The Pennsylvania State University
University Park
Spring 2020

CSE 586: Computer Vision

Project II

MURALI NANDAN NAGARAPU (mzn5322)
SAI AJAY MODUKURI (svm6277)
CHANDU DASARI (cbd5416)
CHANDAN AKITI (cra5302)

05-04-2020

Contents

1. Step 0: Steps to Reproduce

The project is implemented in python3.6 and some 3d visualizations are done using MATLAB support. Please follow the following steps to accurately recreate the results.

1. Base images reside in the folder named- **/images**
2. Running COLMAP produces 3 files named- **images.txt, cameras.txt, point3d.txt**
3. Run the python script in file- **Extract 3D Points.ipynb** to read the **point3d.txt** and generate files within folder **COLMAP 3D Points**-
   (a) 3DPointCloud.txt
   (b) 3DPointCloud.mat
4. Now, running **ransac_fitplane.py** to fit a plane to dominant plane and generate the folder **RANSAC Outputs** for files-
   (a) planeFitPoints.txt - All inlier points
   (b) planeFitPoints.mat
   (c) planeFitCoeff.txt - Coefficients of the plane (a,b,c,d)
   (d) planeFitCoeff.mat
5. Once, the Inlier points are extracted, 3D visualizations and transformations happen in **start.m**. This file takes in required files and first visualizes the 3D point cloud and the inlier points. Thereafter transforms the inlier plane to XY plane. And finally, forms a 3D cube on XY plane and projects it back to the scene coordinates. The resulting cube coordinates are stored in folder named- **3D Box** in the file- **sceneBox3D.mat**
6. Next, **Extract Camera Extrinsic.ipynb** takes in the **imges.txt**, extracts the extrinsic parameters and stores the results in folder named- **Camera Parameters** in the file-
   (a) cameraExtrinsic.txt - Pose parameters for all images
   (b) cameraExtrinsicPoints.txt - (u, v) pixel locations used for reconstruction for every image
7. Similarly, **Extract Camera Intrinsic.ipynb** takes in the **cameras.txt**, extracts the intrinsic parameters and stores the result in the same above folder with filename- **cameraIntrinsic.txt**.
8. Finally, **Augmented Reality.ipynb** aggregated all results to plot-
   (a) some sample base images
   (b) images with (u, v) pixels highlighted used for reconstruction
   (c) images with inliers highlighted
   (d) images with a wireframe of 3D cube placed in scene
   (e) images with 3D solid figure placed in the scene.

   The results of the final 3D cube placement are stored in the folder-**Final images**
9. **We have 42 images, of which very few are shown in report to keep it precise. All the images are also uploaded with the project. Kindly refer that folder for all images.**

2. Step 1: Gathering Images

The project starts with gathering images of a scene from a mobile. Idea is to keep the intrinsic features uniform across all the images. We have taken the pictures of a well lit bedroom scene having dominant plane as a wall. The left side of the scene has many posters and right side is relatively less occupied with series of glow lights (*to give a contrasting pixel intensities in images*). Table 1.1 gives overall idea of the camera and image metadata for this project.

Table 2.1: Image and Camera specifications

Specification	Value
Number of images	42
Image Resolution	4032×3024
Image Bit Depth	24
Camera	iPhone 11 Pro Max
Flash	None

Now, the following figures show a sample of the images captured.

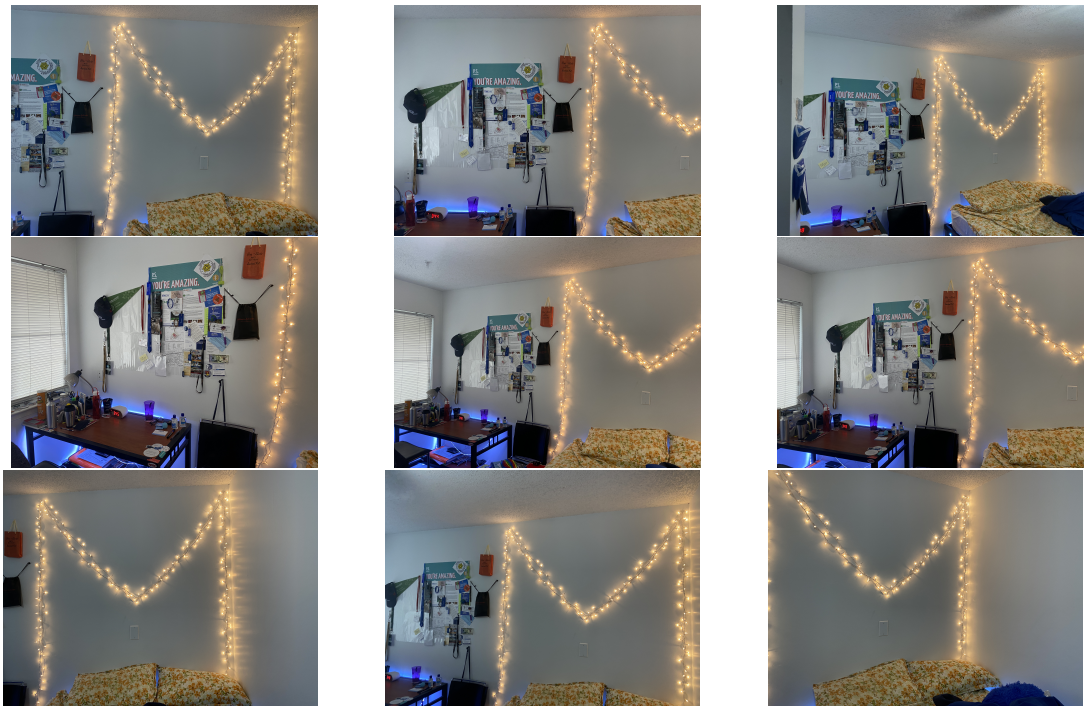


Figure 2.1: Samples from our image captures

3. Step 2: COLMAP

Ran COLMAP with basic configuration with a GPU. Since the images were captured using a single camera, the checkbox was selected to share the camera intrinsic parameters. For the input of 42 images, COLMAP produced the 3D point cloud after sparse reconstruction phase. Figure 3.1 shows the point cloud generated.

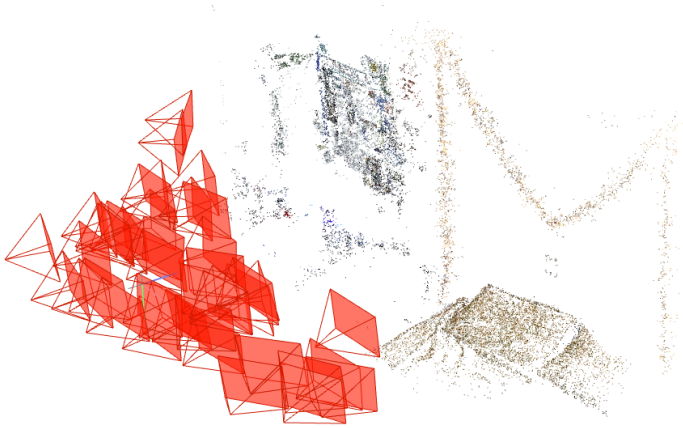


Figure 3.1: COLMAP generated 3D sparse reconstruction

4. Step 3: Reading 3D Point Cloud

COLMAP generates 3 files among which the ***points3D.txt*** has the information of the 3D point cloud generated. We read this file to extract all the 3D points and store it in the following format-

$$\text{The 3D points, } P_{n\times 4} = \begin{bmatrix} Id_1 & U_1 & V_1 & W_1 \\ Id_2 & U_2 & V_2 & W_2 \\ \vdots & \vdots & \ddots & \vdots \\ Id_n & U_n & V_n & W_n \end{bmatrix}$$

Where, Id_i is the point ID assigned by COLMAP and U, V, W are the 3D world coordinates of the points. The following code snippet in Figure 4.1 extract these 3D points and stores them for further operations. And

```
'''
READ THE COLMAP GENERATED POINTS FROM "points3D.txt" AND EXTRACT THE [X,Y,Z] COORDINATES FOR ALL POINTS
'''

with open(filename, "r") as fp:
    allLines = fp.readlines()
    allLines = allLines[3:]
    allPoints = []
    for line in allLines:
        indElements  = line.split(' ')
        POINT3D_ID = int(indElements[0])
        X,Y,Z = float(indElements[1]),float(indElements[2]),float(indElements[3])
        allPoints.append([POINT3D_ID,X,Y,Z])
```

```
with open(os.path.join(outputDir,output_filename), 'wb') as fp:
    pickle.dump(allPoints, fp)
```

```
allPoints = np.asarray(allPoints)
savemat(os.path.join(outputDir,output_filename_mat), {"points":allPoints})
```

Figure 4.1: Extracting 3D point clouds from output of COLMAP

following, Figure 4.2 visualizes the point cloud generated in 3D. We can clearly see the scene objects - *Pillows (blue), Posters (green), Lights (brown), Table(yellow)* etc.

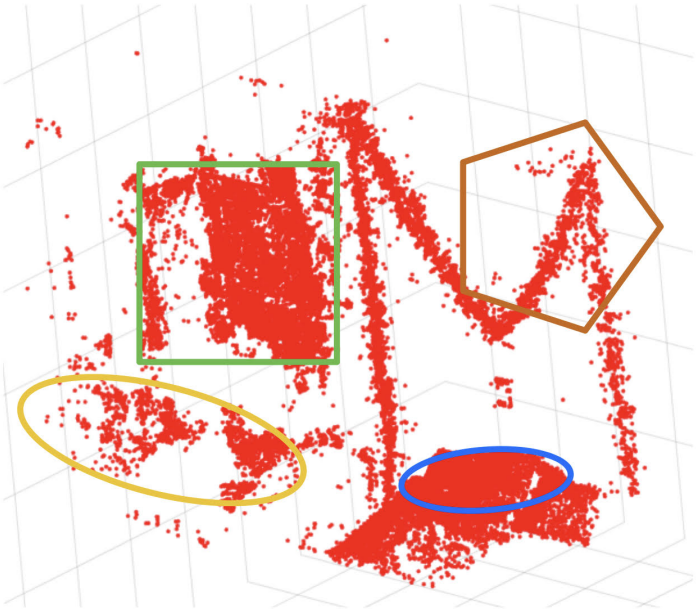


Figure 4.2: Point clouds from the output of COLMAP

5. Step 4: RANSAC

The goal of this step is to identify the dominant plane in this scene. We leverage the 3D points extracted from the Step 3. **By intuition, we know it has to be the wall.** To obtain this, we utilize the RANSAC algorithm to iteratively find this plane. First the local variables are set as follows-

- Total iterations (N): 5000
- Threshold for closeness of inliers: 0.09
- Required ratio of inliers: 0.8
- Number of Points in Minimal set: 3
- Estimation to fit a plane: SVD

With these parameters defined globally, we begin the RANSAC algorithm. Code snippets in the Figure 5.1 shows the main program. Once, the RANSAC algorithm terminates, we obtain the coefficients of the plane equation that was selected by **consensus**. The points thus obtained are stored for further operations.

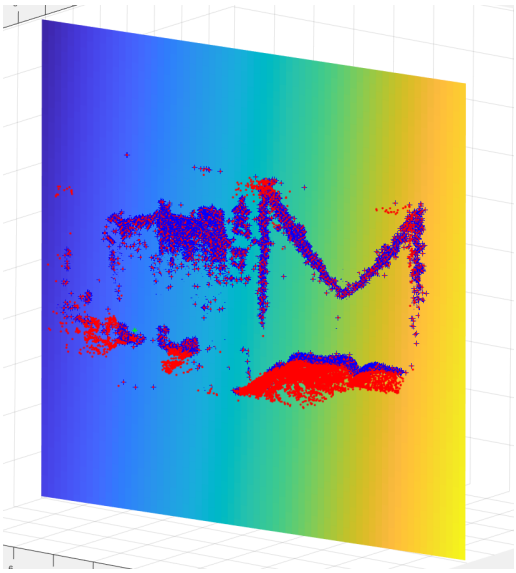
```
def ransac(data, estimate, is_inlier, sample_size, goal_inliers, max_iterations, stop_at_goal=True, random_seed=None):
    soFarBestIC = 0
    soFarBestPlane = None
    random.seed(random_seed)
    soFarBestSetPoints = []
    data = list(data)
    for i in range(max_iterations):
        randomSample = random.sample(data, int(sample_size))
        currPlane = estimate(randomSample)
        currIC = 0

        currSetPoints = []
        for j in range(len(data)):
            if is_inlier(currPlane, data[j]):
                currIC += 1
                currSetPoints.append(data[j])

        print('estimate:', currPlane)
        print('# inliers:', currIC)

        if currIC > soFarBestIC:
            soFarBestIC = currIC
            soFarBestPlane = currPlane
            soFarBestSetPoints = currSetPoints
            if currIC > goal_inliers and stop_at_goal:
                break
    print('took iterations:', i+1, 'best model:', soFarBestPlane, 'explains:', soFarBestIC)
    return soFarBestPlane, soFarBestIC, soFarBestSetPoints
```

Figure 5.1: Implementation of RANSAC Algorithm

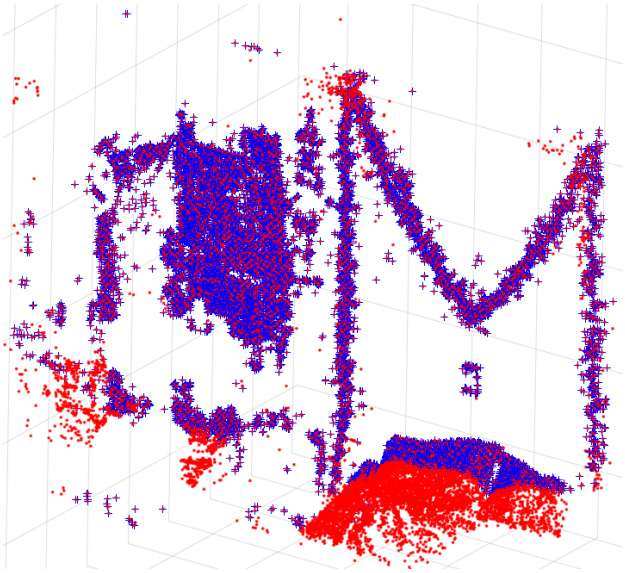
After obtaining the plane equation, we yet ran RANSAC another time with a lower threshold this time to consolidate the coefficients.

6. Step 5: Visualizing the Inliers and Dominant Plane

We visualize the plane selected by the RANSAC algorithm consensus. Figure 6.1a shows the plane obtained form the RANSAC step. The plane is **aligned with the wall** of the scene as expected. Figure 6.1b shows the distinction of inlier (blue) with outliers (red). We see that the points closer to the plane are taken as **inliers**.



(a) Visualizing the plane obtained from RANSAC step



(b) Inlier(blue) - Outlier(red) distinction

7. Step 6: Dominant Plane transformation

We now have the 3D scene coordinates of the inlier points. We will define our coordinate system so that the future Augmented Reality math becomes easier to perform and visualize. Consequently, we will now transform our dominant plane as follows-

1. Shift the plane so as to make it pass through origin - (*Translation*). This is achieved by subtracting the z-intercept from all the points. Consider a plane equation give by-

$$a'x + b'y + c'z + d' = 0 \tag{7.1}$$

   The z-intercept is given by putting $x = 0, y = 0$ which yields $z_0 = -\frac{d'}{c'}$. Thus, we subtract this quantity from all the z coordinates of the points to shift the plane as to pass through origin.

2. Shift the plane so as to bring the origin to the center of inlier points (*Translation*) Considering the new plane equation after previous step given by-

$$ax + by + cz = 0 \tag{7.2}$$

   And the mid point of the inliers lie nearly at-

$$\begin{aligned} x_{mid} &= \min_{x_i \in X} x_i + \frac{\max_{x_i \in X} x_i - \min_{x_i \in X} x_i}{2} \\ y_{mid} &= \min_{y_i \in Y} y_i + \frac{\max_{y_i \in Y} y_i - \min_{y_i \in Y} y_i}{2} \end{aligned}$$

   where X is the set of all x-coordinates and Y is the set of all y-coordinates. So, each inlier point coordinate undergoes the following transformation-

$$\begin{aligned} x &= x - x_{mid} \\ y &= y - y_{mid} \\ z &= z \end{aligned} \tag{7.3}$$

3. Rotate the plane so as to coincide with the XY plane (*Rotation*). Some 3D geometry math is involved here to perform the rotation-

   - The unit vector in the direction of normal vector of our inlier plane is given by-

$$\hat{v} = \left(\frac{a}{\|a, b, c\|}, \frac{b}{\|a, b, c\|}, \frac{c}{\|a, b, c\|} \right)^T \tag{7.4}$$

     The unit vector in the direction of the normal vector of XY plane is given by-

$$\hat{k} = (0, 0, 1)^T \tag{7.5}$$

   - Thus, the angle between the planes can be computed by *dot product* given by-

$$\begin{aligned} \cos\theta &= \frac{\hat{v} \cdot \hat{k}}{\|v\| \, \|k\|} \\ \sin\theta &= 1 - \cos^2\theta \end{aligned} \tag{7.6}$$

     We have found the angle of rotation, we now find the axis of rotation by simply finding the unit vector in the direction of the plane intersection as follows-

$$\begin{aligned} \hat{u} &= \hat{v} \times \hat{k} \\ &= [u_1, u_2, 0]^T \end{aligned} \tag{7.7}$$

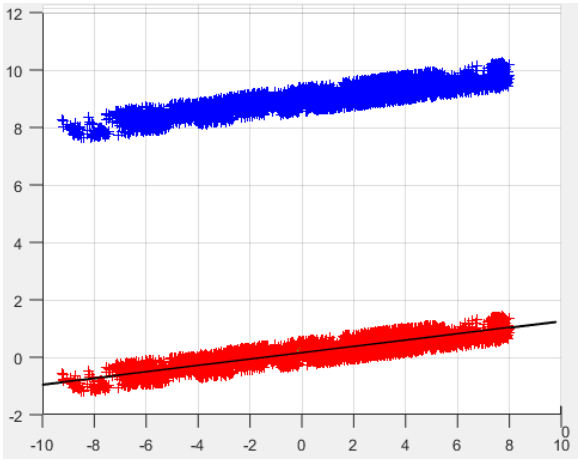
- With the above values, we now define our rotation matrix as follows-

$$M = \begin{bmatrix} \cos\theta + u_1u_2(1-\cos\theta) & u_1u_2(1-\cos\theta) & u_2\sin\theta \\ u_1u_2(1-\cos\theta) & \cos\theta + u_2u_2(1-\cos\theta) & -u_1\sin\theta \\ -u_2\sin\theta & u_1\sin\theta & \cos\theta \end{bmatrix} \tag{7.8}$$

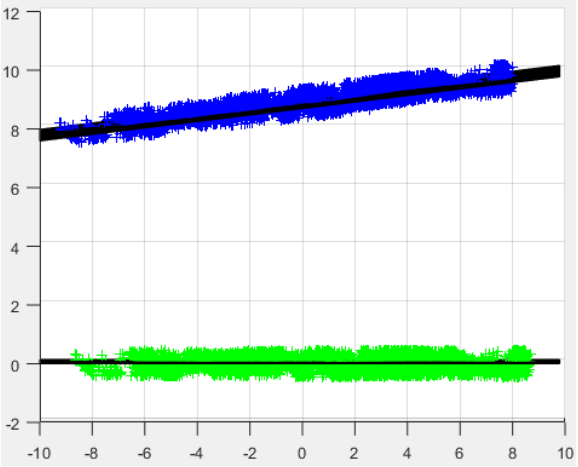
Finally, each of the inlier point is projected (*to be specific rotated*) as follows-

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = M * \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{7.9}$$

We implement a MATLAB routine *transformCoordinates.m* that computes and performs all the above operations. The following figures show the plane of inlier points (*side angle*) to show the transformations-



(a) Translation of the original inliers plane (**Blue**) to pass through origin (**Red**)



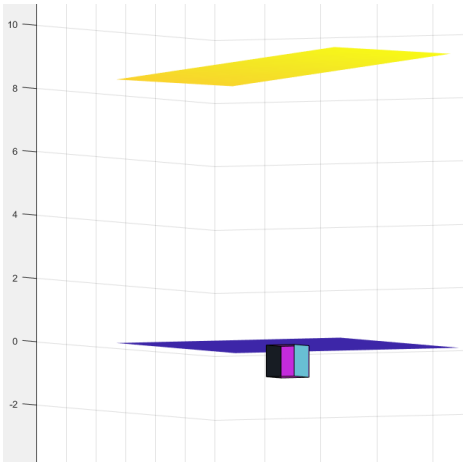
(b) Rotation of the translated inlier points to obtain the final transformation

Figure 7.1: Figure showing the plane of inlier points

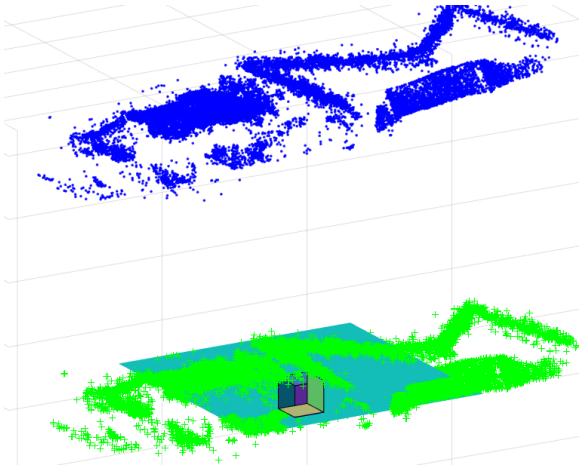
The inlier points will not be completely lie in the XY plane as the dominant plane does not accurately fit all the points.

8. Step 7: 3D Object Creation

Now we have all the inlier points in a coordinate system that we can easily visualize and comprehend. Placing a 3D object (*box*) on this coordinate system is easy as we know the orientation of the XYZ axis. Figure 8.1 shows the 3D cube placed on the XY place with origin lying at the center of one of its faces.



(a) 3D box placed on the XY plane (**Purple**)



(b) 3D box as seen along with the inliers in the XY plane

Figure 8.1: Visualizing the 3D box in XY plane

The vertex coordinates of this 3D box (Cube of size 1) thus can be simple be as follows-

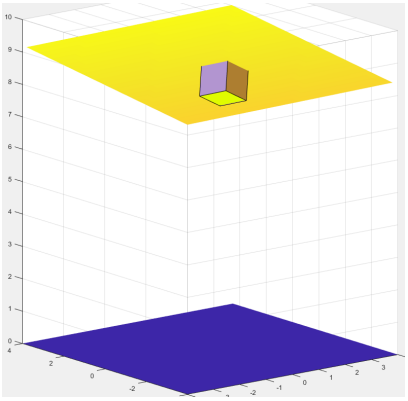
$$
\begin{aligned}
V_1 &= (-0.5, -0.5, 0) \\
V_2 &= (0.5, -0.5, 0) \\
V_3 &= (0.5, 0.5, 0) \\
V_4 &= (-0.5, 0.5, 0.5) \\
V_5 &= (-0.5, -0.5, 1) \\
V_6 &= (0.5, -0.5, 1) \\
V_7 &= (0.5, 0.5, 1) \\
V_8 &= (-0.5, 0.5, 1)
\end{aligned}
$$

Now, we want the 3D box coordinates in the original scene coordinates. So, we transform back the points to the original inlier plane. We perform the operations in reverse order that we performed in the previous step i.e.-

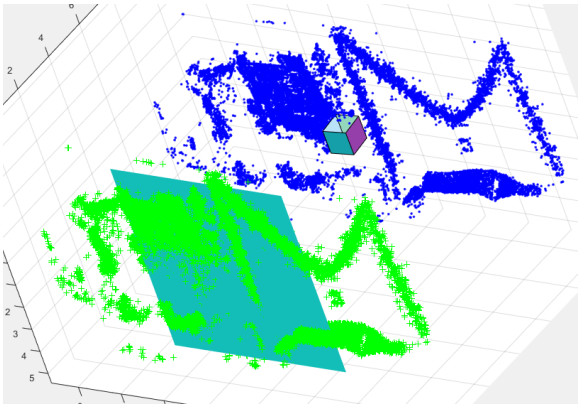
$$
\begin{bmatrix} x \\ y \\ z \end{bmatrix} = M^T * \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} - \begin{bmatrix} x_{mid} \\ y_{mid} \\ z_0 \end{bmatrix} \tag{8.1}
$$

A property that was utilized was that M is a rotation matrix and is **Orthonormal** Thus, $M^{-1} = M^T$.

Finally, the following 2 figures visualize the 3D cube in the original dominant inlier plane.



(a) 3D box placed on the Original inlier plane (**Yellow**)



(b) 3D box as seen along with the original inliers (**Blue**)

Figure 8.2: Visualizing the 3D box in original plane

We have now placed our 3D Augmented Object in the scene.

9. Step 8: Extrinsic and Intrinsic Camera Features

COLMAP also produces the Camera parameters (*Intrinsic and Extrinsic*). We will extract them from **cameras.txt** and **images.txt** respectively. We shall see how these files are formatted and the required information is extracted.

- The Intrinsic Features in the *.txt* file is stored in the following format- *CAMERA_ID, MODEL, WIDTH, HEIGHT, PARAMS* where furthermore, *PARAMS* gives us *PRINCIPLE X* (x_o), *PRINCIPLE Y* (y_o), *FOCAL LENGTH* (ϕ_x, ϕ_y), *SKEW* (γ).
  Here the symbols $x_o, y_o, \phi_x, \phi_y, \gamma$ are the camera intrinsic parameters shown in matrix $\mathbf{K}$ and represented as follows-

$$\mathbf{K} = \begin{bmatrix} \phi_x & \gamma & x_o \\ 0 & \phi_y & y_o \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 3079.75 & 0.0104 & 2016. \\ 0. & 3079.75 & 1512. \\ 0 & 0 & 1 \end{bmatrix} \tag{9.1}$$

- In a much similar way, we write another script to extract the extrinsic parameters (*Pose*) of the camera for individual images. *images.txt* provides for each image, a set of values as- *IMAGE_ID, QW, QX, QY, QZ, TX, TY, TZ, CAMERA_ID, NAME* corresponding to the pose along with *X, Y, POINT3D_ID* representing the (u, v) image pixel locations for points that are utilized for reconstruction process.

With the pose values given to us, we construct the transformation matrix **[R|t]** where **R** is the ro-

```python
with open(inputImagefilename, "r") as fp:
    allLines = fp.readlines()
    allLines = allLines[4:]
    flag     = 0
    image    = {}
    imagePoints = {}
    for line in allLines:
        if flag==0:
            line  = line.split('\n')[0]
            items = line.split(' ')
            IMAGE_ID = int(items[0])
            QW    = float(items[1])
            QX    = float(items[2])
            QY    = float(items[3])
            QZ    = float(items[4])
            TX    = float(items[5])
            TY    = float(items[6])
            TZ    = float(items[7])
            CAMERA_ID = [int(items[8])]
            NAME  = items[9]

            rotationMat = qt.as_rotation_matrix(np.quaternion(QW,QX,QY,QZ))
            translationMat = np.asarray([TX,TY,TZ]).reshape(3,1)
            transformationMat = np.hstack((rotationMat,translationMat))
            image[NAME] = transformationMat

            flag  = 1
        else:
            allthings = line.split(' ')
            points = []
            index = 0
            while index < (len(allthings)-2):
                px = float(allthings[index])
                py = float(allthings[index+1])
                cons = int(allthings[index+2])
                points.append([px,py,cons])
                index = index+3
            imagePoints[NAME] = points
            flag = 0
```

```python
with open(os.path.join(outputCameraFeaturesDir,outputExtrinsicFeatures), 'wb') as fp:
    pickle.dump(image, fp)
with open(os.path.join(outputCameraFeaturesDir,outputExtrinsicPoints),'wb') as fp:
    pickle.dump(imagePoints,fp)
```

Figure 9.1: Code snippet for reading the Extrinsic parameters and pixel locations

tation matrix and **t** is the translation matrix. Rotation matrix can be found by the quarternion given by $[QW,\ QX,\ QY,\ QZ]$ and Translation matrix by $[TX,\ TY,\ TZ]$. These matrices when extracted for each image take by a form as follows-

$$[\mathbf{R}|\mathbf{t}] = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \tag{9.2}$$

Figure 9.1 shows a code snippet of how we achieve this goal of extracting Extrinsic parameters and pixel locations for reconstruction for every image.

For more surety, figure 9.2 highlights the pixel locations utilized for reconstruction by COLMAP for that particular image.

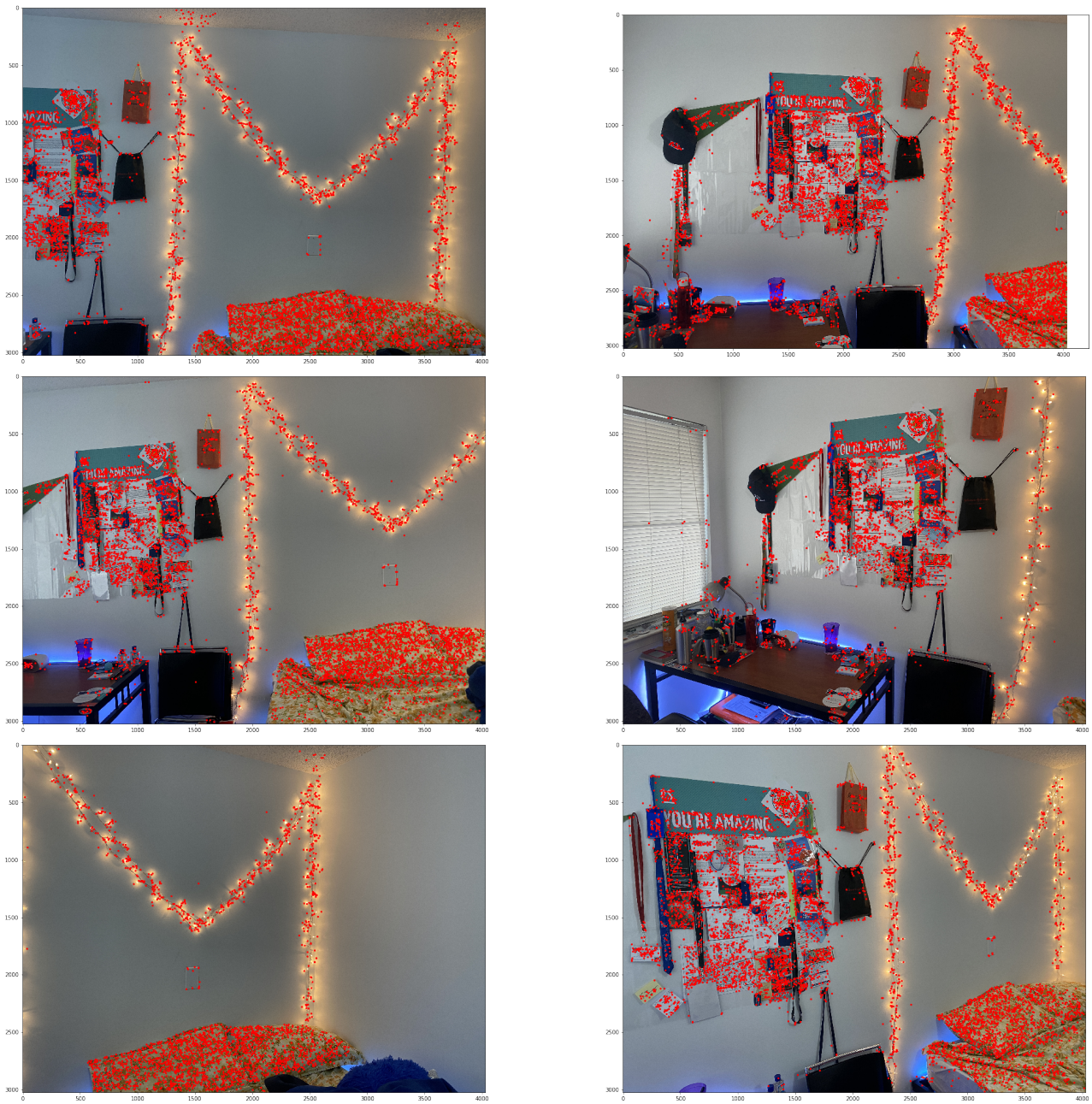


Figure 9.2: Pixel locations utilized for reconstruction by COLMAP

10. Step 9: Convert 3D to 2D

Next step is to convert the given world coordinates to camera coordinates using the intrinsic and pose parameters. This is achieved by the following equation-

$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}_{3\times1} = \begin{bmatrix} \phi_x & \gamma & x_o \\ 0 & \phi_y & y_o \\ 0 & 0 & 1 \end{bmatrix}_{3\times3} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix}_{3\times4} \begin{bmatrix} U \\ V \\ W \\ 1 \end{bmatrix}_{4\times1} \tag{10.1}$$

Where λ is an arbitrary constant. So, given $[U, V, W]$ we can find the corresponding camera pixel locations using the above equation. The following code snippet performs this operation.

```
def convert2D(point3D, K,Rt):
    trans2Dpoint =np.matmul(np.matmul(K,Rt),point3D)
    factor = trans2Dpoint[2][0]
    point2D = np.asarray([trans2Dpoint[0][0]/factor,trans2Dpoint[1][0]/factor])
    return point2D
```

To verify the above function, we projected the inlier points generated by RANSAC back on to the image to see if the points are lying on the dominant plane. The following few figures visualize this verification step confirming that the function is computing accurate results.

Figure 10.1: Verifying the working of 3D to 2D transformation using inlier points

11. Step 10: Back-Projecting the 3D Object

The last step in this project is to draw the 3D augmented box on the image pixel values. We utilize the above written function to convert the scene coordinates of the 3D box to project them to the (u, v) pixel locations of the image. We did this for every image and placed a virtual object in the image. Initially, we tried drawing wire-frame (*Figure 11.1*) for this box and later we fill faces of this cube with filled colors to make it appear as a solid box (*Figure 11.2*).

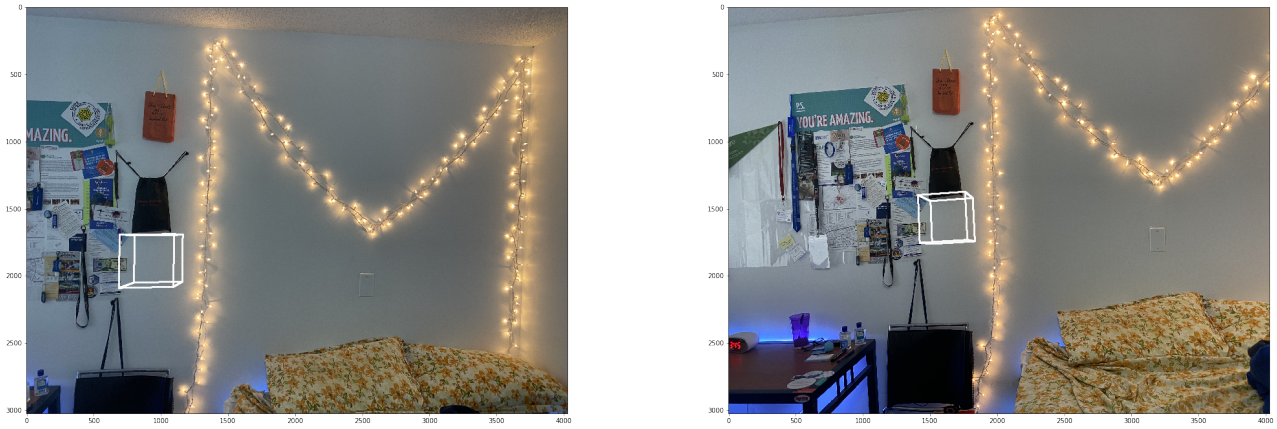


Figure 11.1: Wireframe of 3D cube placed on the wall

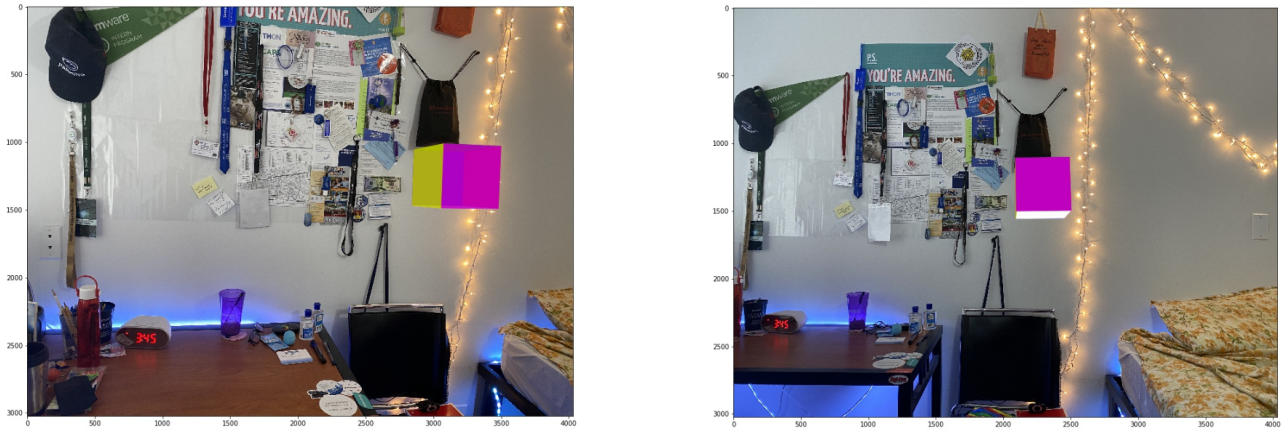


Figure 11.2

Thus, the final 3D cube is now placed on the dominant surface (*wall in our case*) where the cube pixels are overlaid on the image pixels. The colors of the box seem off due to the ordering in which the coloring was done defines the surface that shows in front. The code for drawing this 3D cube on top of every image is in *Augmented Reality.py*.