The Implementation of an Augmented Reality Viewer

Ke LIANG, Lin LI, Yifei XIANG, Lindsey Schwartz

April 30, 2020

Contents

1 Introduction

This project is aimed to do some implementations to realize an augmented reality viewer that displays artificial objects overlaid on the images of a real 3D scene. All the work is based on the available COLMAP software for 3D reconstruction and Matlab. We take the set of the images for the table with the book laying on. Based on the postprocessing on the results for the images from COLMAP, we write the code augmentedRealityViewer.m to finish the 10 steps shown below, including finding the dominant plane, placing a virtual 3D object on the plane, projecting the object into the original images. We pick a cube as our virtual 3D object. The results looks pretty good after following the steps mentioned in the description.

2 Motivation and Goals

The motivation of this term project is to get fully understandings on the 3D virtual reality problems with a simple program. Although it is simple, all the things about the 3D virtual reality problems are covered, such as the registration, projection, and reconstruction.

The goal of this project is to realize an augmented reality viewer that displays artificial objects overlaid on the images of a real 3D scene.

3 Contribution

All of us enrolled in the each parts of the work, including the coding, report and slides in this project. All of us tried step 1 to step 4 individually.

Lindsey and Ke worked more on step 5 to step 8, both for the coding and report, and Lin and Yifei worked more on step 9 and step 10, both on for the coding and report. We also will swift to double check all the things.

4 Related Work

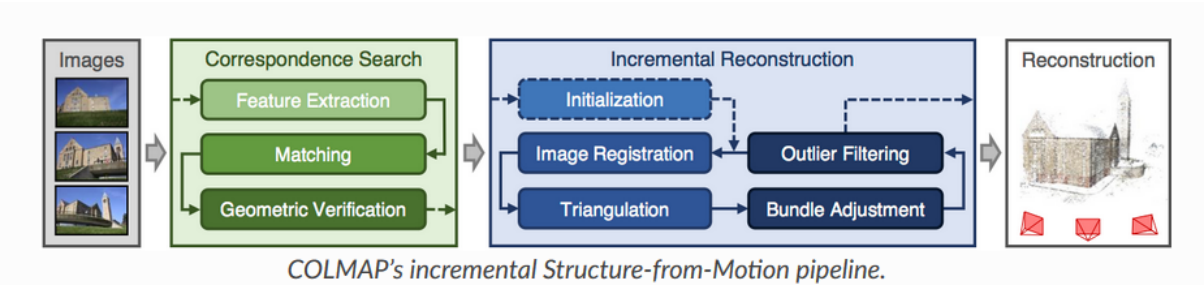

Figure 1: Flowchart for COLMAP [1]

Structure-from-Motion (SfM) is the process of reconstructing 3D structure from its projections into a series of images. The input is a set of overlapping images of the same object, taken from different viewpoints. The output is a 3-D reconstruction of the object,

and the reconstructed intrinsic and extrinsic camera parameters of all images [1] [2]. More information on Structure-from-Motion in general and the algorithms in COLMAP can be found in [1] and [2].

5 Implementation Details and Results

5.1 Step 1

The first step is to collect a set of images that show an area of a 3D scene. We take 9 pictures of a scene which has a dominant planar surface. The table has lots of texture for the COLMAP to find and produce matches between the images. To have the same internal camera parameters, we use a single camera at a single focal length, without zooming in or out between shots. To make things more interesting, we put some games on a table so that it now includes some 3D structure besides the dominant plane, the floor. The images we we used are shown below.

Figure 2: Images taken for step 1

5.2 Step 2

In second step, we downloaded the working version of COLMAP from the *https* : *//colmap .github.io/install.html* and read the documentation and tutorial videos. Then we ran

feature extraction, matching, incremental reconstruction and bundle adjustment to get a "sparse" reconstruction, which is a 3D cloud of points, on our data. This reconstructed point cloud can be seen in Figure 3. In step2, we set all parameters to default and click the box "shared for all images" since we took our pictures at a single focal length.

After reconstruction, we export the sparse 3D point model COLMAP as a set of text files. It contains 3 text files: points3D.txt cameras.txt, and images.txt, which are described in depth in steps 3 and 8, respectively.

Figure 3: "sparse" reconstruction

5.3 Step 3

First we read in the 3D point cloud stored in points3D.txt, which is generated by COLMAP in step 2. After transferring the data to mat format using "cell2mat", we get a 9454 x 26 matrix. We only need the [X,Y,Z] coordinates for each point for what we want to do, and can ignore other fields in the text file. So we only extract the second to fourth columns.

This part of code is under the comment "Read in the file" and "Organize data" in the augmentedRealityViewer.m.

5.4 Step 4

First we write a RANSAC routine to find the largest subset of 3D points that can be described by the equation of a 3D plane in Matlab. According to the outline for RANSAC loop in the lecture, we do RANSAC as shown in *Algorithm*1.

To answer the questions in project description, the minimum number of 3D points needed to fit a 3D plane is 3. We use the Matlab *randsample* function to randomly select three points from the scene upon each iteration. The equation of a plane is given by

$$ax + by + cz + d = 0$$

where (a, b, c) is the normal vector to the plane. We can compute the normal vector by obtaining two parallel vectors, $V1$ and $V2$. $V1$ and $V2$ are computed by subtracting

one sample point from the other two. The cross product of these vectors yields the normal. Once we have the normal, we can solve for d, and thus have the equation of our plane.

To select those "close enough" points pairs, we compute distances and filter those $distance < threshold$, where $threshold = 0.5$. Distance, D, to the plane was calculated as follows

$$D = \frac{ax + by + cz + d}{\sqrt{a^2 + b^2 + c^2}}$$

where a, b, c, and d are known and the points are substituted for x, y, and z. We then total the number of points greater than the the threshold, if this set of points is larger than the current largest set of points, we update the inlier count and replace the current set of inliers with the new set of inliers.

This part of code is under the comment "RANSAC: Find the largest subset of 3D points that can be described by the equation of a 3D plane".

Algorithm 1 RANSAC algorithm

Input:
the [X,Y,Z] coordinates for each point
Initialization:
Set minimal number of points and distance threshold
Initialize global variables ∅
for times = 1 to 10,000 **do**
Randomly select minimal number of samples
Fit the plane using the samples:
–Use two vectors parallel to the plane
–Compute cross product
–Compute d from $ax + by + cz + d = 0$
Map all other points (x_j, y_j) by that transformation to get (x_{pred}, y_{pred})
And compute distance between each pairs
For each distance < threshold, increase inlier count by one
if inlier count is greater than the global one, update the inliers and global inlier count
end for
return the global on plane inlier

5.5 Step 5

We display the 3D points produced by COLMAP and denote the inlier points. The points in the dominant plane are colored green. For those points which are outside the dominant plane, we draw red points. As is shown in Figure 1, we find that the dominant plane is mainly the carpet, and all other objects, such as the outline of the table, is colored in red. This part of code is under the comment "Display" in the augmentedRealityViewer.m.

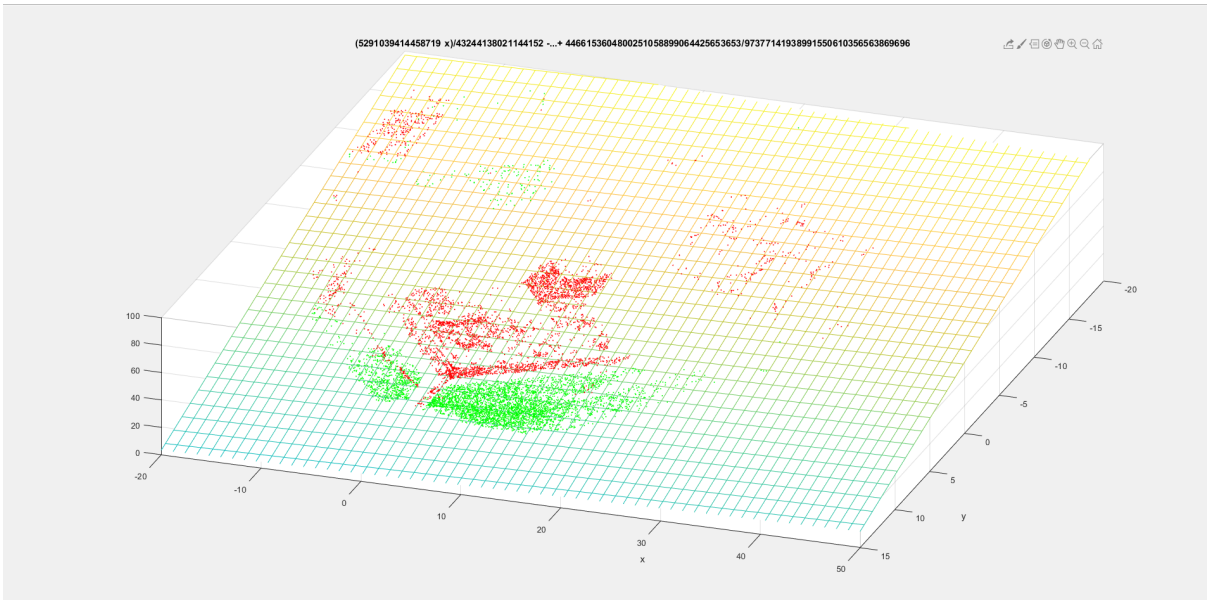

Figure 4: Display of inliers.

5.6 Step 6

In this step, we form a local x-y-z 3D coordinate system where the dominant plane above becomes the z=0 plane. We choose the local origin of this coordinate system so that x=0, y=0 lies roughly in the "middle" of the set of inlier points that lie on the dominant plane.

According to the equation shown below, we build a 3D Euclidean transformation (rotation and translation) that maps each U,V,W in our scene coordinate system into local X,Y,Z coordinates. The transformation matrix contains R matrix and P matrix.

To determine the middle of our inlier points, we compute the half the range in both x and y directions. The transformation matrix is obtained by subtracting our center x, y, and z coordinates from our desired origin (0,0,0). To create the rotation matrix, we need to define the x, y, and z axes. We call the global normal the z-axis. Since we want the x-axis to be perpendicular to the vector [0, 1, 0] and the z-axis, we take the cross product of these two vectors. The y-axis is the cross product of the x-axis and z-axis. The result of this geometric transformation can be seen in Figure 5.

$$\begin{bmatrix} X \\ Y \\ X \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} U \\ V \\ W \\ 1 \end{bmatrix}$$

This part of code is under the comment "Geometric Transformation" in augmentedRealityViewer.m.

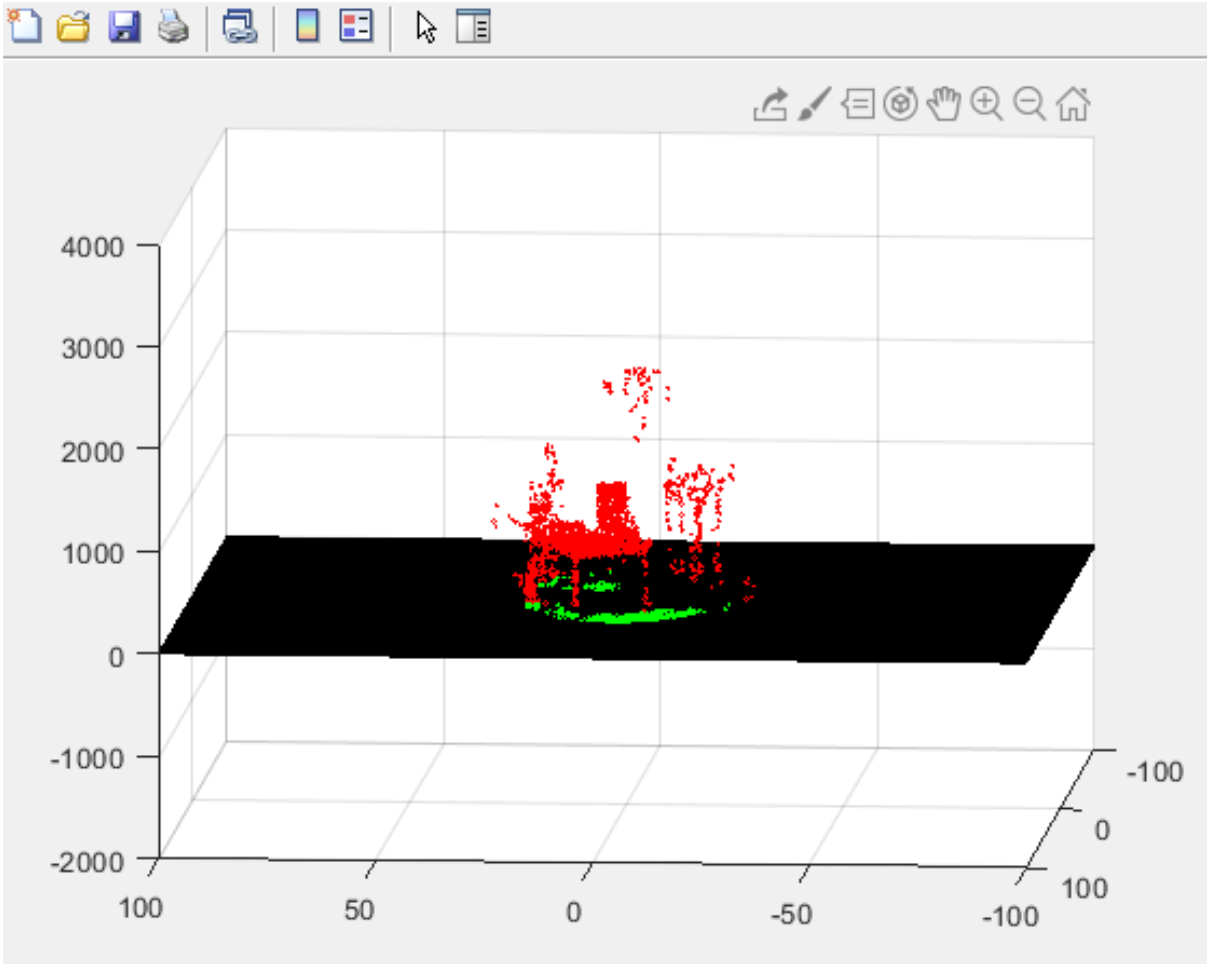

Figure 5: Geometric Transformation

5.7 Step 7

In this step, what we create a virtual object to put in the scene we have.

We create a virtual 3D box to put in the scene. As is shown in Figure 6 and Figure 7, the 3D box is defined with bottom surface lying in our local z=0 plane, so that the center of that rectangular bottom surface is centered at x=0, y=0, z=0.

From the eight corners of the box in x,y,z coordinates, we convert them into scene X,Y,Z coordinates using the transformation matrix we defined in step 6. As is shown in Figures 8 and 9, the eight corner points of the box are lying on the center of our 3D scene.

These parts of the code are under the comments "Create the object and display on plane (simple 3D box)" and "Transform the object points back to scene coordinates".

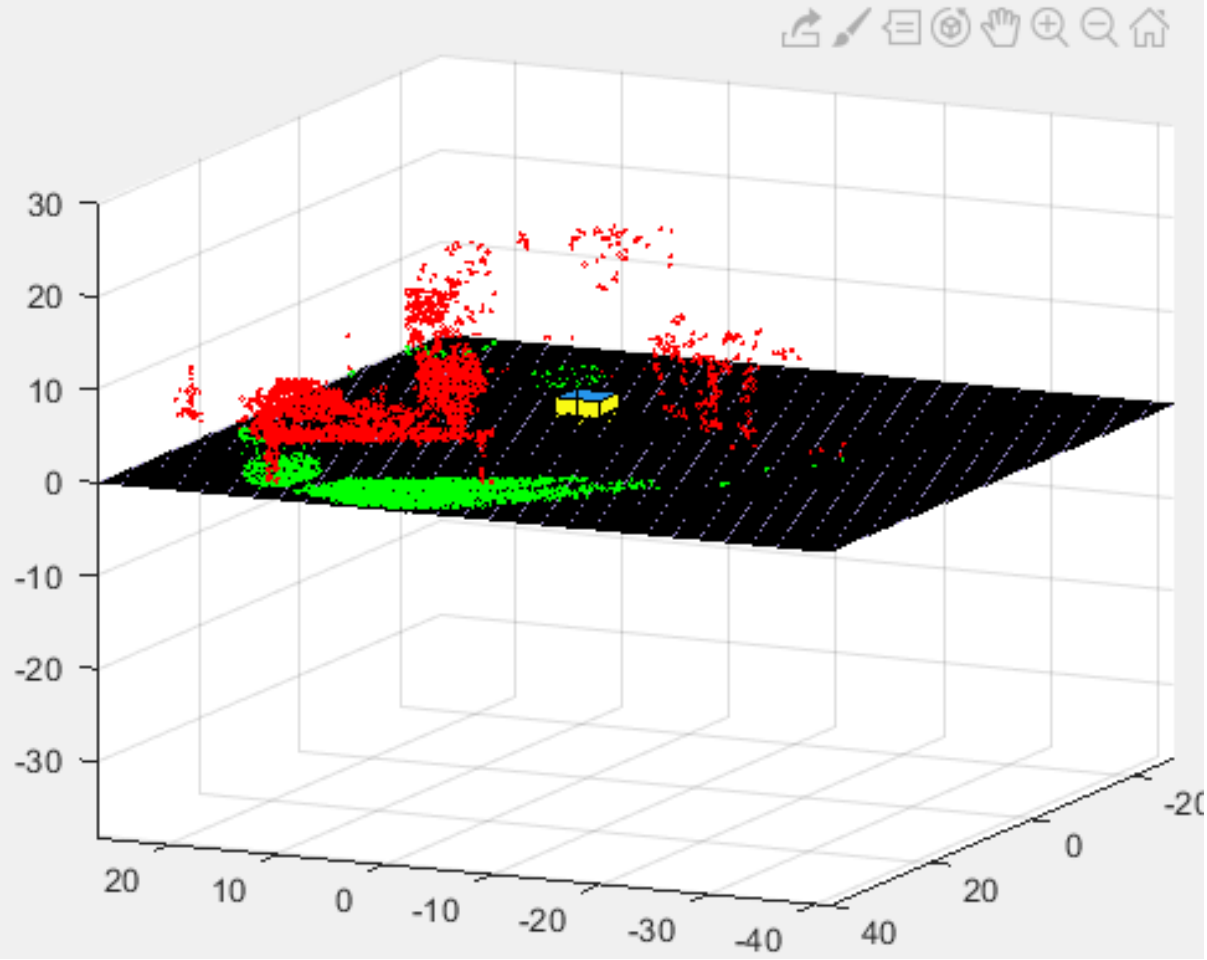

Figure 6: View 1 of the box placed in the scene

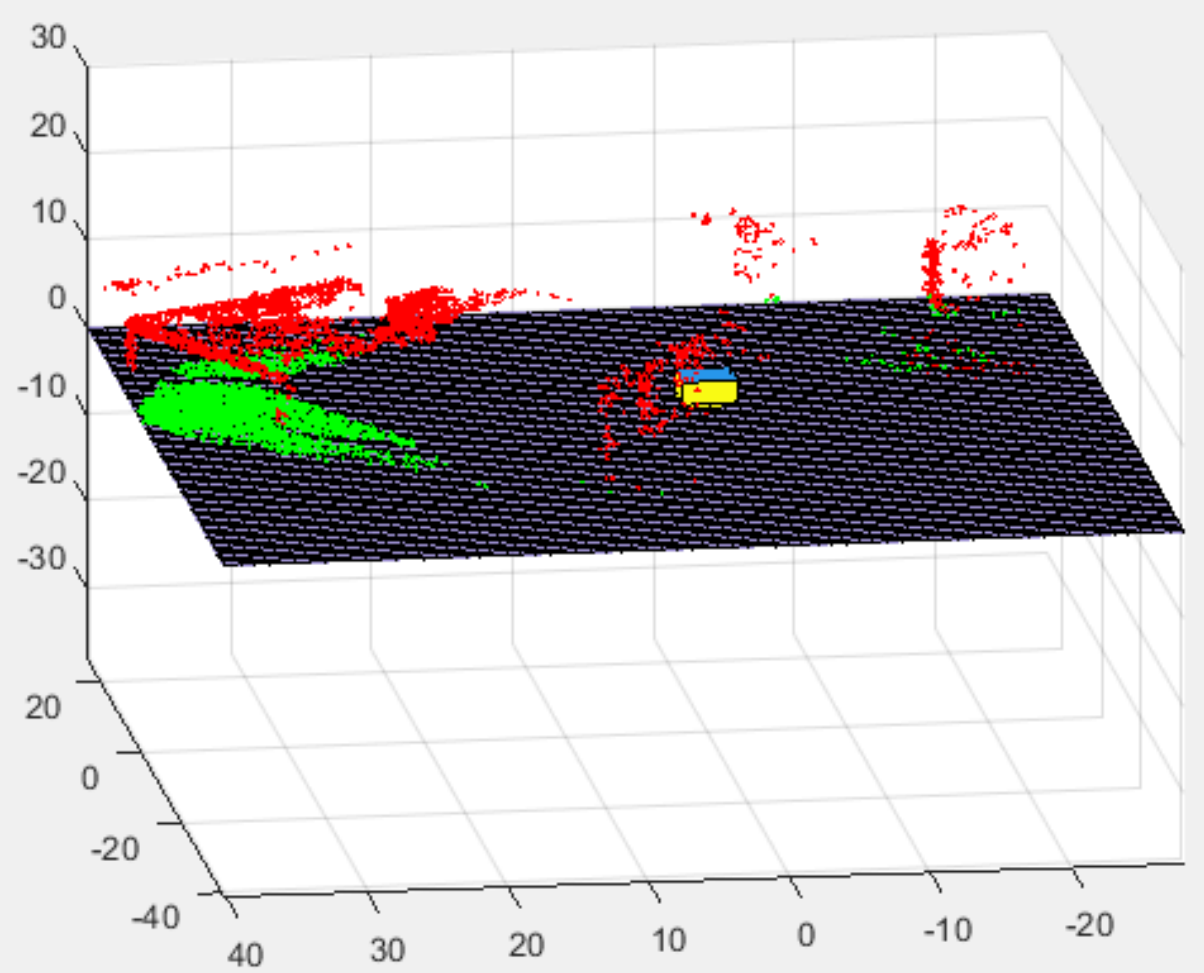

Figure 7: View 2 of the box placed in the scene

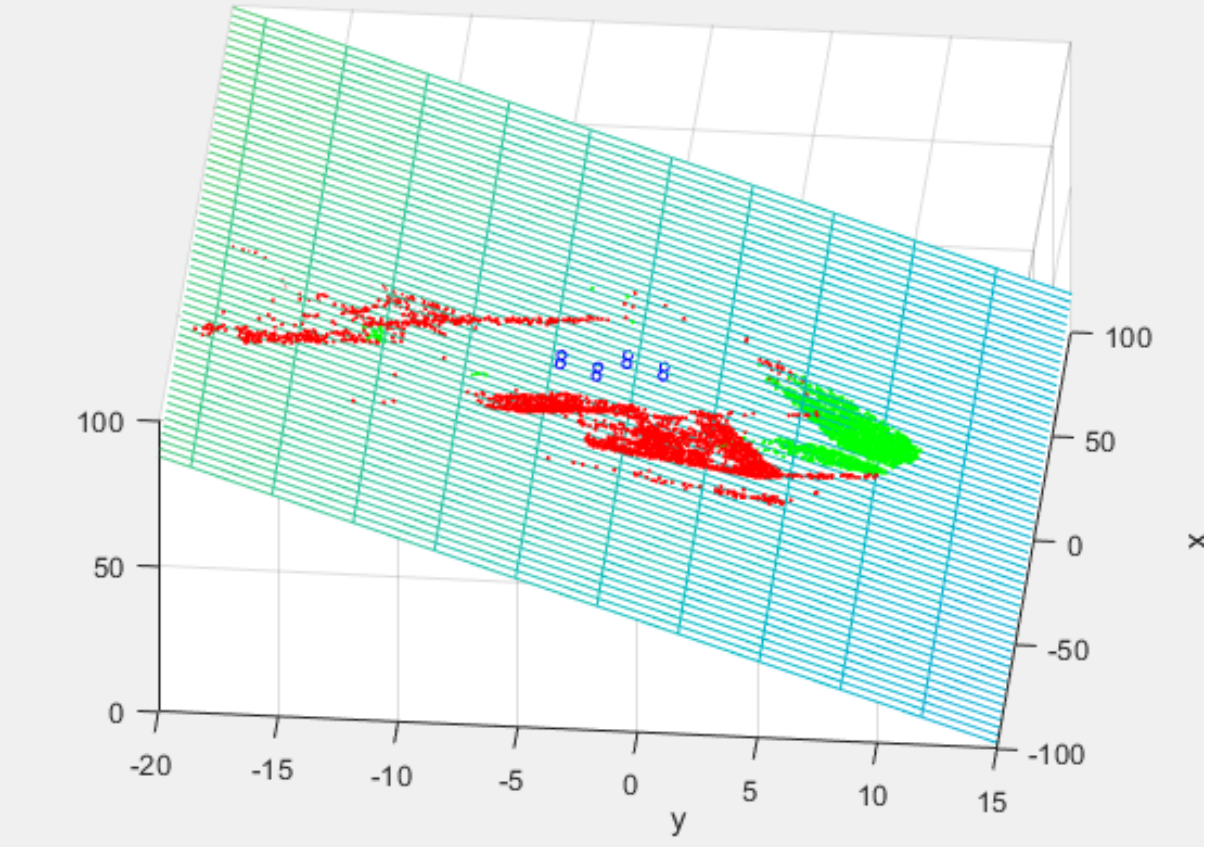

Figure 8: View 1 of the transformed points

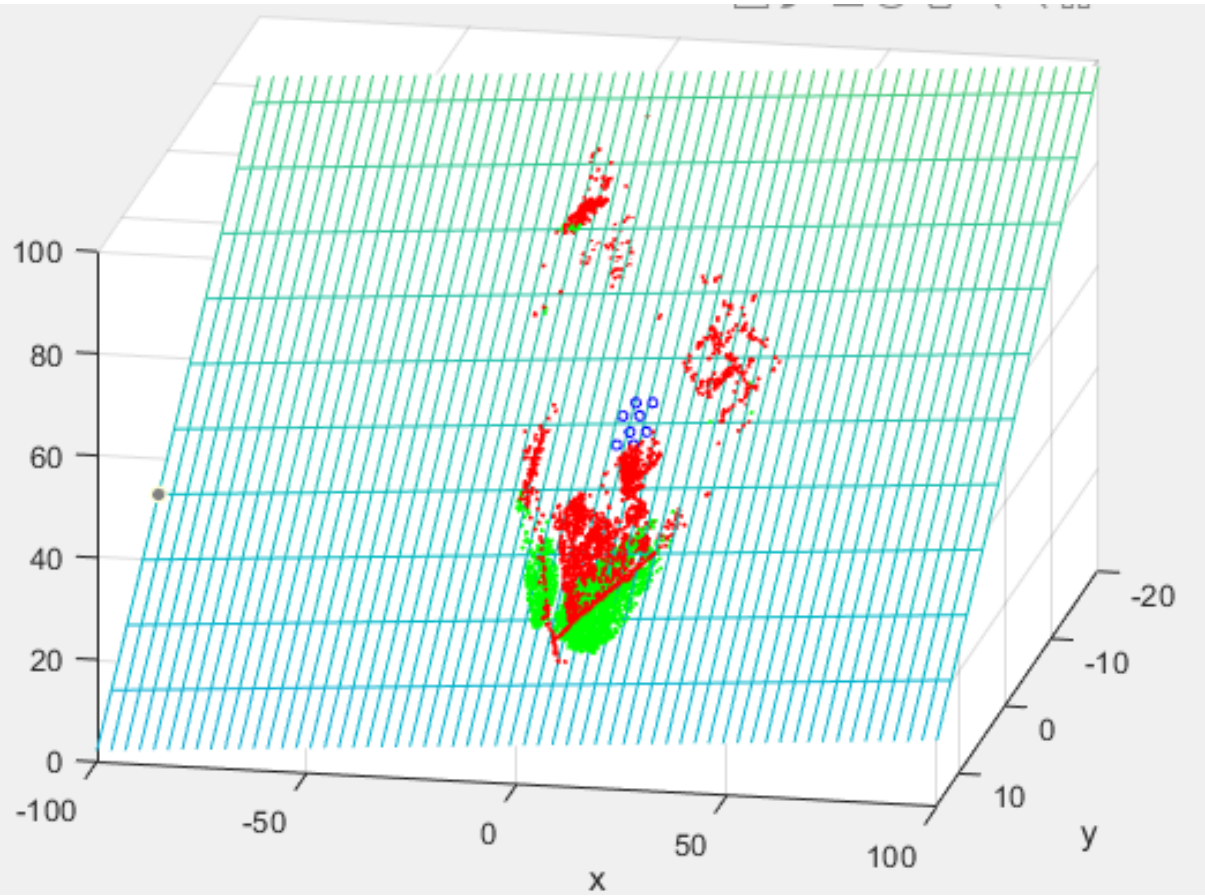

Figure 9: View 2 of the transformed points

5.8 Step 8

In this step, we read in the cameras.txt file with the intrinsic parameters, as well as the images.txt file, which contains the extrinsic parameters.

The cameras file contains many parameters, however the location of the principle point in the x and y-direction and focal length (in pixels) are the only parameters from this file used. The cameras.txt file can be seen in Figure 10. There is only one entry since we selected "same for all images" in COLMAP. Here, the focal length is 3257.93, and the principle point is located at (2016, 1512).

The images file also contains many parameters, but the only ones of interest are those that describe the camera pose; the rotation and translation parameters. The rotation is given as a unit quaternion. To convert this into a 3x3 rotation matrix, we used the Matlab function *quat2rotm*. Since each camera has a different pose, there is an entry in this file for each camera. A portion of this txt file can be viewed in Figure 11.

These parts of the code are under the comments "Read in the camera and image parameters".

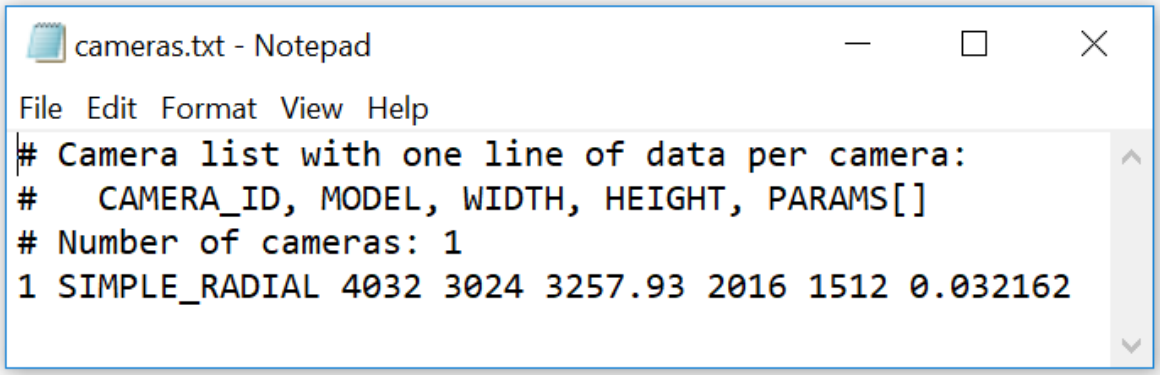

Figure 10: The cameras.txt file

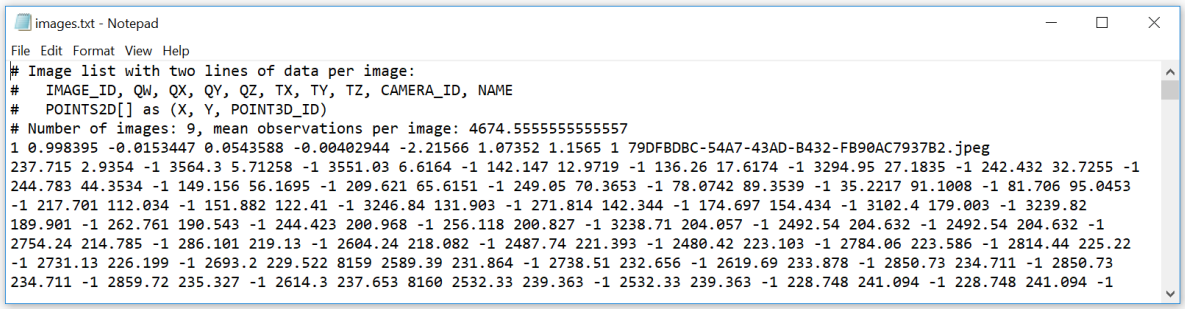

Figure 11: The images.txt file

5.9 Step 9

We write a function named CameraProjection3Dto2D in the augmentedRealityViewer.m to realize this part. The inputs of this function are the rotation matrix, translation vector, the principle point, focal length, and transformed box points. The transformed box points are the result of step 7, while the other parameters are passed in from step 8. This function projects a point in the world to a pixel location in an image according to the following

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{s_x} & 0 & O_x \\ 0 & \frac{1}{s_y} & O_y \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{31} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} U \\ V \\ W \\ 1 \end{bmatrix}$$

where u, v are pixel coordinates that represent the world point (U,V,W). The pixels are assumed to be square, so the scale factors s_x and s_y are set to 1. Principle point, (O_x, O_y), and focal length, f, are given by the camera file. The rotation matrix, $r_{11}...r_{33}$, and translation vector $[c_x...c_z]$ are given by the image file. Once we have these pixel locations, we can use them to display the box. This is shown in step 10.

5.10 Step 10

We placed the virtual box in the scene as shown below in Figure 12. When we ran the code and viewed the results for each image, we noticed that there was an error with one image, as seen in Figure 13. We ran the code three different times and the results can be seen in figures 14 through 16. As you can see, most results appear like Figure 12 which shows the regular box, while there is very few possibilities that the error in Figure 13 will occur. This error is just from the order in which the points are drawn. We draw planes in the following order starting with the plane with the most depth. We assume that this back plane is comprised of the two points with the largest depth (we assume smallest row value) and the two points with the second smallest depth. Then we draw the bottom plane using the four points with the largest row value. The sides, followed by the top of the box are drawn next. And finally the front face is drawn using the points with the two largest and second smallest row values. If we change the order of the points, the error will disappear.

This part of the code is under the comment "Place object in images".

Figure 12: Virtual box placed in our scene

We ran three times using this code, which is shown in Figure 14, Figure 15 and Figure 16. There is only 1 errors in the all 27 pictures and was not reproducible the next time we ran the code three times, but thought it was worth mentioning.

Figure 13: Virtual box placed in our scene with an incorrect ordering of points

Figure 14: Results from the first run through the code

Figure 15: Results from the second run through the code

Figure 16: Results from the third run through the code

6 Conclusion

With this project, we have finished all the steps in the description. Based on this, we have a fully understanding for the 3D vitual reality topic in computer vision area.

In the future, based on this basic idea, we may also achieve virtual reality for some moving objects. And we can also try to combine this with the term project 1, which is about the real-time tracker with SiameseFC-KalmanFilter-CorrelationFilter. Then we may be able to realize the virtual reality of some object moving in a scene and perform modeling.

References

[1] J. L. Schönberger and J.-M. Frahm, "Structure-from-motion revisited," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[2] J. L. Schönberger, E. Zheng, M. Pollefeys, and J.-M. Frahm, "Pixelwise view selection for unstructured multi-view stereo," in *European Conference on Computer Vision (ECCV)*, 2016.