Rohaan Advani -N19682109

PROJECT 4: .

PROBLEM 1: 3D OBJECT GEOMETRY FROM TRIANGULATION:

INTRODUCTION:

In computer vision and computer graphics, perspective projection is a method used to represent a three-dimensional scene onto a two-dimensional plane, mimicking the way human vision perceives depth in the real world. This projection involves transforming 3D points in a scene into 2D points on an image plane. The equations for perspective projection are fundamental for tasks like camera calibration, 3D reconstruction, and augmented reality. Here's an explanation of the perspective projection equations:

Let's consider a 3D point in the world coordinate system, denoted as P = [X, Y, Z]^T, where X, Y, and Z are the coordinates in the 3D space. The camera or image plane is located at a distance f from the camera center along the z-axis.

1. Homogeneous Coordinates: To simplify the mathematical operations, we often use homogeneous coordinates. The 3D point P is represented as P_h = [X, Y, Z, 1]^T.

2. Intrinsic Matrix: The intrinsic matrix, often denoted as K, represents the internal parameters of the camera and is defined as:

K = [f, 0, cx]
[0, f, cy]
[0, 0, 1]

where f is the focal length of the camera, and (cx, cy) are the coordinates of the principal point (the point where the optical axis intersects the image plane).

3. Extrinsic Matrix: The extrinsic matrix, often denoted as [R | t], represents the external parameters of the camera. R is the rotation matrix describing the camera's orientation in the world, and t is the translation vector representing the position of the camera in the world.

4. Perspective Projection: The perspective projection of a 3D point P onto the image plane is given by the equation:

[u, v, 1]^T = (1/Z) . K . [R | t] . P_h

Here, Z is the depth of the 3D point P (distance along the camera's z-axis).

In summary, the perspective projection equations involve combining the intrinsic matrix, extrinsic matrix, and the 3D coordinates of a point to obtain its corresponding 2D pixel

coordinates on the image plane. These equations play a crucial role in various computer vision tasks, enabling the transformation of 3D information into a 2D representation for further analysis and visualization.

Assuming our camera model is a pinhole camera model, this pre-calibration can be simplified. Given below is the conversion of a 3D point in the world coordinate system to respective 2D projection.

$$(x, y, z) \rightarrow (f\frac{x}{z}, f\frac{y}{z})$$

Scene point $\rightarrow$ Image coordinates

Stereo vision, in the context of computer vision, refers to the use of two or more cameras to perceive depth and create a three-dimensional representation of a scene. The human visual system relies on the fact that our eyes are separated by a certain distance, which results in each eye capturing a slightly different view of the world. The brain processes these disparate views to perceive depth and distance. Stereo vision in computer vision attempts to replicate this process using multiple cameras.

1. Stereo Cameras: Stereo vision typically involves a pair of cameras, positioned at a known baseline distance from each other. This baseline mimics the separation between human eyes.

2. Image Capture: Each camera captures an image of the same scene from its viewpoint. Due to the baseline separation, the images from the two cameras will have slightly different perspectives. For experimental purposes, one camera can also be used by shifting by baseline B it to capture 2 images from different viewpoints.

3. Image Rectification: To simplify the stereo matching process, the captured images are often rectified. Rectification transforms the images so that corresponding epipolar lines (lines representing corresponding points in the two images) become parallel. This simplifies the matching process to one dimension.

4. Stereo Correspondence: Stereo correspondence involves finding matching points in the rectified images. Corresponding points are pixels in the two images that represent the same 3D point in the scene. This matching process is often done by comparing pixel intensities or features.

5. Disparity Map: The disparity between corresponding points is calculated. Disparity is the horizontal difference between the pixel coordinates of the corresponding points in the two rectified images. Larger disparities typically correspond to closer objects, while smaller disparities correspond to objects farther away.

6. Depth Calculation: Using the disparity information and the known baseline distance, the depth of each pixel in the scene can be calculated using triangulation. The depth information is crucial for creating a 3D representation of the scene.

7. 3D Reconstruction: The result of stereo vision is often a point cloud, which is a set of 3D points representing the surfaces of objects in the scene. These points can be used for further analysis, object recognition, or other computer vision applications.

Stereo vision is widely used in computer vision for tasks such as 3D reconstruction, object detection, and scene understanding. It is particularly valuable in applications like robotics, autonomous vehicles, augmented reality, and medical imaging. The ability to perceive depth provides richer information about the environment, enabling more sophisticated and accurate computer vision systems.

METHODOLOGY AND DISCUSSION:

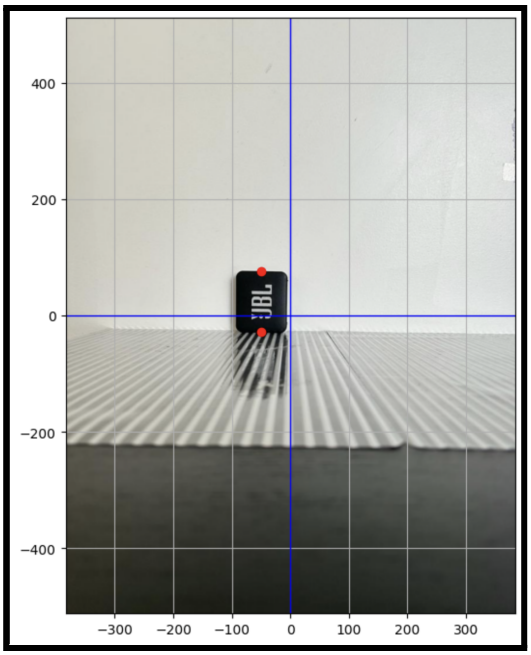
The project was divided into 2 parts. The first part involves simple pre-calibration to find the focal length of our camera. The second part involves taking 2 pictures of an object from a left and right viewpoint as done in stereo vision and using these pictures to find the depth of the object and perform 3D reconstruction of the object.

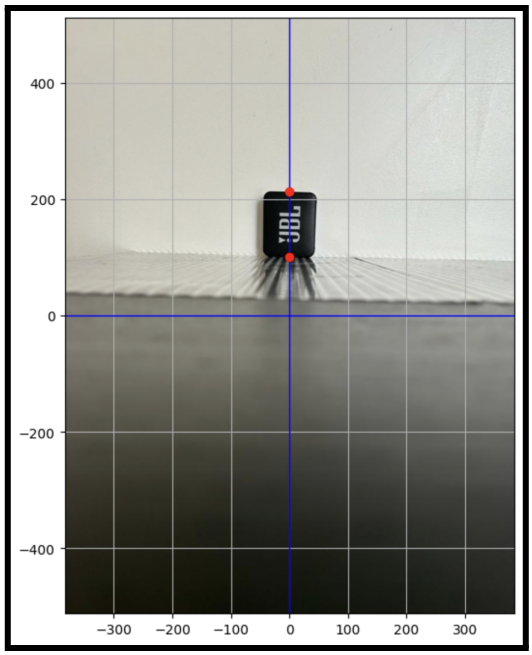
For part 1, choose an object of height X (mm) and place it at a fixed distance Z (mm) from the camera. Take a picture of this object and display it using a plot. As mentioned in the project, assume that the intersection of the optical axis with the sensor is in the middle of the image (let origin be in the middle of the image in the plot).

Experimentally mark points on the plot marking the top and bottom of the object to calculate height x in pixels. Use these values in order to find the focal length using the simple camera pre-calibration formula shown in the introduction. To make sure focal length is accurate you can calculate focal length using multiple pictures and take an average as shown in the images below and the corresponding code in the appendix.

Formula used to calculate the focal length:

f (pixels) = x (pixels) * Z (mm) / X (mm)



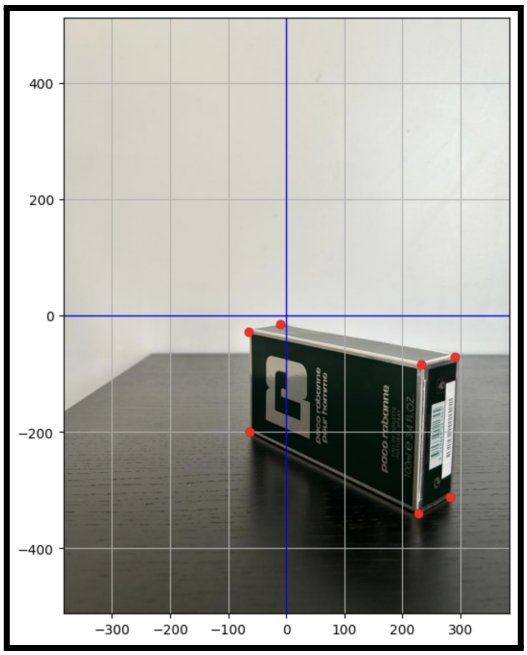


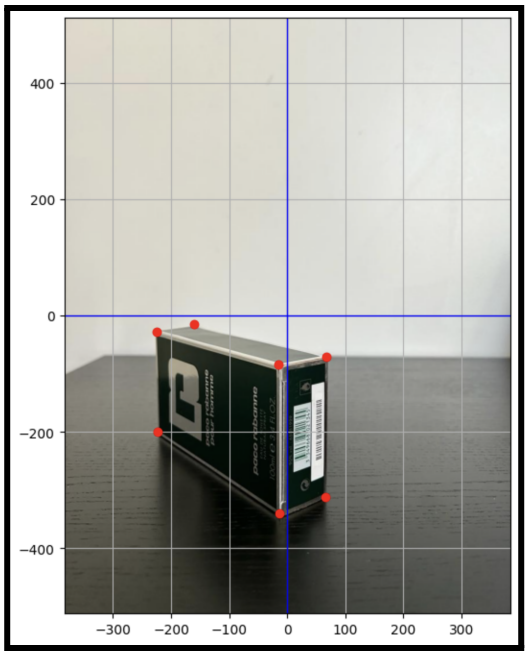
The experimental values were as follows:
X = 86 mm, Z = 650 mm
Test 1: x = 105 pixels, f = 793.6 pixels
Test 2: x = 112 pixels, f = 846.51 pixels
Average Focal Length = 820.05 pixels

Now that the focal length has been calculated, we can move on to the second part. For this part choose a 3D object preferably an object with sharp corners for proving the 3D reconstruction happened accurately. Place it not too far away from the camera and take a picture of the object. Shift camera towards the right in the same plane and take a second picture of the object.

This is a case of parallel stereo vision where we have two images (left and right) created by translation only and no rotation. Translation is the extrinsic change done and the intrinsic parameters of the camera remain the same as part 1, that is the focal length f will remain the same. While shifting the camera towards the right make sure to keep track of how much shift was made. This shift is the baseline B of the stereo vision experiment.

Create plots of the left and right images similar to those done for the part 1 experiment with axes in the center of images. Experimentally find the pixel coordinates of the corners of the object in the image and mark them on the plot as shown below:





In the above images, the baseline B = 70mm. Below are the coordinates of the points the left column represents the points in the left image and the right column represents the corresponding points in the right image which are marked manually:

```
0 [-65, -27] [-225, -27]
1 [-10, -15] [-160, -15]
2 [-63, -199] [-223, -199]
3 [232, -83] [-15, -83]
4 [290, -70] [68, -70]
5 [228, -340] [-13, -340]
6 [283, -311] [66, -311]
```

It can be seen that the above points vary in the x coordinate and the y coordinate remains the same for the pixels between the left and right images as it should be in the case of parallel stereo vision. The difference between the shift in the pixels in the x coordinate is known as the disparity and can be represented by d.

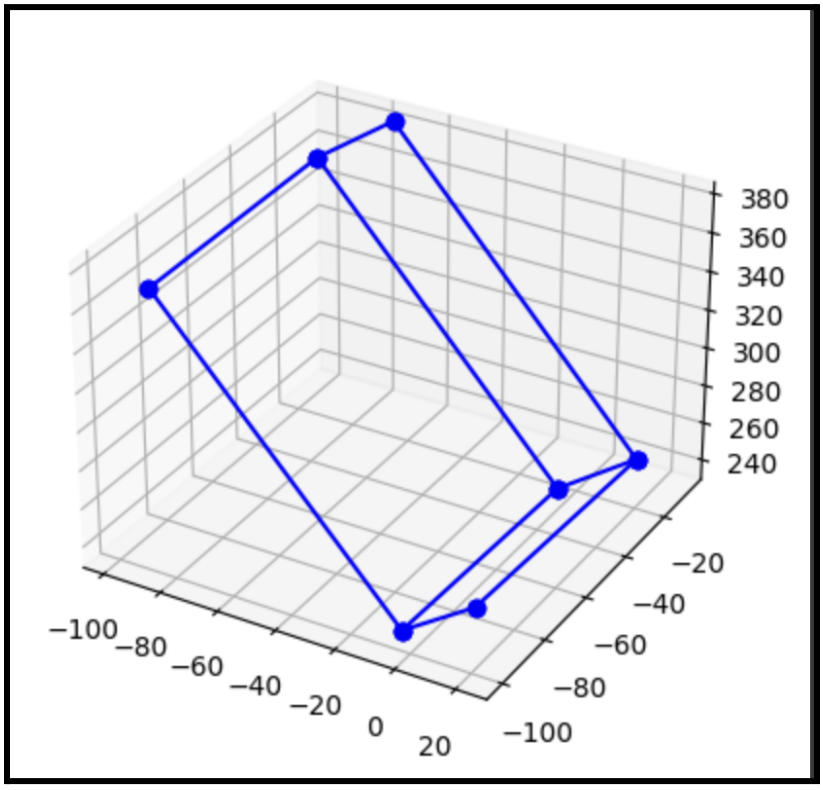
We can calculate values of d as shown in the code in the appendix and use them to calculate the world coordinates of the corners of the object using the below formulae:
Z (mm) = f (pixels) * B (mm) / d (pixels)
X (mm) = Z (mm) * x (pixels) / f (pixels)
Y (mm) = Z (mm) * y (pixels) / f (pixels)
Here x, y corresponds to the pixel coordinates of the corner. Make sure to either take these pixel coordinates from either the left image or the right and not from both.

Shown below are the calculated values for the world coordinates of the 7 corners manually marked by the user in [X, Y, Z] format:

```
0 [-98.4375, -11.812500000000002, 358.77543604651163]
1 [-74.66666666666667, -7.0, 382.6937984496124]
2 [-97.5625, -87.0625, 358.77543604651163]
3 [-4.251012145748987, -23.522267206477732, 232.40514075887393]
4 [21.44144144144144, -22.07207207207207, 258.5768908443327]
5 [-3.7759336099585066, -98.75518672199172, 238.19116086075462]
6 [21.29032258064516, -100.32258064516128, 264.5348837209302]
```

Firstly we use these coordinates to calculate the average depth of the object from the camera that can be done using the average of the Z coordinates in the above image. The average depth of the object in this experiment is equal to 299.13 mm.

The above set of world coordinates can be used to perform a 3D reconstruction of the original object. The code to plot the wireframe structure for the coordinates is in the appendix. Below is the plot for the same:



It can be seen that the 3D reconstruction of the object is cuboidal as is the object. To create this graph we not only need the coordinates calculated above but also the list of

edges that are connected. Below is the list of edges that are connected in the above graph:

```
[(0, 1), (0, 2), (0, 3), (1, 4), (2, 5), (3, 4), (3, 5), (4, 6), (5, 6)]
```

Here the numbers refer to the index of the corners in the coordinates list displayed above.

As a last step we must check whether the 3D reconstruction created is an accurate representation of the actual object. To do this we must measure the actual lengths of the edges and store them in a list. Let the order of these lengths be the same as the above edges list. We will be using the euclidean distance formula as shown in the code in the appendix to calculate the lengths of the edges found experimentally. Below is the results for the actual and experimental lengths of edges on the left and right respectively:

```
32 34.06318736784883
77 75.25508703735582
146 158.0432367215808
146 157.69889839235742
146 153.20968109126352
32 36.703756480685996
77 75.45658292728568
77 78.47714704278427
32 36.397330358930226
```

It can be seen that the lengths of the edges are calculated almost correctly. Thus I think this experiment was a success. The percentage error for the experiment is coming out to be 6.91% between the actual and experimental results. This may be due to human errors since we are doing manual calculations and shifts instead of using an actual stereo.

APPENDIX: PROJECT CODE:

```
import matplotlib.pyplot as plt
import matplotlib.image as img
from mpl_toolkits.mplot3d import Axes3D

Z = 650 # 650 mm
X = 86 # 86 mm

test = img.imread('Test1.jpeg')
plt.figure(figsize=(6, 10))
plt.plot([-400, 400], [0, 0], color="blue", linewidth=1)
plt.plot([0, 0], [-600, 600], color="blue", linewidth=1)
plt.plot(-50, 77, marker='o', color="red")
plt.plot(-50, -28, marker='o', color="red")
plt.grid()
plt.imshow(test, extent=[-test.shape[1]/2., test.shape[1]/2.,
-test.shape[0]/2., test.shape[0]/2. ])

x1 = 105 # 77 - (-28) = 105 pixels
f1 = (x1 * Z) / X # pixels
print('Focal Length Test 1: ', f1)

test = img.imread('Test2.jpeg')
plt.figure(figsize=(6, 10))
plt.plot([-400, 400], [0, 0], color="blue", linewidth=1)
plt.plot([0, 0], [-600, 600], color="blue", linewidth=1)
plt.plot(0, 214, marker='o', color="red")
plt.plot(0, 102, marker='o', color="red")
plt.grid()
plt.imshow(test, extent=[-test.shape[1]/2., test.shape[1]/2.,
-test.shape[0]/2., test.shape[0]/2. ])

x2 = 112 # 214 - 102 = 112 pixels
f2 = (x2 * Z) / X # pixels
print('Focal Length Test 2: ', f2)

f = (f1 + f2)/2. # Avg focal length in pixels
print('Avg Focal Length: ', f)
```

```
B = 70 # 70 mm
points = [[[-65,-27],[-225,-27]], [[-10,-15],[-160,-15]],
[[-63,-199],[-223,-199]], [[232,-83],[-15,-83]], [[290,-70],[68,-70]],
[[228,-340],[-13,-340]], [[283,-311],[66,-311]]]

test = img.imread('Left.jpeg')
plt.figure(figsize=(6, 10))
plt.plot([-400, 400], [0, 0], color="blue", linewidth=1)
plt.plot([0, 0], [-600, 600], color="blue", linewidth=1)
for i in range(len(points)):
      plt.plot(points[i][0][0], points[i][0][1], marker='o', color="red")
plt.grid()
plt.imshow(test, extent=[-test.shape[1]/2., test.shape[1]/2.,
-test.shape[0]/2., test.shape[0]/2. ])

test = img.imread('Right.jpeg')
plt.figure(figsize=(6, 10))
plt.plot([-400, 400], [0, 0], color="blue", linewidth=1)
plt.plot([0, 0], [-600, 600], color="blue", linewidth=1)
for i in range(len(points)):
      plt.plot(points[i][1][0], points[i][1][1], marker='o', color="red")
plt.grid()
plt.imshow(test, extent=[-test.shape[1]/2., test.shape[1]/2.,
-test.shape[0]/2., test.shape[0]/2. ])

def calc_world_coord(B, f, points):
      disparities, world_coordinates = [], []
      for i in range(len(points)):
            left = points[i][0]
            right = points[i][1]
            d = left[0] - right[0]
            disparities.append(d)
            Z = (f * B) / d # mm
            X = (Z * right[0]) / f # mm
            Y = (Z * right[1]) / f #mm
            world_coordinates.append([X,Y,Z])
      return disparities, world_coordinates

depth = 0
no_points = 7
```

```
for i in range(len(points)):
      print(str(i)+' '+str(points[i][0])+' '+str(points[i][1]))
disparities, world_coordinates = calc_world_coord(B, f, points)
for i in range(len(disparities)):
      print(str(i)+' '+str(disparities[i]))
for i in range(len(world_coordinates)):
      print(str(i)+' '+str(world_coordinates[i]))
depth += world_coordinates[i][2]
depth /= no_points
print('Average depth of object = ', depth)

def plot_3d_lines(ax, points, lines):
      for line in lines:
            x = [points[i][0] for i in line]
            y = [points[i][1] for i in line]
            z = [points[i][2] for i in line]
            ax.plot(x, y, z, marker='o', linestyle='-', color='b')

lines_input = '0,1;0,2;0,3;1,4;2,5;3,4;3,5;4,6;5,6'
lines = [tuple(map(int, line.split(','))) for line in
lines_input.split(';')]
print(lines)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
plot_3d_lines(ax, world_coordinates, lines)
plt.show()

def calc_dimensions(world_coordinates, lines):
      dimensions = []
      for i in lines:
            x1, y1, z1 = world_coordinates[i[0]]
            x2, y2, z2 = world_coordinates[i[1]]
            dim = ((x2 - x1)**2 + (y2 - y1)**2 + (z2 - z1)**2)**0.5
            dimensions.append(dim)
      return dimensions

real_dimensions = [32, 77, 146, 146, 146, 32, 77, 77, 32]
edges = 9
err = 0
dimensions = calc_dimensions(world_coordinates, lines)
```

```
for i in range(len(dimensions)):
      print(str(real_dimensions[i])+' '+str(dimensions[i]))
err += abs(real_dimensions[i] - dimensions[i]) / real_dimensions[i]
err /= edges
print('Overall % Error = ', err * 100)
```