EECS 442 Final Project: Structure for Motion

Prakash Kumar
prakashk@umich.edu

Kushal Jaligama
kushalj@umich.edu

Abstract

We pose two methods to construct a 3D point cloud of an object given a set of images of said object at various angles around it. This concept is called structure from motion, and we would like to implement a basic version of it for small objects in a confined space. We will do this by generating a 3D cloud from the relationship between the camera's intrinsic parameters and the world's extrinsic parameters. The first method involves creating disparity maps between stereo images to make dense clouds and the second method involves using epipolar lines and feature extraction to create sparse, more accurate 3D clouds.

1. Introduction

The purpose of this document is to showcase our current progress on implementing Structure from Motion techniques to re-create 3D models from a set of images. Our first steps included looking at the tutorial from Omar Padierna and OpenCV functions in order to recreate a point cloud from stereo images using a disparity map. We tried to implement these similar techniques on a set of more than two images to make a denser, more accurate point cloud based off matching characteristics. However, because disparity maps don't actually match features and just compute disparity between windows, we cannot accurately extract a 3D point cloud from disparity maps alone. Therefore, we looked into generating point clouds from feature extraction rather than disparity maps.

2. Current Progress

In order to create a stereo depth map/point cloud, we did the following steps through two different methods outlined in figure 1.

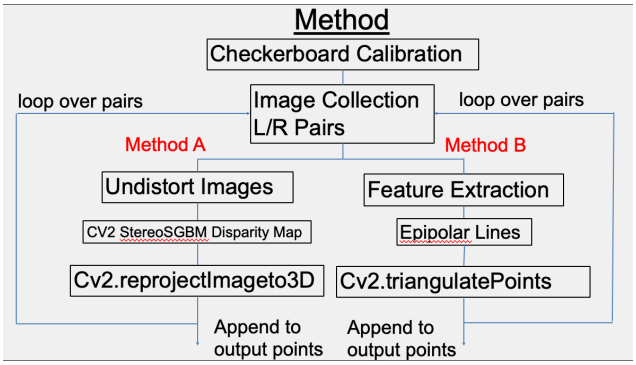

Figure 1: SFM pipeline outlining two different methods

For each method, we will first conduct the following steps:

- Find image sources
- Calibrate the camera using a checkerboard pattern to find distortion coefficients and a camera calibration matrix.

2.1. Finding Image Sources

The first step to reconstruct images into point clouds/models is to collect data. Our current testing has been through using our own phones to take basic stereo images. We attempted using the resources of the Duderstadt design lab to take very structured photos with consistent backgrounds, but we were unable to contact their photogrammetry expert so we decided to take our own set of pictures of objects against a simple background so that any correspondences found from matching correspondences were from the motion of the object itself and not the background. These images were taken using a Google Pixel 1st Gen model. We also collected some images from the eth3D benchmark dataset with camera calibrated matrices so that we wouldn't have to deal with errors in our camera calibration step. An example image collected is shown below.

Figure 2: Example image of user image

2.2. Camera Calibration

We calibrated our cameras with several images of a chessboard pattern. These images were fed into OpenCV's find chessboard library functions. We then refined noise in the image's corner by using subpixeling methods, and then ultimately estimated the camera's intrinsic parameters and distortion model through OpenCV's available camera calibration library function. This calibration procedure can improve if we pass along lots of images of chessboards. In addition, cameras store their focal length information in the EXIF data contained in image files. We used these fx and fy values as opposed to OpenCV's estimations of focal lengths because it's a more concrete figure that the pictures themselves provide. Figure 3 shows an example of an image of a chessboard we used to calibrate the camera. One thing to note is that since the paper was not exactly flat, this could cause issues with the calculation of the camera matrix.

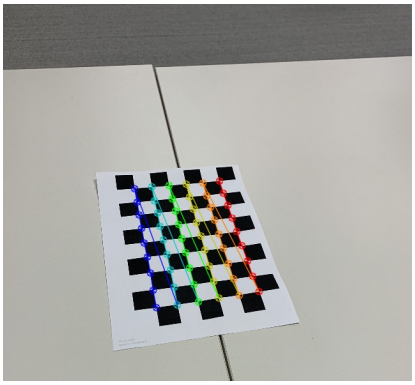

Figure 3: Checkerboard Calibration Image

2.3. Method A, Using OpenCV StereoSGBM

After collecting images and calibration matrices/focal length, we try two different methods for determining the 3D point cloud of the images. The first method we can use is OpenCV's StereoSGBM object in order to compute disparity maps.

2.3.1 Undistort Images

OpenCV calculates for us an estimate of distortion coefficients, drawn from the calibration images. With these, we can undistort an input set of images, namely the stereoscopic images that will be used to generate a point cloud in this project. To do this, we use:

```
img_1_undistorted = cv2.undistort(img_1, K,   d
ist, None, camera_matrix)
```

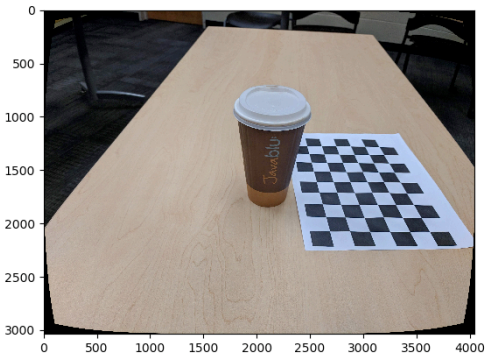

Figure 4: Undistorted Image of Coffee Cup on a Table

2.3.2. Depth Map Generation using Feature Block Matching

We have generated disparity maps using OpenCV's StereoSBGM library functions. There are several parameters in the function that are configurable when gathering a disparity image, namely:

- Number of disparities
- Block size
- Uniqueness ratio
- Speckle window size
- Speckle range
- Maximum pixel distance of features between images
- Window size for how many disparities to clump together

In order to understand how these parameters affected the overall disparity image (and ultimately the point cloud), we created a simple GUI that can manipulate some of these parameters manually. On a test image set of a coffee cup on a table, we were then able to get a disparity map that can clearly identify the cup as having larger disparities than the other objects (as it is the closest object to the camera). Figure 5 shows our simple GUI with block size and window size as enabled parameters to slide around, while witnessing its effects on the overall disparity map. It is unclear what the "perfect" settings are for any given image, but when running on different sets of images, we found that the parameters would have to be tweaked a little bit to get an accurate disparity map, so

having a basic GUI helped find “good enough” parameters by visual inspection.

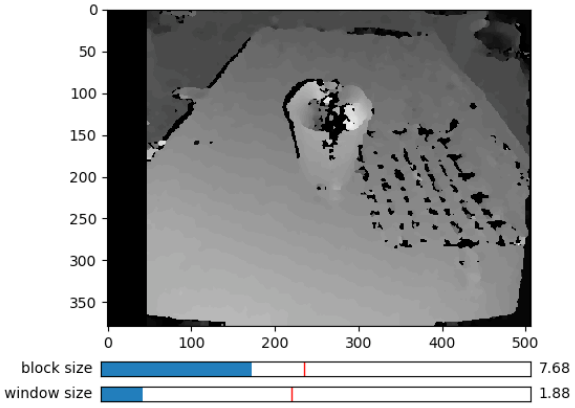

Figure 5: Disparity Map for Coffee cup on a Table

2.3.3. Reprojection of Images into 3D space and Visualization using Open3D

Using cv2.reprojectImageTo3D() and Open3D, we were able to re-create a point cloud of the coffee cup image. Figures 6,7 shows the point cloud that clearly distinguishes the depth of the cup from the depth of the table, as well as the “roundness” of the top of the cup. We found that the disparity map generation often worked better when we also put a calibration checkerboard in the image.

Figure 6: Coffee Cup Point Cloud 2

Figure 7: Coffee Cup Point Cloud 3

2.3.4. Multiple Images

In order to extend this to multiple images, we first started with a simple pipeline that computed disparity maps of the pairs of the images taken directly next to each other, reprojecting them to 3D and adding up all the points together. This naïve approach didn’t work very well, as the 3D space of the same features of the image pairs ended up projecting to different 3D space. The following figures show these incorrect point clouds.

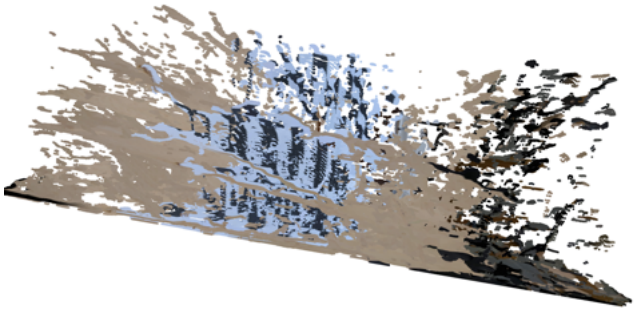

Figure 8: Coffee Cup with Checkerboard Disparity Maps projected to 3D

In order to fix this issue, we must concatenate these point clouds after applying a known transformation between them. We tried to calculate this transform between single images of two different stereo perspectives and apply it to the point cloud generated from the second stereo perspective. This new point cloud would be in the coordinate frame of the point cloud generated from the first stereo perspective A. Then we can overlay the point clouds from perspective A and B to give us a denser reconstruction of the key points of the object from multiple perspectives. An outline of this method is shown in figure 9.

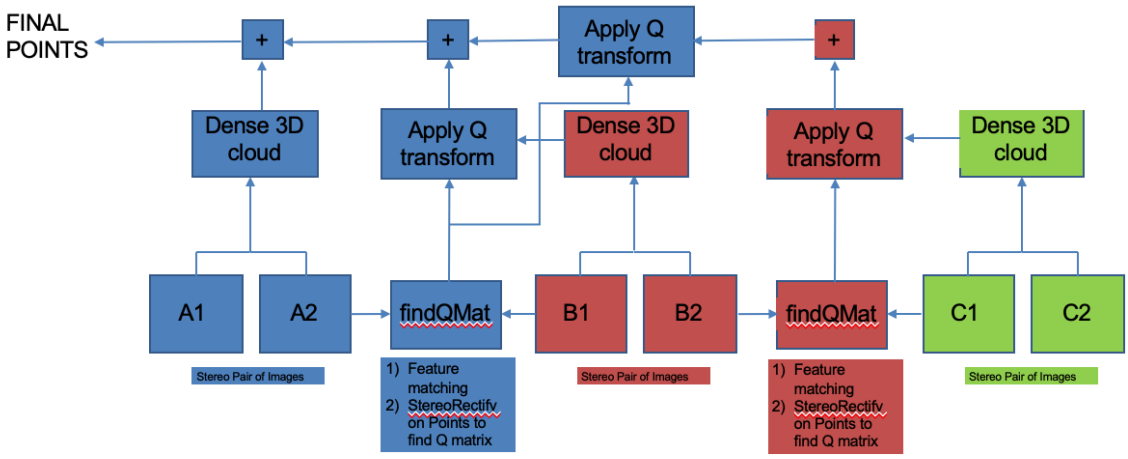

Figure 9: Multi-view 3D stereo reconstruction pipeline

In the process of generating this known transformation, we used OpenCV's stereoRectify function to establish the transformation between two different stereo perspectives and applied it to one of the point clouds. We did this because we wanted to understand how to place point clouds from two perspectives into their respective coordinate frames. StereoRectify needs an R and t between the stereo cameras in order to determine the correct transformation Q. Because we are not actually using stereo cameras and are instead using one camera taking multiple imags, we need to compute what R and t are. To do this, we use epipolar methods similar to the ones in homework5 in order to determine an R and t between each camera that used the image. Using this, the distortion data and camera calibration matrix, we can use stereoRectify() to find a 4x4 Q matrix that can be applied to point clouds as a whole.

2.3.5. Analyzing the Output

We generate the output transformed 3D clouds by calling OpenCV's reprojectTo3D function, which uses our calculated Q. This caused many issues for us, however, and many of the points that were calculated ended up approaching infinity. We were unsure why this happened but came up with a few possible reasons:

- OpenCV's stereoRectify function calls for two images that come from a hypothetical stereo camera, but our images may have been taking at entirely different angles. This could cause issues with determining an accurate Q matrix to begin with.
- By combining the 3D point clouds generated from OpenCV's StereoSGBM and our own code from homework5, perhaps we are missing a normalization step in the data that we cannot see because we don't know how StereoSGBM manipulates the data under the hood. This could mean that we would have to open the source code of OpenCV in order to determine how to continue. Because opening up the OpenCV source code and trying to figure out how it works would be a large undertaking, we decided to try and find a different method to create point clouds with multiple images.

2.4. Method B: Epipolar Method

Method B involves using SIFT to extract features from the two images and then use these matches to find Epipolar lines and ultimately call cv2.Triangulatepoints to plot the output point cloud.

2.4.1 Feature Extraction

We start with feature extraction using SIFT between each of the images. This is done using cv2.xfeatures2d.SIFT.create() on the left and right gray scaled version of the image. This takes far too long for the images input since they are too large, so we first down sample the images by a factor of 8 so that the SIFT matching takes much less time and we still have enough details to maintain the image. We then use opencv's FlannBasedMatcher to run a knnMatch between the descriptors from the left and right images and plot them to make sure the matches are accurate. The following figure shows us matches from 2 stereo images.

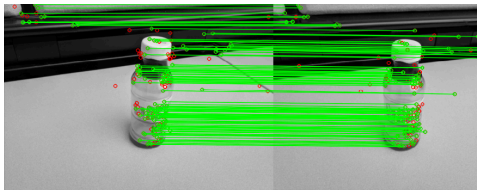

Figure 10: SIFT matches of dunkin bottle

2.4.2 Find Fundamental Matrix and Draw Epipolar Lines

Using the matches, we can solve for the fundamental matrix f and use that to draw the appropriate epipolar lines for the image. The following figure shows the epipolar lines for the above image.

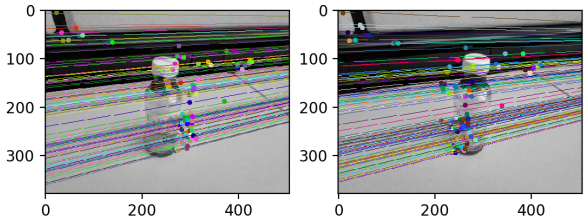

Figure 11: Epipolar lines for dunkin bottle

2.4.3 Epipolar Considerations

Because feature-matching is not always accurate, there are some considerations that we must take when using it to compute epipolar lines. Oftentimes, we would get epipolar lines that would converge at a point on the image, particularly when there were not very many features or sometimes when there were too many features. Figure 12 shows such an image.

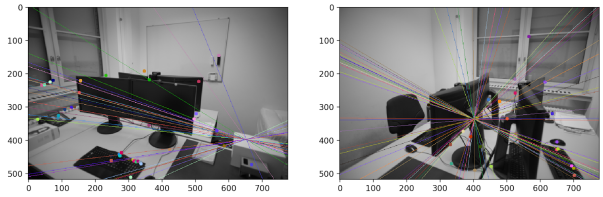

Figure 12: Not very many features due to bad combination of images where epipolar lines converge incorrectl

We noticed that upon downsampling the same image less (to obtain more features), the epipolar lines would converge to a point in the image causing the point clouds to be very incorrect. Although we don't know the cause for this, one solution we found was to use a different algorithm than the 8POINT algorithm and use openCV functions instead of our homework 5 approach. We tried cv2.findFundamentalMat with the cv2.FM_LMEDS least median estimator algorithm instead, which performs slightly better.

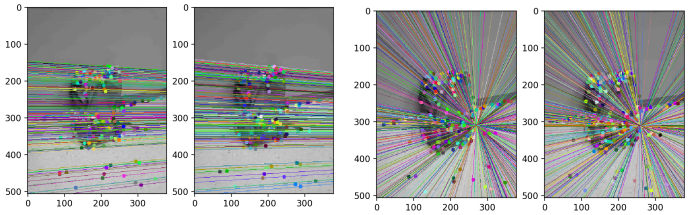

Figure 13: [Left]OpenCV FM_LMEDS [Right] Epipolar "hw5" approach

2.4.4 Use TriangulatePoints and Draw a Point Cloud

Using cv2.triangulatePoints, we can then create a point cloud similar to how we did in homework5. An example point cloud is shown below.

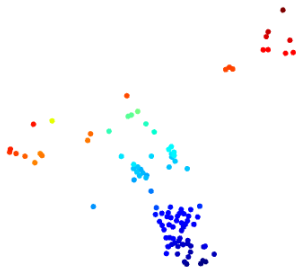

Figure 14: Point Cloud for dunkin bottle. You can see the outline of the bottle in the green/blue colors in the image.

2.4.5 Picking R and t

When using cv2.triangulatePoints, we need to give it the appropriate R and t matrices of the transformation between cameras so that the points that are transformed are in front of the camera. We can estimate R and t by first creating the essential matrix which is K.T @ f @ K (where @ is matrix multiplication, and K is the calibration matrix). Then, by applying cv2.decomposeEssentialMat() on the essential matrix we get two possible rotations and a translation that could be either positive or negative. This gives us four different options for R and T: [(r1, t), (r1, -t), (r2, t), (r2, -t)]. To pick the correct one, we added a simple script that allows a user to visualize all four versions and then pick which one they think is accurate. Picking the correct R and t is crucial to creating the correct point clouds because the outputs when they are incorrect are completely garbage. The following figures show the varying output when R and t are incorrect.

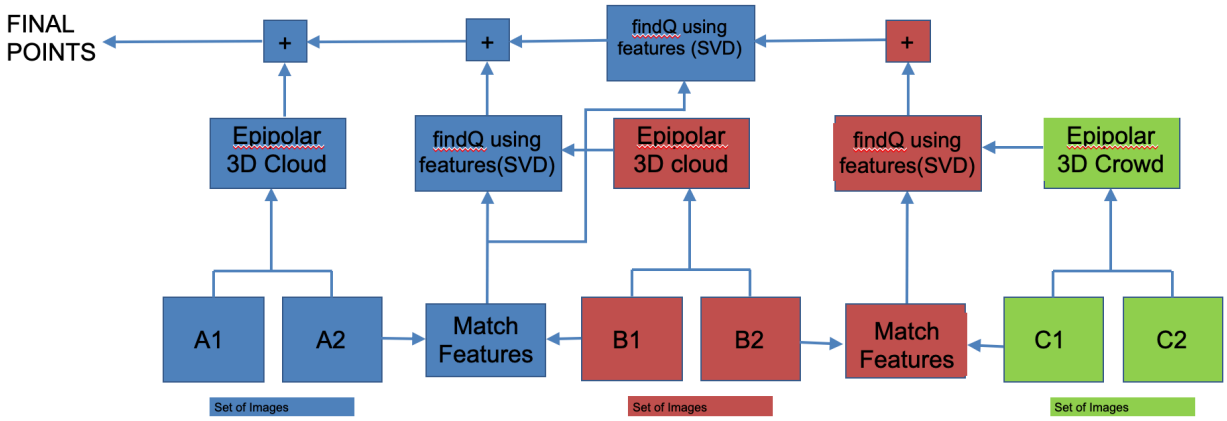

Figure 15: Multi-view 3D stereo reconstruction pipeline with SIFT feature matching

Figure 16: [Left] Correct R and t Point Cloud
[Right] Incorrect R and t Point Cloud

2.4.6 Multiple Images using Epipolar Geometry

The next step in the pipeline is to try to implement the same approach using multiple images. This can be done in a similar manner to the multiple image pipeline we attempted to use in the disparity map method. A figure outlining this method is shown in figure 15.

- Initialize the master point cloud from the one generated from the first stereo image pair.
- loop through every stereo image_pair i:
    - Stereo Image Pair A = image_pairs[i]
    - Stereo Image Pair B = image_pairs[i+1]
    - Calculate the indices of corresponding SIFT features between images A2 and B1 and use those to gather the 3D points in the point clouds associated with those indices (matched points)
    - Calculate transformation Q between point clouds A2 and B1
    - Apply transformation Q to point cloud B
    - If not at the first stereo set:
        - Concatenate point cloud A to the master point cloud.
    - Concatenate point cloud B to the master point cloud.

2.4.7 Estimating Q

We wanted to find the best approximation for a 4x4 transformation matrix Q that would take the 3D data and find a least-squares mapping between one set of 3D points to another. We tried to model this homography off of the same underlying idea that if two points are homogeonous, their cross product must be 0, but quickly ran into some troubles due to the fact you cannot take the cross-product using only two 4D vectors due to dimensionality reasons. This made it difficult for us to find a mapping of homogenous 3D coordinates. Therefore, we tried to find another way to find a mapping matrix and that way was to remove the "homogenous" element and map a 3x4 matrix to the points instead:

$$\begin{bmatrix} u \\ v \\ z \\ w \end{bmatrix} = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u' \\ v' \\ z' \\ w' \end{bmatrix}$$

Therefore,

$$\begin{bmatrix} u \\ v \\ z \end{bmatrix} = \begin{bmatrix} R11 & R12 & R13 & t1 \\ R21 & R22 & R23 & t2 \\ R31 & R32 & R33 & t3 \end{bmatrix} \begin{bmatrix} u' \\ v' \\ z' \\ 1 \end{bmatrix}$$

So

$$\begin{bmatrix} u \\ v \\ z \end{bmatrix} = \begin{bmatrix} R3x3 & t3x1 \end{bmatrix} \begin{bmatrix} u' \\ v' \\ z' \\ 1 \end{bmatrix}$$

In order to do this fitting, we took ICP (Iterative Closest Point Algorithm) that normalizes the points to 0 mean and then computes SVD on A^T*B to ouput U, S and Vt. R and T can then be computed as:

$$R = Vt^T U^T$$
$$T = -RA^T + B^T$$

Where A and B is the mean of each point.
Because we took this estimation from a website Clay Flannigan's github page, we wanted to make sure it was functional in python3 as the code was originally written in python2. After doing the conversion, we ran a test script adapted from nghiaho.com that created 10 3D points and a random transformation of said points and computed their corresponding rotation and translation. After doing the transformation and subtracting the values from the original points, we compute the RMSE error and get a number of $\sim 1.5635 * 10^{-12}$, so we know the estimation is working correctly.

2.4.8 Output (Qualitative Analysis)

The following is the output point clouds given a set of 4 images around a backpack. The following images show the entire pipeline along with the final point cloud.

Figure 17: Four Input Backpack images

We then apply the matching method on each image, following the steps from the algorithm in figure _. After we find the appropriate matches, we find the fundamental and essential matrix to find an acceptable R and t of the camera. After finding the appropriate SIFT matching between sets of images, we find the appropriate rotation and translation in 3D space using the ICP algorithm. The following figure shows such a translation.

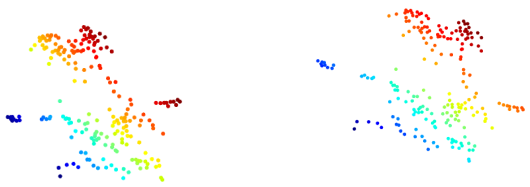

Figure 18: [Left] point cloud for one backpack image along with its [Right] translation post SIFT-Matching

Once we have all the point clouds transformed into the first pair's 3D space, we can simply append them all and view them. Figure 19 shows the point clouds from the 4 images appended together. You can see the cloud is denser than just one image input and that the line that shows where the wall meets the floor is still visibly a line, which means the transformation seemed to work correctly.

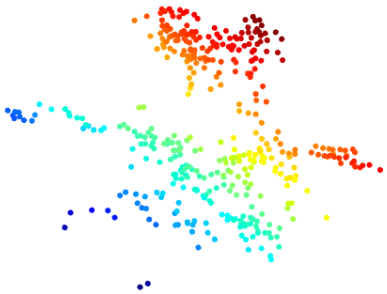

Figure 19: Final appended point cloud

2.4.9 Where things can go wrong

Because we are computing the 3D transformation through SIFT matches, things can go very wrong if there are not enough matches or if there are too many outliers in the matches. We found that by simply changing the ratio in the ratio test for the Flann-based matcher by 0.05, we can drastically change whether or not the point clouds match properly or not. Furthermore, if just one plot generated by our picked R and t value is wrong, this can screw up the entire point cloud as the transformed point clouds will be very far away from each other.

Figure 20: Example of poorly translated/rotated Point Clouds leading to a bad final result

2.4.10 Quantitative Analysis

We wanted to find a way to quantitatively see if our SFM pipeline was working well enough. Visually, we could see that it oftentimes would get outlines of the objects and a small amount of depth, but not much more than that.

Although we could not get the pipeline to do a great job at detecting depth, we tested against a battery box to see if it would find the edges of the box, and use the length of the box and the amount of pixels it was in the model to quantitatively see if we did a good job.

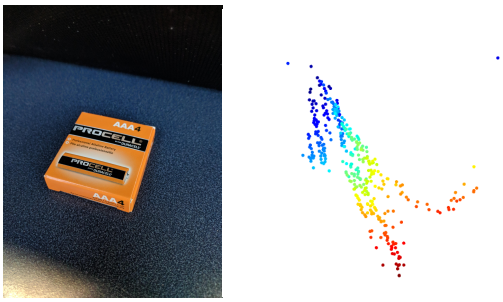

Figure 21: Battery box and it's corresponding point cloud.

It is difficult to measure how well our algorithms create the point cloud based off of this data, since it doesn't even accurately get the entire box, nor can we confirm that our translation/rotation of point clouds is working correctly. This in turn led to an improperly constructed point cloud and prevented us from using measurements of the object's real-life width, height, and lengths as a metric of comparison to the point clouds that we generate.

2.4.11. Using Online Ground-truth Data

Another way to test that our system is working properly is to use online ground-truth data, such as the one from the eth3d dataset. Because many images in this dataset are scenes rather than individual objects, we found that the point cloud generated from them became rather useless as we could not determine which points mapped to which space and could not calculate an error cost for it.

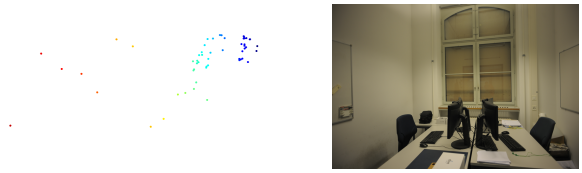

Figure 22: A rather useless sparse point cloud of the office images from the eth3D dataset

Ultimately, we could not find the best way to quantitatively see our progress because our implementation simply wasn't robust enough to create enough points to be able to test its functionality. However, if we did have enough points in our sparsely generated 3D cloud, one could find each point's closest point in the ground-truth 3D points and compute a mean-squared distance between all these pairs to determine a simple cost estimate. There are many problems with this method, one being that it is possible that our 3D cloud is a different coordinate frame (scale, rotation, translation) than the ground-truth dataset. Implementation of this required a lot more effort to obtain accurate empirical results, so we have simply laid out the foundation for obtaining quantitative results.

3. Conclusion and Next Steps

Throughout this paper, we identified two general methods to creating 3D point clouds from 2D images and showed initial results in each method.

Using the disparity map approach, it is very simple to get satisfying results with a dense point cloud of two stereo images. However, we could not implement an approach to adding more images to the pipeline due to constraints on the complexity of OpenCV's documentation and source code. Next steps in this method involve modifying and understanding OpenCV's source code to learn any normalization steps we may have missed in our own pipeline thereby allowing it to perform stereo reconstruction on an asymmetric stereo camera setup.

Using the epipolar approach, we can make sparse point clouds using SIFT matches between two images. We were also able to build a simple pipeline that uses feature matching to determine a transformation between point clouds from different perspectives to get them into a common coordinate frame. We were able to get preliminary results but could not consistently extend these methods to a multi-view stereo case due to the complexity of the parameters to be tuned. Next steps involve testing the pipeline on many feature-rich imagesets while tweaking the many parameters in the pipeline to get a better understanding of their impact. With this, we can determine the optimal parameters for the pipeline to create robust sparse point clouds.

References

[1] [1] 10155043712801134. "Oscar Padierna - Stereo 3D Reconstruction with OpenCV Using an IPhone Camera. Part I." Becoming Human: Artificial Intelligence Magazine, Becoming Human: Artificial Intelligence Magazine, 2 Jan. 2019, becominghuman.ai/stereo-3d-reconstruction-with-opencv-using-an-iphone-camera-part-i-c013907d1ab5.

[2] [2] "Camera Calibration With OpenCV¶." Camera Calibration With OpenCV - OpenCV 2.4.13.7 Documentation, docs.opencv.org/2.4.13.7/doc/tutorials/calib3d/camera_calibration/camera_calibration.html.

[3] [3] "3D Reconstruction from Multiple Images." Wikipedia, Wikimedia Foundation, 16 Mar. 2019, en.wikipedia.org/wiki/3D_reconstruction_from_multiple_images#Projective_reconstruction.

[4] [4] “Structure from Motion.” Wikipedia, Wikimedia Foundation, 7 Mar. 2019, en.wikipedia.org/wiki/Structure_from_motion.
[5] [5] ClayFlannigan. “ClayFlannigan/Icp.” GitHub, github.com/ClayFlannigan/icp/blob/master/icp.py.
[6] [6] “OpenCV Documentation Index.” OpenCV Documentation Index, docs.opencv.org/.
[7] Computer Vision and Geometry Group, and ETH Zurich. “Welcome to the ETH3D Benchmark.” ETH3D Benchmark, www.eth3d.net/.