16-833 Homework 4 Write Up

Exercise 2.1

For the first filter section in function `find_projective_correspondence`, the following criteria must be met in order for filter out indices that are outside of our vertex map of size $w \times h$.

$$0 \leq target\_u < w,\ 0 \leq target\_v < h,\ target\_ds \geq 0$$

```
# TODO: first filter: valid projection
mask = ((target_us < w) & (target_vs < h) & (target_us >= 0) & (target_vs >= 0) & (target_ds >= 0)).astype(bool)
# End of TODO
```

Figure 1: Valid projection filter in ICP

Follow by the first filter, a second filter is implemented to ensure the projection q is in the neighborhood of the original point p, this is determined by a preset threshold of 0.07 unit. Further, $p, q \in \mathbb{R}^3$. Outliers of this condition are discarded.

$$|p - q| < 0.07$$

This step is necessary in order to prevent to much drift in the data when fusing, if the transformed points are not close to the original data, there is no point in fusing the two points together.

```
# TODO: second filter: apply distance threshold
target_points = target_vertex_map[target_vs, target_us]
mask = ((np.linalg.norm((T_source_points - target_points), axis=1)) < dist_diff).astype(bool)
# End of TODO
```

Figure 2: Distance threshold filter in ICP

Exercise 2.2

We are given the following relationship to transform our nonlinear error function into a linear cost function with the state vector $[\alpha, \beta, \gamma, t_x, t_y, t_z]^\top$:

$$\begin{array}{ccc} \text{Non-linear error function} & \rightarrow & \text{Linear error function} \\ n_{q_i}^\top (Rp_i' + t - q_i) & \rightarrow & n_{q_i}^\top ((\delta R)p_i' + \delta t - q_i) \end{array}$$

$$\delta R = \begin{bmatrix} 1 & -\gamma & \beta \\ \gamma & 1 & -\alpha \\ -\beta & \alpha & 1 \end{bmatrix}$$

Expanding the linear error function component-wise:

$$\begin{bmatrix} n_1 & n_2 & n_3 \end{bmatrix} \left(\begin{bmatrix} 1 & -\gamma & \beta \\ \gamma & 1 & -\alpha \\ -\beta & \alpha & 1 \end{bmatrix} \begin{bmatrix} p_x' \\ p_y' \\ p_z' \end{bmatrix} + \begin{bmatrix} \delta t_x \\ \delta t_y \\ \delta t_z \end{bmatrix} + \begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix} \right)$$

Perform matrix multiplication and addition:

$$n_1(p_x' - \gamma p_y' + \beta p_z' + \delta t_x - q_x) + n_2(\gamma p_x' + p_y' - \alpha p_z' + \delta t_y - q_y) + n_3(\beta p_x' + \alpha p_y' + p_z' + \delta t_z - q_z)$$

Using the state stated from above to arrange terms in matrix form:

$$\underbrace{\begin{bmatrix} -n_2 p_z' + n_3 p_y' \\ n_1 p_z' - n_3 p_x' \\ -n_1 p_y' + n_2 p_x' \\ n_1 \\ n_2 \\ n_3 \end{bmatrix}^\top}_{A_i,\ 1\times 6} \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ t_x \\ t_y \\ t_z \end{bmatrix} + \underbrace{[n_1(p_x' - q_x) + n_2(p_y' - q_y) + n_3(p_z' - q_z)]}_{b_i,\ 1\times 1}$$

Furthermore, the first 3 columns of A_i can be rewritten more compactly using the cross-product operator ($[\ \ ]_\times$), with p' being operated according to the emerged pattern:

$$[p']_\times = \begin{bmatrix} 0 & -p'_z & p'_y \\ p'_z & 0 & -p'_x \\ -p'_y & p'_x & 0 \end{bmatrix}, \ n * [p']_\times = \begin{bmatrix} n_1 & n_2 & n_3 \end{bmatrix} \begin{bmatrix} 0 & -p'_z & p'_y \\ p'_z & 0 & -p'_x \\ -p'_y & p'_x & 0 \end{bmatrix} = \begin{bmatrix} -n_2 p'_z + n_3 p'_y \\ n_1 p'_z - n_3 p'_x \\ -n_1 p'_y + n_2 p'_x \end{bmatrix}$$

Finally, replace the result in A_i:

$$\boxed{A_i = \begin{bmatrix} n * [p']_\times & n_1 & n_2 & n_3 \end{bmatrix}, \ b_i = n_1(p'_x - q_x) + n_2(p'_y - q_y) + n_3(p'_z - q_z)}$$

As a note, I am confident that my derivation is correct, but my Python code wants to work with $[n * [p']_\times, -n_1, -n_2, -n_3]$ instead, so I want to be transparent about this for grading.

```python
def vec2skew(w):
    return np.array([[0, -w[2], w[1]],
                     [w[2], 0, -w[0]],
                     [-w[1], w[0], 0]])
```

```python
def build_linear_system(source_points, target_points, target_normals, T):
    M = len(source_points)
    assert len(target_points) == M and len(target_normals) == M

    R = T[:3, :3]
    t = T[:3, 3:]

    p_prime = (R @ source_points.T + t).T
    q = target_points
    n_q = target_normals

    A = np.zeros((M, 6))
    b = np.zeros((M, ))

    # TODO: build the linear system
    for i in range(M):
        A[i, :] = np.hstack(([n_q[i][np.newaxis, :] @ vec2skew(p_prime[i]), -n_q[i][np.newaxis, :]]))
        b[i] = np.dot(n_q[i], (p_prime[i] - q[i]))
    # End of TODO

    return A, b
```

Figure 3: A_i and b_i implemented in `build_linear_system`

Exercise 2.3

Within `solve`, I opted to use QR factorization for a numerical stable solution. `solve` is pasted below for reference. `build_linear_system` is pasted above to better compare with derived system.

```python
def solve(A, b):
    '''
    \param A (6, 6) matrix in the LU formulation, or (N, 6) in the QR formulation
    \param b (6, 1) vector in the LU formulation, or (N, 1) in the QR formulation
    \return delta (6, ) vector by solving the linear system. You may directly use dense solvers from numpy.
    '''
    # TODO: write your relevant solver
    ## Psuedo-inverse
    # return np.linalg.inv(A.T @ A) @ -A.T @ b

    # QR factorization
    Q, R = np.linalg.qr(A)
    d = np.dot(Q.T, b)
    return np.dot(np.linalg.inv(R), d)

    ## LU factorization
    # lu, piv = lu_factor(A.T @ A)
    # return lu_solve((lu, piv), -A.T @ b)
```

Figure 4: QR factorization implemented in `solve`

Below are the result of source and target frame of (frame 10, frame 50) and (frame 10, frame 100) respectively. For frames that are relatively closer to each other – frame 10 & 50, <u>10 iterations</u> of stitching are enough to fuse both camera frames. Result of frame 10 & 50 can be referred in Figure 5. In contrast, farther frames need more iterations to stitch. Frame 10 & 100 used <u>100 iterations</u> in order to arrive at a decent fusion. Result of frame 10 & 100 can be referred in Figure 6. The farther the source and target frames are, the more iterations it will need for the intermediate rotation and translation in order to arrive at a convergence.

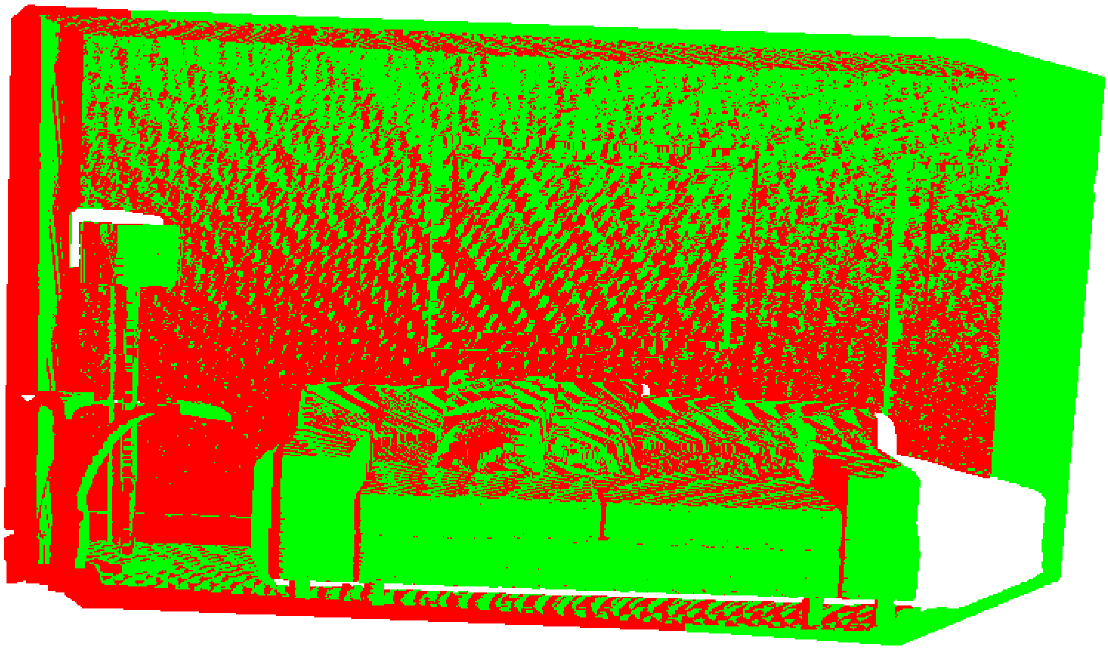

Figure 5: Frame fusion using ICP for frame 10 and frame 50

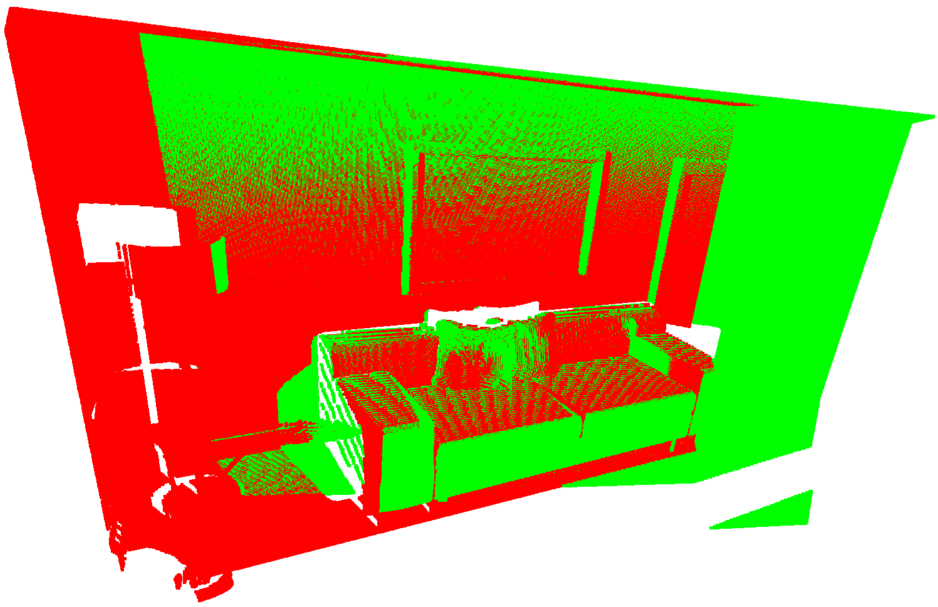

Figure 6: Frame fusion using ICP for frame 10 and frame 100

Exercise 3.1
Similar in ICP, the `filter_pass1` masks all indices that are outside of vertex map, and makes sure the retrieved depths are all greater than 0. `filter_pass2` masks all indices that demonstrate large euclidean distances in between the source and input, additionally, it checks the angle in between the

source normal and input normal through definition of dot product:

$$\theta = \arccos(source\_normal \cdot input\_normal)$$

```python
def filter_pass1(self, us, vs, ds, h, w):
    '''
    TODO: implement the filter function
    \param self The current maintained map, unused
    \param us Putative corresponding u coordinates on an image, (N, 1)
    \param vs Putative corresponding v coordinates on an image, (N, 1)
    \param vs Putative corresponding d depth on an image, (N, 1)
    \param h Height of the image projected to
    \param w Width of the image projected to
    \return mask (N, 1) in bool indicating the valid coordinates
    '''
    return ((us < w) & (vs < h) & (us >= 0) & (vs >= 0) & (ds >= 0)).astype(bool)
```

Figure 7: Projective map filtering in Point-based Fusion

```python
def filter_pass2(self, points, normals, input_points, input_normals,
                 dist_diff, angle_diff):
    '''
    TODO: implement the filter function
    \param self The current maintained map, unused
    \param points Maintained associated points, (M, 3)
    \param normals Maintained associated normals, (M, 3)
    \param input_points Input associated points, (M, 3)
    \param input_normals Input associated normals, (M, 3)
    \param dist_diff Distance difference threshold to filter correspondences by positions
    \param angle_diff Angle difference threshold to filter correspondences by normals
    \return mask (N, 1) in bool indicating the valid correspondences
    '''
    dist_mask = (np.linalg.norm((points - input_points), axis=1)) < dist_diff
    angle_mask = np.abs(np.arccos((normals * input_normals).sum(axis=1))) < angle_diff

    return np.logical_and(dist_mask, angle_mask)
```

Figure 8: Distance and normal threshold filtering in Point-based Fusion

Exercise 3.2

Referencing Equation 1 in [2], the updates in `merge` are the weighted average of the following entities:

$$p \leftarrow \frac{wp + q}{w+1},\ n \leftarrow \frac{wn_p + n_q}{w+1},\ w \leftarrow w+1,\ c \leftarrow \frac{wc + c_{new}}{w+1}$$

where $q = R(p) + t, n_q = R(n_p)$. This process takes incoming transformed points and colors and merges with the existing class attributes to agglomerate into a unified map. Implicitly, each point in $p_i \in \mathbb{R}^3$ has its associated weight vector $w_i \in \mathbb{R}$, normal vector $n_i \in \mathbb{R}^3$, and color vector $c_i \in \mathbb{R}$.

```python
def merge(self, indices, points, normals, colors, R, t):
    '''
    TODO: implement the merge function
    \param self The current maintained map
    \param indices Indices of selected points. Used for IN PLACE modification.
    \param points Input associated points, (N, 3)
    \param normals Input associated normals, (N, 3)
    \param colors Input associated colors, (N, 3)
    \param R rotation from camera (input) to world (map), (3, 3)
    \param t translation from camera (input) to world (map), (3, )
    \return None, update map properties IN PLACE
    '''
    self.points[indices] = (self.weights[indices] * self.points[indices] + (R @ points.T + t).T)/(self.weights[indices] + 1)
    self.normals[indices] = (self.weights[indices] * self.normals[indices] + (R @ normals.T).T)/(self.weights[indices] + 1)
    self.normals[indices] /= np.linalg.norm(self.normals[indices], axis=1, keepdims=True)
    self.colors[indices] = (self.weights[indices] * self.colors[indices] + colors)/(self.weights[indices] + 1)
    self.weights[indices] += 1
```

Figure 9: `merge` in Point-based Fusion

Exercise 3.3
The `add` function is responsible for concatenating points to the existing map to be ready for the next iteration of `merge`. For a set of new inlier points of length N, append associated the transformed point vectors q_i, transformed normal vectors $n_{q,i}$, associated color c_i, and associated weight w_i, where i spans from 1 to N.

```python
def add(self, points, normals, colors, R, t):
    '''
    TODO: implement the add function
    \param self The current maintained map
    \param points Input associated points, (N, 3)
    \param normals Input associated normals, (N, 3)
    \param colors Input associated colors, (N, 3)
    \param R rotation from camera (input) to world (map), (3, 3)
    \param t translation from camera (input) to world (map), (3, )
    \return None, update map properties by concatenation
    '''
    self.points = np.concatenate((self.points, (R @ points.T + t).T))
    self.normals = np.concatenate((self.normals, (R @ normals.T).T))
    self.colors = np.concatenate((self.colors, colors))
    weights = np.ones((len(points), 1))
    self.weights = np.concatenate((self.weights, weights))
```

Figure 10: `add` in Point-based Fusion

Exercise 3.4
After running `fusion.py`, the resulting map and normal map from fusing 200 frames is displayed as below. There are a total of 1,362,143 points in the map at the end of run. The compression ratio is calculated below:

$$\frac{\text{total}}{w \times h \times 200 \text{ frames}} : \underbrace{\frac{1362143}{680 \times 480 \times 200}} = \boxed{0.021}$$

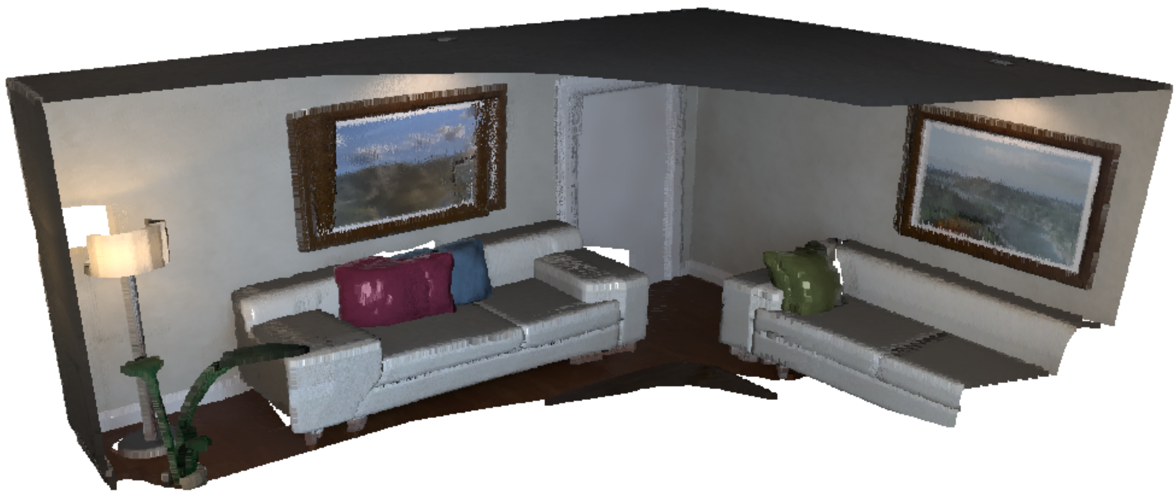

Figure 11: Fused map in Point-based Fusion

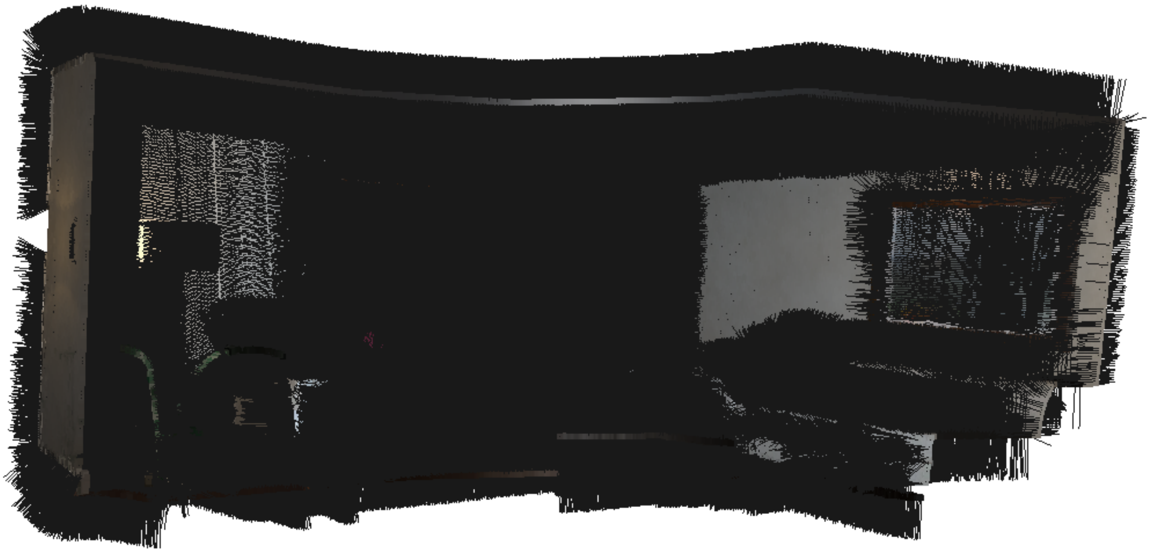

Figure 12: Fused normal map in Point-based Fusion

Exercise 4
The source frame is the RGBD frame and the target frame is the map. I do not think you can switch their roles because RGBD frame since it allows a projection to the vertex map initially, thus it leads to the correct weight (confidence factor) to be counted. The target map is determined by projecting back from the vertex map. In summary, there is a sequential order to the source and target.

The following figure is the map after running `main.py` that utilizes both ICP and point-based fusion. The respective trajectories of ground truth and from `main.py` are displayed below as well.

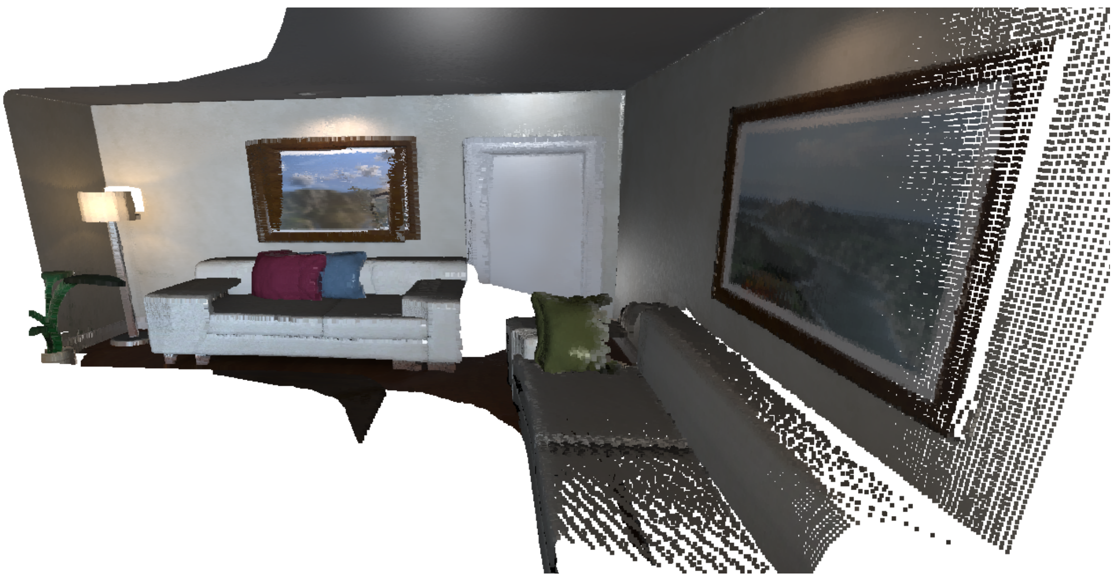

Figure 13: Fused map with ICP and Point-based Fusion

Figure 14: Close up detail of pixel distribution and drift

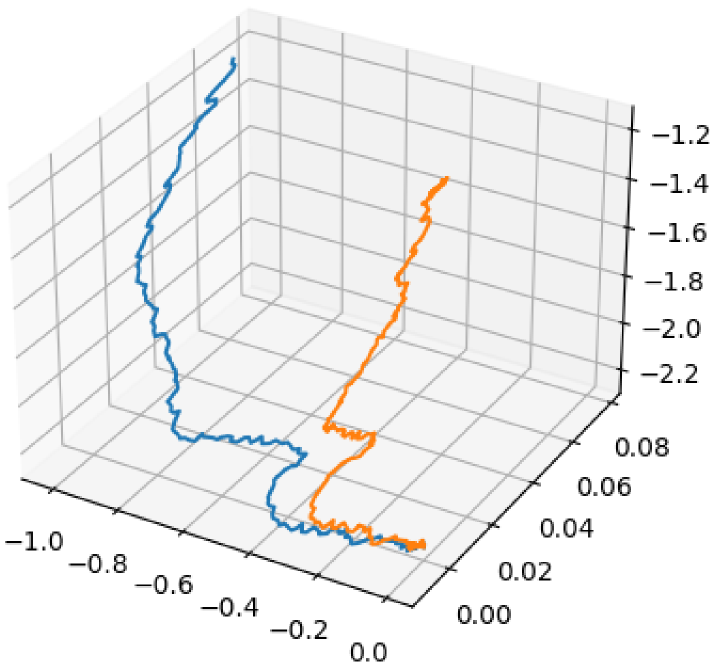

Figure 15: Camera pose estimation from combining ICP and Point-based Fusion. Ground truth shown in blue, pose estimation shown in orange

References

[1] Kaess, M. & Dong, W. *Session 19: KinectFusion and DTAM*, lecture recordings, 16-833 Robot Localization and Mapping, Carnegie Mellon University, delivered 7 April 2021.

[2] Keller, Maik, et al. ”Real-time 3D reconstruction in dynamic scenes using point-based fusion.” *International Conference on 3D vision (3DV)*, 2013. Link: *http://ieeexplore.ieee.org/document/6599048/.*