Project 4: Panoramas

Siddharth Patel

I. P 4.1: Spherical Reprojection

This part of the code, we are using blender to generate images. The images that I have generated using blender with different focal lengths are listed below.

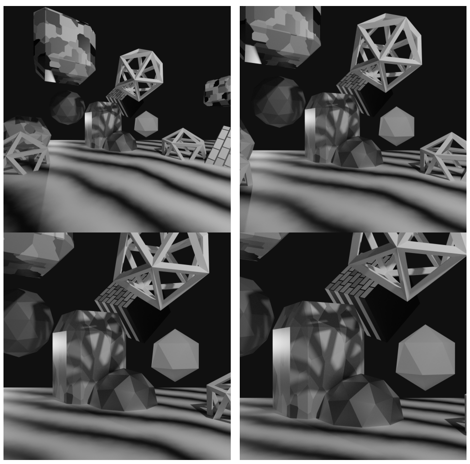

Fig. 1. Blender Images

We basically take 1 image and generate 4 different images with different focal lengths. We are making a panorama by reprojecting the images onto sphere by editing the reproject_image_to_sphere. By researching lecture slides, as asked in the jupyter lab, I got the formula from the slides which is displayed below. To convert the spherical coordinates, you have to find the unit vector for x, y, z.

x = sinθ cosϕ

y = sinϕ

z = cosθ sinϕ

To project images into a plane, the equation of x is f * x/z and y is f * y/z.

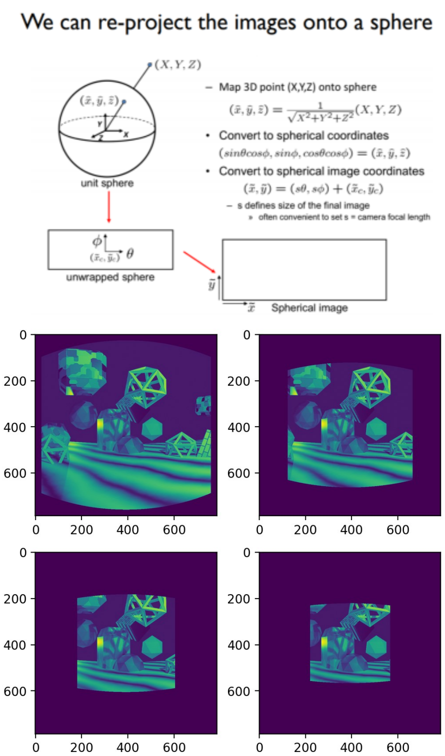

Fig. 2. The Re-projection

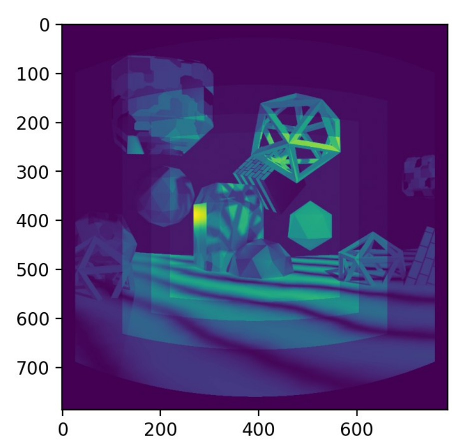

Fig. 3. Panorama

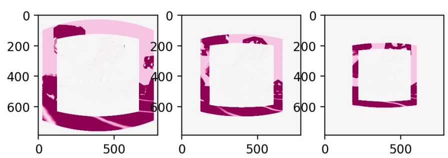

Fig. 4. Difference

II. P 4.2: Panorama Stitching

After generating all the images and re-projecting it into a sphere, you can see the panorama image generated below. By following the thorough explanation on the slides about how to stitch images together to generate a mosaic, I first found the corresponding features in a pair of image. We did feature matching in earlier projects, so the first step similar to that previous project.

In step 2, I computed the transformation between the images through the command, cv.warpPerspective, to move the images in place and by following step 3, I warped the images on top of each other so image 1 will overlay on image 2. Finally, I went ahead to blend the images, so we will not have a visual difference between the matching of the images. In shorter terms, the image will look smooth.

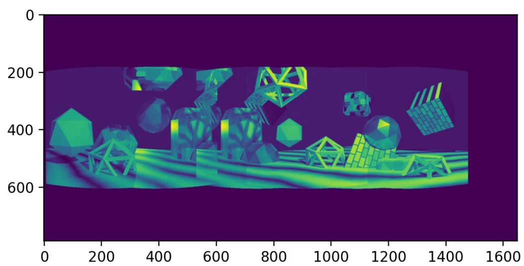

Fig. 5. Combined Image Panorama

III. P4.3.1 Projecting Into Image Space

The goal of 4.3.1 is to basically go over computing image-space point from a 3D point in a scene and the known camera matrices. We are going to apply the get_projected_point function on these two images. What we are going to do is detect the edges of the cubes as you can see below.

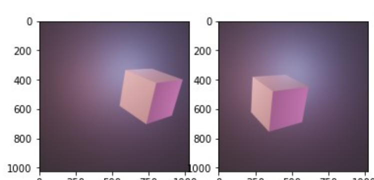

Fig. 6. Image A and B cubes

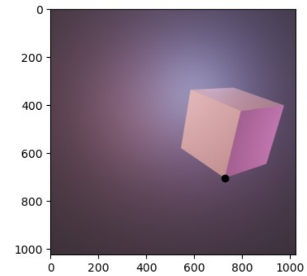

Fig. 7. Image A

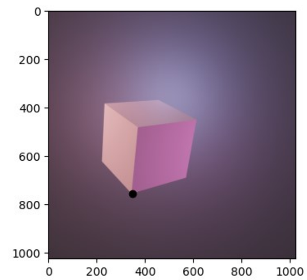

Fig. 8. Image B

A. *Question: What are the camera matrices pa and pb? I will accept either the final matrix, or the matrix written in terms of its component matrices (the intrinsic and extrinsic matrices), as long as these are defined.*

Matrix Pa is below

$$\begin{bmatrix} 1.13780e+03 & 0.00000e+00 & 5.12000e+02 & 2.56000e+03 \\ 0.00000e+00 & 1.13780e+03 & 5.12000e+02 & 2.33244e+03 \\ 0.00000e+00 & 0.00000e+00 & 1.00000e+00 & 5.00000e+00 \end{bmatrix}$$

Matrix Pb is written below

$$\begin{bmatrix} 1.1378e+03 & 0.0000e+00 & 5.1200e+02 & 8.5330e+02 \\ 0.0000e+00 & 1.1378e+03 & 5.1200e+02 & 2.5600e+03 \\ 0.0000e+00 & 0.0000e+00 & 1.0000e+00 & 5.0000e+00 \end{bmatrix}$$

IV. P4.3.2 Determining the Size of the Cube

In section 4.3.2, we will triangulate a point from two correspondences and the steps to do so is given in the slides. First I am picking the two corners of the cube and randomly making an educated guess regarding its x, y image coordinates for both the images, a and b. As you can see, I have also added the code snippet for the section to help better understand my explanation and gain full credit. Using Image A and Image B, and the variety of coordinated generated from the two images, you can see that we are sort of detecting the edges of the cube similar to what we did in previous project.

A. *Question: What is the side length of the cube shown in the two images above?*

The side length of the cube shown in the two images below is 1.1536955839418352.

$$\begin{bmatrix} 1.77195723 & 0.81604985 & 0.15646519 \\ 1.91903588 & -0.24615024 & -0.28324128 \end{bmatrix}$$

```python
import math
def solve_3Dpoint(ImagePointA, ImagePointB): #Uses triangularity to compute 3D point from two 2D image coordinates from two different cameras

    Ap1 = Pa[0,0:]
    Bp1 = Pb[0,0:]
    Ap2 = Pa[1,0:]  #Grabbing rows from camera matricies to be used in A matrix
    Bp2 = Pb[1,0:]
    Ap3 = Pa[2,0:]
    Bp3 = Pb[2,0:]

    A = np.zeros((4,4))
    b = np.zeros((4))

    A[0,0:] = np.dot(ImagePointA[1], Ap3) - Ap2
    A[1,0:] = (Ap1 - np.dot(ImagePointA[0], Ap3))
    A[2,0:] = (np.dot(ImagePointB[1], Bp3) - Bp2) #Building A matrix
    A[3,0:] = (Bp1 - np.dot(ImagePointB[0], Bp3))

    A_T_A = np.dot(A.T, A)
    eigen_values, eigen_vectors = np.linalg.eig(A_T_A)  #Compute 3D coordinate by taking eignevector with smallest eignevalue of A
    h = eigen_vectors[:, np.argmin(eigen_values)]
    h = h/h[3] #Normalize
    return h

ImgPoint1A = (903, 648)
ImgPoint1B = (572, 692)   #Manually selected 2D image points
ImgPoint2A = (975, 405)
ImgPoint2B = (613, 452)

tmp = solve_3Dpoint(ImgPoint1A, ImgPoint1B)
point1_3D = np.array([ tmp[0], tmp[1], tmp[2]]) #Find 3D point of first 2D point
solved1_3D = [point1_3D]
print(point1_3D)

tmp = solve_3Dpoint(ImgPoint2A, ImgPoint2B)
point2_3D = np.array([ tmp[0], tmp[1], tmp[2]]) #Find 3D point of second 2D point
solved2_3D = [point2_3D]
print(point2_3D)

visualize_projected_points(image_a, Pa, solved1_3D)
visualize_projected_points(image_b, Pb, solved1_3D) #Use function from part 1 to confirm our found 3D points are accurate
visualize_projected_points(image_a, Pa, solved2_3D)
visualize_projected_points(image_b, Pb, solved2_3D)

#Compute side length of cube
distanceSide1 = math.sqrt(math.pow(X0[0] - point1_3D[0], 2) + math.pow(X0[1] - point1_3D[1], 2) + math.pow(X0[2] - point1_3D[2], 2))
distanceSide2 = math.sqrt(math.pow(point1_3D[0] - point2_3D[0], 2) + math.pow(point1_3D[1] - point2_3D[1], 2) + math.pow(point1_3D[2] - point2_3D[2], 2))
SideLength = (distanceSide1 + distanceSide2)/2
print(SideLength)
```

Fig. 9. 4.3.2 Code

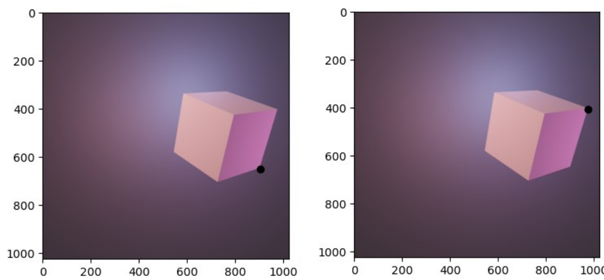

Fig. 10. Image A Point 1 and 2

V. P 4.4: Stereo Patch Matching

Using the patch match stereo, I am going to detect the features matches in the art image given to me in this section. This will create a depth map of the images.

A. Question: The possible feature matches in the second image are along the epipolar line. Since the images are properly rectified, what is the epipolar line in the second image corresponding to coordinate (x_a, y_a) in the first image?

To answer the question, I would first like to explain what epipolar line is which can be defined as the lines intersecting at the epipolar plane and the two image planes. Epipolar lines have such a property that they intersect the baseline at the respective epipoles in the image plane. If your two image planes are parallel to each other then the epipoles are located at infinity and the epipolar lines will be parallel to the x axis of each image plane. Therefore, if the image is rectified, the epipolar line will be parallel on the x and y coordinate in the first image.

B. Question: The left few columns of both depth maps is quite noisy and inaccurate. Give an explanation for why this is the case?

If you look closely, you can see that the depth map is being shifted towards the left by a few pixels. This is the main reason why the columns within the image of both, the ssd and ncc, image are quite noisy and inaccurate on the left.

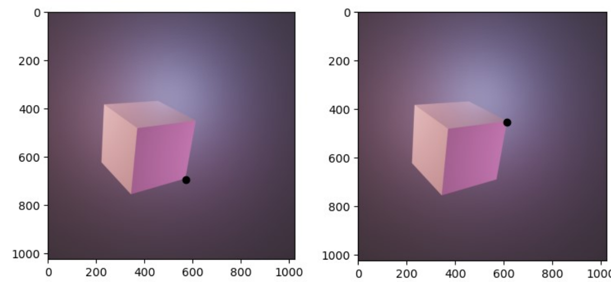

Fig. 11. Image B Point 1 and 2

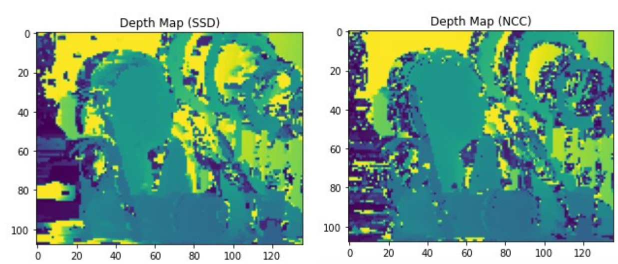

Fig. 12. Depth Map