1 Rigid Surface Tracking and Reconstruction

1.1 Frame-To-Frame (model) alignment

Given a RGB-D camera, we move it over time around your room and wish to reconstruct your room from the recorded RGB-D sequence. How can we link the data between frames? In other words, how can we align these data to reconstruct your room? Assuming the camera moves slowly, then the camera pose at each frame can be estimated by finding the alignment between the current recorded data and the recorded data in the previous frame (or taken from a incrementally build model). This is solvable via **ICP**.

1.1.1 Iterative Closes Point

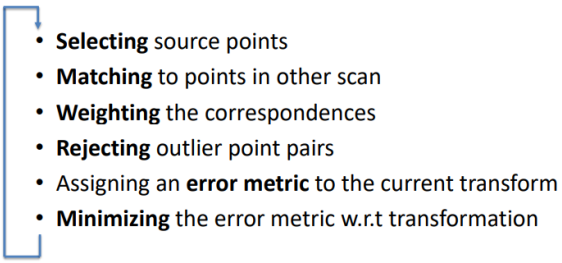
If correct correspondences between consecutive frames are known, then finding the correct relative rotation and translation would be trivial. But the problem is how do we find those correspondences? **In ICP we assume that closest points are the correspondences**. The high-level ICP overview consists of following:

1. Data association.
2. Outlier removal / pruning.
3. Optimization.
4. Iterate

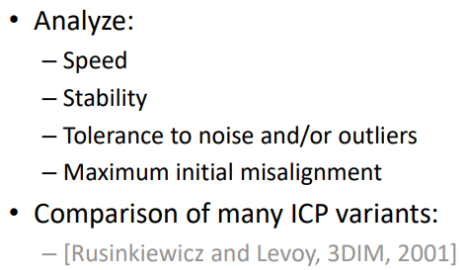
That is, we iterate to find the correct alignment and it converges when poses are close enough. A slightly formalized version of ICP consists of following:

1. Select e.g. 1000 random points.
2. Match each to closest point on other scan, using data structure such as k-d tree.
3. Reject pairs with distance $>$ k times median.
4. Construct error function:
$$E = \sum |Rp_i + t - q_i|^2$$
5. Minimize (has closed form solution)

The algorithm has different stages and there are different choices we can make at each stage.



Each variant has some effect on the speed or robustness of the method. And we will discuss some of them in this lecture:



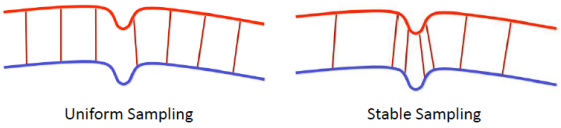
1.1.1.1 Variants on Selecting source points

We can have different choices in this case, such as:

- **Use all points**: (first it would be very slow, second if most of the points fall on the planar surface then we will not cover all the degrees of freedom of the transformation we are solving for).
- **Uniform subsampling**.
- **Random sampling**.
- **Stable sampling**.

The first three options are very straightforward and easy to understand, so we will only discuss the **stable sampling**.

The idea behind stable sampling is that, **we should select samples on the scan that are somehow the most relevant to constraining all the different degrees of freedom of the transformation we are solving for**.



For example, if we had the above situation, we'd want to recognize that the up-and-down component of the motion is well-constrained by almost any point you choose. In contrast, the left-and-right component of the transformation is really only established by points on the little bump in the center. **In other words, by selecting points only on the smooth area we are making the optimization problem unconstrained because there's a strong linear relationship between these correspondences**. So ideally we should focus on capture points around regions that has a high degree of geometry variation rather on planar surfaces in order to make our problem over-constrained and well conditioned.

1.1.1.2 Variants on Matching to points in other scan

Variants of this type can be classified into 2 types:

- **Variants which focus on making matching more stable**: Selecting closest point based only on the distance are often a bad approximation to corresponding point. Instead, we should selecting the closest point that is *compatible* with the source point by metrics such are:
  - Color.
  - Normal.
  - Curvatures, higher-order derivatives, and other local features.

  - Closest points are often bad as corresponding points
  - Improve matching effectiveness by restricting to **compatible** points
    - Color compatibility [Godin et al. 94]
    - Normal compatibility [Pulli 99]
    - Other possibilities: curvature, higher-order derivatives, other local features

- **Variants which focus on making matching process faster**: For structured data such as depth images, the correspondence can be easily established by just projecting the points from one frame to another frame according to the estimated viewpoint. This only works for structured data and points are closest pairs in the projective space.

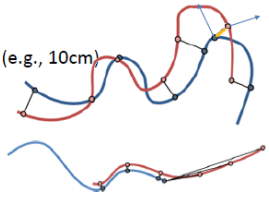
  - Projective (often much better and faster!!)
    - Only works with structured data; e.g., depth images (not on raw point clouds)

1.1.1.3 Variants on Weighting the correspondences

Seems no variants of this type is discussed in the lecture.

1.1.1.4 Variants on Rejecting outlier point pairs

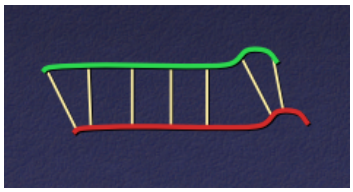
- Rejection based on outliers
  - Distance between depth values > threshold (e.g., 10cm)
  - Variation between normals
- Pruning
  - Prune correspondences to borders



1.1.1.5 Variants on Assigning an error metric to the current transform

There are basically two type of error metrics:

- **Point-to-point metric**.
- **Point-to-plane metric**: This metric is more robust than the point-to-point metric. It allows flat regions to slide along each other as it better captures the notion that sliding two planes along each other doesn't increase the distance between them.



Consider, for example, the above situation. Using the original point-to-point distance the pairs of points in the flat region will prevent the scans from "sliding along each other" to reach the correct transformation as cost will get bigger. Using point-to-plane distance in this case will lead to significantly faster convergence.

1.1.1.6 Variants on Minimizing the error metric w.r.t transformation

If we use point-to-plane as our error metric then the energy function won't has a closed form solution anymore. However, we can "make" the problem linear by assuming that the angles we need to solve for are small, so we can use the first-order approximations to sine and cosine.

- Error function:

$$E := \sum_i \left((R\mathbf{p}_i + t - \mathbf{q}_i) \cdot \mathbf{n}_i\right)^2$$

where R is a rotation matrix, t is a translation vector

- Linearize (i.e., assume $\sin\theta \approx \theta, \cos\theta \approx 1$):

$$E \approx \sum((\mathbf{p}_i - \mathbf{q}_i) \cdot \mathbf{n}_i + \mathbf{r} \cdot (\mathbf{p}_i \times \mathbf{n}_i) + \mathbf{t} \cdot \mathbf{n}_i)^2 \quad \text{where } \mathbf{r} = \begin{pmatrix} r_x \\ r_y \\ r_z \end{pmatrix}$$

- Overconstrained linear system

$$\mathbf{A}x = b,$$

$$\mathbf{A} = \begin{pmatrix} \leftarrow & \mathbf{p}_1 \times \mathbf{n}_1 & \rightarrow & \leftarrow & \mathbf{n}_1 & \rightarrow \\ \leftarrow & \mathbf{p}_2 \times \mathbf{n}_2 & \rightarrow & \leftarrow & \mathbf{n}_2 & \rightarrow \\ & \vdots & & & \vdots & \end{pmatrix}, x = \begin{pmatrix} r_x \\ r_y \\ r_z \\ t_x \\ t_y \\ t_z \end{pmatrix}, b = \begin{pmatrix} -(\mathbf{p}_1 - \mathbf{q}_1) \cdot \mathbf{n}_1 \\ -(\mathbf{p}_2 - \mathbf{q}_2) \cdot \mathbf{n}_2 \\ \vdots \end{pmatrix}$$

- Solve using least squares

$$\mathbf{A}^T\mathbf{A}x = \mathbf{A}^T b$$

But this trick only works for small rotations. So if we are not sure then we can use a regular 2nd order solver to solve it:

- Linearized Point-to-Plane Optimization [Low 04]
  - Only valid for small rotations (linear approximation of the rotation!)

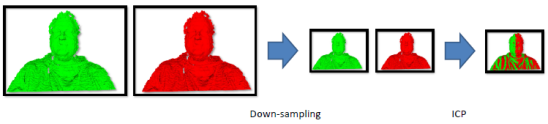
- If you don't know what to do, always use Levenberg-Marquardt!
  - 2nd order method → **quadratic convergences** -> rules them all

$$E_{frame-to-frame}(T) = \sum_i [(Tp_i - q_i) \cdot n_i]^2$$

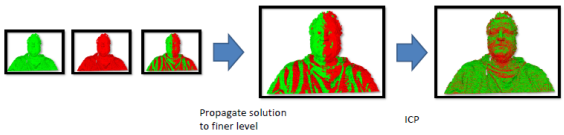
1.1.2 Coarse-to-Fine Strategy

And always remember to use the coarse-to-fine alignment:

- Start on low-resolution depth map
- Then go to finer levels



- Start on low-resolution depth map
- Then go to finer levels

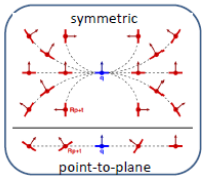


- Start on low-resolution depth map
- Then go to finer levels

➢ Faster convergence
➢ Avoids local minima (down-sampling → smoothing)
➢ Allows faster motions

1.1.3 ICP: Conclusion

- Because the correspondence finding is not differentiable, there is the hack of alternating
- Ideally, you would want to directly align a continuous surface, and differentiate it
- Many ICP variants (e.g., adding a color term)
  - Can also do similar algorithm for RGB-only alignment
- Recently: symmetric ICP [Rusinkiewicz et al 2019]
  - $(p - q) \cdot (n_p + n_q)$



1.2 Reconstruction

So a general pipeline of 3D reconstruction using a RGB-D camera consists in following:

1. Compute poses between all frames
   e.g., frame-to-frame ICP, or frame-to-model ICP

2. Integrate data into a shared model
   e.g., accumulate in a distance field

3. Extract Mesh from implicit representation

We have talked about how to do alignment, i.e. compute poses between all frames, and the mesh extraction is trivial using methods like ray casting or marching cube. **Now let's talk about how to integrate data into a shared model**.

1.2.1 Voxel representation based 3D reconstruction - Volumetric Fusion

Volumetric fusion consists of using a regular grid to store a discretized version of a signed distance function that represents the model. The SDF values stored in the grid is built-up incrementally by "merging" range maps taken from different views.

The basic approach for accumulation of incoming range maps is to project each voxel onto the range map using the estimated camera pose and to evaluate its projective distance. The depth value of a voxel w.r.t a specific range map n is the distance value of the projected pixel defined in the range map n minus the distance of the voxel in the corresponding camera space. The merging step is accomplished by weighted averaging of the prior SDF values in the grid and the incoming one related to the range map. The combining rules give us for each voxel a cumulative signed distance function, $D(x)$, and cumulative weight $W(x)$ defined in the following way:

$$D(x) = \frac{\sum_i w_i(x) d_i(x)}{\sum_i w_i(x)}$$
$$W(x) = \sum_i w_i(x)$$

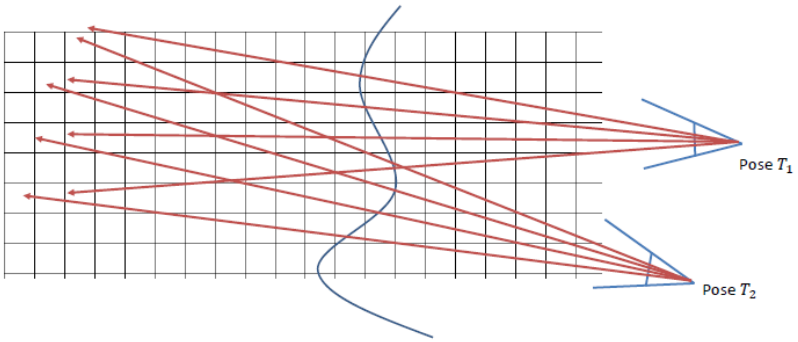
Where $d_i(x)$ and $w_i(x)$ are the signed distance and weight functions from the $i-th$ range image.

Expressed as an incremental calculation, the rules are:

$$D_{i+1}(x) = \frac{W_i(x) D_i(x) + w_{i+1}(x) d_{i+1}(x)}{W_i(x) + w_{i+1}(x)}$$
$$W_{i+1}(x) = W_i(x) + w_{i+1}(x)$$

Where $D_i(x)$ and $W_i(x)$ are the cumulative signed distance and weight functions after integrating the $i-th$ range image.

The isosurface is extracted at $D(x) = 0$.



- Weighted average of signed distance values
- $D(x) = \frac{\sum w_i(x) d_i(x)}{\sum w_i(x)}$
- $W(x) = \sum w_i(x)$
- Depth value $d_i(x)$ at point x
- Weight functions $w_i(x)$
  - E.g., depth-based -> further away -> lower weight
  - E.g., $w_i(x) = \max(w(1 - depthZeroOne), 1)$
- Isosurface $D(x) = 0$

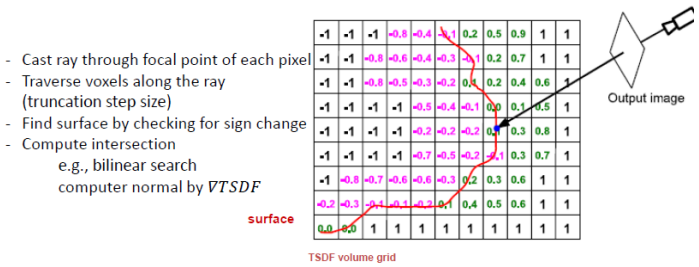
- Running average of signed distance values

- $D_{i+1}(x) = \frac{W_i(x) D_i(x) + w_{i+1}(x) d_{i+1}(x)}{W_i(x) + w_{i+1}(X)}$
- $W_{i+1}(x) = W_i(x) + w_{i+1}(x)$

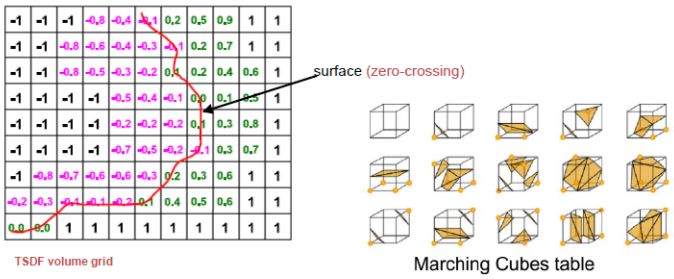
- $W_i(x)$ and $D_i(x)$ are cumulative signed distanced and weight functions after integrating the i-th range image

1.2.1.1 Surface extraction

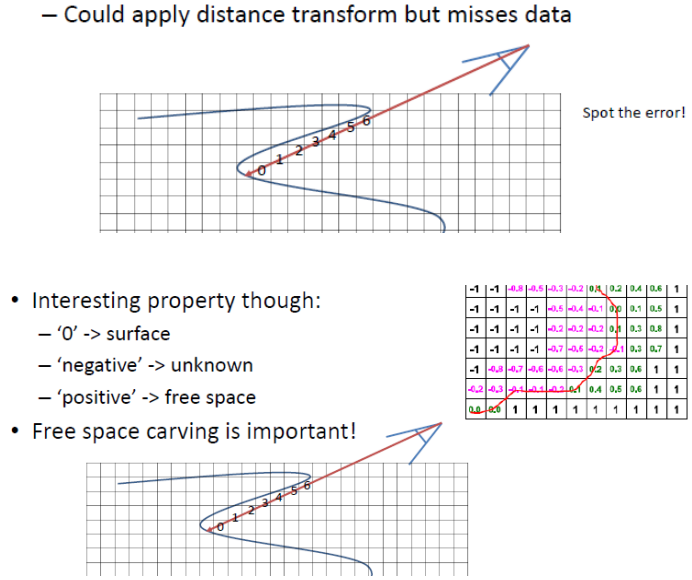
We can either use ray casting:



Or marching cube:



1.2.1.2 Properties of Volumetric Fusion



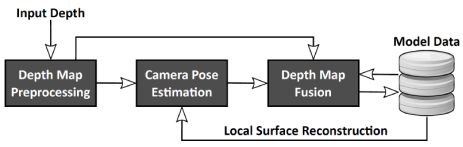
Observe that **free space carving** is important because it can be used to fill holes created by unseen portions of the surface. In the original paper voxels are classified into:

- Near-surface.
- Unseen.
- Empty.

When doing surface extraction the surface between unseen and empty regions will also be extracted.

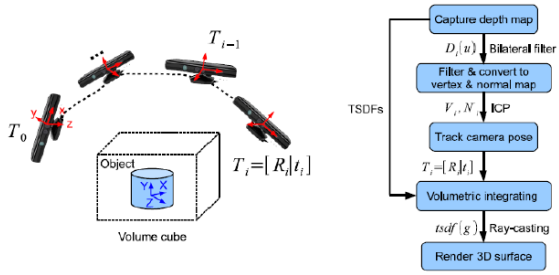
1.2.1.3 General Reconstruction Pipeline

In general volumetric fusion based reconstruction methods follows the following pipeline:



That is:

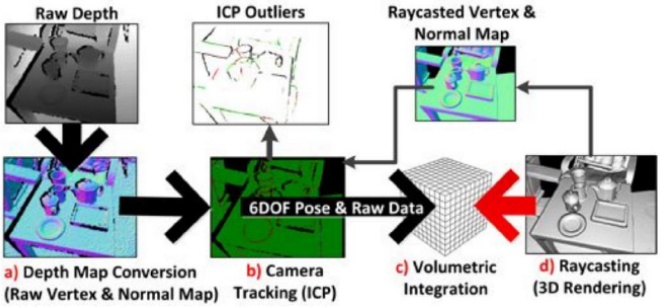
1. Get the input depth map.
2. Preprocess the input depth map to get the normal map and vertex map.
3. Extract surface from the incremental model and use the extract surface and the input depth map to estimate the camera pose.
4. Fuse the input depth map to the model.



- Often frame-to-model tracking rather than frame-to-frame
  – Use ray-casted normal map -> less noise!

Input normal map Ray-casted normal map from SDF

The following picture illustrates the pipeline of the Kinect Fusion approach

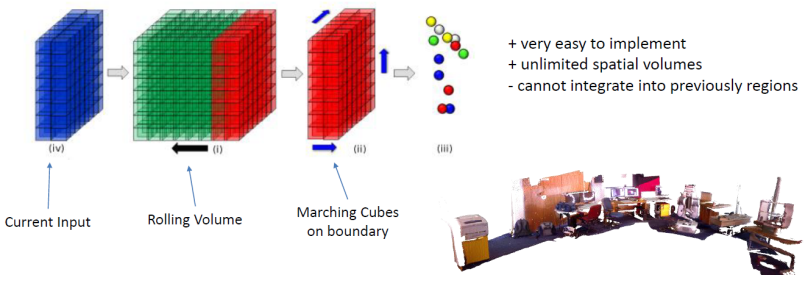


1.2.1.4 Memory Efficient Methods

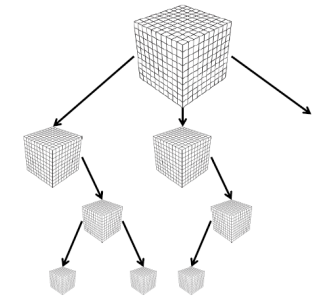
Regular voxel grids are very inefficient in terms of memory consumption and bound to a predefined volume and resolution. In the context of real-time scene reconstruction, where most approaches heavily rely on the processing capability of modern GPUs, the spatial extent and resolution of the voxel grid is typically limited by GPU memory. In order to support larger spatial extents, various approaches have been proposed to improve the memory efficiency of voxel-based representations.

Extended Kinect Fusion This approach consists of a simple dynamic shift of the voxel grid, such that it follows the motion of the camera. The approach converts the part of the volume that is shifted outside the current reconstruction volume to a surface mesh and stores it separately. While this enables larger scanning volumes, it requires heavy out-of-core memory usage and already scanned, and streamed-out surfaces, cannot be arbitrarily re-visited.

- Moving Volume



Hierarchical Fusion This approach uses an adaptive, octree-like data structure to store SDF providing different spatial resolutions. SDF values live only on finest level.



Spatial subdivision with tree
SDF values live only on finest level

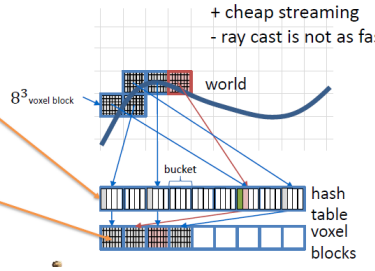
+ efficient storage: large spatial extent, high local res
+ fast lookups
- costly updates during integration (need tree update)
e.g., insertion, removal, etc.

Voxel Hashing This approach represents a virtually infinite scene by a regular grid of smaller voxel blocks of predefined size and resolution, whose spatial locations are addressed with a spatial hash function. Only the voxel blocks that actually contain geometry information are instantiated, and the corresponding indices are stored in a linearized spatial hash. This strategy significantly reduces memory consumption and allows (theoretically) infinite scene sizes. Compared to hierarchical approaches, the main benefit lies in the very efficient data insertion and access, which both is in O(1).

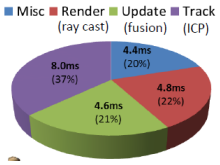
Spatial Hash Function:
$H(x,y,z) = (x \cdot p_1 \oplus y \cdot p_2 \oplus z \cdot p_3) \bmod n$

```
struct HashEntry {
  short position[3];
  short offset;
  int   pointer;
};
```

```
struct Voxel {
  float sdf;
  uchar colorRGB[3];
  uchar weight;
};
```

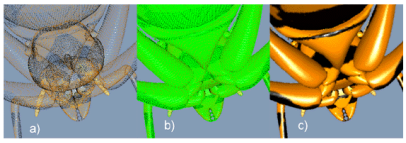


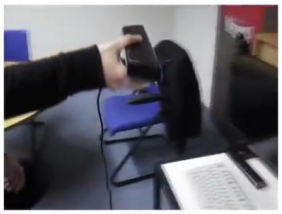
+ efficient storage: large spatial extent, high local res
+ extremely fast updates:
e.g., insertion, removal, etc. is O(1)
+ cheap streaming
- ray cast is not as fast (need to check voxel by voxel)



1.2.2 Point Cloud representation based 3D reconstruction - Point based fusion

- Surfel representation
  - Point + normal + radius
- Rendering through splatting
  - Use graphics pipeline





+ unstructured storage -> large extent and high-res
+ in a way easier for dynamics (although not quite there)
- surface quality slightly less than volumetric SDF

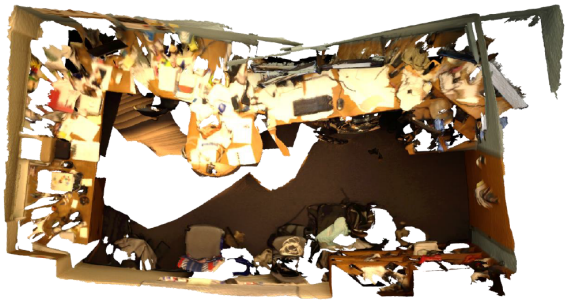
1.2.3 Drift and Loop Closure problems

One severe problem of the frame-to-frame strategies is the accumulation of tracking drift over long scanning sequences.

To reduce this problem, frame-to-model tracking has been extensively used in recent online RGB-D reconstruction frame works. Frame-to-model tracking has two significant advantages over simple frame-to-frame alignment strategy:

- Instead of last frame, a synthetically rendered depth map of the current reconstruction state is employed to anchor the reconstruction, thus drastically reducing temporal tracking drift.
- Second, the stabilized model normals can be utilized instead of the noisy input normals, which leads to higher accuracy tracking and increased robustness.

While frame-to-model tracking significantly reduces temporal tracking drift, it does not completely solve the problem of local error accumulation. This can lead to loop closure problems if already scanned parts of the scene are re-encountered over a different path in the same scanning session. In this case, the previously obtained reconstruction will not match the current depth observations leading to tracking failure and/or double integration of surfaces.

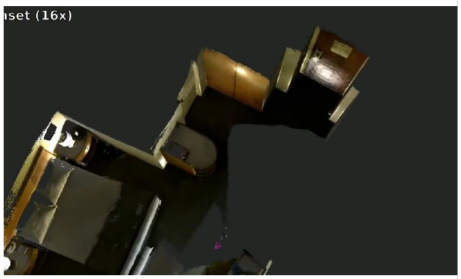


In this lecture we will discuss two methods to solve the the loop closure problem: **Elastic fusion** and **Bundle fusion**.

1.2.3.1 Elastic fusion

- Point-based representation
- Frame-to-model tracking

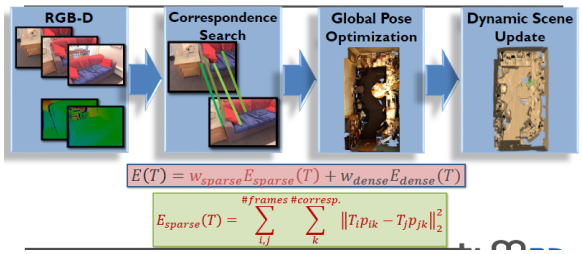
- Explicitly detect loop closures through localization
- Non-rigidly warp representation to close loops



1.2.3.2 Bundle fusion

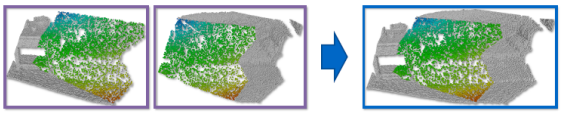
This approach is based on online bundle adjustment and on-the-fly surface re-integration:

- **Bundle Adjustment**: in bundle adjustment we globally optimize the pose of frames using a sparse term and a dense term. The sparse term is just the consistency between keypoints while the dense terms defines the general geometry and color consistency.
- **Re-integration**: once the poses are optimized it will update the model by first de-integrating the frame with wrong estimated pose, and then re-integrating it into the model using the optimized pose in order to get a more accurate model. This process can be easily done as we just need to subtract the SDF value and weight of the corresponding frame from the model.



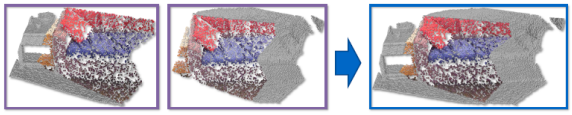
$$E_{dense}(T) = w_{depth}E_{depth}(T) + w_{color}E_{color}(T)$$

$$E_{depth}(T) = \sum_{i,j}^{\#frames} \sum_{k}^{\#pixels} \left\| \left(p_k - T_i^{-1}T_j\pi_d^{-1}(D_j(\pi_d(T_j^{-1}T_ip_k)))\right) \cdot n_k \right\|_2^2$$



$$E_{dense}(T) = w_{depth}E_{depth}(T) + w_{color}E_{color}(T)$$

$$E_{color}(T) = \sum_{i,j}^{\#frames} \sum_{k}^{\#pixels} \left\| VI(\pi_c(p_k)) - VI(\pi_c(T_j^{-1}T_ip_k)) \right\|_2^2$$



- Surface Integration *[Curless and Levoy 96]*

```
Voxel {
    distance;
    color;
    weight;
}
```

$$D(v) = \frac{\sum_i w_i(v)d_i(v)}{\sum_i w_i(v)}$$

$$W(v) = \sum_i w_i(v)$$

- Surface Integration *[Curless and Levoy 96]*

```
Voxel {
    distance;
    color;
    weight;
}
```

$$D'(v) = \frac{W(v)D(v) + w_k(v)d_k(v)}{W(v) + w_k(v)}$$

$$W'(v) = W(v) + w_k(v)$$

- Surface De-integration: remove d_k from weighted average

```
Voxel {
    distance;
    color;
    weight;
}
```

$$D'(v) = \frac{W(v)D(v) - w_k(v)d_k(v)}{W(v) - w_k(v)}$$

$$W'(v) = W(v) - w_k(v)$$


De-integrated
Nießner


Re-integrated with
updated pose