Aidan Fischer CS 532 Homework 1 2/8/2022  
I pledge my honor that I have abided by the Stevens Honor System

**Source Code**

General

main.py – Facilitates the homework problem execution. Centralizes all function execution calls as an organizational measure.

def main():

    H = DLT()

    bilin\_interp("P1/basketball-court.ppm", H,

                 "../Output/P1/basketball-court-warped-500x940.ppm")

dolly\_effect()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

Image.py – General abstraction functions for interaction between numpy and PIL. Written myself for CS558 as a reusable plugin file for all my homework.

def get\_image(path, gray=False):

    image = Image.open(path)

    if gray:

        image = ImageOps.grayscale(image)

    return np.array(image)

def save\_image(image\_array, name):

    image = Image.fromarray(image\_array)

    image.convert("RGB").save(name)

    return

def copy\_image(image):

    return np.copy(image)

Problem 1 Files

DLT.py – Implements DLT algorithm (non-normalized) to estimate the homography matrix. The file also contains a commented-out test function I used during implementation, which is not included here.

correspondences = [

[(246, 51), (1, 1)],

    [(404, 74), (500, 1)],

    [(22, 194), (1, 940)],

    [(280, 280), (500, 940)]

]

# Calculates Ai from the correspondence index

# Assuming wi = 1, wprimei = 1

def A(index):

    thispoint = correspondences[index]

    xi = np.array([thispoint[0][0], thispoint[0][1], 1])

    xprimei = np.array([thispoint[1][0], thispoint[1][1], 1])

    return np.matrix([

        [0, 0, 0,

            -xi[0], -xi[1], -xi[2],

            xprimei[1]\*xi[0], xprimei[1]\*xi[1], xprimei[1]\*xi[2]],

        [xi[0], xi[1], xi[2],

            0, 0, 0, -xprimei[0]\*xi[0],

            -xprimei[0]\*xi[1], -xprimei[0]\*xi[2]]

    ])

# Uses hardcoded correspondences to run DLT

# algorithm and returns homography.

def DLT():

    # Combines steps 1 and 2

    Amat = np.concatenate((A(0), A(1), A(2), A(3)))

    # Step 3

    \_, \_, vh = np.linalg.svd(Amat)

    v = vh.T

    col = v.shape[1]

    h = v[:, col-1]

    # Step 4

    H = np.reshape(h, (3, 3))

    return H

BilinInterp.py – Implements bilinear interpolation algorithm, and outputs resulting image.

# Converts a point in homography form (x, y, w)^T to (x, y) form.

def homog\_to\_pt(homog\_pt):

    return np.array((homog\_pt.T/homog\_pt.T[0, 2])[0, 0:2].flat)

# Inverse warping is utilized.

def bilin\_interp(imagepath, H, outputpath):

    oldimage = get\_image(imagepath).astype(np.float64)

    bounds = oldimage.shape

    newimage = np.zeros((940, 500, 3))

    # Inverse of H used for reverse warping.

    iH = np.linalg.inv(H)

    for r in trange(940):

        for c in range(500):

            x, y = homog\_to\_pt(np.matmul(iH, np.matrix((c, r, 1)).T))

            # Since the values used in pixel calculations here start at 1

            # a negative 1 offset is necessary for array accesses.

            x -= 1

            y -= 1

            if x < 0 or x >= bounds[1] - 1 or y < 0 or y >= bounds[0] - 1:

                newimage[r, c] = [0, 0, 0]

            else:

                i = math.floor(x)

                j = math.floor(y)

                a = x - i

                b = y - j

                newimage[r, c] = (1 - a) \* (1 - b) \* oldimage[j, i] + \

                    a \* (1 - b) \* oldimage[j, i + 1] + \

                    a \* b \* oldimage[j + 1, i + 1] + \

                    (1 - a) \* b \* oldimage[j + 1, i]

    save\_image(newimage.astype(np.uint8), outputpath)

Problem 2 File

DollyEffect.py – Uses the PointCloud2Image and a modified Sample function from the provided SampleCameraPath.py file to create a video containing the dolly effect.

FPS = 15

LEN = 5

FOCAL\_Z\_INIT = 2.90

def PointCloud2Image(M, Sets3DRGB, viewport, filter\_size):

    # setting yp output image

    print("...Initializing 2D image...")

    top = viewport[0]

    left = viewport[1]

    h = viewport[2]

    w = viewport[3]

    bot = top + h + 1

    right = left + w + 1

    output\_image = np.zeros((h+1, w+1, 3))

    for counter in range(len(Sets3DRGB)):

        print("...Projecting point cloud into image plane...")

        # clear drawing area of current layer

        canvas = np.zeros((bot, right, 3))

        # segregate 3D points from color

        dataset = Sets3DRGB[counter]

        P3D = dataset[:3, :]

        color = (dataset[3:6, :]).T

        # form homogeneous 3D points (4xN)

        len\_P = len(P3D[1])

        ones = np.ones((1, len\_P))

        X = np.concatenate((P3D, ones))

        # apply (3x4) projection matrix

        x = np.matmul(M, X)

        # normalize by 3rd homogeneous coordinate

        x = np.around(np.divide(x, np.array([x[2, :], x[2, :], x[2, :]])))

        # truncate image coordinates

        x[:2, :] = np.floor(x[:2, :])

        # determine indices to image points within crop area

        i1 = x[1, :] > top

        i2 = x[0, :] > left

        i3 = x[1, :] < bot

        i4 = x[0, :] < right

        ix = np.logical\_and(i1, np.logical\_and(i2, np.logical\_and(i3, i4)))

        # make reduced copies of image points and cooresponding color

        rx = x[:, ix]

        rcolor = color[ix, :]

        for i in range(len(rx[0])):

            canvas[int(rx[1, i]), int(rx[0, i]), :] = rcolor[i, :]

        # crop canvas to desired output size

        cropped\_canvas = canvas[top:top+h+1, left:left+w+1]

        # filter individual color channels

        shape = cropped\_canvas.shape

        filtered\_cropped\_canvas = np.zeros(shape)

        print("...Running 2D filters...")

        for i in range(3):

            # max filter

            filtered\_cropped\_canvas[:, :, i] = \

                maxfilt(cropped\_canvas[:, :, i], 5)

        # get indices of pixel drawn in the current canvas

        drawn\_pixels = np.sum(filtered\_cropped\_canvas, 2)

        idx = drawn\_pixels != 0

        shape = idx.shape

        shape = (shape[0], shape[1], 3)

        idxx = np.zeros(shape, dtype=bool)

        # make a 3-channel copy of the indices

        idxx[:, :, 0] = idx

        idxx[:, :, 1] = idx

        idxx[:, :, 2] = idx

        # erase canvas drawn pixels from the output image

        output\_image[idxx] = 0

        # sum current canvas on top of output image

        output\_image = output\_image + filtered\_cropped\_canvas

    print("Done")

    return output\_image

def output\_video():

    print("\nStitching final video")

    vw = cv2.VideoWriter("../Output/P2/dolly.wmv", 0, 15, (3073, 2049))

    for step in range(FPS \* LEN):

        vw.write(cv2.imread("P2/RawFrames/{}.jpg".format(step)))

# Modified version of the sample function provided, with dolly

# zoom effect included.

def dolly\_effect():

    # load object file to retrieve data

    file\_p = open("P2/data.obj", 'rb')

    camera\_objs = pickle.load(file\_p)

    # extract objects from object array

    crop\_region = camera\_objs[0].flatten()

    filter\_size = camera\_objs[1].flatten()

    K = camera\_objs[2]

    ForegroundPointCloudRGB = camera\_objs[3]

    BackgroundPointCloudRGB = camera\_objs[4]

    # create variables for computation

    data3DC = (BackgroundPointCloudRGB, ForegroundPointCloudRGB)

    R = np.identity(3)

    move = np.array([0, 0, -2/(FPS \* LEN)]).reshape((3, 1))

    for step in range(FPS \* LEN):

        fname = "P2/RawFrames/{}.jpg".format(step)

        print("\nGenerating {}".format(fname))

        t = step\*move

        t[2] -= 1.25

        if step != 0:

            K[0, 0] \*= (FOCAL\_Z\_INIT - 2 / (FPS \* LEN) \* step) / \

                (FOCAL\_Z\_INIT - 2/(FPS \* LEN) \* (step - 1))

            K[1, 1] \*= (FOCAL\_Z\_INIT - 2 / (FPS \* LEN) \* step) / \

                (FOCAL\_Z\_INIT - 2/(FPS \* LEN) \* (step - 1))

        M = np.matmul(K, (np.hstack((R, t))))

        img = PointCloud2Image(M, data3DC, crop\_region, filter\_size)

        # Convert image values form (0-1) to (0-255)

        # and cahnge type from float64 to float32

        img = 255\*(np.array(img, dtype=np.float32))

        # convert image from RGB to BGR for OpenCV

        img\_bgr = cv2.cvtColor(img, cv2.COLOR\_RGB2BGR)

        # write image to file 'fname'

        cv2.imwrite(fname, img\_bgr)

    output\_video()

**Problem 1**

Points used in the computation

(246, 51), (404, 74), (22, 194), (280, 280)

Output Images

basketball-court-warped-500x940.ppm (assumes specified dimensions were given as height by width)

A picture containing text

Description automatically generated

basketball-court-warped-900x540.ppm (assumed specified dimensions were given as width by height)

A picture containing text, electronics

Description automatically generated

**Further Explanations**

* In problem 1, I wasn’t sure if 940 x 500 as an output size was width by height or height by width, so I included both. The version included in the final source code outputs assuming height by width, which I think is what was intended. Normally, image sizes are specified width by height, contrasted by matrix dimensions which are provided usually as height by width.
* For problem 2, the focal length adjustment was done according the math of pinpoint cameras. Since (x,y) = f/Z(X,Y), if the camera moves, then Z changes to Z’. In order for (x,y) to remain unchanged, then we need to know Z, Z’, and f, in which case the relationship to get the new focal length f’ is f’/Z’ = f/Z -> f’ = fZ’/Z.