3D Object Reconstruction from Multiple 2D Images

1. Abstract

In this project, we are going to perform a process which recovers 3D objects from multiple 2D images that are taken from different angles. This process is also known as 3D reconstruction. This project includes a total of three different pipelines to achieve this goal. The main differences of these pipelines are the methods to compute the depth maps, and will be discussed later in this report. Moreover, we also implemented multiple approaches for some key steps. Therefore, this report will also contain a section to introduce the methodologies that are used, as well as explanation to the experiments that are performed. Finally, a conclusions section will also be included to compare the results of three pipelines.

2. Introduction

Background

In daily life, people like to use cameras to record the scenes or objects in the 3D world. The process inside cameras can be treated as project 3D objects onto a 2D image plane with the depth information lost. On the contrary, 3D reconstruction is the process of reconstructing 3D objects from a group of 2D images that are taken from multiple viewpoints. Basically, it reverts the operation that takes place inside cameras. The 3D reconstruction process projects each individual 2D image back into 3D world coordinates system and merges them together to generate the final output. Specifically, this problem is the process of:

1. Use pre-trained models or specialized algorithms to compute the depth map of an image.
2. Get point clouds for each image and merge them by algorithms, for instance, ICP and rigid body transformation.
3. Assign color to each point cloud in the merged point cloud.

By completion of these steps, a 3D representation of the input object should be generated.

DataSets

This report uses one dataset, which is PASMVS. PASMVS is a comprehensive multi-view stereopsis dataset that consists of accurate synthetic images for four different objects: Teapot, Dragon, Bunny and Armadillo. For each object described, the dataset provides 10 different finishes with “two camera focal lengths, four 3D models of varying geometrical complexity, five high definition, high dynamic range (HDR) environmental textures to replicate photorealistic lighting conditions” (André Broekman, Petrus Johannes Gräbe) rendered from 45 different viewpoints. Moreover, each image also has a text file specifying the intrinsic and extrinsic matrices, which makes it possible to make accurate point clouds . This dataset also includes pre-computed depth maps encoded as pmf files and a function to read these depth maps. We are going to use them to evaluate the performance of the depth maps generated by us.

The PASMVS dataset can be downloaded at https://data.mendeley.com/datasets/fhzfnwsnzf/2.

Literature Review

First pipeline

1. For the model selection, we referred to the error table listed in the following website: https://github.com/ialhashim/DenseDepth. This chart clearly indicates that the model trained by DenseDapth dataset has the lowest error rate and highest accuracy among all other datasets, such as Eigen et al., Laina et al. and MS-CRF. The following benchmarks: https://paperswithcode.com/task/depth-estimation also proved that the DenseDepth model best fits our demand.
2. On the following wikipedia page,https://en.wikipedia.org/wiki/Point_set_registration we find that we can use Iterative Closest Point(ICP) and Rigid body Transformation to find a spatial transformation to align two point clouds. There are also many registration algorithms such as Robust point matching, Robust registration and so on. We use ICP and rigid body transformation since they are the classic and basic algorithms to align point clouds.
3. We implement ICP based on the algorithm step on the following webpage: https://en.wikipedia.org/wiki/Iterative_closest_point.This webpage provides a clear English explanition of the steps for ICP algorithm, which gives us a good start.
4. We find Least-Squares Fitting of two point sets by reading the following paper: "*Least-Squares Fitting of Two 3-D Point Sets*", Arun, K. S. and Huang, T. S. and Blostein, S. D, IEEE Transactions on Pattern Analysis and Machine Intelligence, Volume 9 Issue 5, May 1987
5. We implement 3D rigid body transformation based on http://nghiaho.com/?page_id=671. This website provides the most intuitive introduction and mathematical derivation of the rigid body transformation among all sources that we searched. This website goes through the process of rigid body transformation step by step and makes us easy to follow.

Second pipeline

1. We implement the stereo parallel version based on the knowledge learned in class. This pipeline also provides algorithms for generating depth maps after the images are transformed to parallel versions.

Third pipeline

1. When we computed the fundamental matrix, we referred to the textbook: "*Multiple View Geometry in computer vision, by Richard Hartley and Andrew*" Zisserman. There is a more detailed method to find the foundation matrix using the eight-point algorithm and implement the normalized eight-point algorithm to normalize the Fundamental matrix.
   Furthermore, we found another algorithm that can normalize a fundamental matrix named the algebraic minimization algorithm, and wanted to make a comparison with the normalized eight-point algorithm. However, we did not

find other detailed resources on the Internet, so in the end we did not implement this algorithm.

2. Moreover, we implement ideas from a course note: "*Epipolar Geometry of Stanford University*" that use fundamental matrix to compute left and right homographies, and get more detailed steps on how to implement a normalized eight-point algorithm. https://web.stanford.edu/class/cs231a/course_notes/03-epipolar-geometry.pdf
3. We also refer to how to implement image Rectification on this following website http://www.sci.utah.edu/~gerig/CS6320-S2012/Materials/CS6320-CV-F2012-Rectification.pdf

3. Methodology, Results, and Experiments

First pipeline

Compute Depth Map Using Pre-Trained Model

Theoretical analysis

In the first pipeline, we used a pre-trained model to compute the depth map of a given image. The model chosen is called the DenseDepth model which was trained based on NYU Depth V2 and KITTI data、 sets. Both data include very detailed data in a certain scene. Specifically, the NYU Depth V2 is a data set that contains a variety of indoor scenes. It uses Microsoft Kinect camera to obtain the depth information of the scene. The KITTI dataset is obtained from a well-equipped vehicle. This vehicle has two high resolution colour and grayscale cameras, as well as a laser scanner and GPS localization system. Compared to NYU Depth V2, this dataset contains mainly outdoor scenes.

```
def get_depth(model, img, savePath):
    """
    Generate the depth map of img using pre-trained model.
    :param model: pre-trained model want to use
    :param img: input image
    :param savePath: the file name you wish to save the output file
    """
    # load pre-trained model
    # custom layer BilinearUpSampling2D
    model = load_model(model, custom_objects={'BilinearUpSampling2D': BilinearUpSampling2D, 'depth_loss_function': None}, compile=False)

    # read input image
    im = cv2.imread(img)
    img_read = resize(im, (im.shape[0], im.shape[1]), anti_aliasing=False)

    # use model to predict depth of every pixel
    predict = model.predict(img_read.reshape((1, img_read.shape[0], img_read.shape[1], img_read.shape[2])), batch_size=2)
    # (1 * 288 * 384 * 1) -> (1 * 288 * 384)
    outputs = np.squeeze(1000 / predict, axis=3)

    oriImg = cv2.resize(cv2.imread(img), (im.shape[1] // 2, im.shape[0] // 2))

    # save outputs
    cv2.imwrite(savePath + ' InputImg.png', oriImg)
    cv2.imwrite(savePath + ' DepthImg.png', outputs[0])
```

The above function implements the DenseDepth model approach of computing depth maps. It calls the load_model and predict function from Keras library and saves the outputs to the given file location.

Empirical work and Presentation of Results

To compare the performance of the DenseDepth model using NYU Depth V2 dataset and Kitti dataset, we performed two separate experiments: one on an indoor object scene and the other on an outdoor scene.



Input Image | Kitti Depth Map | NYU Depth Map

Indoor



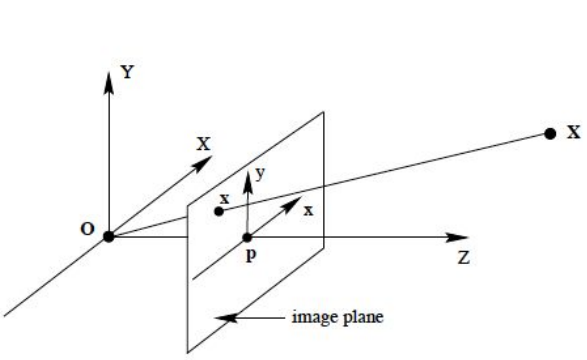
Input Image | Kitti Depth Map | NYU Depth Map

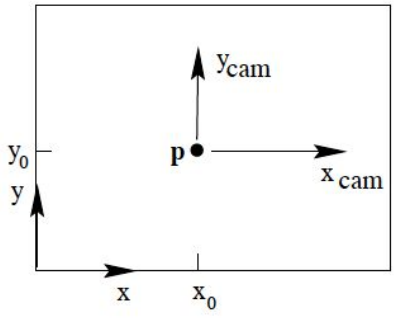
Outdoor

The results are in line with expectations. The NYU model is able to capture more details in the indoor scenes than the Kitti model, and the Kitti model has better performance in outdoor scenes. For example, we labeled one region which should be “infinitely far” from the camera location for both scenes with red circles. Thus, the labeled regions in the depth map should have brighter color. We find that the NYU model and the Kitti model behave more correctly in the indoor scene and the outdoor scene correspondingly.

The github repo of DenseDepth model: https://github.com/ialhashim/DenseDepth

Compute point cloud based on depth map and rgb image





Theoretical analysis

First, we use the reverse process of Perspective Projection to map each pixel in the 2D image plane into 3D points in the camera coordinate system. There is one thing

we need to pay attention to. In the camera coordinate system,the left bottom of the 2D image has coordinates of (x=0,y=0,f), where f is the focal length. And the principle point p has coordinates of (x=px,y=py,f). The positive directions of x-axis and y-axis in the camera coordinates system are left and up accordingly. (As shown in the graph) However, in our traditional way of reading an image, we start at the top left corner of the image, where i =0 and j=0. And the positive directions of x-axis and y-axis are down and right respectively. Thus, the relationship of (i, j) and (x, y) is x = j, y = num_rows-1-i.

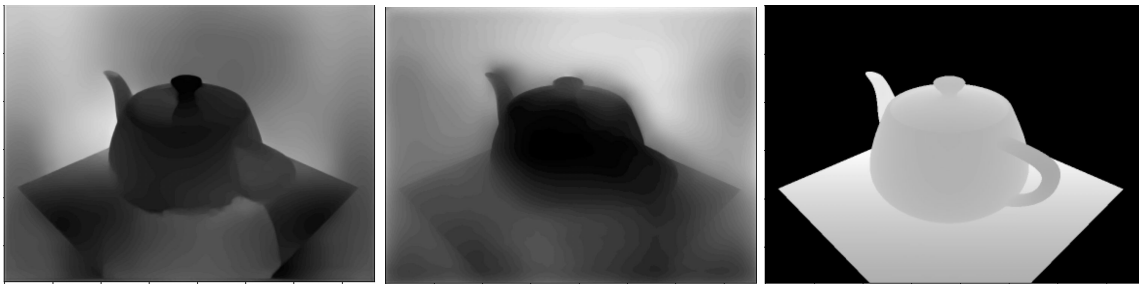
Then we turn the 3D points in the camera coordinates system into the world coordinates system by applying the inverse of extrinsics matrix. The extrinsics matrix for our dataset is a homogenous matrix with shape of 4x4.

```
def compute_point_cloud(imageNumber):
  '''
  Return a N x 6 matrix where N is the number of 3D points with 3
  coordinates and 3 color channel values given depth map, rgb,
  extrinsics and intrinsics of a 2D image.
  '''
  depth, rgb, extrinsics, intrinsics = get_data(imageNumber)
  #Extract RBG channels of rgb image each has shape (rows,cols)
  Red = rgb[:,:,0]
  Green = rgb[:,:,1]
  Blue = rgb[:,:,2]

  rows,cols = depth.shape
  N = rows*cols
  print('number of points is,',rows,cols)
  results = np.zeros((N,6))
  n=0

  for i in range(rows):
    for j in range(cols):
      #pixel coordinates of 2D image plane
      #we switch x and y since origin of image plane is at bottom right
      d = depth[i,j]
      x = j*d
      y = (rows-1-i)*d
      left = np.array([[x],[y],[d]])
      #Find the camera coordinates: check a = Zc(yes)
      camera = np.linalg.inv(intrinsics)@left #shape is (3,1)
      homo_camera = np.array([[camera[0,0]],[camera[1,0]],[camera[2,0]],[1]])
      #Find the world coordinates
      #extrinsics is homogenous extrinsics(4x4)
      world_coord = np.linalg.inv(extrinsics)@homo_camera
      #store in results list
      results[n,0] = world_coord[0,0]
      results[n,1] = world_coord[1,0]
      results[n,2] = -world_coord[2,0]
      #store color channel values
      results[n,3] = Red[i,j]
      results[n,4] = Green[i,j]
      results[n,5] = Blue[i,j]
      n = n+1
  return results
```

Empirical work and Presentation of Results

To compare the quality of depth maps, we generate the ground truth depth map by the code provided in the dataset. Then we normalize the 2 depth maps we want to compare into the range of [0,255] in the following way. We compute the maximum and minimum values of the depth map, and normalize all the depth values into zero and one. Then we multiply 255 to the depth values within [0, 1] so that the original maximum depth has value of 255 at the end, and the original minimum depth has value of 0.

Then, we calculated the mean of the root of the sum of squared difference(SSD) as the error for the depth map we are testing.



Normalized Depth Map (NYU) Normalized Depth Map(Kitti) Normalized Ground truth

Model	Error(mean root of SSD)
NYU	131.8071488
Kitti	146.2901121

From the chart, we find that the depth maps generated by the NYU model are more similar to the ground truth depth maps provided by the dataset.

Input image Depth Map (NYU) Point Cloud

Merge point clouds using Iterative Closest Point Algorithm(ICP)

Theoretical part

In the Iterative Closest Point algorithm, which was first introduced by Chen and Medioni, and Besl and McKay, we fix one point cloud (reference point cloud). We want to transform the source point cloud to best match the fixed point cloud. We implement ICP based on the algorithm step on the following webpage. https://en.wikipedia.org/wiki/Iterative_closest_point

1.First, we use the KD_tree algorithm to find the closest point in the fixed point cloud for every point in the source point cloud. We use built-in functions from scipy.spatial.

```
def KDTree_matching(src, dst):
    '''
    Find the nearest (Euclidean) neighbor in dst(fixed point cloud) for each point in src
    Input:
        src: Nx3 array of points
        dst: Nx3 array of points(fixed point cloud)
    Output:
        distances: Euclidean distances of the nearest neighbor
        indices: dst indices of the nearest neighbor
    '''
    # Find the nearest points for each point in src among the points in dst
    kdtree = KDTree(dst)
    dist, indices = kdtree.query(src, 1)
    #print('dist shape is,',dist.shape) #dist shape is, (110592,)
    #print('indices shape is,', indices.shape)# indices shape is, (110592,)
    return dist, indices
```

We also find a way on stackoverflow to implement the nearest neighbour matching using built-in functions of sklearn.neighbors. By experiments, we find that using NearestNeighbors is much faster than KD_Tree.

```
def nearest_neighbor(src, dst):
    '''
    Find the nearest (Euclidean) neighbor in dst for each point in src
    Input:
        src: Nx3 array of points
        dst: Nx3 array of points
    Output:
        distances: Euclidean distances of the nearest neighbor
        indices: dst indices of the nearest neighbor
    '''
    #Borrowed method in stack-overflow
    assert src.shape == dst.shape

    neigh = NearestNeighbors(n_neighbors=1)
    neigh.fit(dst)
    distances, indices = neigh.kneighbors(src, return_distance=True)
    return distances.ravel(), indices.ravel()
```

2. Then, we use Singular Value Decomposition to generate the optimal rotation and translation matrix given 2 sets of 3D points, which are the matched points(closest

neighbours) for source point cloud and fixed point cloud. We implemented it based on the instructions on the following webpage. http://nghiaho.com/?page_id=671

```
def compute_transform(pc1,pc2):
    '''Compute the transformation(R and T) from pc1(source) to pc2 (reference fixed).
    pc1:  3xnum_points
    pc2: 3xnum_points'''
    #Find the centroids of 2 point clouds.
    centroid1 = calculate_centriod(pc1)
    centroid2 = calculate_centriod(pc2)

    #accumulating a matrix H
    pc1_centered = pc1 - centroid1 #shape 3xnum_points
    pc2_centered = pc2 - centroid2
    H = pc1_centered@np.transpose(pc2_centered) #shape: 3x3
    #Find rotation
    U, S, Vt = np.linalg.svd(H)
    R = Vt.T @ U.T
    # special reflection case
    if np.linalg.det(R) < 0:
        print("reflection occurs")
        Vt[2, :] *= -1
        R = Vt.T @ U.T
    #Translation matrix
    T = centroid2-R@centroid1 #shape:3x1
    return R, T
```

3. We use the optimal rotation matrix and translation matrix found in the previous step to transform the source point cloud. Now, the transformed source point cloud is much closer to the fixed point cloud.
Then, we iteratively find the matched points between 2 point clouds and find the best transformation until the error is less than the threshold. The error is the square root of the average of SSD. The error approximates the distance between each pair of matched points in 2 point clouds.

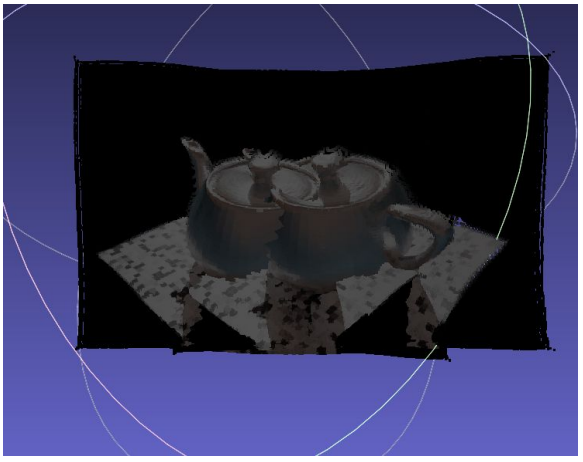
```
def icp(pc1, pc2, threshold, num_iter):
    """Return the transformed source point cloud.
    :param pc1: source point cloud(3xN)
    :param pc2: reference point cloud(fixed)
    :param threshold: Transfomation is the best when SSD less than threshold,
    :param num_iter: maximum number of iteration(in case too time consuming)
    :return: Best transformation from pc1 to pc2
    """
    num_pts = pc1.shape[1]
    iter = 0
    source_pc = pc1
    while iter<=num_iter:
        print('Number of iterations is,',iter)
        # find the nearest neighbors between the current source and destination points
        dist, indices = nearest_neighbor(source_pc.T, pc2.T)
        #find transformation which will best align each source point to its match found
        R, T = compute_transform(source_pc[:,indices.ravel()],pc2)
        new_pc = (R@ source_pc)+T
        # Find the root mean squared error(variant of Sum of squared difference(SSD))
        error = np.square(dist)
        SSD = np.sum(error)
        root_mSSD= np.sqrt(SSD/num_pts)
        print('Error is,',root_mSSD)
        if root_mSSD < threshold:
            break
        else:
            source_pc = new_pc
            #generate transformation for the next iteration
            R,T = compute_transform(source_pc,pc2)
            print("Rotation matrix is,", R)
            print("Translation matrix is,", T)
        iter+=1
    #transformed source pc
    transformed_pc = source_pc
    return R,T, transformed_pc
```

4. Finally, we render the point cloud by adding corresponding RGB values to each point in the 2 point clouds.

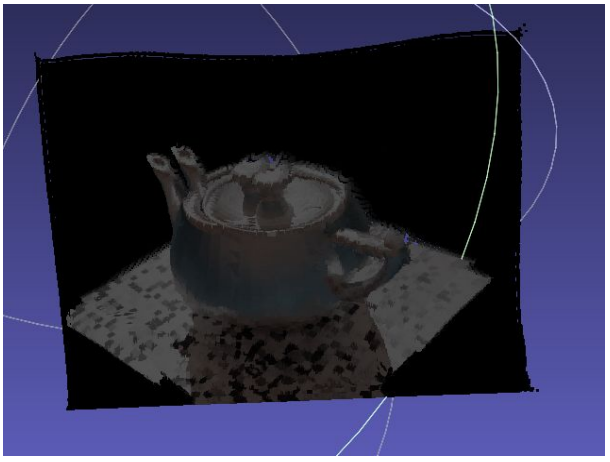
```
def Render_merged_pc(transformed_pc, pc1,pc2):
    """
    Return the merged point cloud(2Nx6) given the best transformation found by ICP.
    :param transformed_pc: source point cloud(3xN)
    :param pc1: source point cloud (Nx6)
    :param pc2: reference point cloud(fixed)(Nx6)
    """
    #Switch the xyz column of pc1 to transformed version
    pc1[:,:3] = transformed_pc.T
    merged_pc  = np.append(pc1,pc2,axis=0)
    return merged_pc
```

Empirical work and Presentation of Results/Failures

When using Nearest Neighbour to match the closest points, the error(which approximates the distance between each pair of matched points in 2 point clouds) is around 1.3 pixels.(We have tried that, with more iterations, the error is always around 1 pixels) The whole algorithm takes around 46.93 seconds to finish 100 iterations.



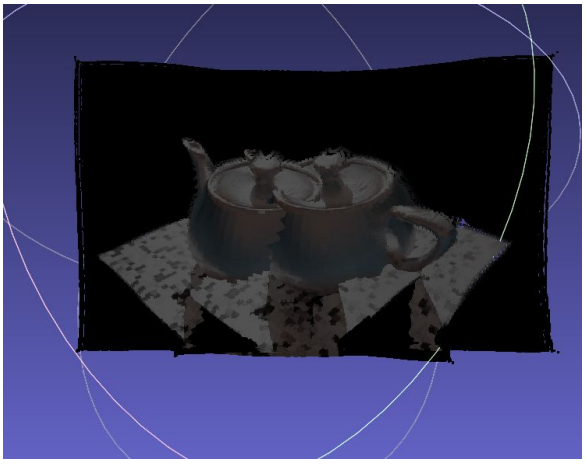
Original point clouds



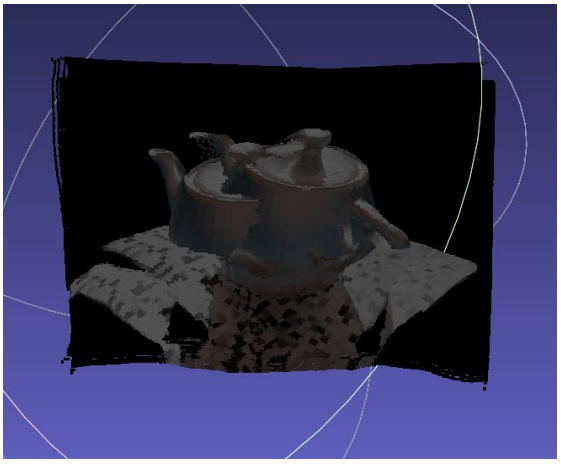
merged point(Nearest Neighbour)

```
Number of iterations is, 100
Error is, 1.370754042270549
The overall algorithm takes 46.932839564seconds.
```

When using KP_Tree to match the closest points, the error is also around 1.3 pixels. However, the quality of the merged point cloud is worse.It may need more iterations to provide a high quality merged point cloud. However, KP_Tree is too time consuming(it requires 1061 seconds to only perform 10 iterations). Therefore, we decide to use Nearest Neighbours to match the closest points.



Original point clouds



merged point(KP_tree)

```
Number of iterations is, 10
Error is, 1.370215251580734
The overall algorithm takes 1061.846858614seconds.
```

Merge Two Point Clouds Using Rigid Body Transformation

Rigid Body transformation is a geometric transformation that keeps the shape of the rigid body unchanged (The distance between any two points on the rigid body does not change).

```python
def rigidTransformation(A, B):
    """
    Match two point clouds using least square lost.

    A' = R*A.T + t, where A' should be close to B.T

    Shape of the rigid bodies will not change after transformatio
    :param A: point cloud A (shape: Nx3)
    :param B: point cloud B (shape: Nx3)
    :return: Return R and t for rigid transformation
    """
    # nx3 -> 3xn
    A = A.T
    B = B.T

    # center and rotate
    center_A = np.mean(A, axis=1).reshape(-1, 1)
    center_B = np.mean(B, axis=1).reshape(-1, 1)
    H = np.matmul((A - center_A), (B - center_B).T)

    U, S, Vt = np.linalg.svd(H)
    Rotation = np.matmul(Vt.T, U.T)

    if np.linalg.det(Rotation) < 0:
        Ur, Sr, Vtr = np.linalg.svd(Rotation)
        Vtr[2, :] *= -1
        Rotation = np.matmul(Vtr.T, Ur.T)

    return Rotation, center_B - np.matmul(Rotation, center_A)
```

Similar to ICP, it also finds the rotation and translation matrix, but in only one shot. The goal of this algorithm is to find the best R and t so that A' = R * A.T + t is close enough to B, where A and B are two input point clouds. In our implementation, we also provide a function to transform A given R and t.

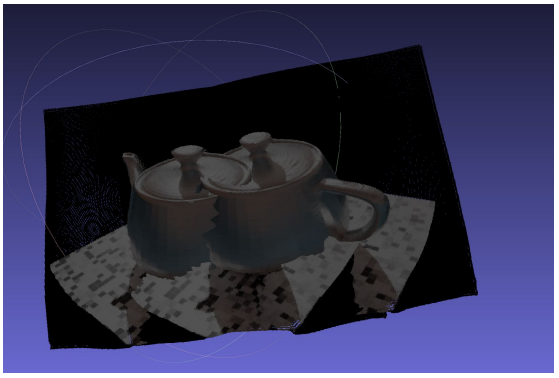
```python
def computeAPrimeT(A, r, t):
    """
    Given r and t for rigid body transformation and compute the transpose of A' (A' = R*A.T + t)
    :param A: point cloud A (shape:Nx3)
    :param r: rotation
    :param t: translation
    :return: transpose of A', which should be close to B (shape Nx3)
    """
    return (np.matmul(r, A.T) + t).T
```

Empirical work and Presentation of Results/Failures

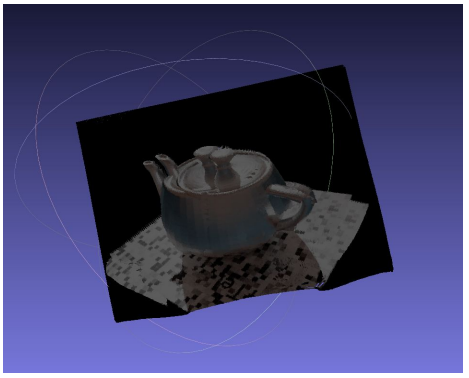
Experiment I

Input A

Input B



Before Merge

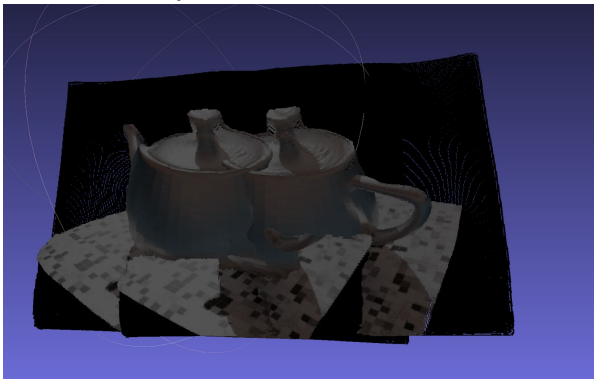


Merged Result

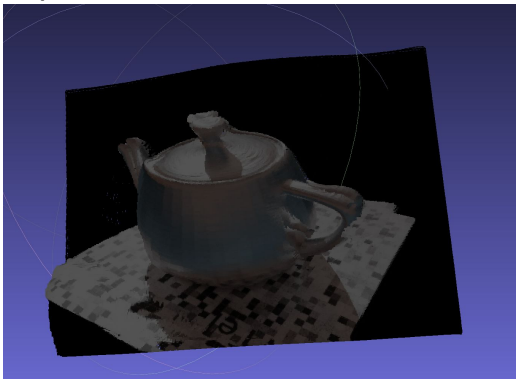
Experiment II

Input A

Input B



Before Merge



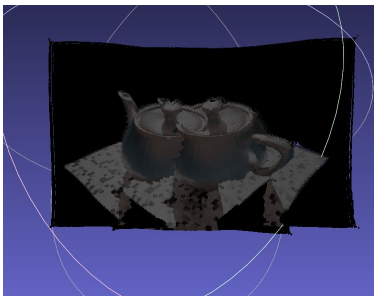
Merged Result

The camera positions input A and input B are off for a little bit for both experiments, and the algorithm is able to general acceptable results within a very limited amount of time (~3.778s for Experiment I and ~3.217s for Experiment II). Especially for experiment II, two point clouds almost aligned perfectly with only small deviations. Notice for both experiments, the shape of the rigid body (the teapots) does not change.

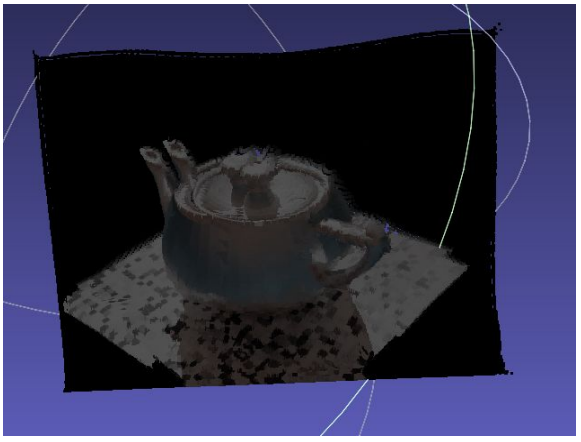
Comparison for ICP and rigid body transformation

Teapot1

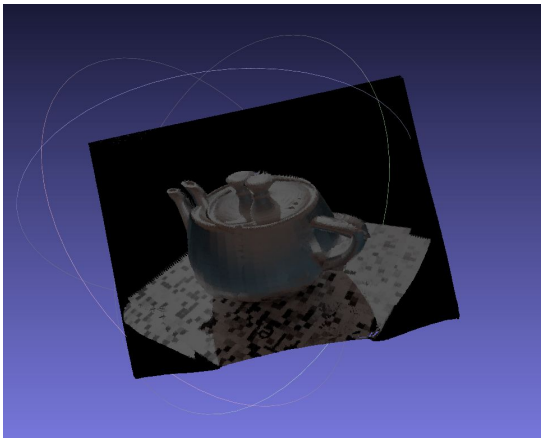
Teapot2



Before Merge



ICP



Rigid Body Transform

Teapot 2

Teapot 3

The performance of ICP algorithm is positively proportional to time, which means it can iteratively improve its result given sufficient amount of time. Whereas, the rigid body transformation can only give one try. No matter how many times are given, the result of rigid body transformation is firm. It is more likely to be a game between

efficiency and accuracy. In the experiments, we actually found out that the result point cloud from early iterations of the ICP algorithm is worse than the rigid body transformation. However, if time is not a constraint, the ICP algorithm will obviously win the game in the long run, which just fits our situation. In the ICP result image displayed above, we set the iteration number to be 100. Although error exists in the c part for both algorithms, the result from the ICP algorithm is able to merge the body part of the teapots better (shorter distance between two inputs).

Second pipeline

In the second pipeline, we generate the disparity map and depth map given 2 images taken by a camera. The camera center only has translation in one dimension(x-axis) in the world coordinates system.

Theoretical part

1. For each pixel(xl,yl) in the left image, find the matched pixel(xr,yl) on the right image by finding the minimum of SSD(matching cost), and then calculate the disparity xl-xr. We only need to search the points with x less than xl on the scanline since 3D points cannot be behind the camera. Also, since the patch size we used may vary from 5x5 to 35x35, we need to do zero padding to images in case of invalid indexes.

```
def disparity_SSD(padded_imgl,padded_imgr,rowi,coli,patch_size,h_pad,w_pad):
    '''Return the disparity "xl-xr" given a point(rowi,coli) in left image(imgl)
    using SSD(sum of Square difference)
    rowi: row index of left image
    coli: col index of left image'''
    left_patch = getValue(padded_imgl, rowi,coli, patch_size, h_pad, w_pad)

    minimum_SSD = float('inf')
    #Consider all columns in right image with the same row index(rowi)
    #Only search columns within (0,xl)
    if coli == 0:
        #on the leftmost column of the image
        disparity = 0
    for col in range(coli):
        right_patch = getValue(padded_imgr, rowi, col, patch_size, h_pad, w_pad)
        #Use Gray images Or add color channel
        SSD = np.sum(np.square(np.subtract(left_patch,right_patch)))
        if SSD < minimum_SSD:
            minimum_SSD = SSD
            #disparity = xl-xr
            disparity = coli-col
    return disparity
```

```python
def zeroPad(img, h_pad, w_pad):
    '''Return a zero padded image with new shape
    row = img.row+h_pad
    col = img.col+w_pad'''
    img_h,img_w = img.shape
    new_h = img_h + h_pad
    new_w = img_w + w_pad
    new_img = np.zeros((new_h, new_w), 'uint8')

    # copy orginal image into new_img with zero padding
    n = 0
    for i in range(h_pad//2, new_h - h_pad//2):
        m = 0
        for j in range(w_pad//2, new_w - w_pad//2):
            new_img[i, j] = img[n, m]
            m = m + 1
        n = n + 1
    print('padded img has shape,',new_img.shape)
    return new_img
```

```python
def getValue(padded_img, x, y,patch_size,h_pad,w_pad):
    '''Return the array of elements in the patch of the padded image
    x,y is the center pixel of the window in the image without padding.
    Parameter: h_pad, w_pad: added rows and columns
    '''
    #the coordinate in the new padded image
    x_pad,y_pad = x+h_pad//2,y+w_pad//2
    side_len = patch_size[0]//2

    result = np.zeros(patch_size[0]*patch_size[0]) #shape: patch_size:5x5
    n = 0
    for i in range(x_pad-side_len, x_pad+side_len+1):
        for j in range(y_pad-side_len,y_pad+side_len+1):
            #print('information,', i,j)
            result[n] = padded_img[i,j]
            n += 1
    return result
```

2. Then, after using Gaussian blur to denoise the original images, we can generate the disparity map with different patch sizes. After that, we generate the depth using the formulas depth = (focal_length x T)/disparity. (As we have discussed in the video representation) Note that when the disparity for a pixel is 0, we define the depth of that pixel to be 255. Also, to avoid cropping values above 255 in the depth map to 255, we normalize the depth into the range of [0,255].

```
def Depthmap(imgl,imgr, f, T,patch_size,h_pad,w_pad):
    '''Return the depth of each pixel in 2D image plane in
    the camera coordinate system(Z_c) and the disparity map of a stereo pair'''
    rows,cols = imgl.shape
    depth_map = np.zeros((rows,cols))
    disparity_map = np.zeros((rows, cols))
    # zero padding for 2 images
    padded_imgl = zeroPad(imgl, h_pad, w_pad)
    padded_imgr = zeroPad(imgr, h_pad, w_pad)

    for i in range(rows):
        for j in range(cols):
            result = disparity_SSD(padded_imgl,padded_imgr,i,j,patch_size,h_pad,w_pad)
            #print('disparity is ', result)
            disparity_map[i,j] = result
            if result == 0:
                depth_map[i,j] = 255
            else:
                depth_map[i,j] = (f*T)/result
    #put depth into range of 0-255
    maxi_depth = np.max(depth_map)
    mini_depth = np.min(depth_map)
    depth_map = ((depth_map-mini_depth)/(maxi_depth-mini_depth))*255
    return disparity_map, depth_map
```

```
def denoise(left,right,resize_shape,window_size,sigma):
    '''Return the denoised gray-scale version of left and right images.
    window_size and sigma are parameters for Gaussian blur'''
    #make sure the color is correct
    left = cv2.cvtColor(left, cv2.COLOR_BGR2RGB)
    right = cv2.cvtColor(right, cv2.COLOR_BGR2RGB)
    left = cv2.cvtColor(left, cv2.COLOR_RGB2GRAY)  # shape:(2000, 2964, 1)
    right = cv2.cvtColor(right, cv2.COLOR_RGB2GRAY)  # shape:(2000, 2964, 1)

    # Resize image to increase the speed of analysis
    left_resized = cv2.resize(left, resize_shape)  # shape:(500, 741, 1)
    right_resized = cv2.resize(right, resize_shape)
    # smooth 2 images with 2D Gaussian filter
    left_blur = cv2.GaussianBlur(left_resized, window_size, sigma)
    right_blur = cv2.GaussianBlur(right_resized, window_size, sigma)
    return left_blur,right_blur
```

Empirical work and Presentation of Results/Failures

Parallel input images:

Left

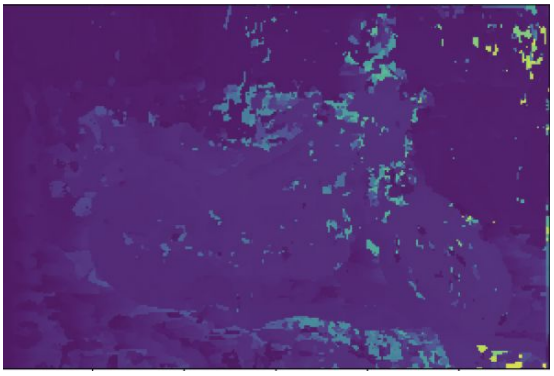
Right

Analysis of the correct appearance of disparity map and depth map

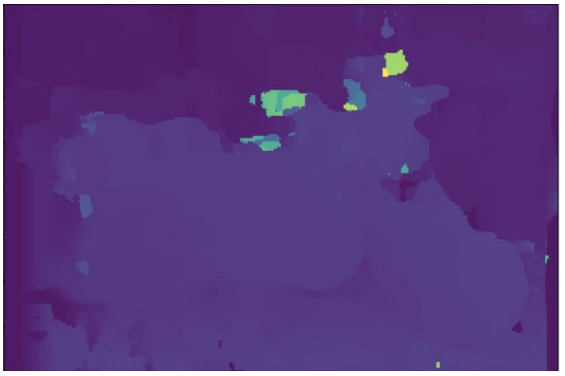
Disparity map	when point is further	depth(Z) is larger	xl-xr is smaller	color is darker
	when point is nearer	depth(Z) is smaller	xl-xr is larger	color is brighter

Depth map	when point is further	depth(Z) is larger	color is brighter
	when point is nearer	depth(Z) is smaller	color is darker

Disparity map:



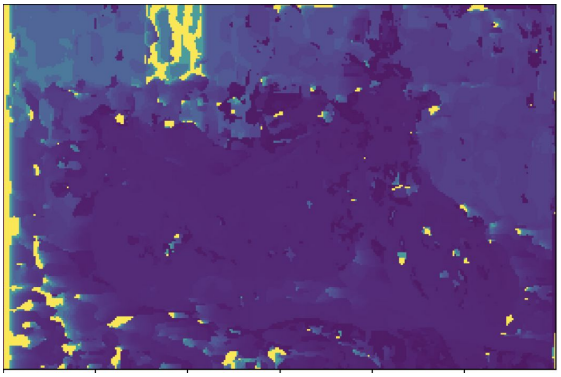
Patch size: 5x5



Patch size: 15x15

Patch size: 35x35

Depth map:



Patch size: 5x5

Patch size: 15x15

Patch size: 35x35

Discussion

The images in the result satisfies with our analysis for the appearance of disparity map and depth map.(As shown in the chart) For example, in the disparity maps, the outline of the motorbike is brighter than the background wall since the motorbike is nearer to the camera center than the wall behind. The depth map, on the contrary, has darker color on the region of the motorbike than the background wall. Therefore, the overall quality of the disparity maps and depth maps is good.

Apart from that, the appearance of the disparity map satisfies the knowledge we learned in lecture. When the patch size is small(5x5), the disparity map shows more detail, but there are some noises. After we increase the patch size to 15x15, the disparity map becomes smoother. But larger patch size, such as 35x35, leads to fewer detail and accuracy loss.

Disadvantages/Failures

We use SSD to find the matched points between 2 images. However, SSD doesn't work well when the pixels in the left images do not exist on the right images. For example, the leftmost columns in the left images disappear on the right image since we translate the left image rightwards. Therefore, the leftmost columns do not have matched points on the right image. We noticed that the leftmost columns for the depth maps have unusual color, which is caused by the defect of this algorithm itself.

Third Pipeline

In the third pipeline, we use the stereo general version. We start with two input images that are not parallel. Firstly, we generate the fundamental matrix by the eight-point algorithm, but after experimentation, it is found that if the fundamental matrix is not normalized, the accuracy of the final depth map is low. So we used two different methods to normalize the fundamental matrix. Then we compute homographies for left and right images by using the normalized fundamental matrix. After that, we use the function wrapPerspective to transform input images into parallel images. Finally, we use SSD to get a Depth map that makes two parallel images as the inputs and use an intrinsic/extrinsic matrix to get the point cloud.

Theoretical part

Computing fundamental matrix

During this process, firslty we use the SIFT algorithm to find the match points between the left and right image. Then, we use the eight-point algorithm and the built-in function SVD to compute the initial fundamental matrix.

```
# Implement the eight-points algorithm by ourselves
def eight_point_alg(points1, points2):
    A = []
    for i in range(points2.shape[0]):
        A.append([points1[i][0]*points2[i][0], points1[i][1]*points2[i][0], points2[i][0], points2[i][1]*points1[i][0],
                  points1[i][1]*points2[i][1], points2[i][1], points1[i][0], points1[i][1], 1])
    A = np.asarray(A)

    # compute F_hat
    U,D,Vt = np.linalg.svd(A, full_matrices=True)
    D = list(D)
    F = Vt[D.index(min(D))].reshape(3, 3)

    # compute F
    Uf, Df, Vft = np.linalg.svd(F, full_matrices=True)
    Df = list(Df)
    Df[Df.index(min(Df))] = 0
    F = np.dot(Uf, np.dot(np.diag(Df), Vft))

    return F
```

Normalize fundamental matrix

a. Normalized eight-point algorithm:

We implemented this algorithm to normalize the initial fundamental matrix from the eight-point algorithm in order to get a more accurate depth map. At first, we normalize the points in the image before constructing the N x 8 matrix in the eight-point algorithm. In other words, we applied for translation and scaling on the image coordinates before we did the eight-point algorithm. The translation is making the origin of the new coordinate system located at the centroid of the image points. And the scaling is that making the mean square distance of the transformed image points from the origin be 2 pixels[8]. In our code, we use two matrices T, T' to represent the processing of translation and scaling:

$$q_i = Tp_i \qquad q_i' = T'p_i'$$

Finally, since the fundamental matrix we computed are for the normalized coordinates, we need to de-normalize it by use the equation:

$$F = T'^{T} F_q T$$

```python
# normalize the fundamental matrix
def normalized_F(p1, p2):
    # normalization
    points1_center_dis = p1 - np.average(p1, axis=0)
    points2_center_dis = p2 - np.average(p2, axis=0)

    # scale = 2 / mean square distance
    # compute two T matrix
    s1 = np.sqrt(2 / (np.sum(points1_center_dis ** 2) / p1.shape[0]))
    T_left = np.array([[s1, 0, -(s1 * np.average(p1, axis=0)[0])],
                       [0, s1, -(s1 * np.average(p1, axis=0)[1])],
                       [0, 0, 1]])
    s2 = np.sqrt(2 / (np.sum(points2_center_dis ** 2) / p1.shape[0]))
    T_right = np.array([[s2, 0, -(s2 * np.average(p2[:, 0:2], axis=0)[0])],
                        [0, s2, -(s2 * np.average(p2[:, 0:2], axis=0)[0])],
                        [0, 0, 1]])
    # compute the fundamental matrix
    F = eight_point_alg(np.transpose(np.dot(T_left, (np.transpose(p1)))),
                        np.transpose(np.dot(T_right, (np.transpose(p2)))))
    # de-normalize F
    F = np.dot(np.dot(np.transpose(T_right), F), T_left)
    return F
```

b. The Gold Standard method :

While we were implementing the normalized eight-point algorithm, we also implemented another improved version of the fundamental matrix normalized method, which was introduced in the book “Multiple View Geometry in computer vision” and called the Gold Standard Method. In this method, the goal is to find the maximum likelihood estimator of the parameters that minimizes the geometric distance:

$$\sum_i d(\mathbf{x}_i, \hat{\mathbf{x}}_i)^2 + d(\mathbf{x}'_i, \hat{\mathbf{x}}'_i)^2$$

Note: $\mathbf{x}_i$ and $\mathbf{x}'_i$ are correspondent points in both input images and $\hat{\mathbf{x}}_i$ and $\hat{\mathbf{x}}'_i$ are their transformations.

This algorithm takes the result of the eight-point algorithm F as an initial guess, and then computes two camera matrices P and P’ based on F. The transformation of the points in the input image is determined by two camera matrices:

$$\hat{\mathbf{x}}_i = \mathrm{P}\widehat{\mathbf{X}}_i,\ \hat{\mathbf{x}}'_i = \mathrm{P}'\widehat{\mathbf{X}}_i$$

Finally, it calls scipy.optimmize.least_square to find the optimal solution for F that minimizes the objective function listed above.

```python
def iteration(F, src, dst, P):
    l = np.dot(np.transpose(F), np.transpose(dst))
    U, D, VT = np.linalg.svd(np.transpose(l))
    e = VT[-1] / VT[-1][2]
    EX = np.asarray([[0, -e[2], e[1]], [e[2], 0, -e[0]], [-e[1], e[0], 0]])
    Pprime = np.concatenate((EX, np.array([[e[0], e[1], e[2]]]).T), axis=1)
    x_hat = []
    x_hat_prime = []
    for i in range(8):
        A = np.array([src[i][0] * P[2].T - P[0].T,
                      src[i][1] * P[2].T - P[1].T,
                      dst[i][0] * Pprime[2].T - Pprime[0].T,
                      dst[i][1] * Pprime[2].T - Pprime[1].T])
        U, D, Vt = np.linalg.svd(A)
        D = list(D)
        Xi = Vt[D.index(min(D))]
        x_hat.append(np.matmul(P, Xi))
        x_hat_prime.append(np.dot(Pprime, Xi))
    return np.asarray(x_hat), np.asarray(x_hat_prime), P, Pprime


def geometricDistance(F, src, dst, P):
    F = np.asarray([[F[0], F[1], F[2]], [F[3], F[4], F[5]],
                    [F[6], F[7], F[8]]])
    x_hat, x_hat_prime, P, Pprime = iteration(F, src, dst, P)
    sum = 0
    for i in range(8):
        sum += scipy.spatial.distance.euclidean(src[i], x_hat[i]) ** 2 + scipy.spatial.distance.euclidean\
            (x_hat[i], x_hat_prime[i].T)
    return sum


def MLEMethod(source, destination, intrinsic):
    F = eight_point_alg(source, destination)
    P = np.concatenate((intrinsic, np.array([[0, 0, 0]]).T), axis=1)
    F_mle = scipy.optimize.least_squares(geometricDistance, np.array([F[0][0], F[0][1], F[0][2], F[1][0],
                                                                      F[1][1], F[1][2], F[2][0], F[2][1], F[2][2]]),
                                         method="trf", args=(source, destination, P))
    return F_mle
```

Homography

The last step of image correction is to use the normalized fundamental matrix to compute the homographies of the left and right images. At the first step, we computed the epipoles of left and right images. Then, we get the transform matrix T and use this matrix to transform the epipoles in order to compute the rotation matrix R:

$$T = \begin{bmatrix} 1 & 0 & -\frac{\text{width}}{2} \\ 0 & 1 & -\frac{\text{height}}{2} \\ 0 & 0 & 1 \end{bmatrix} \qquad R = \begin{bmatrix} \alpha\frac{e_1'}{\sqrt{e_1'^2+e_2'^2}} & \alpha\frac{e_2'}{\sqrt{e_1'^2+e_2'^2}} & 0 \\ -\alpha\frac{e_2'}{\sqrt{e_1'^2+e_2'^2}} & \alpha\frac{e_1'}{\sqrt{e_1'^2+e_2'^2}} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

After we know the transform matrix and rotation matrix, we can compute the right homography by using the equation:

$$H_2 = T^{-1}GRT$$

```
# calculate H2
# first step
# calculate th transform matrix
width = im2.shape[1]
height = im2.shape[0]
T = np.asarray([[1, 0, -1.0 * width / 2], [0, 1, -1.0 * height / 2], [0, 0, 1]])

# translated epipole Te'
E = np.dot(T, e)
if E[0] >= 0:
    alpha = 1
else:
    alpha = -1
# compute rotation matrix
R = np.asarray([[alpha * E[0] / np.sqrt(E[0]**2 + E[1]**2), alpha * E[1] / np.sqrt(E[0]**2 + E[1]**2), 0],
                [- alpha * E[1] / np.sqrt(E[0]**2 + E[1]**2), alpha * E[0] / np.sqrt(E[0]**2 + E[1]**2), 0],
                [0, 0, 1]])
# (f, 0, 0) is epipole line
f = np.dot(R, E)[0]
G = np.asarray([[1, 0, 0], [0, 1, 0], [- 1 / f, 0, 1]])
# H2 = T^-1 * G * R * T
H2 = np.dot(np.dot(np.dot(np.linalg.inv(T), G), R), T)
```

At the second step, we compute the left homography by using the equation:

$$H_1 = H_A H_2 M$$

According to this equation, we need to compute the unknown matrices Ha and M. At first we get the matrix M by implementing:

$$M = [e]_\times F + ev^T$$

The [e]x is a cross product matrix based on epipole and vT = [1 1 1]. Afterwards, we use the way which solves the least-square problem, to compute matrix Ha and finally get the left homographies. After getting left and right homographies, we use function wrapPerspective to transform input images into parallel images.

```
# calculate H1
# H1 = HA* H2 * M
# first step: compute M
# e_m is skew-symmetric
e_m = np.asarray([[0, e[0], -e[1]], [-e[1], 0, e[2]], [e[1], -e[2], 0]])
v = np.array([[1], [1], [1]])
# M = [e] * F + e * v^T
M = np.dot(e_m, F) + np.dot(e.reshape(3, 1), np.transpose(v))

# compute a value for HA
ph1 = np.dot(np.dot(H2, M), np.transpose(points1))
ph2 = np.dot(H2, np.transpose(points2))
# least square problem Wa = b
W = np.transpose(ph1)
for i in range(W.shape[0]):
    W[i][0] /= W[i][2]
    W[i][1] /= W[i][2]
    W[i][2] /= W[i][2]
b = np.transpose(ph2)[:, 0]

# least square problem
a = np.linalg.lstsq(W, b)[0]

# Get HA
HA = np.asarray([a, [0, 1, 0], [0, 0, 1]])
H1 = np.dot(np.dot(HA, H2), M)
```

Depth map:

After we get left and right parallel images, we use the way in the second pipeline to compute the disparity map and depth map, but in this part, we change the value of the focal length and T. Finally, we use the algorithm in the first pipeline and the depth map obtained in the previous step to compute the point cloud.

Empirical work and Presentation of Results/Failures

Parallel images that computed by the stereo general version:


original left image

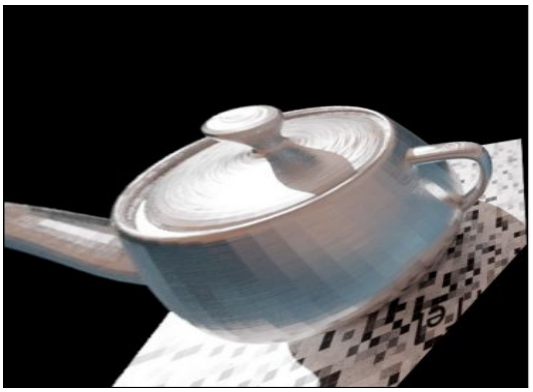

original right image

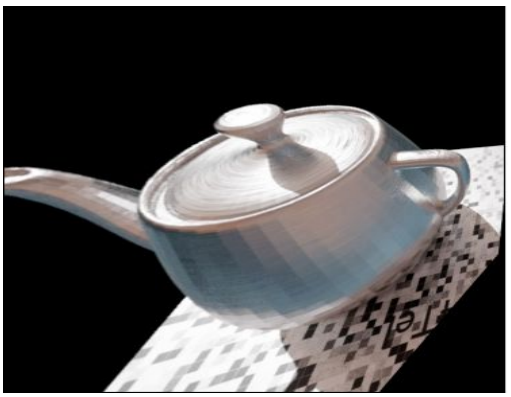
Normalized fundamental matrix by using normalized eight-point algorithm:


Left parallel image


Right parallel image

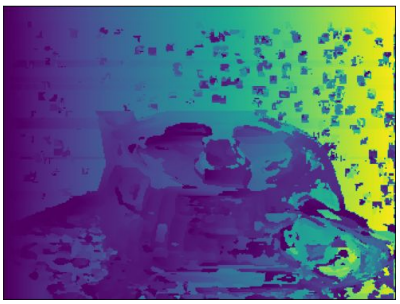
Normalized fundamental matrix by using the Gold Standard method:


Left parallel image


Right parallel image

After we compared two sets of parallel images obtained by using different normalized fundamental matrix algorithms. We found that the parallel images obtained with the normalized eight-point algorithm have higher accuracy. Because the entire item in the image still maintains the shape of the original image, but the parallel images obtained by the Gold Standard method are deformed compared to the original image

Result of the disparity map and depth map


Normalized Disparity map

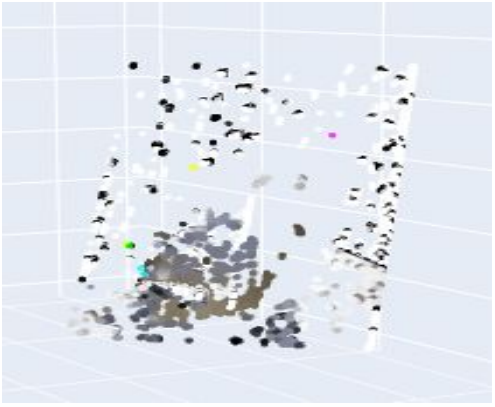

Normalized Depth map


Normalized Ground truth

We also compute the error for our depth map in this pipeline based on the ground truth depth map that calculated the mean of the root of the SSD as the error for the depth map we are testing.

Version	Error(mean root of SSD)
Stereo general	260.9973
NYU	131.8071488
Kitti	146.2901121

From the above table, it shows that the depth map generated from the stereo general version is not better than the depth map computed by the deep learning models. The reason may be that SSD would work badly when matching points between left and right images are not accurate.


Point cloud

In general, the disparity map and depth map can show a clear outline of the teacup, but because the original image still has noise pixels, there are many irregular small pieces around the teapot. For the made point cloud, the result is not satisfactory, it is difficult to see the 3D image of the teapot and the accuracy is low.

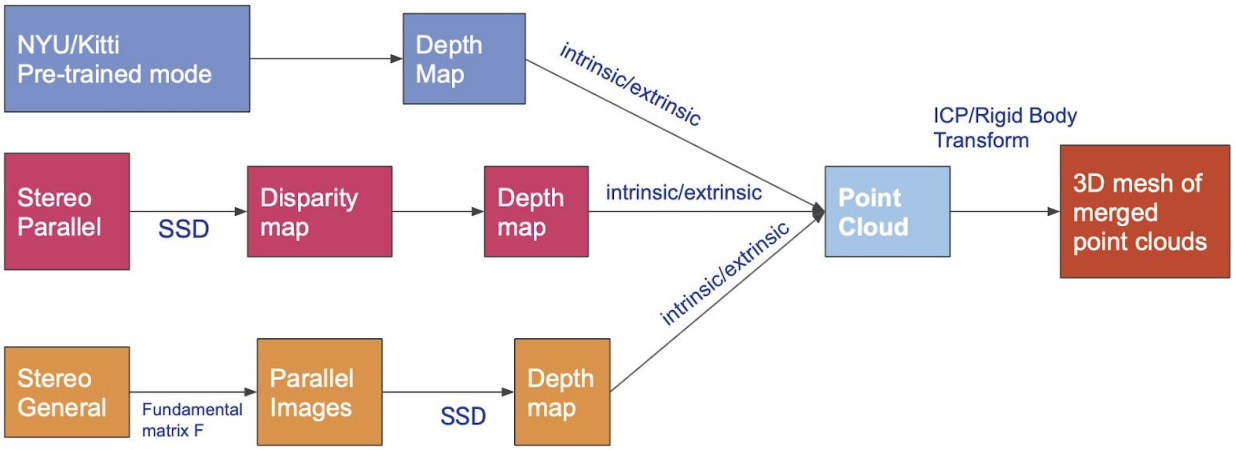
4. Conclusion

From all the experiments and comparisons we performed, we can conclude the following things:

1. The depth maps computed by the deep learning models are better than the depth maps that generated from general (stereo) methods. This is due to the fact that SSD doesn't work well when matching pixels in the left image that are not existing in the right image. Another issue with general (stereo) methods is, there is not much detail in the depth image generated with a larger patch size, while the result depth map will be too noisy if the patch size gets smaller.
2. The ICP method can provide more accurate results than the rigid body transformation approach with a loss in time efficiency, since ICP iteratively refines the transformation. We also implemented two approaches to match the closest points of the two point clouds: NearestNeighbors and KD_tree. The NearestNeighbors approach is significantly faster than the KD_tree approach in all the experiments performed.
3. Among two normalization algorithms implemented in the third pipeline, the fundamental matrix computed by the Normalized Eight-Point algorithm will eventually lead to more accurate depth maps than the Gold Standard Method.

We also tried to reconstruct the surface of the input objects given computed point clouds using existing library functions, such as poisson and ball point method from open3d library. However, the results of these reconstructions were far away from our expectation, thus we decided to not include them in this report. On the other hand, this also yields one potential future work for us, which is to implement surface reconstruction algorithms by ourselves and compare them with the open3d library implementations.

Finally, we plot the summary chart again to give one last review of our three pipelines.



5. Authors' Contributions

Tianjiao He:
Find the dataset with camera intrinsics/extrinsics matrix, generate point clouds based on depth map, ICP implementation, the second pipeline(stereo parallel version), test the quality of depth map given ground truth, all the empirical work and result presentations related to my previous work

Jiaheng Li:
Main distribution in third pipeline: Eight-point algorithm and normalized eight-point implementation, computing left and right homographies by using fundamental matrix, and Transform two images to be parallel, computing depth map and Point cloud based on stereo parallel version

Yujie Wu:
Preprocess dataset, use pre-trained model to compute depth map and point cloud and compare the performance of two pre-trained models; initial implementation of ICP and 8-point algorithm, but rewrote by teammates later; rigid body transformation; another way to normalize the fundamental matrix: the gold standard method (Maximum likelihood estimator of fundamental matrix)

References

[1] André Broekman, Petrus J. Gräbe, PASMVS: a dataset for multi-view stereopsis training and reconstruction applications
https://data.mendeley.com/datasets/fhzfnwsnzf/2
[2]Ibraheem Alhashim and Peter Wonka, High Quality Monocular Depth Estimation via Transfer Learning (arXiv 2018) https://github.com/ialhashim/DenseDepth
[3]Wikipedia webpage for point set registration
https://en.wikipedia.org/wiki/Point_set_registration
[4]Wikipedia webpage for point set registrationICP
https://en.wikipedia.org/wiki/Iterative_closest_point
[5] Arun, K. S. and Huang, T. S. and Blostein, S. D, IEEE Transactions on Pattern Analysis and Machine Intelligence, Volume 9 Issue 5, May 1987, "*Least-Squares Fitting of Two Point Sets*"
[6]Nghia Ho, "Finding Optimal Rotation and Translation between Corresponding 3D Points"
[7] Richard Hartley and Andrew Zisserman,"Multiple View Geometry in computer vision""
[8] Kenji Hata and Silvio Savarese, "Epipolar Geometry",
https://web.stanford.edu/class/cs231a/course_notes/03-epipolar-geometry.pdf