Implement KinFu: Real-Time Dense Reconstruction and Device Tracking

Zhejin Huang
zhejinh@stanford.edu

Abstract

This project implements Kinect Fusion [1], a dense scene reconstruction and device tracking pipeline that could run in real-time on mobile devices. Results are evaluated with respect to device tracking quality, reconstruction quality, and performance (wall time). Then the pipeline is optimized on all three aspects and further optimizations are proposed.

1. Introduction and Related Work

Reconstructing 3D objects and scenes from multiple images with depth information is a popular topic in computer vision. Moreover, with the thriving of the augmented reality industry in recent years, performing 3D room scene reconstruction on mobile devices in real-time has been of more interest. Kinect Fusion (or KinFu [2]) is a state-of-the-art pipeline adopted by the AR industry nowadays.

Compared to other 3D reconstruction methods such as Space Carving [3] and Structure from Motion methods, KinFu has its advantages. Space carving typically works with simple convex objects and with a calibrated camera and scene in a lab (to get precise extrinsic), while KinFu is designed to work with complex scenes without having to know camera extrinsic. Instead, it estimates the device pose at each frame. Structure from Motion methods such as Tomasi and Kanade factorization [4] needs a set of human-labeled correspondences on all images (which means the pipeline cannot run in real-time) and the results suffer from affine ambiguity, while KinFu could perform reconstruction in real-time without ambiguity and with no requirements on correspondences.

Although KinFu has these advantages, it needs to satisfy two conditions. Firstly, instead of color images consumed by other methods, the inputs of the KinFu pipeline are depth maps (or RGB-D images). This precondition could be fulfilled with the recent advances of affordable real-time structured light depth sensors such as Kinect [5]. Secondly, KinFu assumes the device only performs a small amount of rigid transformation between consecutive frames. This assumption holds because the sensor captures depth frames at a high frame rate (30 fps).

2. Problem Statement

2.1. Problem

Given a sequence of depth map frames associated with timestamps and intrinsic, the system estimates the 6 DOF device associated with each frame's timestamp and reconstructs the scene as Truncated Signed Distance Function (or TSDF).

2.2. Dataset

TUM RGB-D dataset [6] is used for this project. I used depth map and ground truth trajectory while ignoring color images since this project's scope includes device tracking and scene reconstruction without registering color information. There are multiple sequences provided by the dataset, and for this report, I used *freiburg1_xyz* and *freiburg3_cabinet*.

2.3. Evaluation

The real-time tracking-and-reconstructing system is evaluated on three aspects: device poses estimation accuracy, reconstruction quality, and performance (latency, or wall time per frame).

In order to evaluate device pose estimation quality, I adopted the evaluation tools [7] from the TUM dataset by converting the scripts from Python2 to Python3 for environment compatibility. The script *evaluate_rpe.py* computes the relative pose error between the ground truth and estimated trajectories. And the script *evaluate_ate.py* computes absolute trajectory error. Both scripts align the two trajectories using the method of Horn (closed form), so the errors are independent of the coordinate system, which means two trajectories off by a rigid transformation has no error. For the evaluation script to work properly, the output trajectory of the system should be a list of 6 DOF poses associated with timestamps to follow the same format as the ground truth trajectory. Each 6 DOF pose is represented by seven values with three indicating the 3D translation and four indicating the 3D rotation in quaternion [8] format. Note that the ground truth trajectory poses are not necessarily captured at the same time as depth maps. So, it is necessary to align the timestamps when computing the error.

To evaluate reconstruction quality, it makes the most sense to visualize and view the 3D scene from different angles since there is no 3D ground truth in the dataset. It is easy to check whether the reconstruction is successful and whether there are any artifacts through visualization.

In order to evaluate performance which is important for a system to run in real time, I measured the average wall time per (end-to-end) frame in milliseconds. Python3 has its built-in clock library to achieve wall-time measurement.

3. Technical Approach

The approach consists of four components: Surface Measurement, Pose Estimation, TSDF Update, and Surface Prediction. I'll first describe these four components independently and then put them together and illustrate the first 2 frames of the pipeline.

3.1. Surface Measurement

The Surface Measurement component at frame k takes a depth map D_k (with shape (h, w)) as input and computes the vertex map V_k and normal map N_k from the frame k's point of view (device pose). Given intrinsic and extrinsic (at frame k), it is easy to construct a set of points in 3D from a depth map. V_k is a vertex map of shape (h, w, 3) where each slot stores a vector of 3 to indicate the corresponding point's 3D location. N_k is the normal map of shape (h, w, 3) where each slot also stores a vector of 3 to indicate surface normal corresponding to the vertex. The surface normal of $V_k[i][j]$ is computed by the cross product of two neighboring edges and normalization of the result's length:

$N_k[i][j] = (V_k[i+1][j] - V_k[i][j]) \times (V_k[i][j+1] - V_k[i][j])$

$N_k[i][j] = N_k[i][j] / \| N_k[i][j]\|$

3.2. Pose Estimation

Pose Estimation at frame k takes vertex map V_k and normal map N_k of this frame (from Surface Measurement) and estimated vertex map V'_{k-1} and estimated normal map N'_{k-1} (from Surface Prediction executed in the previous frame, which will be discussed in section 3.4) along with $T_{g,k-1}$, the device pose at frame k-1. The annotation g and k-1 mean the rigid transformation from the global coordinate system g to the device's coordinate system at frame k-1. With these inputs, and with the assumption that the rigid transformation between frame k-1 and frame k is small, we could use the point-to-plane ICP (Iterative Closest Point) algorithm to find the transformation between vertex maps and normal maps. Then we could apply this transformation on $T_{g,k-1}$ to get $T_{g,k}$.

3.3. TSDF Update

TSDF is a representation of the scene structure using voxel grids with each voxel storing two values: the truncated signed distance (TSD) of that voxel to the closest surface and the weight of that voxel. The paper uses TSDF to represent the scene for two reasons. Firstly, a high-resolution TSDF representation could represent the scene with good quality. Given a TSDF representation, we could use the marching cube algorithm [9] to get the mesh of the surfaces, or we could also use volume ray casting [10] to render the scene from any camera pose (used by Surface Prediction in section 3.4). Secondly, it is efficient to fuse two TSDFs by computing the weighted TSD on each voxel. In this case, the weight is the number of observations for each voxel. A voxel with weight zero means that the voxel is uninitialized. An uninitialized voxel's SDF will be initialized on its first observation and maintain the average of all observations afterward.

TSDF Update component executed at frame k takes the device current pose $T_{g,k}$ and depth map D_k as inputs. It first computes the current frame's TSDF representation S'_k and then fuses (as discussed above) it with global TSDF representation S_{k-1} to get S_k.

To get S'_k, we project each voxel to the depth image plane (with intrinsic from the dataset and extrinsic from Pose Estimation) and sample its depth value. Then we truncate the depth values (according to predetermined truncate parameters) to get the voxels' TSD values. It is obvious that this process approximates, instead of accurately computing the distance to each surface, the TSD value for each voxel by projecting it to the depth map. Practically, this approximation works well because it still accurately represents surfaces, which are what we care about. Plus, the inaccuracy introduced by this approximation is averaged out by multiple views.

3.4. Surface Prediction

The Surface Prediction component takes the device pose $T_{g,k}$ and global TSDF S_k as inputs to compute the estimated vertex map V'_k and estimated normal map N'_k at the current frame pose. Both maps are consumed by Pose Estimation during the next frame (described in section 3.2).

V'_k and N'_k are rendered using the volume ray casting algorithm. For each image location (i, j) on a device whose pose is defined by $T_{g,k}$, we could cast a ray from the camera center through the image location outwards into the world. Along this ray, we could search from the intersection voxel by a step size of voxel size and find the first pair of voxels whose TSD values change from positive to negative. We then tri-linearly interpolate the voxels based on TSD values to get the surface intersection point location where the TSD value becomes zero. After getting V'_k using this way, we then access the neighboring voxels to get the gradient and construct N'_k based on that.

3.5. KinFu system

Figure 1 illustrates the system at frame 0 and frame 1. For frame 0, the Surface Measurement and the Pose Estimation components are skipped. The camera coordinate system of the frame 0 is initialized as the global coordinate, which means that $T_{g,0}$ is an identical transformation. In the TSDF Update, which is the first step, frame 0's TSDF S'_0 is estimated based on D_0 used as an initialization for global TSDF S_0. Next, Surface Prediction takes S_0 to render V'_0 and N'_0 from pose $T_{g,0}$. V'_0 and N'_0 are stored for frame 1 to use.

For frame 1, the Surface Measurement takes D_1 to compute V_1 and N_1. Next step, the Pose Estimation takes V'_0 and N'_0 from frame 0 along with V_1 and N_1 from the Surface Measurement to get $T_{g,1}$ based on $T_{g,0}$. Then TSDF S_0 is updated to S_1 by fusing the current frame's TSDF S'_1 computed from $T_{g,1}$ and D_1. Lastly, Surface Prediction takes S_1 to render V'_1 and N'_1.

The processes of frame 1 are repeated from frame 2 until the last frame. From the system's perspective, it takes a sequence of depth map frames D_0, D_1, …, D_N as input and outputs the device pose $T_{g,0}$, $T_{g,1}$, …, $T_{g,N}$ and scene S_0, S_1, …, S_N in real-time.

4. Data pre-processing and evaluation

4.1. Dataset

In order to make the pipeline run, the first step is to download a sequence from TUM datasets and uncompress it locally in my repository's [11] *data/datasets/* folder. In each data sequence's folder, depth maps are in the *depth/* folder and are named as their timestamps of sensor exposure time. The ground truth trajectory exists in *groundtruth.txt* as a list of 6DOF poses associated with their timestamps. Note that the timestamps of the depth maps and the poses are not aligned. Other files such as RGN images (e.g., Figure 2 is from the *freiburg1_xyz* sequence) are not used for this project.

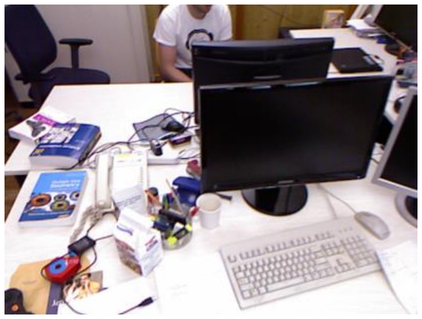
Figure 2: An RGB image from *freiburg1_xyz*

4.2. Input pre-processing

Utilities I implemented for data pre-processing exist in *data/* folder and in *pipeline.py*.

The inputs for the pipeline are depth frames only. At a run time frame, a depth frame (e.g., Figure 3), along with its timestamp, is loaded to the pipeline and the depth values are scaled according to TUM dataset specification. Camera intrinsics (which are used to convert a depth map to a 3D point cloud) are set according to TUM specifications.

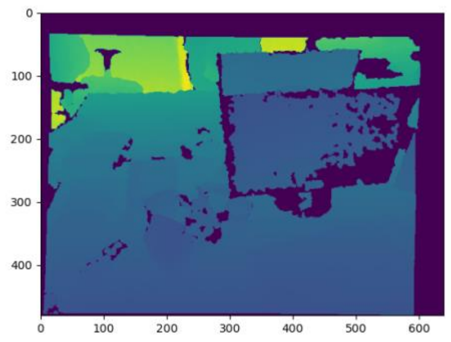

Figure 3: A Depth map from *freiburg1_xyz*

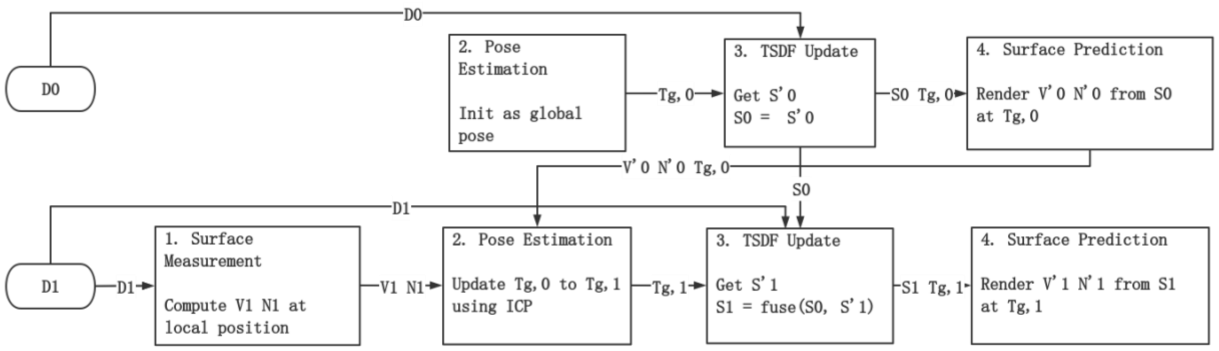

Figure 1: KinFu system flow

The paper suggests using a bilateral filter to smooth the depth maps. I experimented and decided to use filter size 15 and sigma value 75 and applied the filter on disparity (or 1 / depth) space. Figure 4 shows a point cloud computed from an unfiltered depth map and Figure 5 shows a point cloud from a filtered depth map.

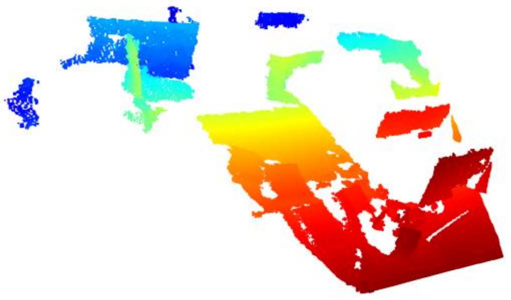
Figure 4: Point cloud from unfiltered depth

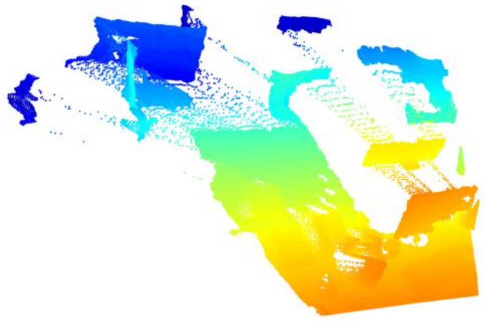
Figure 5: Point cloud from filtered depth

As we can see from the above figures, the filtered depth map is smoother with fewer holes. An artifact of filtering is that there are points flying in the air around the edges where the depth values discontinue. However, with multiple views, those points will be corrected by updating the TSDF, which removes any points that are floating above an existing surface. So, the artifact won't impact the final reconstruction result.

4.3. Output evaluation

Utilities for trajectory evaluation are in the *eval/* folder of my repository. They are originally from the TUM dataset and adopted to Python3 by me. Scripts *evaluate_ate.py* and *evaluate_ate.py* are used for this project.

To evaluate reconstruction quality, I extracted the point clouds from the reconstructed TSDF representation and visualized them from any point of view.

To evaluate performance, I measured each component in every frame using the built-in clock.

5. Experiments and Optimizations

5.1. Pipeline implementation

The implementations of the four components and their utilities are in the *algorithm/* folder of my repository.

For the Surface Measurement component, I computed the Vertex map in NumPy, then loaded it into an Open3D point cloud and used Open3D to compute the Normal map. Computing the Normal Map using Open3D is to ensure that it is aligned with the one computed from the Surface Prediction. The code for this component is in *camera.py* and *measurement.py*.

For Pose Estimation, I adopted Open3D's point-to-plane ICP [12], which implements a non-linear iterative method to match two point clouds and compute the transformation matrix between them. After getting the 4x4 transformation matrix (first translate, then rotate) for each frame's pose, I converted it to a trajectory with format *[tx ty tz qx qy qz qw]*, where tx, ty, tz represents the 3 DOF translation and qx, qy, qz, qw is a quaternion representation of the 3 DOF rotation. I exercised the Pose Estimation component in *algorithm/pose.py* and implemented trajectory conversion in *data/trajprocessor.py*.

For TSDF Update and Surface prediction, I adopted Open3D's implementation [13] based on a Hash Map of voxel grid blocks. Each block consists of NxNxN voxels (N = 16 in my case) and is created and registered to the world coordinate system at run time. This Hash Map implementation guarantees both high performance (low compute and space complexity) and runtime extensibility. Utilities for creating and updating TSDF are in *algorithm/update.py* and utilities for ray casting TSDF are in *algorithm/prediction.py*.

The overall pipeline which integrates data pre-processing and the four components is in *pipeline.py*. With the dataset and environment setup, the pipeline could be run by a command *Python3 pipeline.py* from the root folder of my repository.

5.2. Performance Optimization

After getting the pipeline to run, the first issue was that it ran too slowly. The wall time for each frame is 123.9s on average, which means it would take over 30 hours to run through a dataset with 1000 frames.

I used the built-in clock to break down the wall time of each component, and found that the bottleneck is on Pose Estimation, which took 110.4s per frame on average (see Table 1). Note that the complexity of ICP is $O(N^2)$ where N is the number of points in the point cloud. In this case, N is approximately 3e5 given the 640 x 480 depth map, and N^2 is around 9e10, which is too high for a personal computer's CPU and DRAM to operate in real-time.

To cut down the computing on Pose Estimation, I down-sampled the point clouds from Surface Measurement and Surface Prediction evenly on x and y dimensions by 10 times. Thus, the point clouds used for ICP had only 1% points compared to the previous implementation. With this change, Pose Estimation's wall time decreased from 110.4s to 133ms, and other components were also hugely sped-up due to less computing and less memory contention. The results of this optimization are shown below in Table 1.

	Overall	Depth Streaming	Surface Measurement	Pose Estimation	TSDF Update	Surface Prediction
Before	123.9 s	6.3 ms	7.0 s	110.4 s	29.7 ms	6.4 s
After	987 ms	3.6 ms	10.9 ms	133.385 ms	24.4 ms	814.1 ms

Table 1: Wall time before and after optimization

It is worth mentioning that after reducing the number of points, ICP still works as expected, which means that the pipeline does not need such a dense point cloud for pose prediction. In my code, the parameter *down_sample_factor* in *camera.py* controls the down-sampling factor. Setting the factor to larger than 10 could also work, but since Pose Estimation is no longer the bottleneck (now it is Surface Prediction), I kept it to 10.

With the overall pipeline able to run at 1 FPS, it takes around 20 minutes to evaluate one dataset, which is good enough for this project's purpose. A potential (and most beneficial) future optimization is running the pipeline on GPU to fully utilize its parallelism on per-pixel operations and matrix multiplications to achieve real-time processing (30 FPS, which is the interval depth map is taken). To achieve real-time processing, converting all the NumPy tensors used in the pipeline to Open3D tensors and resolving the compatibility issues are necessary.

5.3. Reconstruction Optimization

After getting the pipeline running at 1 FPS, I could evaluate the quality of the pipeline. However, the reconstruction failed quickly after 10 frames because of noise. I got the reconstruction results visualized below. The left image shows the reconstruction result after 5 frames, and the right one shows the result after 10 frames. As we can see, the reconstruction quickly became too noisy after only 5 frames.

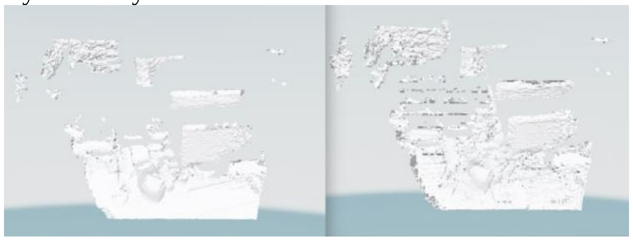

Figure 6: Noisy results at frame 5 and 10

This issue was due to a low depth recall rate. More than 20% of the depth values in a depth map are invalid due to either sensor failure or occlusion. I resolved this issue by removing the invalid depths values in both Pose estimation component and TSDF update component. After filtering out bad depth values, the pipeline was able to run for a larger number of frames without failure. The reconstruction result at the frame 50 is shown in Figure 7.

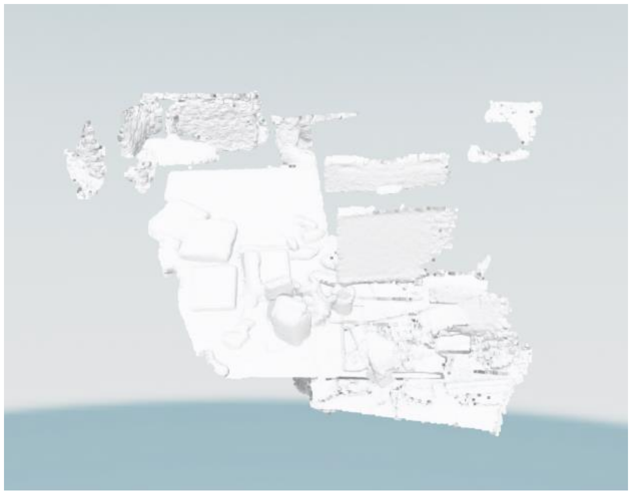

Figure 7: Result at frame 50 without invalid depths

5.4. Pose Optimization

However, with the above optimizations, another artifact still impacted the system significantly. At around frame 100, the estimated pose suddenly drifted away and was never able to come back. As shown in Figure 8 below, the blue curves showed the estimated trajectory and the red curves showed the ground truth trajectory at frame 50, 100, 150 and 200. Note that the estimated and ground truth trajectories are aligned using the method of Horn, which caused an estimated trajectory to transform to a different position when it is off by a lot from the ground truth trajectory. Even worse, when we look at the reconstruction result at frame 100, the scene is reconstructed twice in the air (due to TSDF update from a totally incorrect pose). The pipeline is largely impacted by Pose Estimation failure.

I was able to find the reason for this artifact when I ran the pipeline on another sequence *freiburg3_cabinet* from which I observed the pose estimation suddenly failed when a new surface not observed from the initial point of view appearred in the point of view. After the first pose estimation failure happened, it brought errors to reconstruction results, which further propagated to worse pose estimation. So, in this pipeline, once there is a slightly inaccurate pose estimated, it would quickly blow up without recovery and the pipeline would fail.

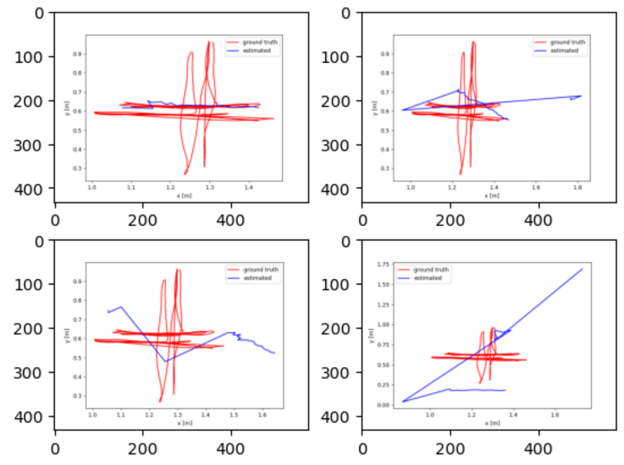
Figure 8: Trajectory at frame 50, 100, 150, 200

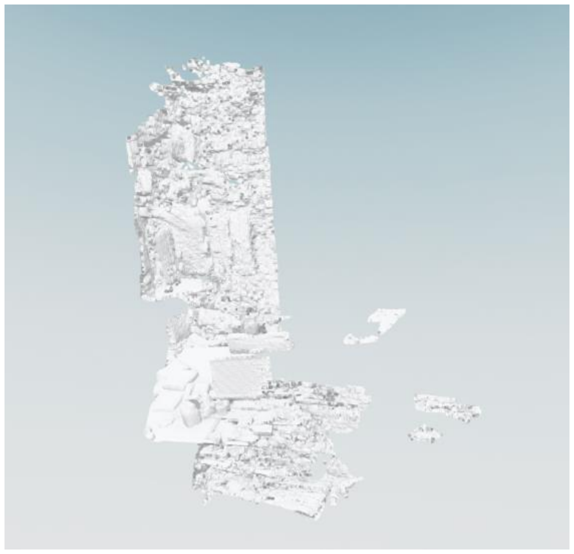
Figure 9: Bad result at frame 100

To mitigate this issue, I turned to estimate the pose transformation from the previous frame to the current frame, instead of from the global frame to the current one. This change has two benefits: firstly, ICP on two consecutive frames is easier to converge; secondly, it is easier to match to point clouds precisely and smoothly from two consecutive. The resulting trajectories at frames 50, 100, 150 and 200 are shown in Figure 10. And the reconstruction result after all the 798 frames is shown in Figure 11. Both pose estimation and scene reconstruction were significantly improved. Whether to estimate the pose from global or from the previous frame is controlled by the parameter *ICP_from_global* in *pipeline.py*.

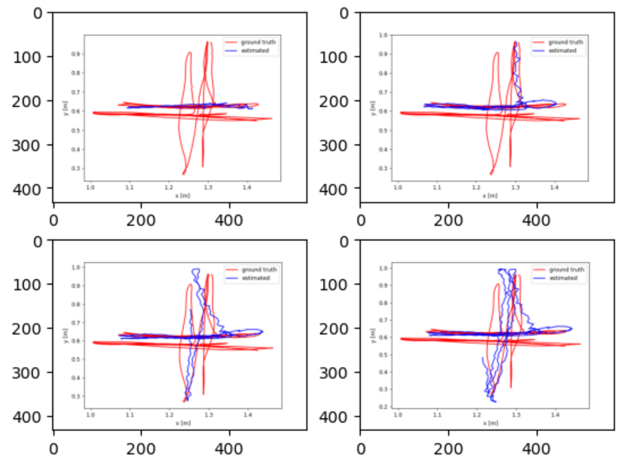
Figure 10: Trajectory at frame 100, 200, 300, 400

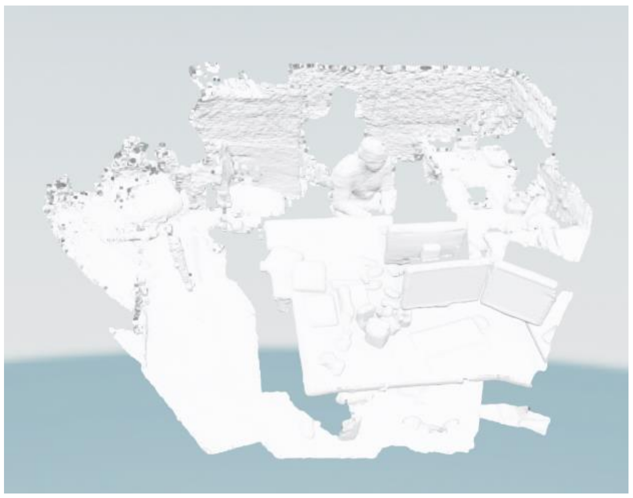
Figure 11: Good result at the last (798th) frame

5.5. Potential Solutions for loop closure

The pipeline has successfully performed reconstruction and pose estimation on *freiburg1_xyz* sequence. In this sequence, the device moves either horizontally or vertically back and forth in an office scene without a loop. How about moving the device around a loop? The *freiburg3_cabinet* sequence in TUM dataset is captured by moving the device around an object along a loop. By running the pipeline on this sequence, I got the trajectory results for frames 250, 500, 750 and 1000 in Figure 12.

At up to frame 500 (first row right), the pose was pretty good, while around frame 750 (second row left) there was a significant deviation. This deviation happened when the camera looped back and saw the surface it had seen before. The error accumulated through Pose Estimations from all the previous frames makes another reconstruction of the same surface overlayed on the already reconstructed one, which makes the reconstruction no longer reasonable. Once this reconstruction inconsistency happens, the error will suddenly be propagated to Pose Estimation and back to TSDF Update, blowing up the error of the system.

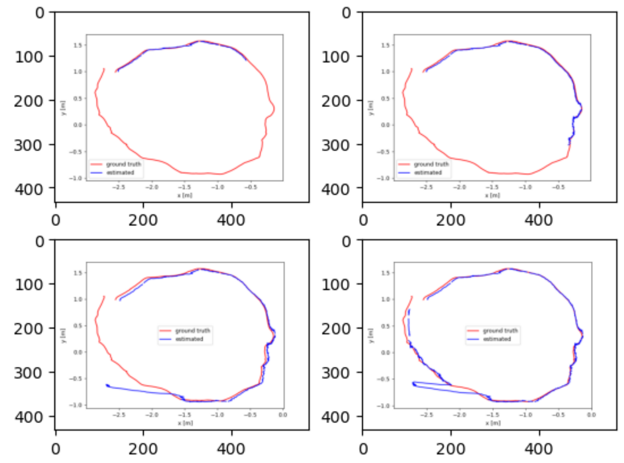

Figure 12: trajectory at frame 250,500,750,1000

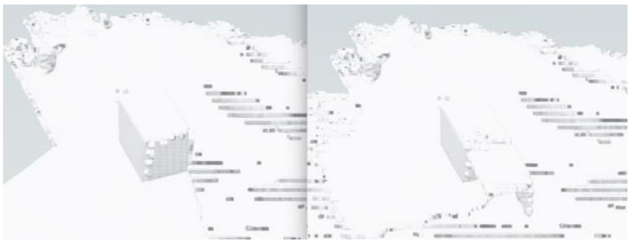

Figure 13: reconstruction results at frame 500 and 750

To resolve this issue, a straightforward idea is to use higher quality pose estimation instead of estimating from ICP. To validate this idea, I tried to blend ground-truth poses (queried from depth map timestamps, implemented in *data/trajprocessor.py*) into the estimated poses controlled by a ratio (which is the parameter *gt_traj_ratio* in *pipeline.py*) of frames. However, the reconstruction quality did not look well. The reason is likely that the frames that I used ground truth pose as extrinsic might have had a point cloud misalignment with the previous frame (since the previous pose is estimated from ICP but the current pose is from ground truth). And this misalignment causes a similar system issue as introduced by loop closure.

Still, the direction above is valid if high-quality poses could be better integrated. Overall, there are three potential solutions for the loop closure issue. Firstly, for the pose estimation algorithm, using a standalone SLAM system with higher quality is better than estimating from ICP. Secondly, for points used for ICP, instead of using an evenly sampled depth map, we could detect meaningful and representative feature points, track them across frames and perform ICP based on point clouds from those points. Thirdly, for loop closure handling, when the system detects a loop closure, it should adjust the trajectory in the past to make sure the loop is smoothed and remove the incorrect reconstructions based on those drifted poses. With this handling, the system would be able to converge.

6. Conclusions

In this project, I implemented the KinFu pipeline and evaluated its performance, device tracking quality and reconstruction quality. Based on these evaluations, I optimized performance by reducing the number of points used for ICP. Then I reduced the reconstruction noise by filtering out invalid depths from ICP and TSDF Update. After that, I made the device tracking able to run more robustly by ICP on the frame delta. Lastly, I tried to solve the issue brought about by loop closure using high-quality device pose. The specific implementation was not successful, but the direction needs to be better explored.

Furthermore, I proposed a list of potential future optimizations to make the system more performant and more robust. For performance, it would be promising to run the whole pipeline on GPU to achieve real-time processing. For device tracking, it is better to use more robust feature points (instead of randomly sampled points from the depth map) for ICP; or to use a more robust SLAM pipeline for Pose Estimation. For reconstruction quality at already visited scenes (such as loop closure), removing and updating the part of TSDF representation based on the incorrect pose is necessary.

As an engineer working on power and performance optimizations for algorithms used in Augmented Reality, I've been hoping to get more context on how the algorithms work, what the artifacts are like and how they are resolved. Scene Reconstruction and Pose Estimation are two of the most important building blocks for Augmented Reality, and KinFu is the foundation of the Scene Understanding solutions that most companies are working on. By working on this project, I'm able to get those contexts and learned how to use Open3D as an efficient tool for demos and scientific experiments.

References

[1] R. A. Newcombe et al., "KinectFusion: Real-time dense surface mapping and tracking," 2011 10th IEEE International Symposium on Mixed and Augmented Reality, Basel, Switzerland, 2011, pp. 127-136, doi: 10.1109/ISMAR.2011.6092378.

[2] https://docs.opencv.org/4.x/d8/d1f/classcv_1_1kinfu_1_1KinFu.html

[3] Kutulakos, K.N., Seitz, S.M. A Theory of Shape by Space Carving. *International Journal of Computer Vision* 38, 199–218 (2000).

[4] Tomasi, Carlo, and Takeo Kanade. "Shape and motion from image streams: a factorization method." *International journal of computer vision* 9.2 (1992): 137-154.

[5] Z. Zhang, "Microsoft Kinect Sensor and Its Effect," in *IEEE MultiMedia*, vol. 19, no. 2, pp. 4-10, Feb. 2012, doi: 10.1109/MMUL.2012.24.

[6] https://vision.in.tum.de/data/datasets/rgbd-dataset

[7] https://vision.in.tum.de/data/datasets/rgbd-dataset/tools

[8] https://en.wikipedia.org/wiki/Quaternion

[9] https://web.cs.wpi.edu/~matt/courses/cs563/talks/powwie/p1/ray-cast.htm

[10] H. Ray, H. Pfister, D. Silver and T. A. Cook, "Ray casting architectures for volume visualization," in *IEEE Transactions on Visualization and Computer Graphics*, vol. 5, no. 3, pp. 210-223, July-Sept. 1999, doi: 10.1109/2945.795213.

[11] My repository: https://github.com/YHHHCF/KinectFusion

[12] http://www.open3d.org/docs/release/tutorial/pipelines/icp_registration.html

[13] http://www.open3d.org/docs/latest/python_api/open3d.t.geometry.VoxelBlockGrid.html