure 5. Optimization using *statistical_outlier_removal*

Finally, I compare my best 3D constructing result between normal grayscale images and processed images using CLAHE. With normal grayscale images, the 3D constructions retains better details for visible features (e.g the stripes on the stuff animal). However, when using CLAHE to increase contrast and sharpness of the images before computing, the 3D constructions I get has more points and includes more features on the images (the beak of the stuff animal has more shape while normal grayscale images, this is almost invisible). However, details of other detected features do not retain great details (the stripes disappeared).

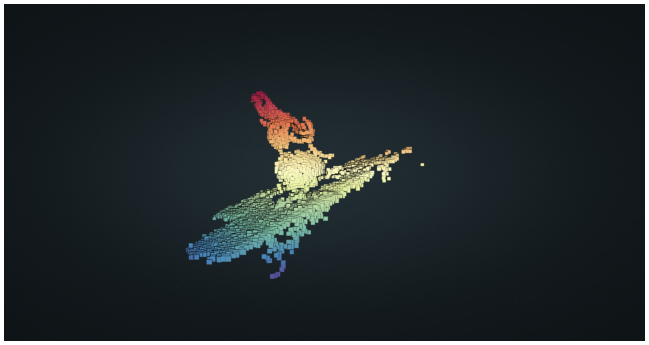

Figure 6. 3D construction for images with CLAHE processing

5 References

[1] Feature Matching https://docs.opencv.org/4.5.4/dc/dc3/tutorial_py_matcher.html

[2] Adaptive Histogram Equalization https://docs.opencv.org/4.x/d5/daf/tutorial_py_histogram_equalization.html

[3] Point Cloud Outlier Removal http://www.open3d.org/docs/latest/tutorial/Advanced/pointcloud_outlier_removal.html