Gabriel Valachi (101068875)
Alexander Cornish (101053176)

COMP4102A - Final Project Report

Abstract

The idea behind this project was to take a video of a 3D object, specifically a toy car, and apply a specific technique to generate a 3D representation of the object from the 2D video frames. The object would be placed on a turntable, and the video would record the object being turned, capturing every angle of the object except from above. We initially wanted to use a method involving corner detection and stereoscopy, and use this information to generate a 3D model, but we ultimately ended up settling for a Structure from Motion (SfM) approach to generate a point cloud accurate to all dimensions except for scale. Unfortunately, we significantly underestimated the complexity of SfM 3D reconstruction using more than two cameras, and the difficulty of finding matching features between two images, and the shapes of the point clouds generated by our project code are highly inaccurate. We also had to rely on more OpenCV functions than we would have liked; we initially wanted to perform the processes behind said functions ourselves, but this focus was abandoned in an attempt to make our existing pipeline work. We therefore consider this project a failure. Although this project was unsuccessful, we will still discuss the details behind the implementation to show our intended process.

Introduction

Our 3D reconstruction algorithm is based on a two-image SfM algorithm, extended to use many images. To use this algorithm, the only equipment necessary is a camera capable of taking videos and looking at the object, a turntable on which the object is to be placed and rotated, and a white background.

First, the user must place the object on the turntable and the camera close to the turntable, with the object entirely within the camera's view space. The user then rotates the turntable 360 degrees, capturing all angles of the object to be reconstructed. When the video is finished, the user runs the 3D reconstructor on the video file they have generated, and a point cloud is to be generated and output for the user to see in an interactive scatterplot. This point cloud is supposed to be accurate up to scale. However, we have not been able to generate an accurate point cloud from an image sequence, the reason for this believed to be due to errors in the algorithm design. The project makes heavy use of OpenCV, a free and open-source computer vision library.

This 3D reconstruction application would be very useful for 3D modelers, who would like to create models that are accurate up to scale. Although the algorithm does not generate a 3D model in itself, it can be used to generate a point cloud with points at accurate positions relative to each other, from which a model can be constructed or refined. The scanning algorithm can be used to generate a collision mesh, which may have uses in augmented reality applications that involve interaction between software and real entities.

The system is challenging to design and use for several reasons. One of the most important challenges to be addressed is the need for accurate feature matching. If there are false positives in the image, the final point cloud will also be inaccurate; this is why the scanned object needs to be on a white turntable and shown against a white background. The object being scanned should also have contrasting, hard-to-confuse features to prevent false positives. The very specific scanning conditions may heavily limit the project's uses. Additionally, pose estimation is another major challenge; since each image pair is independent from each other, it can be difficult to accurately determine the camera's successive poses from the image alone, although one can estimate successive camera poses based on previous poses.

Background

Our sources have primarily been related to two-camera SfM systems. To understand the theory behind SfM, we have consulted the COMP4102A (Computer Vision, Carleton University) lecture by professor Rosa Azami on 3D reconstruction [1]. We have also consulted the lecture by professor Ioannis Gkioulekas for his 16-385 Computer Vision (Carnegie Mellon University) lecture discussing SfM techniques [2]. To better understand how one would actually implement an SfM system, we have also consulted two GitHub repositories for clarification: one by a user named Ashok93 [3], which involved using multiple images, and another one by a user named hsuanhauliu [4], which used only two pictures.

The two-image SfM lectures and the code repository by hsuanhauliu were used in order to give us a basic understanding of SfM, and to give us understanding behind the theory of estimating relative point positions in 3D space given intrinsic camera attributes and correspondences in two images. We concluded that in order to extend the two-camera SfM approach to multiple cameras, we needed camera pose estimations to be influenced by previous pose estimations (except for the first pair of frames); the GitHub repository by Ashok93 showed a way of accomplishing this, which is described in the next section.

Approach

The approach we have taken for our 3D reconstruction algorithm does not vary much from a typical SfM-based 3D reconstruction algorithm. We opted to take a SfM approach since we believed the incremental methods used in SfM would allow for high-fidelity point clouds, and because of the relatively low use complexity - taking a video or sequence of photos of an object - necessary to generate such point clouds. While these benefits are true, it turns out we neglected a critical step in the SfM pipeline: bundle adjustment. As 3D points are estimated from 2D correspondences and more information is acquired, it is necessary to adjust all previous estimations of not only the 3D points, but also the fundamental and essential matrices used to derive point locations. We believe this lack of bundle adjustment is the reason our project generates extremely inaccurate point clouds. The rest of the pipeline, excluding the bundle adjustment, is detailed below. Please note that the camera calibration code is written in C++, while the 3D reconstruction code is written in Python due to ease of calculation and list manipulation; both processes are in separate files, the former being in `calibrate-camera.cpp` and the latter being in `project.py`.

For the recording of the object being turned, the turntable must be white so that none of its features are matched and considered part of the object. Good lighting that eliminates shadows on the scanned object is recommended. It is recommended that the object have many unique, distinguishable features so as to prevent false positives. The object may be at any angle in the first frame, but it must be rotated 360 degrees so that all angles of the object may be captured. The user's hand must not be visible in the rotation, or else its features may accidentally be scanned. Any camera, even a common smartphone camera, is adequate; however, the camera should have a fast shutter speed to prevent motion blur and allow for better matching between frames, and it must remain stationary during the whole recording.

Before the 3D reconstruction can begin, there are two necessary prerequisites. The first is conversion from a video or animated GIF image to an image sequence. This is not strictly necessary, as the video/image source can be read frame-by-frame (with some frames skipped, depending on the image interval), but converting the video/GIF image and reading frames as they are needed is much more memory-efficient than storing an entire video/GIF image in memory. This task is trivial; all that needs to be done is read the input image source frame-by-frame, and write each frame to a file in a directory. When the 3D reconstruction begins, the corresponding images from the image sequence directory are read.

The second prerequisite is camera calibration, in order to estimate the camera matrix and distortion coefficients. The camera can be calibrated by taking several photos of a checkerboard at different angles. A checkerboard generates many easily-distinguishable features, and OpenCV provides an algorithm to detect a chessboard in a picture given its width and height, in tiles minus one. After finding the correspondences of the chessboard in several images, the camera can be calibrated using the OpenCV function `cv::calibrateCameraRO()`. This function returns an estimate of the camera's intrinsic parameters in the camera matrix, and it also returns the distortion coefficients that can be used to undistort a picture (including detected points in that picture). This camera matrix and the distortion coefficients are then written to a file, which is fed into the 3D reconstruction program, run by the user after finishing the recording of the object being rotated on the turntable.

The last step in the process is the SfM-based 3D reconstruction algorithm itself. The following is done for every two frames, with a certain frame interval between both frames.

The first step is feature detection between both frames. An ORB (Oriented FAST and rotated BRIEF) feature detector is run on both frames in order to identify keypoints and descriptors, and then a brute-force matcher using Hamming normalization is run on the lists of found descriptors from both images, in order to detect correspondences between features of both images. We have had better 3D point estimation results (on the between-frames level) when the decimal portions of the correspondence coordinates are truncated, so this is done by converting the returned lists of correspondences to integers and then back to floating-point numbers.

The next step is to use the found correspondences to calculate the fundamental matrix, and then use the fundamental matrix to calculate the essential matrix. OpenCV already provides a function (`cv2.findFundamentalMat()`) to calculate the fundamental matrix, and also returns an outlier mask (but unfortunately, at the time of writing, using the outlier mask has been disabled due to programming issues caused by using it). To calculate the fundamental matrix, the RANSAC algorithm is used, as it tends to generate good approximations provided that less than 50% of the correspondences are outliers (ie. false positives). Since the images we have used tended to have less than 50% of correspondences as outliers, RANSAC seemed to be appropriate over the eight-point algorithm. Sometimes the algorithm will return not one, but three solutions to the fundamental matrix; in this case, only the first solution is used. After the fundamental matrix is calculated, the essential matrix is calculated like so, where E is the essential matrix, F is the fundamental matrix, and K is the camera matrix (used twice, since there is only one type of camera in use):

$$E = K^T F K$$

In order to keep the relative scaling of points between image pairs accurate, the fundamental matrix and the essential matrix are calculated only once, for the first image pair.

With correspondences found and the essential matrix calculated, the camera's pose can be estimated and the 3D points can be triangulated from the 2D correspondences. We used the OpenCV function `cv2.recoverPose()` to return R and t, where R is the camera's rotation matrix and t is the camera's translation vector. Combined, they typically create a 3x4 camera transformation matrix:

$$M = [R \mid t]$$

In order to take into account previous transformations, the current rotation matrix is multiplied with the previous rotation matrix (let this matrix be R_{prev}), and the previous translation vector (let this vector be t_{prev}) is added to the current one multiplied by the rotation vector R':

$$M_{prev} = [R_{prev} \mid t_{prev}]$$
$$R' = R * R_{prev}$$
$$t' = t + R' * t_{prev}$$
$$M' = [R' \mid t']$$

Note that R_{prev} is initialized to the identity matrix, and t_{prev} is initialized to a zero vector, meaning that:

$$M_0 = [I \mid 0]$$

Using the camera matrices, the projections for the “left” (ie. previous or first) and the “right” (ie. next) images are calculated by multiplying M_{prev} and M' each with their respective camera matrix, which is K since the same camera is used for all images:

$$P_{prev} = K * M_{prev}$$
$$P' = K * M'$$

After this calculation, the previous rotation/translation matrix is set to the current rotation/translation matrix:

$$M_{prev} = M'$$

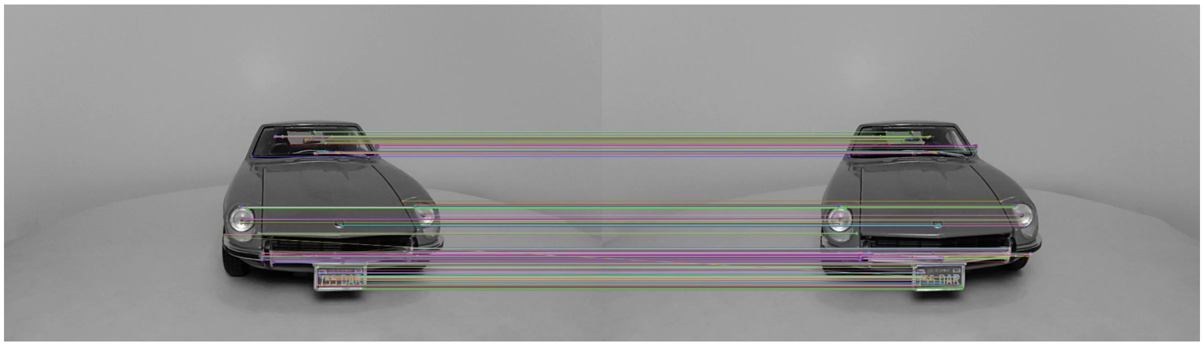
Now that the projections have been calculated, the 3D points in homogeneous coordinates can be estimated using triangulation, which can be done using the OpenCV function `cv2.triangulatePoints()`. This function takes as input P_{prev}, P', and the correspondences between both images found earlier.

Finally, the 3D points are converted from homogeneous coordinates to Cartesian coordinates by dividing each point’s X, Y, and Z coordinates by the respective fourth coordinate. These 3D points for this image pair are plotted on a 3D scatterplot, which is saved to disk and is only used for verification purposes. The points are also added to a final list of points, which comprises the final point cloud. When all images are done being processed, this final point cloud is shown on an interactive 3D scatterplot, which is then saved to disk as an image.

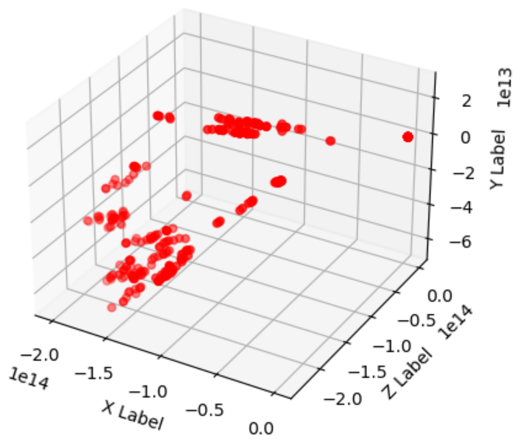
Results

Consider an example of a video showing a toy car rotated on a turntable and "scanned". The following is an image of the car at an angle:

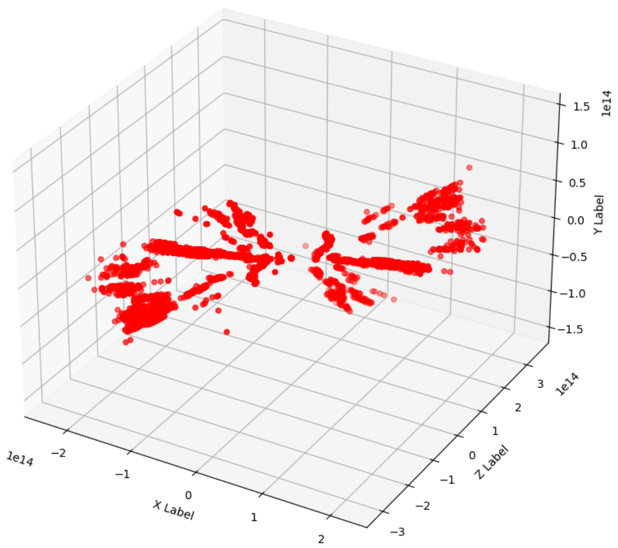
The frame interval is 2 frames skipped between images used. Features matched between the first and third frames:



Point cloud generated by this image pair:

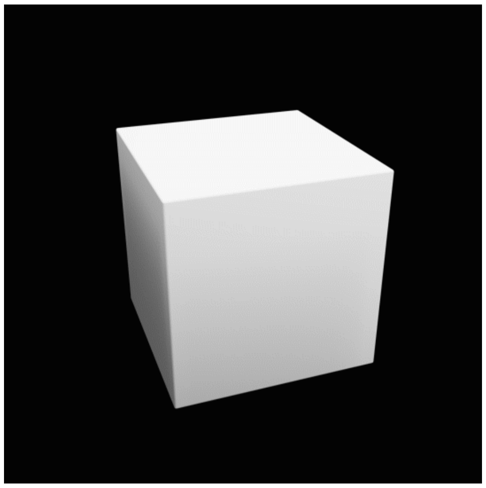


Final point cloud for the toy car:

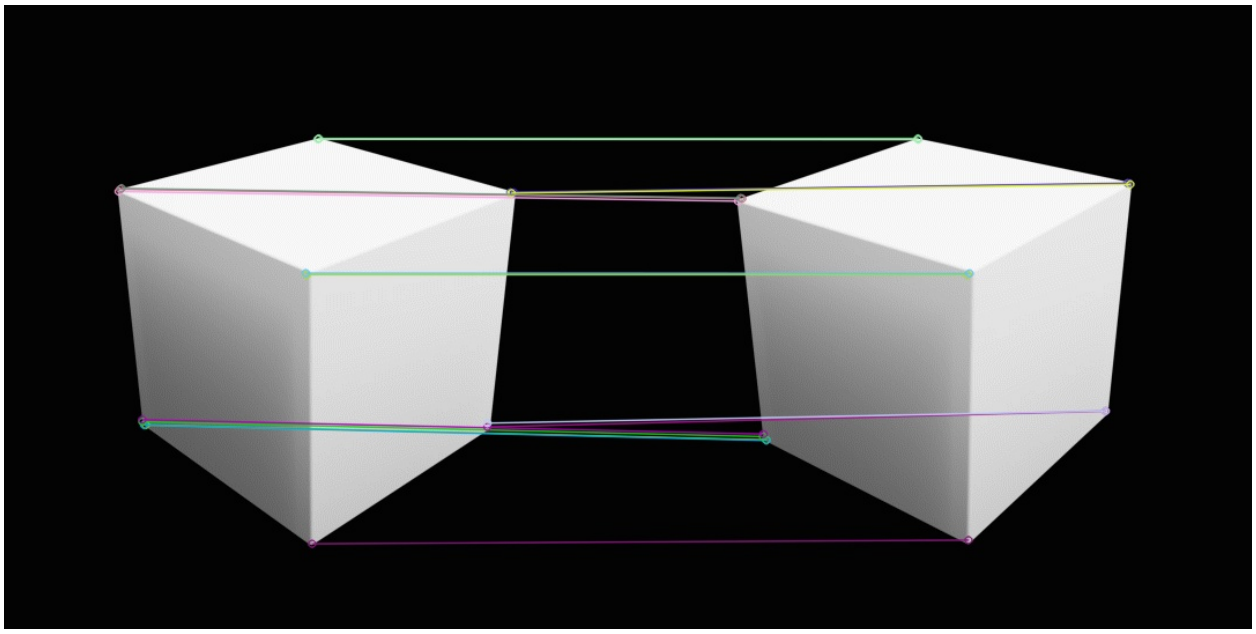


As is visible above, the final point cloud is highly inaccurate, and in this case it is possible the initial point cloud is also inaccurate. Note once again that outliers are not filtered after the fundamental matrix is calculated, so this may have exacerbated inaccuracies in the point cloud.

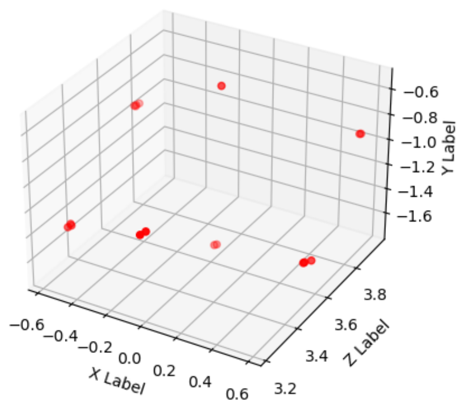
To illustrate the inaccuracy of the system more clearly, consider an example using the following GIF image of a rotating cube:



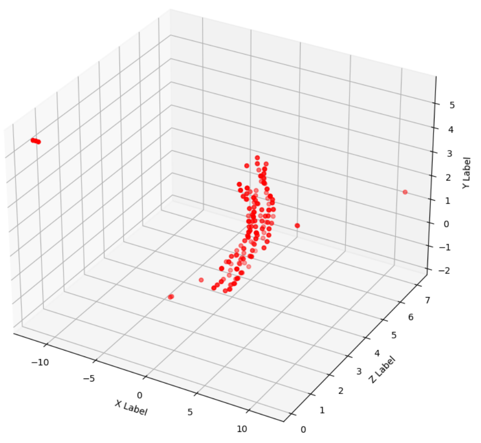
Features matched between the first and third frames (once again, the image interval is 2 frames skipped):



Point cloud generated by this image pair (note how it is accurate, but the result is rotated 180 degrees along the Z axis, though this is not so obvious given the angle from which the cloud is seen):



Final point cloud for the rotating cube (which, as you can see, does not even remotely resemble a cube):



List of Work

Both group members performed equal work.

GitHub Page

https://github.com/CornUkOpia/4102

References

[1] R. Azami. COMP4102A. Class Lecture, Topic: “Reconstruction and Model Building.” School of Computer Science, Carleton University, Ottawa, ON, March 31, 2021.

[2] I. Gkioulekas. 16-385. Class Lecture, Topic: “Structure from motion.” The Robotics Institute, Carnegie Mellon University, Pittsburg, PA, March 4, 2019.

[3] Ashok93. “GitHub - Ashok93/Structure-From-Motion-SFM-: Structure from Motion (Sfm) in Python using OpenCV,” January 1, 2019. [Online]. Available: https://github.com/Ashok93/Structure-From-Motion-SFM- [Accessed March 31, 2021].

[4] hsuanhauliu. “GitHub - hsuanhauliu/structure-from-motion-with-OpenCV: Exercise structure from motion pipeline with OpenCV.” May 25, 2019. [Online]. Available: https://github.com/hsuanhauliu/structure-from-motion-with-OpenCV [Accessed February 15, 2021].