Homework 2

Team Members:

- Atharv Subhekar
- Prasad Naik

Problem 1

Code:

```
# ========== rank transform function ==========
def rankTransform(inputImage, windowSize = 5):
  """
    input:
        - input image
        - window size

    function:
        - computes rank transform on the input image

    returns:
        - rank transformed image
  """

  inputImage = cv2.cvtColor(inputImage, cv2.COLOR_BGR2GRAY)
  rankTransform = np.zeros(inputImage.shape)

  # looping through image to compute rank transform
  for i in range(inputImage.shape[0] - windowSize):
    for j in range(inputImage.shape[1] - windowSize):
        # extracting windows from input image
      currentWindow = inputImage[i: i + windowSize, j: j + windowSize]
      currentWindowRavel = currentWindow.ravel()
      # computing rank of center element of the window
      rank = np.where(currentWindowRavel < currentWindowRavel[(windowSize**2) //
2])[0].shape[0]
      rankTransform[i + 2, j + 2] = rank
```

```
  return rankTransform

# ========== disparity map function ==========
def computeDisparityMap(leftImage, rightImage, windowSize = 15):
  """
  input:
      - left teddy image
      - right teddy image
      - window size

  function:
      - computes rank transform on the input images using the rankTransform function
      - computes disparity map using the rank transformed images
      - disparity range is predefined from 0 to 63

  returns:
      - disparity map
  """

  # computing rank transform of images
  print("\nComputing Rank Transforms of Left and Right Images")
  leftImageRankTransform = rankTransform(leftImage, 5)
  rightImageRankTransform = rankTransform(rightImage, 5)
  print("Computing Rank Transforms of Left and Right Images...Done")

  # allocating memory for disparity map
  disparityMap = np.zeros(leftImageRankTransform.shape)

  print("\nComputing Disparity Map...This may take a while")
  # looping through the images with the given window size to compute disparity map
  for i in range(disparityMap.shape[0]):
    for j in range(disparityMap.shape[1]):
      # dispVal stores the disparity value. maxDisparity stores the highest disparity
recorded yet
      dispVal = -1
```

```
      maxDisparity = 99999
      for d in range(64):
        # creating windows
        currentWindowLeft = leftImageRankTransform[i: i + windowSize, j + d: j + d +
windowSize]
        currentWindowRight = rightImageRankTransform[i: i + windowSize, j: j +
windowSize]

        # taking sum of absolute differences in left and right image windows
        if currentWindowLeft.shape[1] == currentWindowRight.shape[1]:
          absoluteVal = abs(currentWindowLeft - currentWindowRight).sum()
        else:
          x = np.zeros((currentWindowRight.shape[0], abs(currentWindowLeft.shape[1] -
currentWindowRight.shape[1])))
          currentWindowLeft = np.append(currentWindowLeft, x, axis = 1)
          absoluteVal = abs(currentWindowLeft - currentWindowRight).sum()

        # updating disparity parameters and assigning values to disparity map
        if absoluteVal < maxDisparity:
          dispVal = d
          maxDisparity = absoluteVal
      disparityMap[i, j] = dispVal

  print("Computing Disparity Map...Done")
  return disparityMap

# ========== error rate function ==========
def errorRate(groundTruth, predictedMap):
  """
  input:
      - ground truth disparity map
      - predicted disparity map

  function:
      - loops through the two images
```

```
      - divides the ground truth value at each pixel by 4 and compares
      to the corresponding value in the predicted disparity map
      - if the difference is greater than 1, then records an error
      - computes error percentage

    returns:
      - error percentage
    """
    groundTruth = cv2.cvtColor(groundTruth, cv2.COLOR_BGR2GRAY)
    errorPixelCount = 0

    # calculating error rate between ground truth and predicted disparity map
    for i in range(groundTruth.shape[0]):
      for j in range(groundTruth.shape[1]):
        if abs(round(groundTruth[i, j] / 4) - predictedMap[i, j]) > 1:
          errorPixelCount += 1

    # calculating error percentage
    error = (errorPixelCount / (groundTruth.shape[0] * groundTruth.shape[1])) * 100
    return error

# ========== main function ==========
def main():
    # reading images
    teddyLeft = cv2.imread('teddy/teddyL.pgm')
    teddyRight = cv2.imread('teddy/teddyR.pgm')
    groundTruth = cv2.imread('teddy/disp2.pgm')

    # computing disparity maps and error rate (3 x 3)
    print("\n===== Computing Disparity Map and Error Rate (3 x 3
Window)...Please Wait ======")

    mapDisparity3 = computeDisparityMap(teddyLeft, teddyRight, 3)

    print("\nSaving Disparity (3 x 3) Map | Filename: win3Disparity.pgm")
```

```
    cv2.imwrite('win3Disparity.pgm', mapDisparity3)
    print("File Saving...Done")

    print("\nComputing Error (3 x 3 Disparity Map)")
    error3 = errorRate(groundTruth, mapDisparity3)
    print(f"Error Rate (3 x 3 Disparity Map): {error3}")

    print("\n===== Computing Disparity Map and Error Rate (3 x 3 Window)...Done ======")

    # computing disparity maps and error rate (15 x 15)
    print("\n===== Computing Disparity Map and Error Rate (15 x 15 Window)...Please Wait =====")
    mapDisparity15 = computeDisparityMap(teddyLeft, teddyRight, 15)

    print("\nSaving Disparity (15 x 15) Map | Filename: win15Disparity.pgm")
    cv2.imwrite('win15Disparity.pgm', mapDisparity15)
    print("File Saving...Done")

    print("\nComputing Error (15 x 15 Disparity Map)")
    error15 = errorRate(groundTruth, mapDisparity15)
    print(f"Error Rate (15 x 15 Disparity Map): {error15}")

    print("\n===== Computing Disparity Map and Error Rate (15 x 15 Window)...Done =====")

if __name__ == "__main__":
    main()
```

Results:

Figure 1 Disparity Map (3 x 3 Window)

Figure 2 Disparity Map (15 x 15 window)

Error Rates:

Disparity Map (3 x 3 Window): 64.8622

Disparity Map (15 x 15 Window): 51.2035

Problem 2

Code:

```
"""
Authors:
    - Atharv Subhekar (CWID: 20015840)
    - Prasad Naik (CWID: 20016345)
"""
# ========== importing libraries ==========
import numpy as np
import pickle
import time
import sys
import cv2
#from scipy.signal import medfilt
#from scipy.ndimage.filters import maximum_filter as maxfilt

# ========== point cloud to image function ==========
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(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 + cropped_canvas

print("Done")
return output_image
```

```
# ========== camera intrinsic scale function ==========
def scale_intrinsics(K, scale = 1.0):
    """
    input:
        - Camera intrinsic matrix
        - Scale Factor

    function:
        - scales the camera instrinsic parameters

    returns:
        - returns scales camera instrinsic matrix
    """
    K[0, 0] = K[0, 0] * scale
    K[1,1] = K[1, 1] * scale
    K[0, 2] = K[0, 2] * scale
    K[1, 2] = K[1, 2] * scale

    return K

# ========== image rendering function ==========
def SampleCameraPath():
    """
    function:
```

```
    - configures intrinsic and extrinsic parameters to render
    images and computes disparity maps

returns:
    - saves disparity maps
"""
# load object file to retrieve data
file_p = open("data/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]

# parameters
scale = 1/4
disparities = []
imgArray = []

# scale K to desired width/height
K = scale_intrinsics(K, scale = scale)
f = K[0, 0]
```

```
crop_region[2] = crop_region[2] * scale
crop_region[3] = crop_region[3] * scale

# create variables for computation
data3DC = (BackgroundPointCloudRGB,ForegroundPointCloudRGB)
R = np.identity(3)
move = np.array([-0.1, 0, 0]).reshape((3,1))

# get average foreground Z distance (should be ~4m + zoom)
Z_avg = np.mean(ForegroundPointCloudRGB[2,:]) + move[0, 0]

for step in range(8):
    tic = time.time()

    fname = "SampleOutput{}.jpg".format(step)
    print("\nGenerating {}".format(fname))

    # configuring extrinsic parameters
    t = step * move
    M = np.matmul(K,(np.hstack((R,t))))

    # updating intrinsic parameters
    K[0, 2] += t[0,0]
    K[1, 2] += t[0,0]
```

```
# rendering images
img = PointCloud2Image(M,data3DC,crop_region,filter_size)


# calculating disparity
disparities.append((abs(t[0, 0]) * K[0, 0]) / Z_avg)


# 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 RGB to GrayScale
imgGray = np.zeros((img.shape[0], img.shape[1]))
for i in range(imgGray.shape[0]):
    for j in range(imgGray.shape[1]):
        imgGray[i, j] = (0.6 * img[i, j, 0]) + (0.59 * img[i, j, 1]) + (0.11 * img[i, j,
2])


# write image to file 'fname'
cv2.imwrite(fname,imgGray)
imgArray.append(imgGray)


toc = time.time()
toc = toc-tic
print("{0:.4g} s".format(toc))
```

```
    for i in range(1, len(imgArray)):
        disparityMap = computeDisparityMap(imgArray[0], imgArray[1], 15,
int(disparities[i]))
        cv2.imwrite(f"DisparityMap{i}.jpg", disparityMap**2)


# ========== rank transform function ==========
def rankTransform(inputImage, windowSize = 5):
  """
    input:
        - input image
        - window size


    function:
        - computes rank transform on the input image


    returns:
        - rank transformed image
  """


  #inputImage = cv2.cvtColor(inputImage, cv2.COLOR_BGR2GRAY)
  rankTransform = np.zeros(inputImage.shape)


  # looping through image to compute rank transform
  for i in range(inputImage.shape[0] - windowSize):
    for j in range(inputImage.shape[1] - windowSize):
```

```
            # extracting windows from input image
            currentWindow = inputImage[i: i + windowSize, j: j + windowSize]
            currentWindowRavel = currentWindow.ravel()
            # computing rank of center element of the window
            rank = np.where(currentWindowRavel < currentWindowRavel[(windowSize**2) //
2])[0].shape[0]
            rankTransform[i + 2, j + 2] = rank

    return rankTransform


# ========== disparity map function ==========
def computeDisparityMap(leftImage, rightImage, windowSize = 15, disparityRange = 0):
    """
    input:
        - left teddy image
        - right teddy image
        - window size

    function:
        - computes rank transform on the input images using the rankTransform function
        - computes disparity map using the rank transformed images
        - disparity range is predefined from 0 to 63

    returns:
        - disparity map
```

```
"""

# computing rank transform of images
print("\nComputing Rank Transforms of Left and Right Images")
leftImageRankTransform = rankTransform(leftImage, 5)
rightImageRankTransform = rankTransform(rightImage, 5)
print("Computing Rank Transforms of Left and Right Images...Done")

# allocating memory for disparity map
disparityMap = np.zeros(leftImageRankTransform.shape)

print(f"\nComputing Disparity Map (Disparity Range = {disparityRange})...This may take a while")
# looping through the images with the given window size to compute disparity map
for i in range(disparityMap.shape[0]):
  for j in range(disparityMap.shape[1]):
    # dispVal stores the disparity value. maxDisparity stores the highest disparity recorded yet
    dispVal = -1
    maxDisparity = 99999
    for d in range(disparityRange):
      # creating windows
      currentWindowLeft = leftImageRankTransform[i: i + windowSize, j + d: j + d + windowSize]
      currentWindowRight = rightImageRankTransform[i: i + windowSize, j: j + windowSize]
```

```
        # taking sum of absolute differences in left and right image windows
        if currentWindowLeft.shape[1] == currentWindowRight.shape[1]:
          absoluteVal = abs(currentWindowLeft - currentWindowRight).sum()
        else:
          x = np.zeros((currentWindowRight.shape[0], abs(currentWindowLeft.shape[1] -
currentWindowRight.shape[1])))
          currentWindowLeft = np.append(currentWindowLeft, x, axis = 1)
          absoluteVal = abs(currentWindowLeft - currentWindowRight).sum()

        # updating disparity parameters and assigning values to disparity map
        if absoluteVal < maxDisparity:
          dispVal = d
          maxDisparity = absoluteVal
      disparityMap[i, j] = dispVal

  print("Computing Disparity Map...Done")
  return disparityMap

# ========== error rate function ==========
def errorRate(groundTruth, predictedMap):
  """
  input:
      - ground truth disparity map
      - predicted disparity map
```

```
    function:
        - loops through the two images
        - divides the ground truth value at each pixel by 4 and compares
        to the corresponding value in the predicted disparity map
        - if the difference is greater than 1, then records an error
        - computes error percentage

    returns:
        - error percentage
    """
    groundTruth = cv2.cvtColor(groundTruth, cv2.COLOR_BGR2GRAY)
    errorPixelCount = 0

    # calculating error rate between ground truth and predicted disparity map
    for i in range(groundTruth.shape[0]):
      for j in range(groundTruth.shape[1]):
        if abs(round(groundTruth[i, j] / 4) - predictedMap[i, j]) > 1:
          errorPixelCount += 1

    # calculating error percentage
    error = (errorPixelCount / (groundTruth.shape[0] * groundTruth.shape[1])) * 100
    return error


# ========== main function ==========
```

```
def main():

    # reading images

    teddyLeft = cv2.imread('SampleOutput0.jpg')

    teddyRight = cv2.imread('SampleOutput1.jpg')

    #groundTruth = cv2.imread('teddy/disp2.pgm')


    # computing disparity maps and error rate (3 x 3)

    #print("\n===== Computing Disparity Map and Error Rate (3 x 3
Window)...Please Wait ======")


    #mapDisparity3 = computeDisparityMap(teddyLeft, teddyRight, 3)


    #print("\nSaving Disparity (3 x 3) Map | Filename: win3Disparity.pgm")

    #cv2.imwrite('win3Disparity.pgm', mapDisparity3)

    #print("File Saving...Done")


    #print("\nComputing Error (3 x 3 Disparity Map)")

    #error3 = errorRate(groundTruth, mapDisparity3)

    #print(f"Error Rate (3 x 3 Disparity Map): {error3}")


    print("\n===== Computing Disparity Map and Error Rate (3 x 3 Window)...Done
======")


    # computing disparity maps and error rate (15 x 15)

    print("\n===== Computing Disparity Map and Error Rate (15 x 15
Window)...Please Wait =====")
```

```
    mapDisparity15 = computeDisparityMap(teddyLeft, teddyRight, 15)


    print("\nSaving Disparity (15 x 15) Map | Filename: win15Disparity.pgm")
    cv2.imwrite('win15Disparity.pgm', mapDisparity15)
    print("File Saving...Done")


    #print("\nComputing Error (15 x 15 Disparity Map)")
    #error15 = errorRate(groundTruth, mapDisparity15)
    #print(f"Error Rate (15 x 15 Disparity Map): {error15}")


    print("\n===== Computing Disparity Map and Error Rate (15 x 15 Window)...Done =====")


def main():
    SampleCameraPath()


if __name__ == "__main__":
    main()
```

Analysis:

- We render the images by translating the camera center to the right by 0.1 units. Hence, our stereo baseline is 0.1.
- Using this baseline, focal length from the camera intrinsic matrix, and the average Z value from the foreground point cloud information, we compute the disparity between the first rendered image and each of the other rendered images and compute the disparity maps between these pairs of images.
- The disparity ranges obtained for each pair are mentioned along with the maps below.

Results:

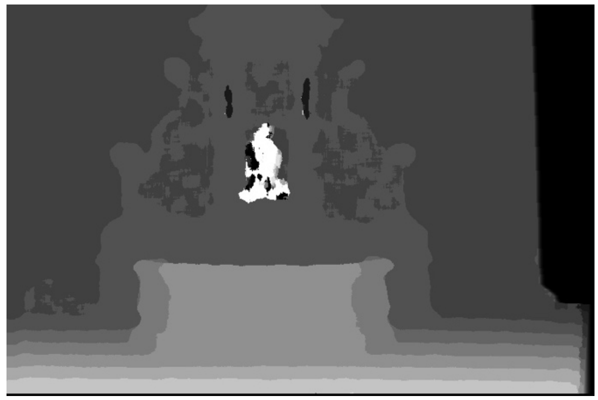

Figure 3 Disparity Range = 0 - 17

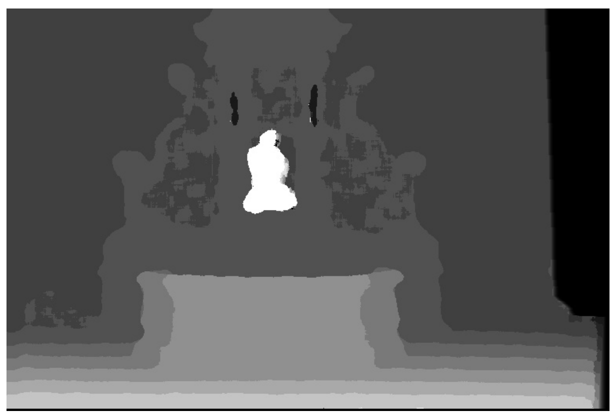

Figure 4 Disparity Range = 0 - 35

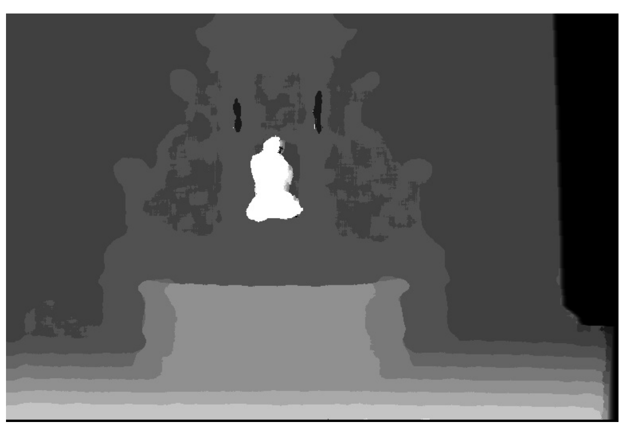

Figure 5 Disparity Range = 0 - 53

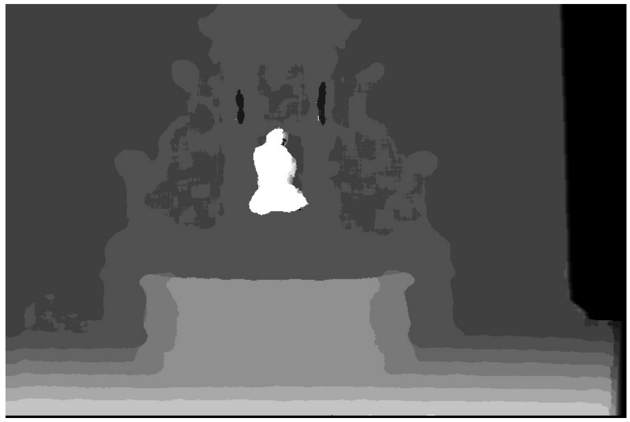

Figure 6 Disparity Range = 0 - 70

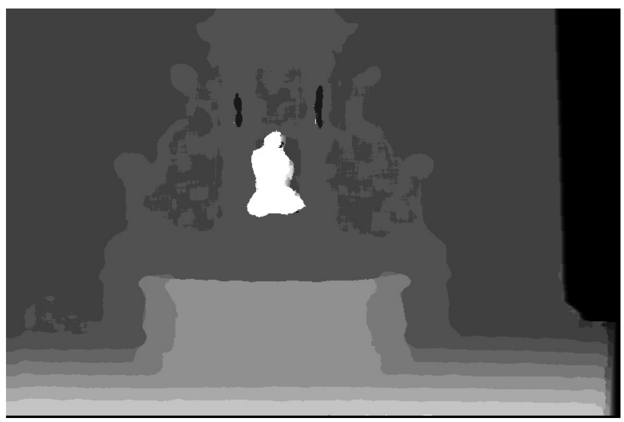

Figure 7 Disparity Range = 0 - 88

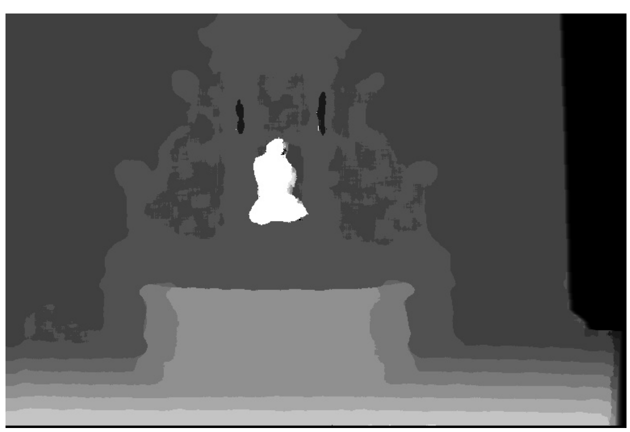

Figure 8 Disparity Range = 0 - 108

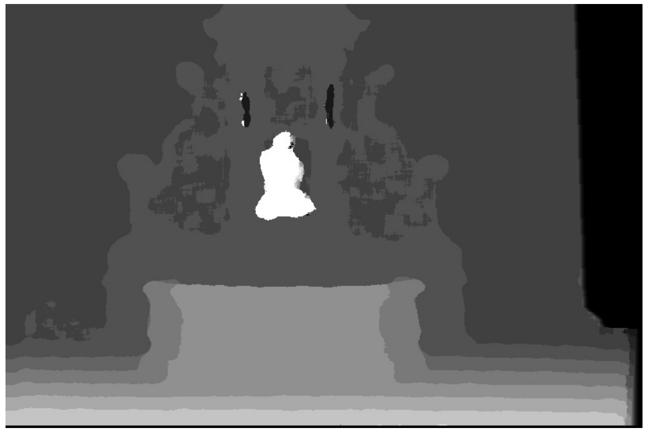

Figure 9 Disparity Range = 0 - 123