CS 423 Project Report

Introduction

In this project, I aimed to reconstruct a 3D world within a closed system observed by four distinct cameras. In order to achieve this, I first created a 3D voxel grid between the cameras. I chose the dimensions of the grid so that it covers the intersection of all cameras' fields of view. I have chosen the dimensions of one voxel so that it doesn't take too long to complete but gives enough information to be able to calculate the volume. I then traced a light ray through the pinhole of each camera and found where the light coming from the voxel would hit the sensor in 3D camera coordinates. Once I found the camera coordinates where the light ray hits, I converted that to image pixel coordinates to find which pixel in the original image that voxel corresponds to. I did this for all cameras and checked if the voxel hits an image pixel that is a part of the object (fish) in the object mask. I made each camera vote if the voxel is a part of the object, if 3 or 4 out of the 4 cameras vote positively, I marked that voxel. After all the voxels are voted, I visualized the voted voxels in a 3D plot that shows the found object inside the 3D voxel grid.

I have implemented this in two different ways. I iterated through the voxels using nested for loops in one implementation and in the other one, I used matrices for my calculations. Both can be found in my submission. Note that the one with matrices take some more time to run.

This report consists of 4 other sections. The first section will be the design where I will explain how the code is structured and what each part of the code does. In the second section, I will give implementation details for each of these parts. In the third section, I will present you the results I got from the project. In the last section, I will give my concluding remarks and mention some other approaches.

Design

Original (Without Matrices):

In this implementation, my code can be separated into 3. The first part is where I define the necessary parameters and fetch the image matrices from the image masks for each fish. The necessary parameters include the world coordinates of the focal points of each camera, the height and width of each sensor, and the resolution of each sensor. To fetch the matrices from the images, I used OpenCV.

The second part is where I create a voxel grid and iterate through each voxel in that grid. For each iteration, I find where the corresponding voxel falls on the image plane of each camera. I use a function called find_projection_on_sensor for this which will be explained in the third part. After I find the pixels where the corresponding voxel hits the image plane for each camera, I check the matrix of the image mask that is taken by the corresponding camera to see if the voxel hits the pixel which is a part of the object (fish). Since the object is in 3D and the sensor is in 2D, the cameras do not have the depth information. However, we can derive

this information by using the images of all cameras. Since the cameras are aligned so that they are rotated 90 degrees according to the origin, they can see different dimensions of the object. If we combine this information we can determine the depth of the object as well. How we combine is that each camera casts a vote for all of the voxels. If a voxel is voted positively by 3 or more cameras, we mark that voxel positively which means that we determine that this voxel is a part of the object. After this, the voxel grid is visualized where the marked pixels can be seen in red.

The third part is the main logic which is embedded in a function called find_projection_on_sensor. We provide the coordinates of the focal point of the camera and the coordinates of the pixel that is going to be projected onto the image plane. The function first converts these coordinates to camera coordinates. In order to do this, the coordinates should be rotated according to the rotation of the camera. How the rotation is done will be explained in the Implementation Details section. After the rotation, the coordinates will be translated according to the focal point of the camera (the focal point of the camera will be the origin of the camera space.). Now that the points are converted to the camera coordinates, we can determine at which point in the sensor the point falls using triangle similarity. Once the point is found, it is converted into the pixel coordinates.

With Matrices:

In the one with matrices, I use the same structure. There are differences in the second and the third part. In the second instead of iterating through the voxels using a nested for loop, I use np.meshgrid to create the voxels. I preferred using a for loop in the original implementation in order not to keep all voxels in memory.

The differences in the third part will be discussed in the later section.

Implementation Details

Original (Without Matrices):

Let me start with the first part where necessary parameters are defined and the image matrices are constructed using OpenCV. First, I define the given parameters like focal length, camera distances, sensor width and height, and sensor resolution. To be able to convert mms to pixels, I defined a coefficient named mm_to_pixel_coefficient. This is simply the division of the width of the sensor by the number of pixels in the x dimension (1920/11.3). Then I defined the focal coordinates of each camera in order to distinguish between them. Next, I fetched the image matrices using the imread function of OpenCV.

```
camera_0_image = cv2.imread("fish-1/frame9_cam0_msk.jpg")
```

This function returns a 3 dimensional matrix where the first two dimensions are x and y coordinates of the pixel and the last dimension corresponds to the channel information. The output image image has three channels but since the image masks are black and white, all the channels have equal intensity. This completes the first part.

In the second part, I first had to determine the sizes of the voxel grid and the dimensions of the voxels. To make sure that the grid covers the intersection of all cameras' fields of view, I determined the grid to be a cube that has 840mm width since each camera is 420mm away from the origin. The choice for the voxel size was determined through trial and error. There is a trade-off between the execution time of the program and the amount of 3D information we can capture with the voxel. 10.5mms was a fine sweet spot for me. It is also very convenient since 840 is divisible by 10.5. I describe each voxel by its center point coordinate. These coordinates will range from -409.5 to 409.5 for x, y and z dimensions. I have written a nested for loops as below to iterate through all voxels in the grid.

```
    for voxel_x in range(int(total_grid_size_mm[0]/voxal_size_mm[0])):
        for voxel_y in range(int(total_grid_size_mm[1]/voxal_size_mm[1])):
            for voxel_z in range(int(total_grid_size_mm[2]/voxal_size_mm[2])):
                x_mm = voxel_x*voxal_size_mm[0]-(total_grid_size_mm[0]/2 -
voxal_size_mm[0])
                y_mm = voxel_y*voxal_size_mm[1]-(total_grid_size_mm[1]/2 -
voxal_size_mm[1])
                z_mm = voxel_z*voxal_size_mm[2]-(total_grid_size_mm[2]/2 -
voxal_size_mm[2])

                voxel_mm = [x_mm, y_mm, z_mm]
```

For each camera, I called the function find_projection_on_sensor to find the pixels that the voxel falls onto by giving the coordinate of the focal point of the camera and the voxel center coordinate.

```
camera_0_projection_pixel =
find_projection_on_sensor(camera_0_focal_coordinate_mm, voxel_mm)
```

After finding the projections, I checked if the corresponding pixel has an intensity above 180 (this number is chosen by intuition) in the image mask for that camera. If it has, that means this point is a part of the object in the image plane and we vote that voxel. One problematic case is when the output of the find_projection_on_sensor function does not return an x, y pair that is in the range of the camera resolution. This means that the light ray from the voxel that is traced through the focal point does not hit the image plane. In this case, we can't vote this voxel.

```
if camera_0_projection_pixel[0] < 1920  and camera_0_projection_pixel[0] >= 0
and camera_0_projection_pixel[1] >= 0 and camera_0_projection_pixel[1] < 1200:
      if
camera_0_image[camera_0_projection_pixel[1]][camera_0_projection_pixel[0]][0]
> 180:
                        votes += 1
```

After all the cameras vote on the voxel, we check if the number of votes is greater or equal to 3. If it is so, we mark this voxel by adding it to an array called positively_voted_voxels.

```
if votes >= 3:
      positively_voted_voxels.append(voxel_mm)
```

We do this for each voxel and when the process is over, we convert this array into a numpy array. Finally, we create 3D plot using mathplotlib to show the marked voxels inside the voxel grid. We do this by scattering the elements of the positively_voted_voxels array on the graph.

```
fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')

    ax.scatter(positively_voted_voxels[:, 0], positively_voted_voxels[:, 1],
positively_voted_voxels[:, 2], c='r', marker='o', label='Marked Voxels')
```

After this, we can find the volume of the fish by multiplying the volume of one voxel by the number of voxels in the positively_voted_voxels array.

In the last part, I will explain the details of the find_projection_on_sensor function. As mentioned previously, we give the coordinates of the focal point of the camera and the coordinates of the pixel (the center point of the voxel). We determine which camera we are working with by checking its focal point coordinates. After we determine which camera it is, we convert focal point coordinates and pixel coordinates to camera coordinates. Since the cameras are rotated, we have to rotate the coordinates first. The rotation is a bit complicated. I determined camera 1 as the reference since it is the front camera. I observed that the x-axis for camera 1 corresponds to the -y-axis of camera 0 (top camera). So the negative of the x coordinate of a point in the world coordinates corresponds to the y coordinate of the point according to camera 0. I determined the rotated coordinates by visualizing the rotations between cameras. This is possibly the most challenging part of the project. Once I found the rotated coordinates of the focal point and the pixel, I needed to apply translation to the pixel coordinates according to the focal point of the camera. This can simply be done by subtracting the rotated focal point's x coordinate from the rotated pixel.

```
pixel_location_camera_mm_x = rotated_pixel_location_mm[0] -
rotated_focal_coordinate_mm[0]
```

Once we found the camera coordinates we can finally find where the voxel hits the image plane. We can do this thanks to triangle similarity. We can calculate the similarity ratio by dividing the focal length by the x coordinate of the pixel. This is the key to finding the x and y coordinates of the point on which the voxel will fall. The z dimension of the pixel coordinate will determine the x dimension of the pixel on the sensor while the y dimension will determine the y dimension of the pixel on the sensor. With this, we found the camera coordinates of the pixel that the voxel falls on. We now have to convert this to the image pixel coordinates. To do that, we first multiply both dimensions with the mm_to_pixel_coefficient. The problem now is that we took the origin of the image plane as 0,0. However, it should be 960, 600 since the top left pixel is 0,0. Because of this, we add 960 to the x coordinate and 600 to the y coordinate.

Finally, we round this number since pixel coordinates have to be discrete and now we return this coordinate pair.

```
sensor_projection_x_pixel =
round(sensor_projection_z_mm*mm_to_pixel_coefficient + 1920/2)
sensor_projection_y_pixel =
round(sensor_projection_y_mm*mm_to_pixel_coefficient + 1200/2)
```

With Matrices:

In this implementation, different from the original, I used np.meshgrid to create the voxel grid.

```
    range_values = np.arange(-409.5, 410.0, 10.5)
    voxel_x, voxel_y, voxel_z = np.meshgrid(range_values, range_values,
range_values)
    voxels = np.column_stack((voxel_x.ravel(), voxel_y.ravel(),
voxel_z.ravel()))
```

Here range values are used to determine the range of the voxels. Since the total size is 840, the centers go through -409.5 to 409.5. 10.5 is the voxel dimension, that's why the units are 10.5 units apart. Np.meshgrid is used for creating sıuch grids. Thanks to that, I was able to construct a voxel matrix that contains each combination of the x, y, z values inside the range.

Most of the change was in the third part. For converting the pixel coordinates to camera coordinates, I created a camera conversion matrix specific to each camera. This matrix deals with the translation and rotation all by itself.

```
camera_0_conversion_matrix = [[0, 1, 0, -focal_distance_to_origin_mm], [-1, 0,
0, 0], [0, 0, 1, 0]]
camera_0_conversion_matrix = np.array(camera_0_conversion_matrix)
pixel_camera_location_mm = np.dot(camera_0_conversion_matrix,
np.append(world_pixel_location_mm, 1))
```

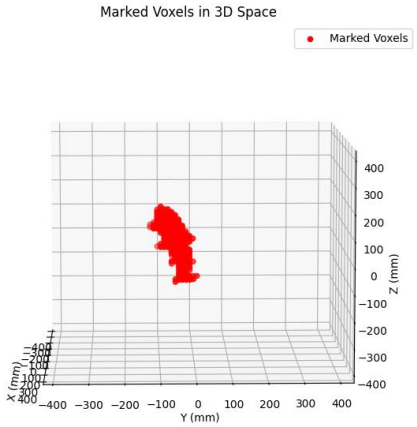
The last 1 that is added to the pixel coordinates vector is to deal with the translation. As you can see, the focal distance to the origin is added as the last computation.

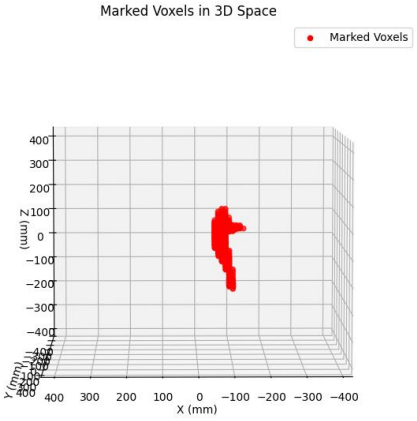
Similarly, I created a conversion matrix to convert the pixel coordinate into the corresponding image plane pixel coordinates. This deals with the projection and the triangle similarity calculations.

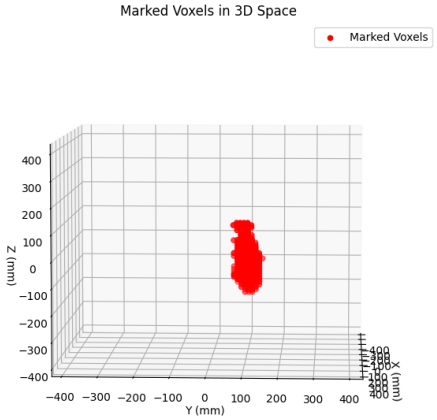
```
    image_plane_conversion_matrix = [[0, 0,
similarity_ratio*mm_to_pixel_coefficient, 960], [0,
similarity_ratio*mm_to_pixel_coefficient, 0, 600]]
    image_plane_conversion_matrix = np.array(image_plane_conversion_matrix)
    sensor_projection = np.round(np.dot(image_plane_conversion_matrix,
np.append(pixel_camera_location_mm, 1)))
```

You can see that the coordinates are multiplied with the similarity ratio and the coefficient automatically. This also handles the addition of 960 and 600 to move the origin.

Results







Above, you can see images that show what the 3D plot looks like. These plots correspond to fish1, fish2, and fish3 respectively. I tried to get the screenshots from the best angles possible. You can run the code yourself to look at them from different angles.

```
Volume of Fish 1 (cm^3): 636.69375
Volume of Fish 2 (cm^3): 575.339625
Volume of Fish 3 (cm^3): 606.5955
```

The volumes of the fish can be seen above. Due to the different orientations of the limbs, the volumes might be calculated differently. However, we can see that the results are pretty close. Note that the calculated volumes might not be exact since this is just an approximation. We can make it more accurate by decreasing the voxel size but as this will increase the number of voxels quadratically, it will affect the execution time a lot.

Conclusion

I believe my program meets the requirements of the projects as it correctly identifies the fish and visualizes them with a 3D plot. Before I conclude the report, I will talk about some possible optimizations on the algorithm. In this algorithm, we blindly go through all the voxels to determine if it is a part of the object. We instead determine regions of interest where the object might potentially be. If we can roughly find the boundaries that are filled by the object, we can assume that the voxels inside are also a part of the object. This way we can decrease the needed calculations drastically.