Augmented Reality Viewer

Li Yu[1], Shrey Nigam[2], Shivansh Rao[1], and Vikas Kumar[2]

[1]College of Information Science and Technology, Penn State University, PA, USA
{luy133, shivanshrao}@psu.edu
[2]Electrical Engineering and Computer Science Department, Penn State University, PA, USA
{sqn5300, vuk160}@psu.edu

Abstract

The goal of this project is to implement an augmented reality viewer that displays virtual objects overlaid on the images of a 3-Dimensional custom scene. The goal is achieved in 10 different steps, all of which are mentioned in this report. The main step of our approach is to recover the sparse point cloud from the multi-view images, for which we use the COLMAP open source implementation. Once the 3D point cloud of a real 3D scene is recovered, we place a virtual object on the dominant plane of the scene. The virtual object is then projected back to the original images to display how the virtual object overlays on the original images of a real 3D scene. The motivation of this work is to create a framework similar to the real-time Augmented Reality application available on iOS or ARCore on Android. (We use Python as the developing language throughout the project.)

1. Real Scene Images

For the first step we record a video of a living room and extract its frames as the input of the project. There are a total of 332 frames, with dimension 1920x1080. Figure 1 shows some frames of the video. The dominant planar surface is the ground plane (floor) along with also having the scene of a sofa and a chair for better visualization.

Figure 1. Sample frames from input video

2. Generating 3D sparse point cloud

After collecting the multi-view images in step-1, our next step is to generate a 3D point cloud corresponding to the input frames. A theoretical explanation is given in Section 13. This is a key step since the generated 3D point cloud is necessary to proceed to the next steps. For this we use COLMAP open source implementation [3]. For the scope of this project we generate a sparse point cloud instead of a dense point cloud (which requires GPU and more computation time). The following steps are used to generate the sparse point cloud:

- Install the COLMAP open source implementation [3] and input the multi-view images.
- Go to File, then select New Project and create a database.md file and select the images folder.
- After loading the images, proceed to the *"Processing"* option and select feature extraction by selecting SIMPLE_RADIAL as the camera model and click on the checkbox : shared for all images, followed by feature matching. See figures 2 for more details.
- Finally export the 3D sparse model as a set of text files including cameras.txt, points3D.txt, and images.txt.

The output of this step can be seen in Figure 3, from which we can notice the reconstructed scene (with the sofa in orange color and the carpet floor too). See Appendix 13 for details on how COLMAP works.

3. Reading point cloud

In the third step we read the 3D point coordinates from the points3D.txt file generated by COLMAP. These will be the world coordinates for us. It contains the information of all reconstructed 3D points in the dataset using one line per

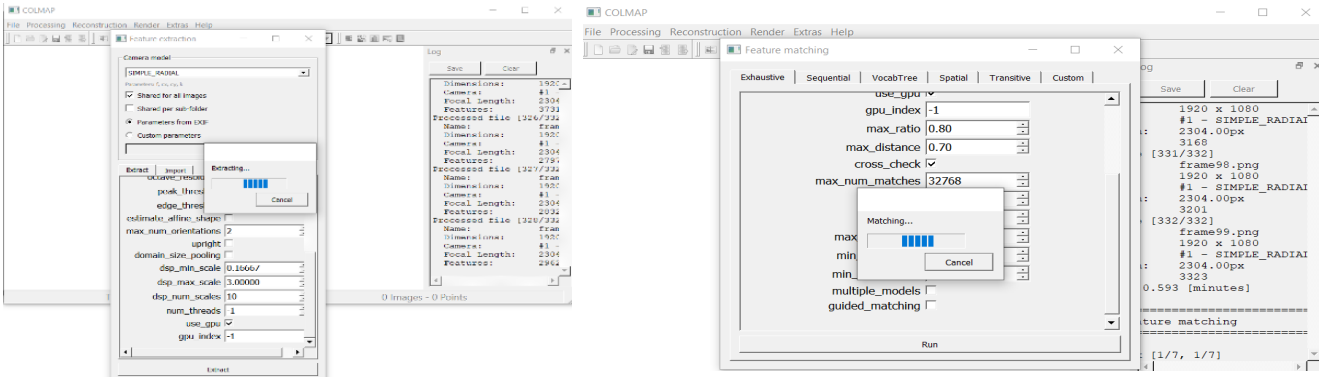

Figure 2. Figure taken from COLMAP GUI.

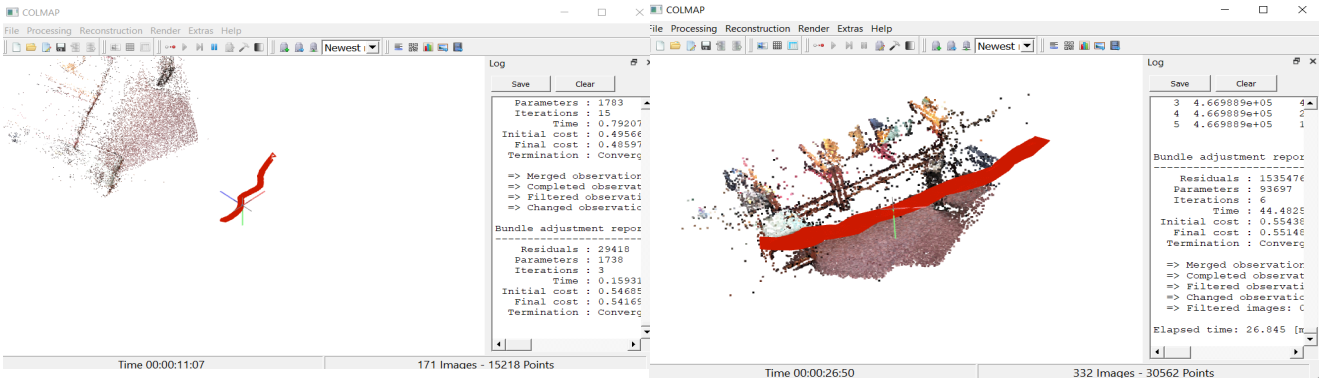

Figure 3. Figure displaying the incremental reconstruction of the input frames. The figure on left is after processing half of the frames, and the figure on the right is after the completion of the reconstruction.

point. More details can be found on the official COLMAP documentation [3].

4. Apply RANSAC [1]

In the fourth step we have to write a custom RANSAC routine based on the lecture notes. Before writing it, there are a few points which we intend to mention:

- Minimum number of 3D points needed to fit a 3D plane is 3, since the degree of freedom of a 3D plane is 3. Because of this, we randomly sample 3 points to fit a plane at each iteration.

- We formulate a 3D plane as: a * x + b * y + c * z + d = 0, where x, y, and z are the 3D axis. Fitting the plane is a problem of least square fitting of the four parameters (a, b, c, d) with given 3 set of 3D coordinates (X, Y, Z).

- We define the distance of a point to the 3D plane as the geometrical distance of the point to the plane:

$$dist = \frac{|a * x + b * y + c * z + d|}{\sqrt{a^2 + b^2 + c^2}}$$

  If parameters (a, b, c) are normalized, then the distance can be reduced to $dist = |a * x + b * y + c * z + d|$.

- A distance threshold of 0.1 is manually set to determine if a point is an "inlier" to the plane. To choose the proper threshold, we first experiment with a starter threshold and observe the resulted plane, gradually change the value until a plane with almost all inliers included.

- At each iteration, we keep track of indices of inliers to the plane and update the set of inliers with largest size.

- After manipulating the distance threshold, we find the proportion of outliers is roughly one third of total number of points. Combined with the fact that the sampling size s equals 3, we calculate the number of iterations based on the following equation and decide $N = 20$ is enough for yielding a good 3D plane.

$$N = \frac{log(1-p)}{log(1-(1-e)^s)}$$

The complete implementation of RANSAC algorithm is shown in Figure 4.

```python
def fit_plane(data):
    (rows, cols) = data.shape
    G = np.ones((rows, 3))
    G[:, 0] = data[:, 0]  #X
    G[:, 1] = data[:, 1]  #Y
    Z = data[:, 2]
    (a, b, c), resid, rank, s = np.linalg.lstsq(G, Z, rcond=None)
    x = np.array([a, b, -1, c])
    norm = np.linalg.norm([a, b, -1])
    x /= norm
    return x

def is_inlier(model, point, threshold_inlier):
    point_withone = np.array([*point, 1])
    dist = abs(np.sum(np.multiply(model, point_withone)))
    return dist < threshold_inlier

def get_inplane_point_idx(model, data, threshold_inlier):
    point_idx = []
    for i, point in enumerate(data):
        if is_inlier(model, point, threshold_inlier):
            point_idx.append(i)
    point_idx = np.array(point_idx)
    return point_idx

def ransac(data, sample_size, num_iters, threshold_inlier, num_points):
    """ RANSAC algorithm to find a 3d plane
    :param data: np array, Nx3
    :param sample_size: number of points at fitting
    :param num_iters: number of iterations
    :param threshold_inlier: threshold of point-to-plane distance
    :param num_points: minimum number of points a plane should include
    :return: best model, in the form (a, b, c, d) such that ax + by + cz + d = 0
             best count, number of points of the dominant plane
    """
    best_model = None
    best_count = num_points
    for i in range(num_iters):
        # randomly choose sample points from data
        sample_idx = np.random.permutation(np.arange(data.shape[0]))
        sample = data[sample_idx[:sample_size], :]
        sample_left = data[sample_idx[sample_size:], :]
        # fit a plane
        model = fit_plane(sample)

        # get number of points in plane
        point_idx = get_inplane_point_idx(model, sample_left, threshold_inlier)
        count = point_idx.shape[0]
        print('iter {}: model param {}, in plane point count {}'.format(i, model, count))

        # check if count > num_points
        if count > num_points:
            model = fit_plane(np.vstack([sample, sample_left[point_idx, :]]))
            if count > best_count:
                best_count = count
                best_model = model

    return best_model, best_count
```

Figure 4. RANSAC implementation.

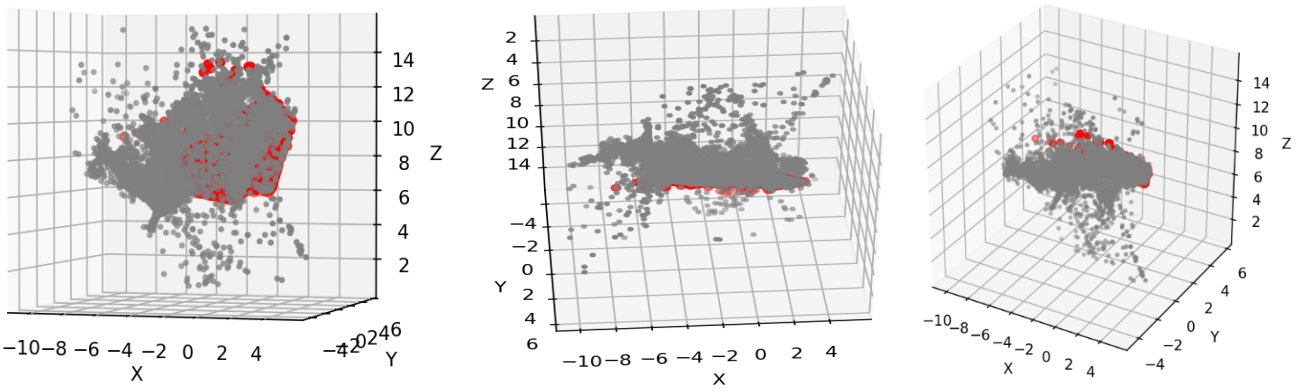

Figure 5. Inplane points are indicated in red colour and 3D points from COLMAP are shown in gray color.

5. Display Inplane points

In step 5 we will display both the 3D point cloud generated by COLMAP software [3] and the set of inlier points detected by RANSAC algorithm explained in step 4. If we look at figure 1 closely, we can observe that the dominant plane is the ground plane. RANSAC should ideally find the dominant plane, and set of inlier points should lie on the same plane. Figure 5 shows the inplane points (red color) from different views, along with the total 3D points (grey color). The number of total inplane points is 18950, whereas the number of outplane points is 11530. Inplane having a high proportion of points indicates that the RANSAC routine written by us is returning the correct plane as dominant plane. It is corresponding to the ground plane from our captured video. A better visual of the inplane points with dominant plane can be seen in Figure 7.

6. Local Coordinate System

In step 6, we have to form a local xyz coordinate system where the dominant plane found in step 4 becomes z=0 plane. In order to form the local coordinate system, we have to find the transformation between two 3D coordinate systems. This is fulfilled by realizing that we can instead find the transformation between two unit vectors in the two 3D coordinate systems such that one would match the other if we transform the coordinate system. A good choice of unit vector is the normal vector of the dominant plane, which we have already found in the scene X,Y,Z coordinates. The second unit vector will naturally be the normal of xy plane of local coordinates where we wish to place the plane.

In order to place (x=0,y=0) in the middle of the set of inlier points (dominant plane), we calculate the center of these points by averaging their coordinates. This center, which can also be viewed as the vector from (X=0,Y=0,Z=0) to the center of the dominant plane, will be the translation distance (negative here) of all the points in X,Y,Z coordinates before they are applied with rotation. This distance will be used again when we transform from local x,y,z coordinates to scene X,Y,Z coordinates in step 7.

To calculate the transformation matrix from the normal of dominant plane in X,Y,Z to the normal in x,y,z, we refer to the method mentioned in the following link. It computes the rotation matrix for two arbitrary vectors so that makes it easy for us to transform between X,Y,Z and x,y,z back and forth. Figure 7 displays the local xyz coordinate system containing the dominant plane in blue colour. Note that it is at z=0 which is xy plane. This is correct, since the dominant plane had to lie on xy plane of local coordinate system. The complete implementation of transformation (translation and rotation) is shown in Figure 6.

7. Create Virtual Object

As part of step 7 we first create a virtual box in local x,y,z coordinates. The coordinates of eight corners are set such that the box lies in the local z=0 plane and the center of the rectangular bottom surface is centered at (x=0,y=0,z=0). The eight coordinates are (1,1,0), (-1,1,0), (-1,-1,0), (1,-1,0), (1,1,1), (-1,1,1), (-1,-1,1), (1,-1,1). They form a box with size 2x2x1 in x,y,z dimensions.

We then transform the eight corners from local coordinate system x,y,z back to the scene X,Y,Z coordinates. The rotation matrix is calculated using the function `get_rotation_matrix` by rotating normal of local plane to normal of scene plane. After rotation, a translation is needed to place the box back to the dominant plane in X,Y,Z coordinates. The distance has been computed in step 6. Figure 8 shows the virtual object created in blue color in local coordinate system and also in scene coordinate system. It is clear that in local coordinate system, the dominant plane lies at z=0 (xy plane), whereas in the scene coordinates it not at the same location. This confirms that the new coordinate system is not local coordinate system, instead it is the oringinal scene coordinate system.

```python
def translate(data, dist):
    return data + dist

def get_vector_normal(model):
    """ Compute the vector normal of plane: ax + by + cz + d = 0
    :param model: a, b, c, d
    :return: normalized vector normal
    """
    normal = np.zeros(3)
    normal[:] = model[:3]
    norm = np.linalg.norm(normal)
    normal /= norm
    return normal

def get_rotation_matrix(a, b):
    """ Compute the rotation matrix from vector a to b
    :param a: normalized, (3,)
    :param b: normalized, (3,)
    :return: R, such that Ra = b
    """
    # find axis and angle using cross product and dot product
    v = np.cross(a, b)    # axis
    s = np.linalg.norm(v) # sine of angle
    c = np.dot(a, b)      # cosine of angle
    # compute skew-symmetric cross-product matrix of v
    v1, v2, v3 = v
    vx = np.array([[0, -v3, v2],
                   [v3, 0, -v1],
                   [-v2, v1, 0]])
    # compute rotation matrix
    R = np.identity(3) + vx + np.dot(vx, vx) * (1 / (1 + c))
    return R

def rotate_point(point, R):
    return np.dot(R, point)

def rotate(data, R):
    return np.apply_along_axis(rotate_point, 1, data, R)
```

Figure 6. Euclidean transformation implementation.

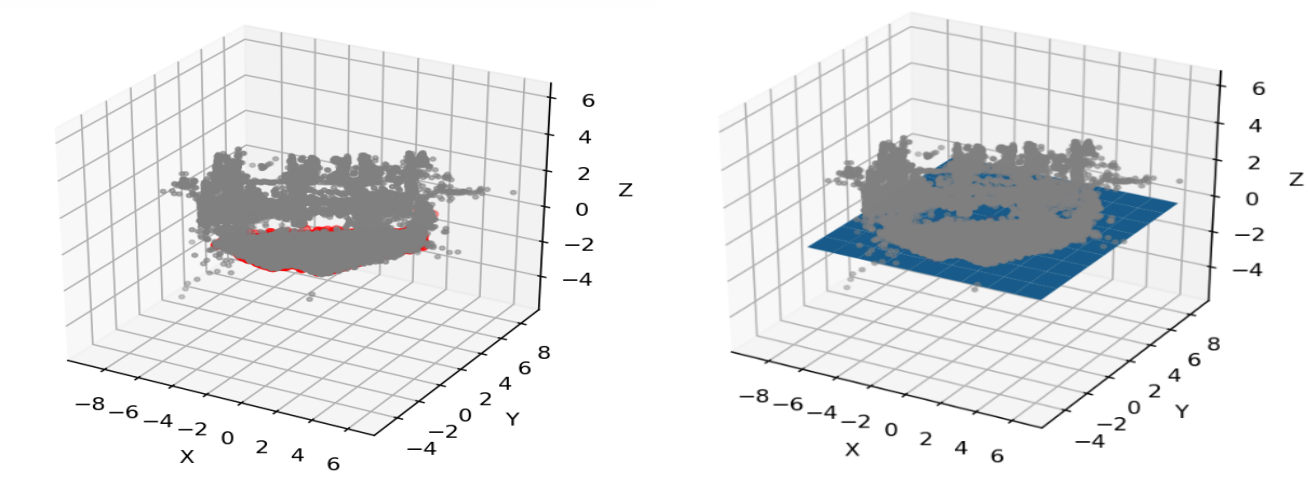

Figure 7. Local XYZ coordinate system with the dominant plane highlighted in blue, outlier points in gray, and inlier points in red.

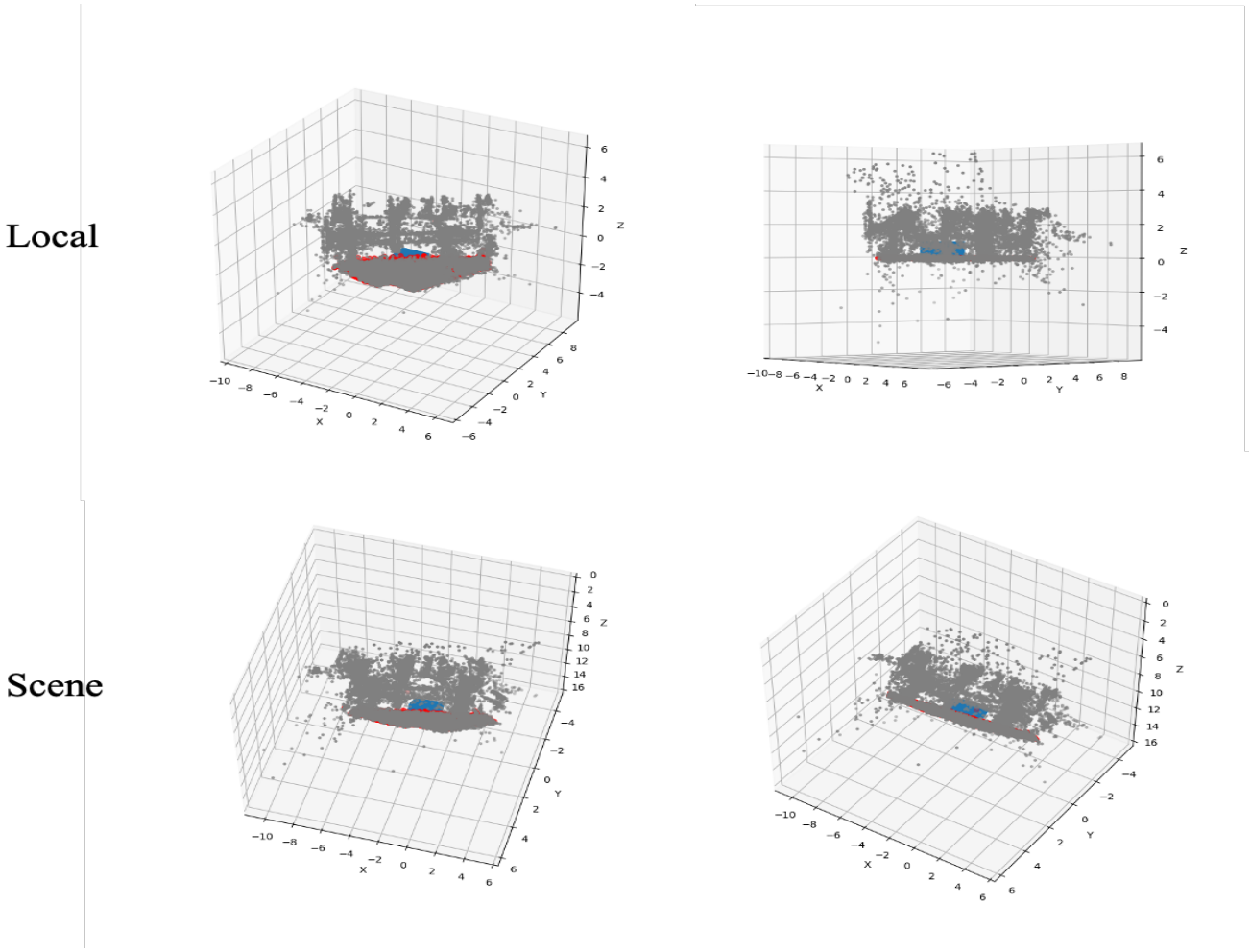

Figure 8. Blue box object in local (first row) and scene (second row) coordinate systems.

8. Read Intrinsic and Extrinsic Parameters

In step 8 we read the points from cameras.txt and images.txt file generated from the COLMAP after the reconstruction. Cameras.txt file contains the intrinsic parameters of all reconstructed cameras in the dataset using one line per camera, whereas the images.txt file contains the pose and keypoints of all reconstructed images in the dataset using two lines per image.

Specifically for our project, cameras.txt has only one set of parameters for a simple radial camera model. It contains the information WIDTH and HEIGHT which is the dimension of photos, and also the intrinsic parameters: f, cx, cy, and k. Parameter f is the focal length; cx and cy is the position of principle point which here is the center of film plane; k is the parameter of distortion of the camera.

Images.txt contains for each image the extrinsic parameters corresponding to camera poses. The information for rotation comes in the form of quaternion (QW, QX, QY, QZ), which will later be converted to rotation matrix (check function `qvec2rotmat` in `projection.py`, it is excerpted from the source code of COLMAP). The information for translation comes directly with three values TX,TY,TZ that we can readily use in the next step.

9. Forward projection

In step 9, we follow lecture notes on camera model and divide the forward projection into three stages: world to camera, camera to film, film to pixels. The detailed calculations are explained below. All the functions in this step can be found in the file `projection.py`, including the conversion from quaternion to rotation matrix.

World to camera conversion is the conversion from world coordinates to camera coordinates, which can be expressed as:

$$P_C = [R|t]P_W$$

where P_W should be homogeneous world coordinates with dimension 4x1 and P_C is the corresponding point in camera coordinates with dimension 3x1. Both R and t are

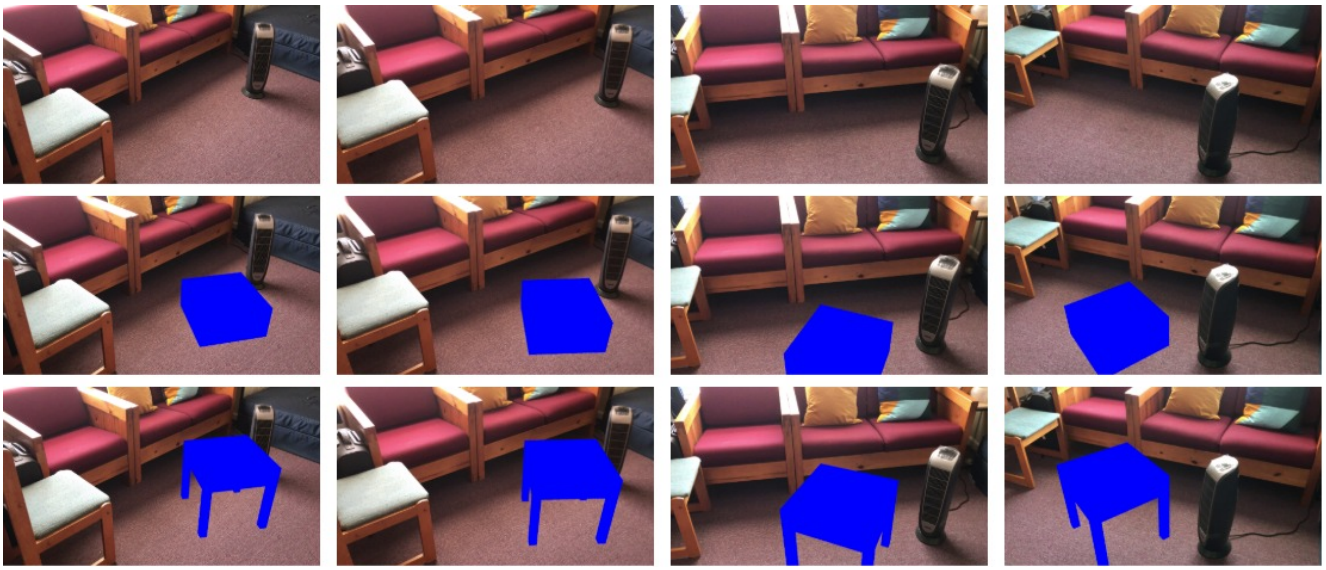

Figure 9. Output frames showing a simple blue box in second row, and a complex blue table in third row. The first row displays corresponding input frames.

retrieved from the information in images.txt.

Camera to film is the perspective projection from 3D camera coordinates to 2D film coordinates, which can be expressed as:

$$P_f = K_1 P_C$$

where P_C is the camera coordinates resulted from last procedure, P_f is the film coordinates with dimension 2x1, and K_1 represents perspective projection matrix with dimension 3x3. K_1 is a diagonal matrix with its main diagonal values to be (f,f,1). One thing worth of notice is the use of distortion parameter k read from `cameras.txt`. We have checked the source code of COLMAP to find the use of k and rewrite the forward projection code in python.

Film to pixels is simply the offset between film plane to pixel coordinates, which can be expressed as:

$$P_u = K_2 P_f$$

where P_u is the final 2D pixel coordinate, and K_2 is the offset matrix that adds (cx, cy) to P_f. Note that multiplying K_2 with K_1 will get the matrix K, as in the lecture notes.

10. Project box to original images

In step 10, which is the last step we follow the procedures in step 9, and project the corners of the 3D virtual box into the image and draw it overlaid over the original pixel values. To get a better visualization of the box in images, we also calculate the depths of each corners, with the function `calc_depth` in `projection.py`. We sort the faces of the box by the min depth of 4 corners forming each face, and draw the faces in reverse order, so that faces with minimum depth are placed in front. The output results are shown in Figure 9. We put a simple blue box as well as a complex blue table overlaid on the original images. It can be seen from Figure 9 that both the virtual objects are on the ground plane, which confirms to the correctness of our implementation since ground plane is the dominant plane in our scene.

11. Work Distribution

Li Yu started the project by capturing the real scene multi-view images and Shivansh Rao generated the point clouds corresponding to the images. Both Li and Shivansh started writing the RANSAC routines and code for local coordinate systems and transformation between scene coordinate system and local coordinate system. These codes were written in a modular fashion so that it could be used as it is in the future steps. Both their implementations were cross-checked and at the end the better implementation was used. Vikas Kumar helped in creating the virtual object (both box and complex table) and used previous implementation to transform local coordinates system back to scene coordinate system. Vikas also read the intrinsic and extrinsic parameters to convert initial world coordinates to pixel coordinates for the forward projection step. Finally, Shrey Nigam found the results of the frames overlaid with the virtual box and contributed in the report writing part, power point presentation, and preparing the contents in an organized manner. At the end, Shivansh Rao recorded the audio for the video to be submitted. Overall, all the members had roughly contributed equally to the project.

12. Conclusion

After performing all the 10 steps as mentioned in this report, we can conclude from the output shown in Figure 9 that the virtual object that we place is overlaid on the ground plane of the images. This shows that our model is functioning in a correct manner, since if we observe the video we captured in step 1, the dominant plane is indeed the ground plane. A few more things to note from this project is that COLMAP [3] indeed provides a highly efficient open source implementation for 3D scene reconstruction, however we only use it for sparse reconstruction and not dense reconstruction.

13. Appendix

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. Typically, Structure-from-Motion systems divide this process into three stages i.e. feature detection and extraction, feature matching & geometric verification and structure & motion reconstruction.

In the first step, feature detection/extraction finds sparse feature points in the image and describes their appearance using a numerical descriptor (SIFT [2] and its derivatives). COLMAP [3] imports images and performs feature detection/extraction in one step in order to only load images from disk once.

In the second step, feature matching and geometric verification finds correspondences between the feature points in different images. We use exhaustive matching which matches each image against every other image. The authors introduce scene graph augmentation (geometric verification strategy). The method uses homography inliners and inliners of essential matrix to distinguish between pure-rotation (panoramic) and planar scenes. Only non-panoramic and calibrated image pairs are used as seed for initialization process (third step). The output of this stage is a so-called scene graph with images as nodes and verified pairs of images as edges.

After producing the scene graph in the previous two steps, we can start the incremental reconstruction process. COLMAP first loads all extracted data from the database into memory and seeds the reconstruction from an initial image pair. This stage includes incremental reconstruction procedures such as image registration, triangulation (observing existing scene points), bundle adjustment (uncertainties pertaining to camera pose). The authors use next best view selection strategy to ensure that the rest of the reconstruction process is not affected by mis-registrations and faulty triangulations. Hence, the candidate images chosen for the next best view are the images with at least one triangulated point. To ensure uniform distribution, the candidate image is discretized into a grid with a fixed number of bins in each dimension (each grid has either empty or full state). The score (dependent on bin size) is accumulated over multiple bin sizes to find the next best view. Additionally, the authors proposed a robust triangulation method to deal with outlier contamination. Bundle Adjustment (BA) is performed after each step on most-connected images instead at a global level. Since BA can be a bottleneck, COLMAP partitions the scene into several camera groups, instead of clustering several cameras into one map.

References

[1] M. A. Fischler and R. C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981. 2

[2] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004. 8

[3] J. L. Schonberger and J.-M. Frahm. Structure-from-motion revisited. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4104–4113, 2016. 1, 2, 4, 8