Computer Vision II (EE 554/CSE 586) Project 2: Augmented Reality

August 7, 2020

Sweekar Sudhakara, Sai Raghav Reddy Keesara, Srikanth Banagere Manjunatha and Laxmaan Balaji

Abstract

Augmented Reality (AR) is a technology that superimposes a computer-generated image on a user's view of the real world, thus providing a composite view. The augmented reality can be broken down into; marker based, marker less, projection based and superimposition based augmented reality. Some of the applications of augmented reality include like visual art, industrial engineering, archaeology and entertainment. In this project, we build an augmented reality viewer which resides an artificial computer generated object onto real-world environment. The steps in the AR's implementation included: 1) 3D reconstruction using COLMAP for input images, 2) COLMAP output processing to find the the dominant plane of the scene using RANSAC algorithm, 3) 3D to 2D camera projection to reside an artificial 3D object onto the plane and further depict it onto the original images to show the AR output.

Contents

1 Introduction and Motivation

Augmented Reality (AR) is a technology that superimposes a computer-generated image on a user's view of the real world, thus providing a composite view enabling an enhanced view of artificial perspective with the real world. The augmented reality can be defined as a system that exhibits three fundamental properties: 1) Combination of real & virtual worlds, 2) Real-time interaction, and 3) Accurate 3D registration of virtual and real objects. The added information can be constructive, or destructive. This experience is seamlessly interwoven with the physical world such that it is perceived as an immersive aspect of the real environment. Some of the application of the augmented reality includes 1) Medical imaging i.e. recreating anatomical structures virtually to aid surgeries, 2) Advertising i.e. depict the items to be sold in a much cleaner manner to the customer, 3) Military i.e. the virtual environment can be created to avoid enemy striking, etc. Augmented reality, in general, helps to create an artificial environment to depict information more convincingly that was lacking originally in the real world scene.

Main Idea:

In this project, we build an augmented reality viewer which resides an artificial computer generated object onto real-world environment.

2 Related Work

Schonberger et al. [1] introduced Structure from Motion (SfM) algorithm for 3D reconstructed of a scene from un-ordered images considering different viewpoints. There are several algorithms that exist as a part of the SfM algorithm, which mainly includes incremental SfM Algorithm. The incremental SfM algorithm can be defined as a pipeline that includes image correspondence component and an iterative reconstruction component in a sequential pattern. The pipeline works in two different stages. The first stage starts by extracting features from the images and this is done by the image correspondence component. Further, this component also performs matching and geometric verification for each of the images. In the second stage, the reconstruction component takes the output from the previous stage as input and gives the model a selected two-view reconstruction prior to registering new images. Further, the scene points are determined employing triangulation, outliers filtering and modification of reconstruction output by bundle adjustment (BA) techniques.

The authors employ SIFT (Scale Invariant Feature Transform) algorithm to extract the invariant features from the images, along with matching to find the scene overlap and geometric verification to find the possible match of features across the images based on homography and epipolar. During the reconstruction phase, the dense location from the scene graph is chosen i.e. the location with high overlapping features. Further, the accurate pose is estimated using image registration based on best image selection technique. Next, using triangulation a new scene point is added to the already existing list of points. Lastly, using bundle adjustment the re-projection error is minimized for every iteration by altering the camera parameters along with the point parameters. In this project, we have used the COLMAP software for reconstruction in 3D considering several un-ordered images.

3 Approach

3.1 Dataset Collection

We have collected 27 images of a floral patterned bed with objects placed on it. We have used a single phone camera with same internal camera parameters (i.e., with same zoom etc.,) to take pictures from different orientations. We expected the dominant plane to be the flat surface of the cot which is in fact the case as per step 4. A sample of 4 images is shown in Figure 1

Figure 1: Sample of 4/27 images from the dataset

3.2 3D Reconstruction using COLMAP

We used COLMAP software to perform feature extraction, matching, incremental reconstruction and bundle adjustment on our images to get a sparse reconstruction of the scene. The output is a cloud of 3D points that are "interesting". We have specified that the camera parameters are same for all the images by checking the box 'Shared for all images' while extracting features. We have exported the model as text to get cameras.txt, images.txt, points3D.txt The cameras.txt has the camera's internal parameters which are the camera id, camera model, focal length and the principal point values.

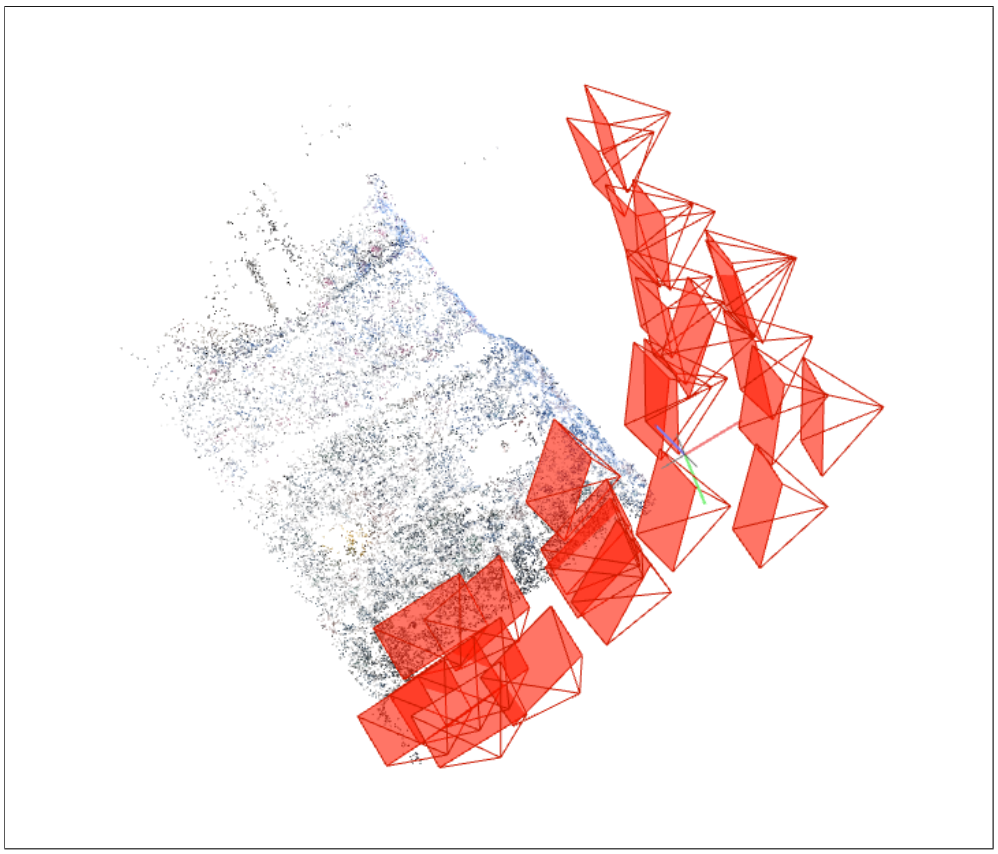

Figure 2: COLMAP output of 3D feature points

The images.txt file contains the camera's external parameters which include the pose and keypoints of all the reconstructed images in the dataset. The pose is represented using a rotation quaternion and a translation vector for each image in the file.
The points3D.txt file contains the X, Y and Z coordinates for each of the point in the point cloud generated from COLMAP. The 3D point model we have obtained for our reconstructed scene is shown in Figure 2.

3.3 Extraction of 3D scene points

We have extracted the X, Y and Z coordinates for each point from the 3Dpoints.txt file generated by COLMAP and stored these point coordinate values in a matrix of dimensions (31499×3). The first few lines of 3Dpoints.txt are shown in Listing 1

```
1 # 3D point list with one line of data per point:
2 #   POINT3D_ID, X, Y, Z, R, G, B, ERROR, TRACK[] as (IMAGE_ID,
    POINT2D_IDX)
3 # Number of points: 31499, mean track length: 4.35801
4 38292 -4.34025 -0.124632 8.10536 152 122 140 2.34242 20 6654 19 4327 26
     5776 21 7270 15 4291 14 2776 18 485
5 38291 -2.18974 -0.485819 8.67149 120 140 168 1.16344 25 665 16 3244 8
    329 7 442 1 365 11 497 13 641 19 644 21 837 18 532 22 1301 27 1467
```

Listing 1: Points3D.txt example

	1	2	3
1	-6.5020	1.6867	9.1669
2	1.0737	0.2889	11.1339
3	4.4963	0.6905	10.9592
4	-1.1280	0.7557	10.1690
5	-6.5793	1.5840	9.3775
6	-0.0929	0.2573	10.9419
7	0.1323	3.6055	10.0924

Figure 3: Sample Snippet of the (X,Y,Z) coordinates

We extract the attributes from the points3D.txt file and only parse the columns 2 to 4 which correspond to the (X,Y,Z) coordinates. The code snippet extracting the 3 coordinates can be found from Listing 2.

```
%%
%   READ POINTS FROM POINTS3D.TXT
%%
points3D = readmatrix('points3D.txt', 'delimiter', ' ');
pts_xyz = points3D(:,2:4);
```

Listing 2: reading 3D scene points into a matrix

A sample snip of the 3D coordinates of (X,Y,Z) can be found from the figure 3. The extracted 3D coordinates are further used for processing to obtain the Inlier points and the Outlier points using the RANSAC routine, which is explained in detail in the next step.

3.4 Implementing RANSAC for fitting plane through 3D points

The basic intuition is to fit a plane that represents most of the extracted 3D points. We need minimum 3 points to fit a plane. The RANSAC procedure is implemented as follows:

1. Maintain a global inlier count and initialize it to ZERO.

2. Select three points in random from the total set of 3D points

3. Fit a plane passing through these three points using Least Square approach.

4. With the best fit plane, find all the points lying close to this plane, based on a certain threshold distance. These are the set of inlier points corresponding to this best fit plane passing through the randomly selected three points. The threshold is found based on the table 4 when measurement error is known to be Gaussian with mean 0 and variance s^2.

m	Model	d^2
1	Line, fundamental matrix	3.84 s^2
2	Homography, camera matrix	5.99 s^2

Figure 4: Deciding the threshold

Often the threshold is chosen empirically. In our case, we have chosen a threshold of "**0.5**".

5. If the total number of inlier points exceeds the global inlier count, update the inlier count to this new total and store these inlier indices or points. otherwise, repeat the same procedure by selecting new set of three random points.

6. Repeat the same procedure for N times, where N is chosen based on the distributing of samples (based on the number of good and bad samples). We choose the value of N based on the equation 1

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

where "p" is the desired probability that we get a good sample, s is the number of points in a sample and e is the probability that a point is an outlier.

7. After looping for N times, we have found our set of Inlier indices/points and also found the best fit plane that passes through the randomly selected three points which resulted in the choosing of these inlier points.

We use these set of inlier points and find a new plane which best fits all these global inlier points, the best fit plane equations and the three points lying on this best fit plane for our future steps to find the normal to this plane, etc. The further steps are explained in detail in the coming steps. The outline of the RANSAC procedure is portrayed in the block diagram 5. The remaining points apart from the inlier points are the outlier points.

3.5 Finding inlier Points and dominant plane

Figure 6 contains the output of RANSAC. The inlier points are denoted in red and the outliers in blue. The dominant plane is found by randomly initializing a normal vector $\hat{n}$ and minimizing the objective

$$\Sigma|\hat{n}^T x|\forall x \in Inliers$$

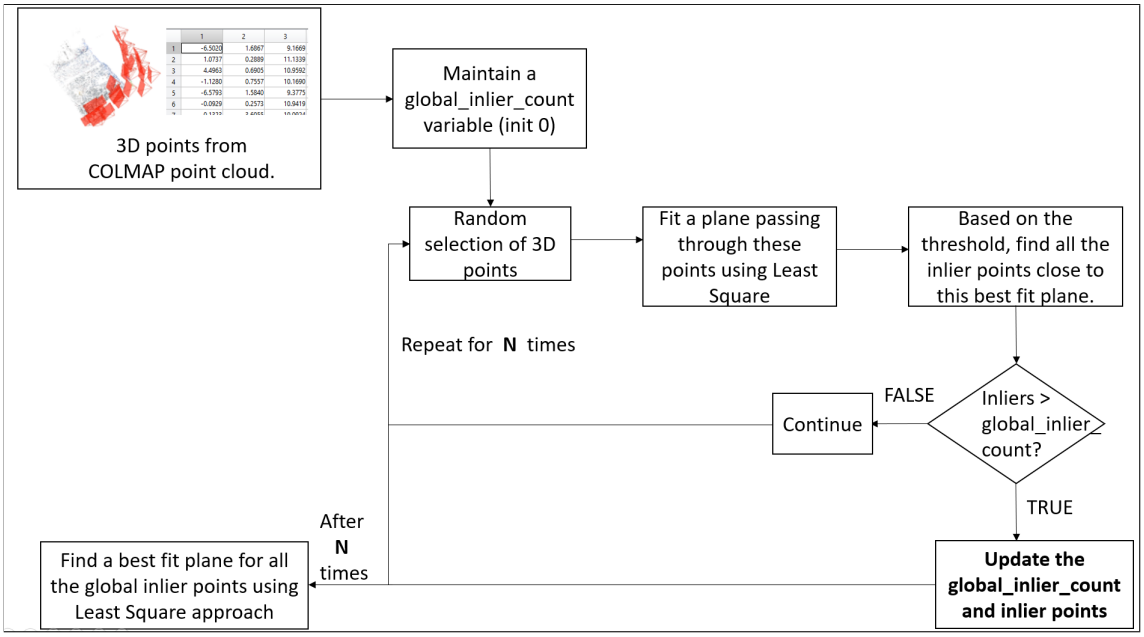

Figure 5: Basic block diagram of the RANSAC procedure to extract the set of inlier points

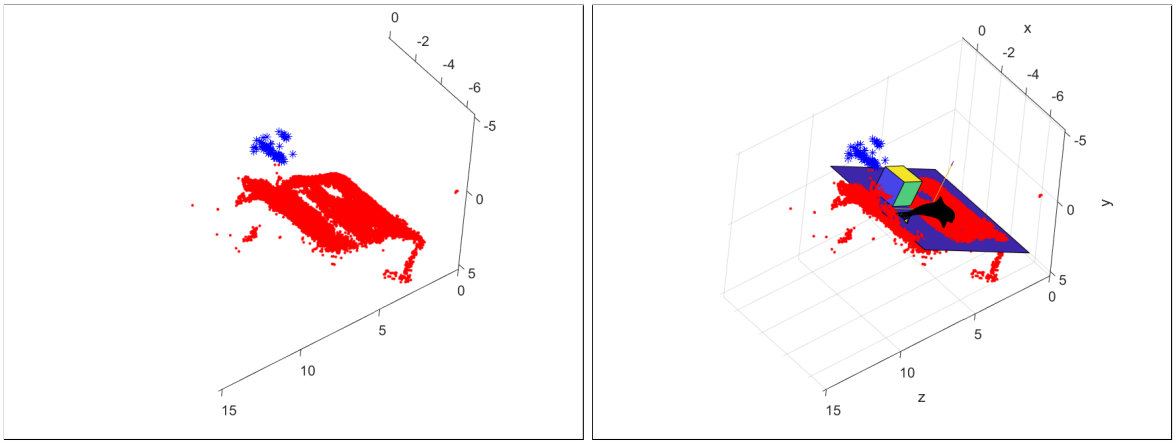

Figure 6: Inlier points (in red) as point cloud (left) and with objects and dominant plane drawn in (right)

3.6 Creating a Local Coordinate System

This process is broken down into several sub steps:

3.6.1 Choosing the orgin

The origin is chosen as the centroid of the inlier point cloud

$$\bar{x} = \frac{\Sigma_{x \in I} x}{|I|}$$

where I is the set of inliers.

3.6.2 Obtaining the rotation and translation matrices

The normal for the scene coordinate system is the Z axis $\begin{bmatrix}0\\0\\1\end{bmatrix}$ and the normal obtained in step 3.5 $\hat{n} = \begin{bmatrix}n_1\\n_2\\n_3\end{bmatrix}$. The angle between them is given by $\theta = \cos^{-1}(\hat{n} \cdot \hat{k})$. As they both are unit vectors, we do not need to divide by the magnitude. To align both planes, we need to rotate $\hat{n}$ by θ onto the Z axis by an axis perpendicular to both i.e. $(\hat{a} = \hat{n} \times \hat{k})$. This rotation will also suffice to align their corresponding planes.

$$\hat{a} = \hat{n} \times \hat{k} = \begin{bmatrix}a_x & a_y & a_z\end{bmatrix} \tag{2}$$

$$A = \begin{bmatrix}0 & -a_z & a_y\\ a_z & 0 & -a_x\\ -a_y & a_x & 0\end{bmatrix} \text{ is the cross product matrix such that} A\overrightarrow{x} = \hat{a} \times \overrightarrow{x} \tag{3}$$

$$R = I + \sin\theta A + (1 - \cos\theta)A^2 \tag{4}$$

Rodigues' formula outlined in Eqn 4 allows us to define the matrix R given $\hat{a}$ and θ. The matlab function `axang2rotm` yields the homogeneous version of the same matrix. The translation vector is given by $\tau = -R^T\bar{x}$

The transform $M = R_h\tau_h$ (where the subscript h denotes the homeogeneous form) and M^{-1} convert from scene coordinates to local coordinates and vice versa, respectively.

```
%%
%   OBTAIN TRANSFORM MATRIX TO CONVERT FROM LOCAL <---> SCENE SYSTEMS
%%
params = fit_plane_for_given_pts(inliers');
a = params(1);
b = params(2);
c = params(3);

v = [a; b; c];


v = v/norm(v);
k = [0; 0; 1];

theta = acos(v'*k);  %angle between nomal to global plane (Z axis) and
                     %plane normal
a = cross(v,k);
a = a/norm(a);
A = [0 -a(3) a(2); a(3) 0 -a(1); -a(2) a(1) 0];
%Rot = axang2rotm( [cross(v,k)' theta]); % rotate by theta by axis
    perpendicular
                                        % to both normals

```

```
Rot = eye(3) + sin(theta)* A + (1 - cos(theta))* (A*A);
scene2local = trvec2tform(-mean_xyz*Rot') * rotm2tform(Rot);
```

Listing 3: generating local coordinate system

3.7 Creating a Virtual Object (Cube & a Dolphin)

Two objects are designed: a dolphin (the faces and vertices obtained from PyTorch 3D's resources) and a cube.

Each object is denoted as a list of faces and a list of vertices. Local transformations are applied to the respective vertices to bring the two objects to the desired orientation. Then, the corresponding vertex and face lists are concatenated into a global vertex list and a global face list, containing details of all objects in the scene.

M^{-1} as listed in section 3.6.2 is then used to transform the vertex list from local to scene coordinates.

```


%%
%   DEFINE BOTH OBJECTS
%%
A = 1.5;
t_object = [0 2*A 0];



verts1 = cart2hom(readmatrix('vertices.txt', 'delimiter', ' ')); %read
    dolphin

dolphin_center = mean(verts1,1);
verts1 = verts1 * trvec2tform(-hom2cart(dolphin_center))' * rotm2tform
    (5*eye(3));
local_transform = get_transform_mat(120,0,0,-1.5,4,2.25,"deg")*...
    get_transform_mat(0,0,0,0,0,3,"deg"); % set dolphin on plane

faces1 = readmatrix('faces.txt', 'delimiter', ' '); % read dolphin
    faces
verts1 = (local2scene * local_transform * verts1')';
verts1 = hom2cart(verts1);




%%% align the cube and draw
[verts2,faces2] = draw_cube(A);
 verts2 = (local2scene * get_transform_mat(0,0,95,1,2,0,"deg") * verts2
    ')';
 verts2 = hom2cart(verts2);
```

Listing 4: Building both objects

3.8 Extraction of camera parameters

We have extracted camera internal parameters from **'cameras.txt'** and stored them in a matrix of dimensions $3X4$ (K matrix). and extracted external camera parameters for each image from **'images.txt'** which helps construct the pose (The $[R|t]$ matrix) of dimensions $3X4X27$ which contains 27 $3X4$ $[R|t]$-Matrices for all of the 27 images. The corresponding code snippet is shown below in 5.

```
1
2 saveas(gcf,"outs/pointcloud_objects.png");
3
4 %%
5 %   READ CAMERA PARAMS DEFINE R_T and K
6 %%
7 cam_params = readmatrix('cameras.txt', 'delimiter', ' ','NumHeaderLines
    ',3);
8 width = cam_params(3) ;
9 height = cam_params(4) ;
10 focal_length = cam_params(5) ;
11 delta_x = cam_params(6) ;
12 delta_y = cam_params(7) ;
13 skew = cam_params(8);
14
15 phi_x = focal_length; % / width;
16 phi_y = focal_length; % / height;
17
18 K = [phi_x, skew, delta_x; 0, phi_y, delta_y; 0, 0, 1];
19
20 camera_mat = readmatrix('images.txt', 'delimiter', ' ');
21 %1 SIMPLE_RADIAL 4000 3000 3036.32 2000 1500 0.0249925
22
23 new_camera_mat = zeros(27,37338);
24 for i = 1:27
25 new_camera_mat(i,:) = camera_mat(2*i - 1,:);
26 end
27
28 Q_matrix = new_camera_mat(:,2:5);
29 T_matrix = new_camera_mat(:,6:8);
30
31 R_matrix = quat2rotm(Q_matrix);
32
33 R_T_matrix = zeros(3,4,27);
34 for i = 1:27
```

Listing 5: Extracting Camera Parameters

3.9 Projecting scene (3D) coordinates to Image (2D) coordinates

Simple radial camera model follows the equation-

$$\lambda \times \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} = \begin{bmatrix} \phi_x & \gamma & \delta_x \\ 0 & \phi_y & \delta_y \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} w_{11} & w_{12} & w_{13} & \tau_x \\ w_{21} & w_{22} & w_{23} & \tau_y \\ w_{31} & w_{32} & w_{33} & \tau_z \end{bmatrix} \times \begin{bmatrix} u \\ v \\ w \\ 1 \end{bmatrix} \tag{5}$$

where the above equation can be compactly represented as -

$$\lambda \times p = \mathrm{K}[\mathrm{R} \mid \mathrm{t}]P_w \tag{6}$$

where,

$$p = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{7}$$

$$\mathrm{K} = \begin{bmatrix} \phi_x & \gamma & \delta_x \\ 0 & \phi_y & \delta_y \\ 0 & 0 & 1 \end{bmatrix} \tag{8}$$

$$\mathrm{K}[\mathrm{R} \mid \mathrm{t}] = \begin{bmatrix} w_{11} & w_{12} & w_{13} & \tau_x \\ w_{21} & w_{22} & w_{23} & \tau_y \\ w_{31} & w_{32} & w_{33} & \tau_z \end{bmatrix} \tag{9}$$

$$P_w = \begin{bmatrix} u_i \\ v_i \\ w_i \\ 1 \end{bmatrix} \tag{10}$$

where the matrix K is computed from intrinsic camera parameters and K[R | t] matrices as computed from image parameters as described in the previous step and finally project a given set of 3D (X,Y,Z) world points into a set of 2D (col, row) pixel locations using equation 9. Here we snow the snippet in the code that is responsible for it below 6.

Listing 6: Projecting Scene coordinates to image coordinates

3.10 Projecting virtual object onto images

As outlined in section 3.9, for an image i, we have a transformation $\Gamma_i = K[R|t]_i$ that maps a 3D scene coordinate to a 2D image pixel coordinate, such that $\lambda P_c = \Gamma_i P_w$ for a scalar λ.

By applying Γ_i to all vertices in our object, we project it onto the image. However, when filling in surfaces, we still have to account for hidden surfaces. The code for projection is in Listing 7

```
1
2     %cube 1
3     set_2d1 =K * R_T_matrix(:,:,i)* cart2hom(verts)';
4     pimage2d1 = hom2cart(set_2d1');
5     depths1 = get_Z_buffer(cart2hom(verts)',set_2d1,K,R_T_matrix(:,:,i)
      );
6     maxval = max(depths1);
7     depths1 = abs(depths1 - maxval);
8     depth_2d1 = [pimage2d1 depths1'];
9
10
11
12
13    figure
14    imshow(strcat('imgs\',list_of_files(i).name));
15    hold on;
16
17
18
19    patch('Vertices',depth_2d1,'Faces',faces,...
```

Listing 7: Projecting object onto image

3.10.1 Z buffering

We calculate the camera view angle as

$$\overrightarrow{u} = R_i^T K^{-1} P_c$$

Using $\hat{u} = \frac{\overrightarrow{u}}{||\overrightarrow{u}||}$, we calculate the depth of each point P_w as:

$$d = \hat{u} \cdot P_w$$

We draw all surfaces incorporating this depth component leading to better occlusion of hidden surfaces.

NOTE : Arbitrarily picking P_c to calculate $\hat{u}$ could lead to a vector that is not parallel to to the normal to the image plane. This phenomenon is shown in Fig. 7. The actual depth of the point is 5 units, along the pink vector from the Camera position C to the point E. The magnitude of this vector $\overrightarrow{CE}$ is **5 units**, which is the **true depth of** P_c, denoted as d_{true}. However, we see that we calculate the magnitude of the red vector in our method.

This is an acceptable metric as well, because if $\hat{u}$ makes an angle α with the normal to the image plane, the true depth is given by $d_{true} = d \cos \alpha$. As long as we fix the vector $\hat{u}$ and calculate all depths using the same vector, this metric holds.

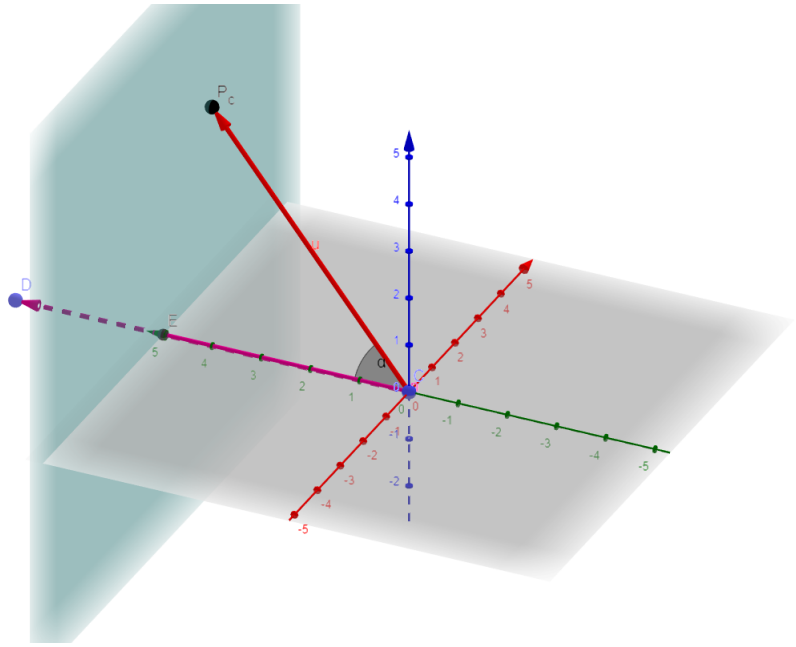

Figure 7: Depth calculation

```
function [zbuffer] = get_Z_buffer(points_3d,points_2d,K,R_t)
%UNTITLED Summary of this function goes here
%   Detailed explanation goes here
% points 3d of shape (4,N)
R = R_t(:,1:3);
T = R_t(:,4);

pt = mean(points_2d,2); %use one point to get u
u = R'*pinv(K)* (pt/norm(pt));

u_hat = u / norm(u);

zbuffer = u_hat' * points_3d(1:3,:);
end
```

Listing 8: Calculating depths

4 Results & Discussion

In this section we will be presenting the results of our project in the figures 8. In our implementation, we work on inserting 2 3D images, namely a 3D cube and a 3D image of dolphin. In the results, we can observe the 3D depth that has been taken care based on the Z-buffering approach discussed in section 3.10.1. We can observe the colors of the faces of the cube in the exact order, even with the movement of camera. Also, we can observe the placement of the 3D image of dolphin next to the cube. One of the key things to notice from the results is handling of the depth information effectively, where, in cases where cube is in front of dolphin, the cube covers the dolphin partially just as expected.

The observed results of the code are generated in the subfolder "outs" after running the main script "project2.m". With each run, we randomly choose color for each face of

Figure 8: Sample result figures

the cube. Also, in our implementation, we have included a segment that handles local transformations to rotate, scale and translate the objects before placing them in the scene. The code snippet that performs this feature is included in the Listing 4.

5 Conclusion

In this project, we have implemented an Augmented Reality (AR) system that projects an artificial 3D object onto a real world environment scene. For the AR system, we have built the RANSAC function to determine the dominant plane. In addition, we have also implemented 3D to 2D transformation for the in-liers and projected a artificial object in 3D onto the dominant plane i.e. to the real world environment scene.

6 Code

Our implementation of the current paper can be found at:
https://github.com/Laxmaan/3D-Reconstruction

References

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