

ImMesh: An Immediate LiDAR Localization and Meshing Framework

Jiarong Lin*, Chongjian Yuan*, Yixi Cai, Haotian Li, Yunfan Ren, Yuying Zou, Xiaoping Hong, and Fu Zhang

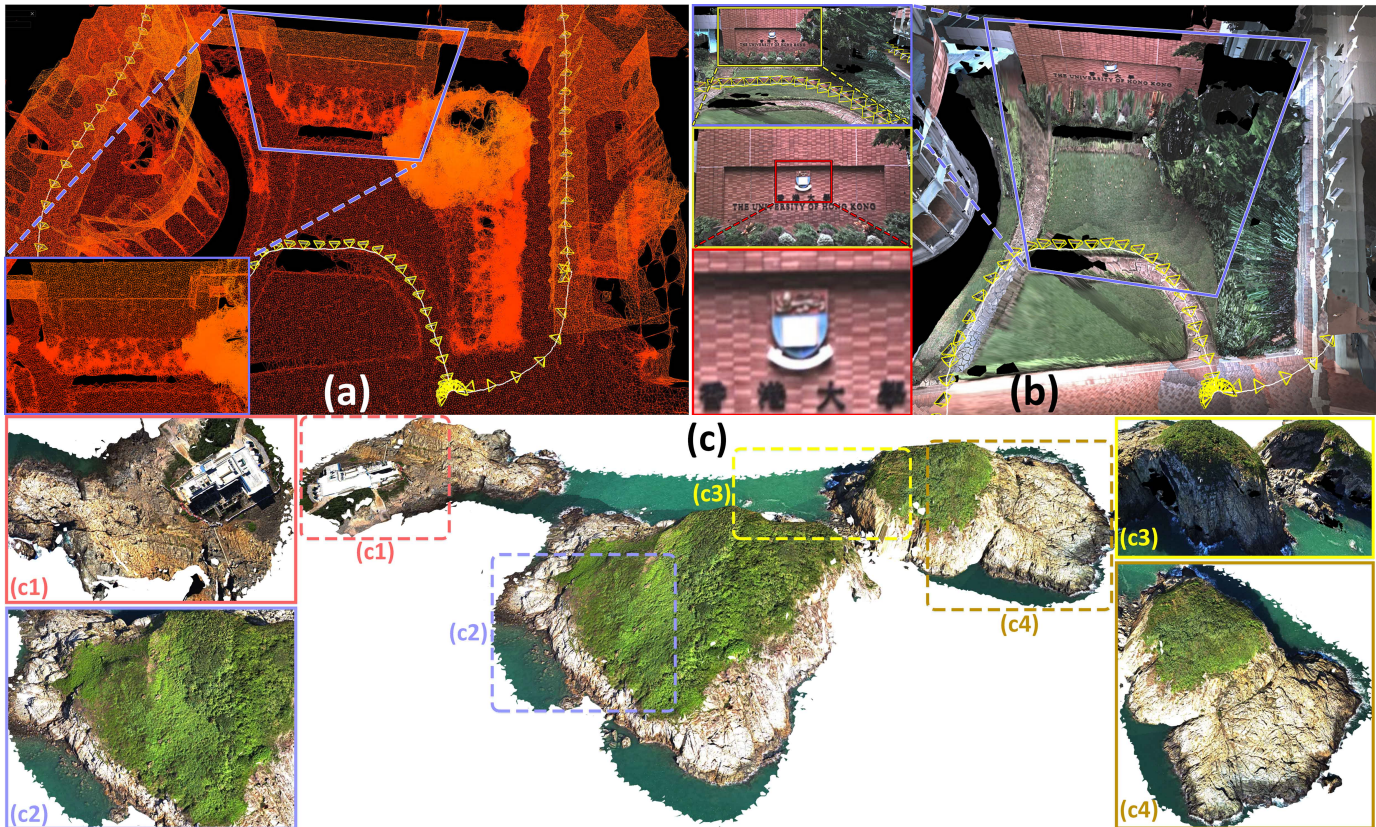


Fig. 1: (a) shows the triangle mesh that is online reconstructed by our proposed work ImMesh, where the white path is our sampling trajectory, and the yellow frustums are the estimated sensor pose. In (b), we use the estimated camera poses (the yellow frustums) of R³LIVE for texturing the mesh with the collected images. Based on ImMesh, we developed a lossless texture reconstruction application, with one of our results shown in (c). Our accompanying video that shows details of this work is available on YouTube: youtu.be/pzT2fMwz428.

Abstract—In this paper, we propose a novel LiDAR-(inertial) odometry and mapping framework to achieve the goal of simultaneous localization and meshing in real-time. This proposed framework termed ImMesh comprises four tightly-coupled modules: receiver, localization, meshing, and broadcaster. The localization module first utilizes the preprocessed sensor data from the receiver, estimates the sensor pose online by registering LiDAR scans to maps, and dynamically grows the map. Then, our meshing module takes the registered LiDAR scan for incrementally reconstructing the triangle mesh on the fly. Finally, the real-time odometry, map, and mesh are published via our broadcaster. The primary contribution of this work is the meshing module, which represents a scene by an efficient voxel structure, performs fast finding of voxels observed by new scans, and incrementally reconstructs triangle facets in each voxel. This voxel-wise meshing operation is delicately designed

for the purpose of efficiency; it first performs a dimension reduction by projecting 3D points to a 2D local plane contained in the voxel, and then executes the meshing operation with pull, commit and push steps for incremental reconstruction of triangle facets. To the best of our knowledge, this is the first work in literature that can reconstruct online the triangle mesh of large-scale scenes, just relying on a standard CPU without GPU acceleration. To share our findings and make contributions to the community, we make our code publicly available on our GitHub: github.com/hku-mars/ImMesh.

Index Terms—Mapping, 3D reconstruction, SLAM

I. INTRODUCTION

RECENTLY, the wide emergence of 3D applications such as metaverse [1, 2], VR/AR [3], video games, and physical simulator [4, 5] has enriched human lifestyle and boosted productive efficiency by providing a virtual environment that resembles the real world. These applications are built upon triangle meshes that represent the complex geometry of real-world scenes. Triangle mesh is the collection of vertices and triangle facets, which serves as a fundamental tool for object modeling in most existing 3D applications. It can not only significantly simplify the process and boost the speed of rendering [6, 7] and ray-tracing [8], but also

Manuscript received February 5, 2023; revised July 23, 2023; accepted September 18, 2023. This work is supported by the University Grants Committee of Hong Kong General Research Fund (project number 17206421) and DJI Donation. (Corresponding author: Fu Zhang.)

*These two authors contribute equally to this work.

J. Lin, C. Yuan, Y. Cai and F. Zhang are with the Department of Mechanical Engineering, The University of Hong Kong, Hong Kong SAR, China. {jiarong.lin, ycjl, yixicai, haotianli, renyf, zyycici, fuzhang}@connect.hku.hk

X. Hong are with the School of System Design and Intelligent Manufacturing, Southern University of Science and Technology, Shenzhen, People's Republic of China. {hongxp}@sustech.edu.cn

play an irreplaceable role in collision detection [9, 10], rigid-body dynamics [11, 12], dense mapping and surveying [13], sensor simulation [14, 15], etc. However, most of the existing mesh is manufactured by skillful 3D modelers with the help of computer-aided design (CAD) software (e.g., Solidworks [16], blender [17], etc.), which limits the mass production of large-scene meshing. Hence, developing an efficient mesh method that could reconstruct large scenes in real-time draws increasing research interests and serves as a hot topic in the community of 3D reconstruction.

Performing mesh reconstruction in real-time is particularly important in practical usages. Firstly, online mesh reconstruction makes data collection effective by providing a live preview, which is essential to give users a reference. Especially for non-expert users, a live preview can provide feedback about which parts of the scene have already been reconstructed in good quality and where additional data is needed. Secondly, online mesh reconstruction can immediately output the mesh of the scene once data collection is complete, saving additional post-processing time of offline mesh reconstruction and boosting the productivity of mass production. Thirdly, it is particularly important for those real-time applications, especially fully autonomous robotic applications. A real-time update of mesh can provide better maps with denser representation and higher accuracy, enabling the agent to navigate itself better.

Reconstructing the mesh of large scenes from sensor measurements in real-time remains one of the most challenging problems in computer graphics, 3D vision, and robotics, which require reconstructing the surfaces of scenes with triangle facets adjacently connected by edges. This challenging problem needs to build the geometry structure with very high accuracy, and the triangle facet should be reconstructed on surfaces that actually exist in the real world. Besides, a good mesh reconstruction method should also suppress the appearance of holes on the reconstructed surface and avoid the reconstruction of triangle silver (i.e., the noodle-like triangles with an acute shard angle). Real-time mesh reconstruction in large scenes is even more challenging as it further requires the reconstruction to operate efficiently and incrementally.

In this work, we propose a real-time mesh reconstruction framework termed ImMesh to achieve the goal of simultaneous localization and meshing on the fly. ImMesh is a well-engineered system comprised of four tightly-coupled modules delicately designed for efficiency and accuracy. Among them, we implement a novel mesh reconstruction method in our meshing module. Specifically, our meshing module first utilizes the voxels for partitioning the 3D space and allows fast finding of voxels that contain points of new scans. Then, the voxel-wise 3D meshing problem is converted into a 2D one by performing dimension reduction for efficient meshing. Finally, the triangle facets are incrementally reconstructed with the voxel-wise mesh pull, commit and push steps. To the best of our knowledge, this is the first work in literature to reconstruct the triangle mesh of large-scale scenes online with a standard CPU. The main contributions of our work are:

- We propose ImMesh, a novel SLAM framework designed to achieve simultaneous localization and mesh reconstruction using a LiDAR sensor. ImMesh is built upon our

previous work VoxelMap [18], and incorporates a novel mesh reconstruction method. This proposed approach can efficiently and incrementally reconstruct the mesh of scenes online, achieving real-time performance in large-scale scenarios on a standard desktop CPU.

- We comprehensively evaluated ImMesh’s runtime performance and meshing accuracy using real-world and synthetic data, by comparing our runtime performance and meshing accuracy against existing baselines to assess its effectiveness.
- We additionally demonstrate how real-time meshing can be applied in potential applications by presenting two practical examples: point cloud reinforcement and lossless texture reconstruction (see Fig. 1(b and c)).
- We make ImMesh publicly available on our GitHub: github.com/hku-mars/ImMesh for sharing our findings and making contributions to the community,

II. RELATED WORKS

In this section, we discuss the related works of mesh reconstruction based on 3D point clouds, which are closely related to this work. Depending on whether the reconstruction processes can perform online, we categorize existing mesh reconstruction methods into two classes: offline methods and online methods.

A. Offline mesh reconstruction

The offline methods usually require a global map in prior, for example, the full registered point cloud of the scene. Then, a global mesh reconstruction process is used to build the mesh. In this category, the most notable works include: methods based on Poisson surface reconstruction (Poisson-based), and methods based on Delaunay tetrahedralization (i.e., 3D Delaunay triangulation) and graph cut (Delaunay-based).

1) *Poisson surface reconstruction (Poisson-based)*: Given a set of 3D points with oriented normals that are sampled on the surface of a 3D model, the basic idea of Poisson surface reconstruction [19, 20] is to cast the problem of mesh reconstruction as an optimization problem, which solves for an approximate indicator function of the inferred solid whose gradient best matches the input normals. Then, the continuous isosurface (i.e., the triangle mesh) is extracted from the indicator function using the method [21, 22], similar to adaptations of the Marching Cubes [23] with octree representations.

Benefiting from this implicit representation, where the mesh is extracted from the indicator function instead of being estimated directly, Poisson surface reconstruction can produce a watertight manifold mesh and is resilient to scanner noise, misalignment, and missing data. Hence, in the communities of graphics and vision, these types of methods [19, 20, 24] have been widely used for reconstructing the mesh from given 3D scanned data.

2) *Delaunay triangulation and graph cut (Delaunay-based)*: In the category of offline mesh reconstruction methods, approaches [25]–[27] based on Delaunay tetrahedralization and graph cut have also been widely used for generating

the mesh, relying on the reconstructed 3D point cloud and the sensor’s poses. The basic idea of this class of methods is first to build a tetrahedral decomposition of 3D space by computing the 3D Delaunay triangulation of the 3D point set. Then, the Delaunay tetrahedra were labeled as two classes (i.e., “inside” or “outside”) with the globally optimal label assignment (i.e., the graph cut). Finally, the triangle mesh can be extracted as the interface between these two classes.

Besides these two classes of methods, there are other offline mesh reconstruction methods, such as the ball-pivoting algorithm [28]. This algorithm works by pivoting a ball of fixed radius around each point in the point cloud and constructing a triangle whenever three balls overlap. [29] involves extracting the curve skeleton using Laplacian-based contraction, and then reconstructing the surface with the skeleton-assisted topology. However, these methods are often not the first choice due to various limitations such as robustness, accuracy, and efficiency when compared to Poisson- and Delaunay-based methods [30].

Unlike these offline mesh reconstruction methods, our proposed work ImMesh can perform online in an incremental manner without the complete point cloud of the scene. Besides, ImMesh also achieves a satisfactory meshing accuracy that is higher than Poisson-based methods and slightly lower than Delaunay-based methods (see our experimental results in Section VIII-C).

B. Online mesh reconstruction

1) *Voxel volume-based methods (TSDF-based)*: The online mesh reconstruction method is predominated by TSDF-based methods, which represent the scene in a voxel volumetric theme. These methods implicitly reconstruct the mesh in a two-step pipeline, which first establishes the truncated signed distance to the closest surface of voxels, then extracts the continuous triangle mesh by leveraging the Marching Cubes algorithm [23] from volumes. TSDF-based methods are popularized by KinectFusion [31], with many follow-up works focused on scaling this approach to larger scenes [32, 33], adding multi-resolution capability [34, 35], and improving efficiency [36]–[38]. Since these classes of methods can be easily implemented with parallelism, they can achieve real-time performance with the acceleration of GPUs.

Compared to these methods, our work ImMesh shows several advantages: Firstly, in ImMesh, the triangle mesh is directly reconstructed from the point cloud in one step, while for TSDF-based methods, the mesh is implicitly built in a two-step pipeline (i.e., SDF update followed by a mesh extraction). Secondly, ImMesh can output the mesh in scan rate (i.e., sensor sampling rate), while the mesh extraction of TSDF-based methods is usually at a lower rate. Thirdly, ImMesh achieves real-time performance by running on a standard CPU, while TSDF-based methods need GPU acceleration for real-time SDF updates. Lastly, TSDF-based methods require adequate observation for the calculation of the SDF of each voxel w.r.t. the closest surface, which needs the data to be sampled by a depth sensor of high resolution and moving at a low speed. On the contrary, our work exploits high-accuracy LiDAR points for meshing and is robust to points data of low density.

2) *Surfel-based mesh reconstruction*: Besides TSDF-based methods, another popular approach is representing the scene with a set of points or surfels (e.g., oriented discs). For example, in work [33, 39, 40], the maps are reconstructed with point-based representation, and its “surface” is rendered with the approaches of “point-based rendering” that originated from the communities of computer graphics [41]–[43]. Besides, in work [44], the high-quality map is reconstructed with surfel-based representations (i.e., use patches). Such forms of mapping representation are popularized in works [45]–[48]. To reconstruct a dense map, these classes of methods need a large number of points or tiny patches to represent the surface of the models, which is an inefficient representation with high usage of system memory and computation resources. In contrast, our work reconstructs the surface of models with triangle mesh, which uses triangle facets of proper size adjacently connected by edges. It is the most efficient solid-model representation that has been widely adopted in most modern 3D software.

Compared with the works reviewed above, our proposed work is in a class by itself, which contains the following advantages:

- It is an online mesh reconstruction method that reconstructs the triangle mesh in an incremental manner. It can achieve real-time performance in large-scale scenes (e.g., traveling length reaches 7.5 km) by just running on a standard desktop CPU.
- It explicitly reconstructs the triangle mesh by directly taking the registered LiDAR points as meshing vertices, performing the voxel-wise meshing operation as each new LiDAR scan is registered.
- It is delicately designed for the purpose of efficiency and achieves satisfactory meshing precision comparable to existing high-accuracy offline methods.

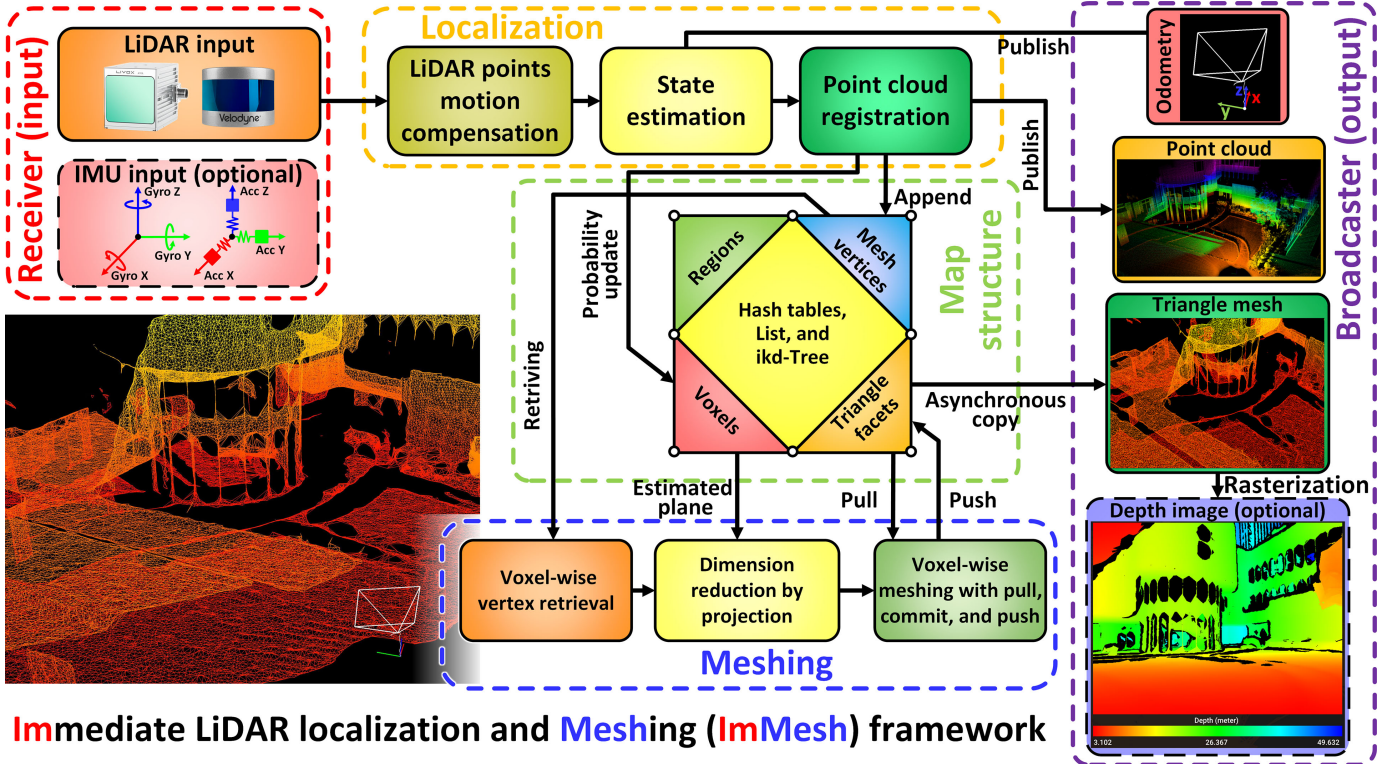
III. SYSTEM OVERVIEW

Fig. 2 depicts the overview of our proposed system (ImMesh), which consists of a map structure and four modules that work jointly to achieve the goal of simultaneous localization and meshing in real-time. As shown in Fig. 2, from left to right are: *receiver* (in red), *localization* (in orange), *map structure* (in green), *meshing* (in blue) and *broadcaster* (in purple).

In the rest sections, we will first introduce our *map structures* in Section IV, showing the detail of the data structures used in other modules. Next, we will introduce our receiver and localization module in Section V. Then, we will present how our *meshing* modules work in Section VI. Finally, in Section VII, we will introduce the *broadcaster* module, which publishes the localization and meshing results to other applications.

IV. MAP STRUCTURE

As shown by the *map structure* (in green) in Fig. 2, we designed four data types, including mesh vertices, triangle facets, regions, and voxels, as well as two data structures: a hash table for efficient data lookup and an incremental



Immediate LiDAR localization and Meshing (ImMesh) framework

Fig. 2: This figure shows the overview of our proposed work ImMesh, which utilizes the raw input sensor data to achieve the goal of simultaneous localization and meshing. It is constituted by four tightly-coupled modules and a map structure, from left (input) to right (output) are: *receiver* (in red), *localization* (in orange), *map structure* (in green), *meshing* (in blue) and *broadcaster* (in purple).

kd-tree (ikd-tree) for k nearest neighbors (kNN) search and downsampling.

The relationship among these map structures is depicted in Fig. 3, where we partition the 3D space into two types of volumetric grids: regions and voxels. Triangle facets are stored inside the regions containing them and are also indexed in a global hash table of triangle facets, and mesh vertices are stored inside the voxels containing them and are also indexed in a global list of vertices. Additionally, we maintain two hash tables to facilitate the efficient lookup of regions and voxels.

A. Data types: Region, voxel, triangle facet, and mesh vertex

1) *Region R*: Region have a much larger size S_R (e.g., $S_R = 10.0\text{m}$) compared to voxel's size S_O (e.g., $S_O = 0.4\text{m}$). They contain triangle facets whose centers are located inside, allowing for the *broadcaster* to asynchronously copy these triangle facets. Additionally, each region has a status flag f_R to identify its syncing status, which can be either *Sync-required* or *Synced*. This status indicates the update flag related to the data synchronization of triangle facets.

2) *Voxel O*: Voxels enable the *meshing* module to efficiently retrieve all in-voxel mesh vertices for voxel-wise meshing operations. Each voxel O_i also has a status flag f_O indicating whether it has new points appended. Specifically, O_i is marked as *Activated* if new mesh vertices are registered from the latest LiDAR scan. The *Activated* flag is reset to *Deactivated* after the voxel-wise meshing operation has been performed on this voxel.

3) *Triangle facet T*: In our work, triangle facets are stored in regions. A triangle facet describes a small surface that exists in the reconstructed scene. It is maintained online by our *meshing* module and is asynchronously copied to the *broadcaster* module for publishing. For a triangle facet T , it is constituted by the following elements: 1) The sorted indices $Pts_id(T)$ of three mesh vertices that form this triangle: $Pts_id(T) = \{i, j, k\}$, $i < j < k$. 2) The center $Center(T)$ and normal $Norm(T)$ (both in the global reference frame) of this facet.

4) *Mesh vertex V*: In ImMesh, mesh vertices are the points that constitute the geometric structure (shape) of mesh. For the i -th vertex V_i , it contains the following elements: 1) The unique index (id) of this vertex $Id(V_i)$ in the global list containing all the vertices in the map. 2) Its 3D position $Pos(V_i) \in \mathbb{R}^3$ in the global frame. 3) The list $Tri_list(V_i)$ of triangles facets whose vertices contain V_i .

B. Data structure: Hash tables and Incremental kd-Tree (ikd-Tree)

In our work, we leverage a global list for accessing mesh vertices by indices. Besides, we employ two data structures (i.e., hash tables and incremental kd-tree (ikd-Tree)) for efficiently managing our four data types. Specifically, we leverage the hash tables for efficient lookup of regions, voxels, and triangle facets, and maintain an ikd-Tree to enable the fast kNN search of mesh vertices.

1) *Hash tables*: To facilitate efficient lookup of the data types (i.e., regions, voxels, and triangle facets), and avoid excessive memory consumption from allocating regular data

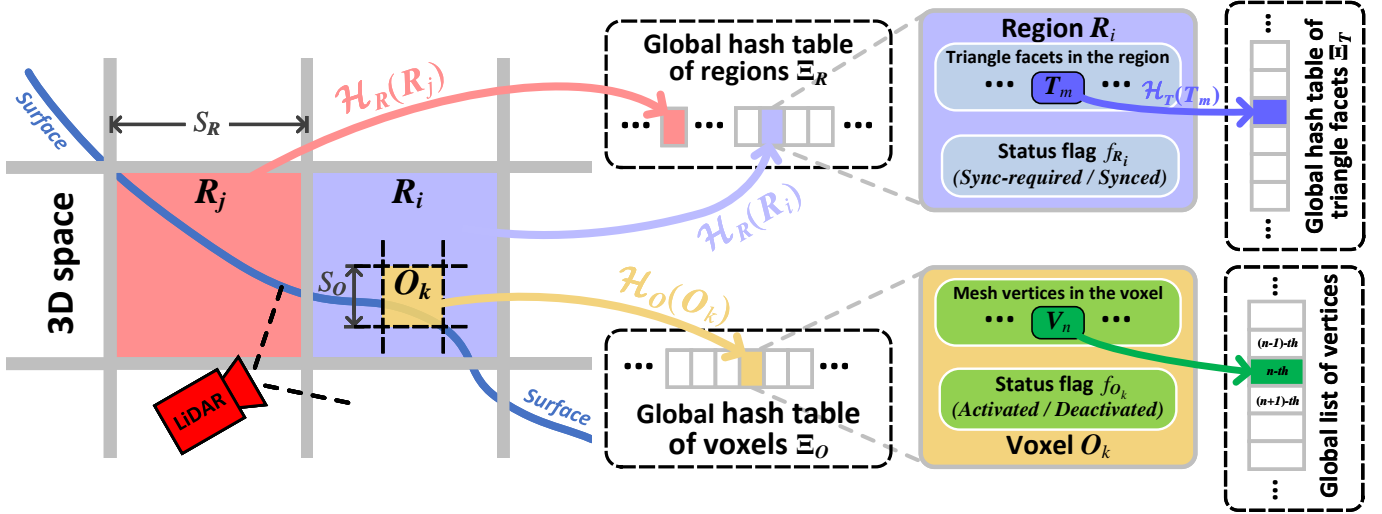


Fig. 3: In ImMesh, we partition the 3D space into two types of volumetric grids: regions and voxels. Triangle facets are stored inside the regions, and mesh vertices are stored inside the voxels. Additionally, we maintain three hash tables to facilitate efficient lookup of these data types.

structures in continuous memory space, we employ a spatial hashing scheme. This scheme allows us to compactly store, access, and update the data structure by mapping them into a hash table using appropriate hash functions, as illustrated in Fig. 3.

Given a 3D vector $\mathbf{p} = [x, y, z]^T \in \mathbb{R}^3$, its corresponding hash key $\mathcal{H}(\mathbf{p})$ is calculated via the 3D hash function $\text{Hash}(x, y, z)$, shown as below:

$$\mathcal{H}(\mathbf{p}) = \text{Hash}(x, y, z) = \text{Int_Hash}(x_i, y_i, z_i) \quad (1)$$

$$= \text{Mod}((x_i \cdot p_1) \oplus (y_i \cdot p_2) \oplus (z_i \cdot p_3), n) \quad (2)$$

$$x_i = \text{Round}(x * 100/S), \quad y_i = \text{Round}(y * 100/S) \quad (3)$$

$$z_i = \text{Round}(z * 100/S)$$

where x_i, y_i, z_i are the corresponding integer-rounded coordinates, S is size of a region (i.e., S_R) or voxel (i.e., S_O), \oplus is the XOR operation, and function $\text{Mod}(a, b)$ is the calculation of integer a modulus another integer b . p_1, p_2, p_3 are three large prime numbers for reducing the collision probability [33, 49], n is the hash table size. In our work, we set the value of p_1, p_2, p_3 and n as 116101, 37199, 93911 and 201326611, respectively.

In our map structure, we maintain three independent hash tables for regions, voxels, and triangle facets, denoted as: Ξ_R , Ξ_O , and Ξ_T , respectively. For a region \mathbf{R}_i , a voxel \mathbf{O}_j , and a triangle facet \mathbf{T}_k , they are mapped to hash tables (i.e., Ξ_R , Ξ_O , and Ξ_T) through the hash keys $\mathcal{H}_R(\mathbf{R}_i)$, $\mathcal{H}_O(\mathbf{O}_j)$, and $\mathcal{H}_T(\mathbf{T}_k)$ are calculated as below:

$$\mathbf{R} \mapsto \Xi_R : \mathcal{H}_R(\mathbf{R}_i) = \mathcal{H}(\mathbf{p}_i), \quad \mathbf{p}_i \in \mathbb{R}^3 \quad (4)$$

$$\mathbf{O} \mapsto \Xi_O : \mathcal{H}_O(\mathbf{O}_j) = \mathcal{H}(\mathbf{p}_j), \quad \mathbf{p}_j \in \mathbb{R}^3 \quad (5)$$

$$\mathbf{T} \mapsto \Xi_T : \mathcal{H}_T(\mathbf{T}_k) = \text{Int_Hash}(\text{Pts_id}(\mathbf{T}_k)) \quad (6)$$

where \mathbf{p}_i (and \mathbf{p}_j) can be any point that located inside region \mathbf{R}_i (and voxel \mathbf{O}_j). The hash function $\mathcal{H}_R(\cdot)$ in (4) and $\mathcal{H}_O(\cdot)$ in (5) are distinguished with different container's size S in (3).

Besides, we use function $\Psi(\cdot)$ to denote the retrieval of \mathbf{R}_i , \mathbf{O}_j , and \mathbf{T}_k from the hash tables, shown as follows:

$$\mathbf{R} \leftarrow \Xi_R : \mathbf{R}_i = \Psi(\Xi_R, \mathcal{H}_R(\mathbf{R}_i)) \quad (7)$$

$$\mathbf{O} \leftarrow \Xi_O : \mathbf{O}_j = \Psi(\Xi_O, \mathcal{H}_O(\mathbf{O}_j)) \quad (8)$$

$$\mathbf{T} \leftarrow \Xi_T : \mathbf{T}_k = \Psi(\Xi_T, \mathcal{H}_T(\mathbf{T}_k)) \quad (9)$$

Notice that the hash table is unstructured, indicating that neighboring regions (or voxels) are not stored spatially but in different parts of the buckets, as illustrated by two neighboring regions \mathbf{R}_i and \mathbf{R}_j in Fig. 3.

Lastly, for resolving the possible hash collision (i.e., two pieces of data in a hash table share the same hash value), we adopt the technique in [33], using the implementation of `unordered_map` container [50] in C++ standard library (std) [51].

2) *Incremental kd-Tree (ikd-Tree)*: We maintain an incremental kd-tree to enable the fast kNN search of mesh vertices. The ikd-Tree is proposed in our previous work [52, 53], which is an efficient dynamic space partition data structure for fast kNN search. Unlike existing static kd-trees (e.g., kd-tree implemented in PCL [54] and FLANN [55]) that require rebuilding the entire tree at each update, ikd-Tree achieves lower computation time by updating the tree with newly coming points in an incremental manner. In ImMesh, we use the ikd-Tree for: 1) ensuring that the distance between any two mesh vertices remains larger than the minimum value ξ , thereby maintaining the triangle mesh at a proper resolution. 2) enabling the vertex dilation operation in our voxel-wise meshing operation to erode the gaps between neighbor voxels.

V. RECEIVER AND LOCALIZATION

The *receiver* module is designed for processing and packaging the input sensor data. As shown in the red box of Fig. 2, our *receiver* module receives the streaming of LiDAR data from live or offline recorded files, processes the data to a unified data format (i.e., customized point cloud data) that make

ImMesh compatible with LiDARs of different manufacturers, scanning mechanisms (i.e., mechanical spinning, solid-state) and point cloud density (e.g., 64-, 32-, 16-lines, etc.). Besides, if the IMU source is available, our *input* module will also package the IMU measurements within a LiDAR frame by referring to the sampling time.

The *localization* module utilizes the input data stream of *receiver* module, reuses the voxels for estimating the sensor poses of 6 DoF by registering the points to planes in voxels in real-time. Our *localization* module is built upon our previous work VoxelMap [18], which represents the environment with the probabilistic planes and estimating pose with an iterated Kalman filter.

A. Voxel map construction

Our *localization* is built by representing the environment with probabilistic planes, which accounts for both LiDAR measurement noises and sensor pose estimation errors, and constructs the voxel-volumetric maps in a coarse-to-fine adaptive resolution manner. Since the main focus of this work is on meshing, we only discuss those processes in *localization* module that are closely related to our *meshing* module. For the detailed modeling and analysis of LiDAR's measurement noise and sensor estimation errors, we recommend our readers to our previous work VoxelMap [18].

For each LiDAR point, we first compensate the in-frame motion distortion with an IMU backward propagation introduced in [52]. Denoting ${}^L\mathbf{p}_i$ the i -th LiDAR point after motion compensation, it is registered to the world frame as ${}^W\mathbf{p}_i$ with the estimated sensor pose $({}^W_L\mathbf{R}, {}^W_L\mathbf{t}) \in SE(3)$:

$${}^W\mathbf{p}_i = {}^W_L\mathbf{R}{}^L\mathbf{p}_i + {}^W_L\mathbf{t} \quad (10)$$

The registered LiDAR point ${}^W\mathbf{p}_i$ is stored inside the voxels. Given all points ${}^W\mathbf{p}_i$ ($i = 1, \dots, N$) inside a voxel \mathbf{O} , the points covariance matrix \mathbf{A} is

$$\bar{\mathbf{p}} = \frac{1}{N} \sum_{i=1}^N {}^W\mathbf{p}_i, \quad \mathbf{A} = \frac{1}{N} \sum_{i=1}^N ({}^W\mathbf{p}_i - \bar{\mathbf{p}}) ({}^W\mathbf{p}_i - \bar{\mathbf{p}})^T \quad (11)$$

where the symmetric matrix \mathbf{A} depicted the distribution of all points. Perform the eigenvalue decomposition of matrix \mathbf{A} :

$$\mathbf{A}\mathbf{U} = \begin{bmatrix} \lambda_1 & & \\ & \lambda_2 & \\ & & \lambda_3 \end{bmatrix} [\mathbf{u}_1 \quad \mathbf{u}_2 \quad \mathbf{u}_3], \quad \lambda_1 \geq \lambda_2 \geq \lambda_3 \quad (12)$$

where $\lambda_1, \lambda_2, \lambda_3$ are the eigenvalues and $\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3$ are the correspondent eigenvectors. In our *meshing* module, we use these calculated eigenvectors of voxel \mathbf{O} for performing the dimension reduction through projection, as we will discuss in Section VI-D.

In our *localization* module, voxel \mathbf{O} might be subdivided into smaller sub-voxels to construct possible planar features at finer resolutions for robust localization in unstructured environments. Then, the sensor pose $({}^W_L\mathbf{R}, {}^W_L\mathbf{t})$ is estimated by minimizing the point-to-plane residual. While this paper primarily focuses on our mesh reconstruction method, we refer readers to our previous work [18] for more details on the implementation of our *localization* module, including the voxel subdivision and state estimation.

B. Point cloud registration

With the estimated sensor pose $({}^W_L\mathbf{R}, {}^W_L\mathbf{t})$, we perform the point cloud registration for transforming each measurement point ${}^L\mathbf{p}_i$ from the LiDAR frame to the global frame (i.e., the first LiDAR frame) with (10). This registered point cloud is then used for: 1) publishing to other applications with our *broadcaster*. 2) updating the voxel map (detailed in [18]). 3) appending to *map structure* that serves as the mesh vertices for shaping the geometry structure of our online reconstructed triangle mesh.

If a new registered point does not lie on an existing voxel \mathbf{O} (or region \mathbf{R}), a new voxel (or region) will be created and added to the hash table $\Xi_{\mathbf{O}}$ (or $\Xi_{\mathbf{R}}$). Subsequently, the newly registered point will be included in the newly constructed voxel.

1) *Append of mesh vertices*: The registered LiDAR points are also used for forming the meshing vertices in *map structure*. To be detailed, we first leverage a voxel-grid filter to downsample the newly registered LiDAR point cloud. Then, to avoid the appearance of tiny triangles in reconstructing the mesh, we leverage the ikd-Tree for keeping the minimum distance ξ between any of two meshing vertices. That is, for each register LiDAR point ${}^W\mathbf{p}_i$ in the global frame, we search for the nearest mesh vertex in *map structure* with ikd-Tree. If the Euclidean distance between this point and the searched vertex is smaller than ξ , we will discard this point. Otherwise, this point will be used for: 1) constructing a new mesh vertex \mathbf{V}_i , where i is the unique index indicating that \mathbf{V}_i is the i -th appended vertex. 2) adding the vertex \mathbf{V}_i to the ikd-Tree. 3) pushing back \mathbf{V}_i to the vertex array of the voxel \mathbf{O}_j that \mathbf{V}_i lies in. After, the status flag $f_{\mathbf{O}_j}$ of \mathbf{O}_j is set as *activated* for notifying the meshing module for performing the voxel-wise meshing operation.

VI. MESHING

In ImMesh, our meshing module takes the registered LiDAR scan for incrementally reconstructing the triangle mesh on the fly. We explicitly reconstruct the triangle mesh by directly utilizing 3D registered LiDAR points as mesh vertices enabled by two facts of LiDAR sensors: 1) The points sampled by LiDAR and registered via the LiDAR odometry and mapping [18] have very high positional accuracy. Hence, they can accurately shape the geometric structure of the mesh. 2) A LiDAR measurement point naturally lies on the surface of the detected object, with two other points in the same plane that can form a triangle facet to represent its underlying surface.

A. Goals and requirements

With the accurate mesh vertices appended from the point cloud registration in Section V-B, the problem of online mesh reconstruction is converted to another goal, which is to seek a proper way of real-time reconstructing the triangle facets with a growing 3D point set. This new problem is barely researched to date. Given a set of growing 3D points, our *meshing* module is designed to incrementally reconstruct the triangle facets considering the following four requirements:

Firstly, precision is our primary consideration. For each reconstructed triangle facet representing the surface of the scene, we require it to lie on an existing plane.

Secondly, the reconstructed mesh should be hole-less. In the dense reconstruction of the surface triangle mesh, the appearance of holes is unacceptable since they lead to the wrong rendering results, where surfaces behind a real object are rendered.

Thirdly, the reconstruction of triangle mesh should avoid constructing sliver triangles. A sliver triangle (i.e., the noodle-like triangle), as defined in the communities of computer graphics [56], is a thin triangle whose area is nearly zero, an undesired property in the field of computer graphics. For example, these noodle-like triangles would cause some errors in the numerical analysis on them [57]. Besides, these unfavorable properties cause troubles in the pipelines of rendering (e.g., rasterization, texturing, and anti-aliasing [6, 7, 58]), which leads to the loss of accuracy in calculating (e.g., depth testing, interpolation, etc.) the pixel values distributed near the sharp angle [7, 59, 60].

Lastly, the complexity of triangle mesh reconstruction should be computationally efficient to meet the requirement of real-time applications. The time consumption of each meshing process should not exceed the sampling duration of two consecutive LiDAR frames.

B. Challenges and approaches

To achieve our goals of dense incremental meshing with the four requirements listed above, our system is proposed based on a deep analysis of the challenges. The challenges and corresponding scientific approaches are briefed below:

The first challenge is that the global map is continuously grown by the newly registered LiDAR points, with each update of a LiDAR scan only affecting parts of the scene. Hence, an incremental mesh reconstruction method should be able to process only those parts of the scene with new points. In our work, we incrementally perform the mesh reconstruction with a mechanism similar to *git* [61]. For each incremental mesh update, we first retrieve the data of the voxels with new mesh vertices appended via the *pull* step (detailed in Section VI-E1). Then, an efficient voxel-wise meshing algorithm is executed to reconstruct the mesh with these data. The incremental modifications of newly reconstructed results w.r.t. pulled results are calculated in our *commit* step (detailed in Section VI-E2). Finally, these incremental modifications are merged to the global map via our *push* step (detailed in Section VI-E3).

Given a set of 3D vertices, the second challenge is how to correctly and efficiently reconstruct the triangle facets representing the surfaces of the scene. Since it is hard to directly reconstruct mesh from these mesh vertices in 3D space, our work performs the meshing operation in 2D. To be detailed, for vertices located in a voxel \mathbf{O} , we first project them into a proper plane (i.e., the estimated plane given by the *localization* module). The mesh of these 2D points is constructed using the 2D meshing algorithms and is recovered back to 3D (detailed in Section VI-D2).

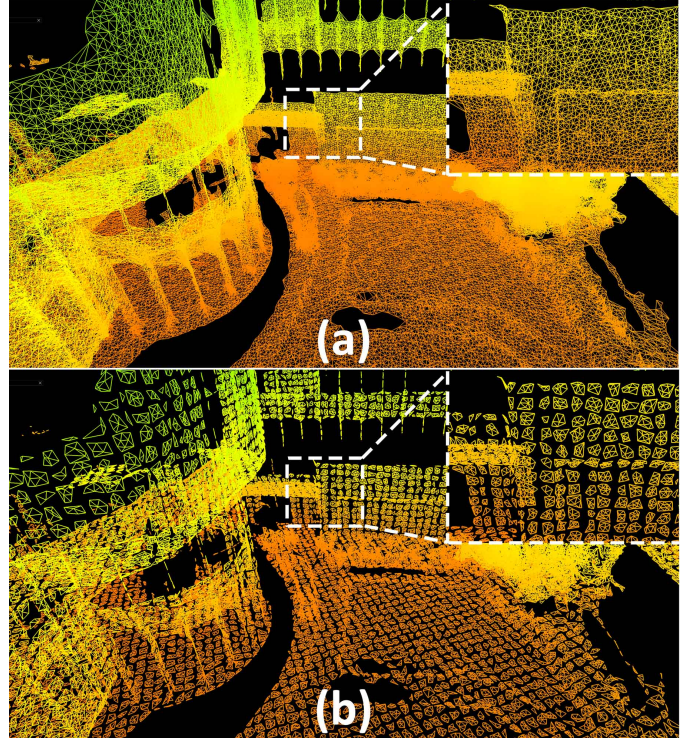


Fig. 4: The comparisons of mesh reconstruction with (a) and without (b) the vertex dilation.

C. Voxel-wise vertex retrieval

1) *Retrieval of in-voxel vertices*: To reconstruct the triangle mesh incrementally, the first step is to retrieve the vertices that need to mesh with the newly added points. ImMesh uses voxels for dividing the 3D space, and uses the flag $f_{\mathbf{O}}$ of each voxel \mathbf{O} for identifying whether \mathbf{O} has newly appended mesh vertices (i.e., *activated* voxel).

Take an *activated* voxel \mathbf{O}_i as an example. We perform a voxel-wise meshing operation to reconstruct the triangle facets with all in-voxel vertices. For all vertices inside the voxel \mathbf{O}_i , we denote them as $\mathcal{V}_i^{\text{In}} = \{\mathbf{V}_{j_1}, \mathbf{V}_{j_2}, \dots, \mathbf{V}_{j_m}\}$.

2) *Vertex dilation*: In practice, if we perform the meshing operation with only the in-voxel mesh vertices, the gaps between neighborhood voxels will appear due to the absence of triangles facets across voxels, as shown in Fig. 4(b). Motivated by morphological operations (e.g., dilation and erosion) in digital image processing [62], we perform the 3D point cloud dilation for adding neighborhood points of $\mathcal{V}_i^{\text{In}}$ to erode the gaps between voxels, as shown in Fig. 4(a).

For vertex $\mathbf{V}_{i_j} \in \mathcal{V}_i^{\text{In}}$, we perform the radius-search operation by leveraging the *ikd-Tree* [53] for searching the nearest vertices of \mathbf{V}_{i_j} with their Euclidean distance smaller than a given value d_r (usually set as 1/4 of the size of a voxel). Using $\tilde{\mathcal{V}}_{i_j}$ to denote the searched neighbor vertices of \mathbf{V}_{i_j} , we have:

$$\forall \mathbf{V} \in \tilde{\mathcal{V}}_{i_j}, \quad \|\text{Pos}(\mathbf{V}) - \text{Pos}(\mathbf{V}_{i_j})\| \leq d_r. \quad (13)$$

We enumerate each $\mathbf{V}_{i_j} \in \mathcal{V}_i^{\text{In}}$ and union the corresponding $\tilde{\mathcal{V}}_{i_j}$ into \mathcal{V}_i (excluding duplicated vertices), which is the set of dilated vertices. The full algorithm of our voxel-wise vertex retrieval is shown in Algorithm 1.

Algorithm 1: Voxel-wise vertex retrieval of \mathbf{O}_i

Input : The *activated* voxel \mathbf{O}_i
Output: The retrieved vertex set \mathcal{V}_i
Start : Copy all in-voxel pointer list to $\mathcal{V}_i^{\text{in}}$.
 $\mathcal{V}_i = \mathcal{V}_i^{\text{in}}$.
1 **foreach** $\mathbf{V}_{i_j} \in \mathcal{V}_i^{\text{in}}$ **do**
2 $\hat{\mathcal{V}}_{i_j} = \text{RadiusSearch}(\mathbf{V}_{i_j}, d_r)$
3 **foreach** $\mathbf{V} \in \hat{\mathcal{V}}_{i_j}$ **do**
4 **if** $\mathbf{V} \notin \mathcal{V}_i$ **then**
5 $\mathcal{V}_i = \mathcal{V}_i \cup \mathbf{V}$

Return: The retrived vertex set \mathcal{V}_i after dilation

D. Dimension reduction through projection

With the mesh vertices \mathcal{V}_i retrieved from Algorithm 1, we introduce the voxel-wise mesh reconstruction.

1) *Projection of 3D vertices on a 2D plane:* Since it is hard to directly mesh in real-time with \mathcal{V}_i , which is distributed in 3D space, we simplify the 3D meshing problem into a 2D one by projecting \mathcal{V}_i on a suitable plane. This dimension reduction by projection is inspired by two key observations: 1) Every LiDAR point can be viewed as lying on a small local surface around it. Hence, for vertices \mathcal{V}_i retrieved from Algorithm 1 that are distributed in a small area (i.e., inside a voxel \mathbf{O}_i), they tend to form a planar-like point cluster. 2) For these planar-like point clusters, we can approximately mesh them in a 2D view on their lying surface. To preserve the 3D space spanned by \mathcal{V}_i to the best extent, the plane (\mathbf{n}, \mathbf{q}) suitable for projection should be formed by the two principal components of \mathcal{V}_i , which is essentially the plane fitted from \mathcal{V}_i and has already been calculated in our *localization* module in Section V-A. The norm \mathbf{n} of the plane is the eigenvector \mathbf{u}_3 that corresponds to the minimum eigenvalue λ_3 in (12), which is the eigendecomposition of point covariance matrix \mathbf{A} in voxel \mathbf{O}_i . \mathbf{q} is the center points inside \mathbf{O}_i .

For each vertex $\mathbf{V}_{i_j} \in \mathcal{V}_i$, we project it to plane (\mathbf{n}, \mathbf{q}). The resultant 2D point \mathbf{p}_{i_j} is calculated as:

$$\mathbf{p}_{i_j} = [\phi, \rho]^T \in \mathbb{R}^2 \quad (14)$$

$$\phi = (\text{Pos}(\mathbf{V}_{i_j}) - \mathbf{q})^T \mathbf{u}_1, \quad \rho = (\text{Pos}(\mathbf{V}_{i_j}) - \mathbf{q})^T \mathbf{u}_2 \quad (15)$$

where $\mathbf{u}_1, \mathbf{u}_2$ are the other two eigenvectors in (12). We use $\mathcal{P}_i = \{\mathbf{p}_{i_1}, \mathbf{p}_{i_2}, \dots, \mathbf{p}_{i_m}\}$ to denote the 2D point set after projected onto the plane.

2) *Two-dimensional Delaunay triangulation:* After the projection, the dimension of 3D meshing problem is reduced to a 2D one, which can be solved by 2D Delaunay triangulation.

As introduced in [63, 64], a Delaunay triangulation $\text{Del}(\mathcal{P})$ for a 2D point set $\mathcal{P} = \{\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_m\}$ is a triangulation such that no point in \mathcal{P} is inside the circumcircle of any triangle. Using $\mathcal{T} = \text{Del}(\mathcal{P})$ to denote the triangle facets after triangulation, \mathcal{T} has the following properties: 1) Any of two facets are either disjoint or share a lower dimensional face (i.e., edge or point). 2) The set of facets in \mathcal{T} is connected with adjacency relation. 3) The domain $\mathbf{P}_{\mathcal{T}}$, which is the union of facets in \mathcal{T} , has no singularity¹. With these three useful

¹The union $\mathbf{U}_{\mathcal{T}}$ of all simplices in \mathcal{T} is called the domain of \mathcal{T} . A point in the domain of \mathcal{T} is said to be singular if its surrounding in $\mathbf{P}_{\mathcal{T}}$ is neither a topological ball nor a topological disc (view https://doc.cgal.org/latest/Triangulation_2/index.html of [63] for detail).

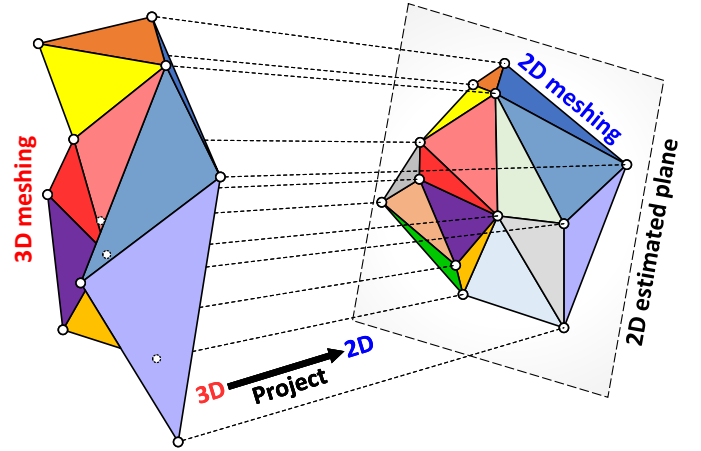


Fig. 5: In ImMesh, we reduce the 3D meshing problem to a 2D one by projecting the 3D vertices onto their principal plane.

properties, the 2D Delaunay triangulation has been widely applied for reconstructing dense facets with a given 2D point set (e.g., [65]).

Considering our requirements in Section VI-A, we chose Delaunay triangulation to reconstruct the mesh for its remarkable properties as follows. Firstly, it is a 2D triangulation providing mesh with no hole left in the convex hull of \mathcal{P} , which satisfies our first requirement. Secondly, it naturally avoids sliver triangles by maximizing the minimum angles of the triangles in triangulation, which meets our second requirement. Finally, it is a fast algorithm suitable for real-time requirements. The algorithm complexity of n points is $\mathcal{O}(n \log(n))$ in 2D (p.s. $\mathcal{O}(n^2)$ in 3D) [66].

Denote the triangle facets after the Delaunay triangulation of \mathcal{P}_i (from Section VI-D) as $\mathcal{T}_i = \text{Del}(\mathcal{P}_i) = \{\mathbf{T}_{i_1}, \mathbf{T}_{i_2}, \dots, \mathbf{T}_{i_n}\}$. For each triangle facets $\mathbf{T}_{i_j} \in \mathcal{T}_i$, we retrieve the indices of its three vertices with: $\{\alpha, \beta, \gamma\} = \text{Pts_id}(\mathbf{T}_{i_j})$, indicating that this triangle is formed with 2D points $\{\mathbf{p}_{i_\alpha}, \mathbf{p}_{i_\beta}, \mathbf{p}_{i_\gamma}\}$. Returning back to 3D space, we constitute a triangle facet \mathbf{T}_{i_j} with vertices $\{\mathbf{V}_{i_\alpha}, \mathbf{V}_{i_\beta}, \mathbf{V}_{i_\gamma}\}$, as shown in Fig. 5. Then, the center $\text{Center}(\mathbf{T}_{i_j})$ and norm $\text{Norm}(\mathbf{T}_{i_j})$ of \mathbf{T}_{i_j} are calculated as below:

$$\text{Center}(\mathbf{T}_{i_j}) = (\text{Pos}(\mathbf{V}_{i_\alpha}) + \text{Pos}(\mathbf{V}_{i_\beta}) + \text{Pos}(\mathbf{V}_{i_\gamma})) / 3 \quad (16)$$

$$\text{Norm}(\mathbf{T}_{i_j}) = \mathbf{n} / (\|\mathbf{n}\|) \quad (17)$$

$$\mathbf{n} = (\text{Pos}(\mathbf{V}_{i_\alpha}) - \text{Pos}(\mathbf{V}_{i_\beta})) \times (\text{Pos}(\mathbf{V}_{i_\gamma}) - \text{Pos}(\mathbf{V}_{i_\beta})) \quad (18)$$

Additionally, to ensure proper face orientation for identifying the front-back face, which is crucial for various computer graphics applications such as front-back face culling, lighting, and shading, we adjust the normal of \mathbf{T}_{i_j} to make it always face towards the current LiDAR position by:

$$\text{If : } ((\mathbf{W}_L \mathbf{t} - \text{Center}(\mathbf{T}_{i_j}))^T \text{Norm}(\mathbf{T}_{i_j}) < 0 \quad (19)$$

$$\text{Then : } \text{Norm}(\mathbf{T}_{i_j}) = -\text{Norm}(\mathbf{T}_{i_j}) \quad (20)$$

where $\mathbf{W}_L \mathbf{t}$ is the LiDAR position of current scan, which is estimated in our *localization* module. Furthermore, if the normal is flipped in (20), we will change the indices of \mathbf{T}_{i_j} from $\{\alpha, \beta, \gamma\}$ to $\{\beta, \alpha, \gamma\}$ when publishing this facet in our *broadcaster*, which is necessary to ensure the correct normal orientation in certain rendering engines (e.g., in [67]).

E. Voxel-wise meshing with pull, commit, and push

With the triangle facets \mathcal{T}_i newly constructed by the voxel-wise meshing operation, we incrementally merge \mathcal{T}_i to the existing triangle facets in the voxel currently saved in *map structure*. This update is designed with a mechanism similar to *git* [61] (a version control software) that includes *pull*, *commit*, and *push* steps.

1) *Pull*: The pull operation aims to retrieve existing triangle facets $\mathcal{T}_i^{\text{Pull}}$ in the i -th $L2$ voxel. Given vertices \mathcal{V}_i in the voxel, which is obtained from Algorithm 1, we retrieve the triangle facets $\mathcal{T}_i^{\text{Pull}}$ from the *map structure* as shown in Algorithm 2.

Algorithm 2: Voxel-wise mesh pull.

Input : The retrieved vertex set \mathcal{V}_i from Algorithm 1

Output: Existing triangles facets in the voxel $\mathcal{T}_i^{\text{Pull}}$

Start : $\mathcal{T}_i^{\text{Pull}} = \{\text{null}\}$

```

1 foreach  $\mathbf{V}_j \in \mathcal{V}_i$  do
2   Get triangles having vertex  $\mathbf{V}_j$ :  $\mathcal{T}_{\mathbf{V}_j} = \text{Tri\_List}(\mathbf{V}_j)$ 
3   foreach  $\mathbf{T}_k \in \mathcal{T}_{\mathbf{V}_j}$  do
4     Get all vertices of  $\mathbf{T}_k$ :  $\{\alpha, \beta, \gamma\} = \text{Pts\_id}(\mathbf{T}_k)$ 
5     if  $(\mathbf{V}_\alpha \in \mathcal{V}_i)$  and  $(\mathbf{V}_\beta \in \mathcal{V}_i)$  and  $(\mathbf{V}_\gamma \in \mathcal{V}_i)$  then
6        $\mathcal{T}_i^{\text{Pull}} = \mathcal{T}_i^{\text{Pull}} \cup \mathbf{T}_k$ 

```

Return: $\mathcal{T}_i^{\text{Pull}}$

2) *Commit*: In this step, we incrementally update the newly reconstructed triangle facets \mathcal{T}_i (in Section VI-D2) to the existing facets $\mathcal{T}_i^{\text{Pull}}$ (from Algorithm 2). These incremental updates are summarized into an array of mesh facets to be added $\mathcal{T}_i^{\text{Add}}$ and an array of mesh facets to be erased $\mathcal{T}_i^{\text{Erase}}$. The detailed processes of this commit step are shown in Algorithm 3.

Algorithm 3: Voxel-wise mesh commit.

Input : The pulled triangle facets $\mathcal{T}_i^{\text{Pull}}$ from Algorithm 2

The reconstructed triangle facets \mathcal{T}_i

Output: The triangle facets to be added $\mathcal{T}_i^{\text{Add}}$.

The triangle facets to be erased $\mathcal{T}_i^{\text{Erase}}$.

Start : $\mathcal{T}_i^{\text{Add}} = \{\text{null}\}$, $\mathcal{T}_i^{\text{Erase}} = \{\text{null}\}$

```

1 foreach  $\mathbf{T}_j \in \mathcal{T}_i$  do
2   if  $\mathbf{T}_j \notin \mathcal{T}_i^{\text{Pull}}$  then
3      $\mathcal{T}_i^{\text{Add}} = \mathcal{T}_i^{\text{Add}} \cup \mathbf{T}_j$ 
4 foreach  $\mathbf{T}_j \in \mathcal{T}_i^{\text{Pull}}$  do
5   if  $\mathbf{T}_j \notin \mathcal{T}_i$  then
6      $\mathcal{T}_i^{\text{Erase}} = \mathcal{T}_i^{\text{Erase}} \cup \mathbf{T}_j$ 

```

Return: The triangle facets to be added $\mathcal{T}_i^{\text{Add}}$ and erased $\mathcal{T}_i^{\text{Erase}}$.

3) *Push*: With the incremental modification $\mathcal{T}_i^{\text{Erase}}$ and $\mathcal{T}_i^{\text{Add}}$ from the previous *commit* step, we perform the addition and erasion operations of triangle facets in *push* step by: 1) constructing (or deleting) the triangle facet structures (as defined in Section IV-A3). 2) adding (or removing) the pointer to these facet structures to other data structures (i.e., mesh

Algorithm 4: Voxel-wise mesh push.

Input : The triangle facets that need to be erased $\mathcal{T}_i^{\text{Erase}}$.

The triangle facets that need to be added $\mathcal{T}_i^{\text{Add}}$.

```

1 Function Add_triangle( $\mathbf{T}_j$ ):
2   Get vertex indices  $\{\alpha, \beta, \gamma\} = \text{Pts\_id}(\mathbf{T}_j)$ 
3   Find the region  $\mathbf{R}_k$  with Center( $\mathbf{T}_j$ ) via (7).
4   Set the status flag  $f_{\mathbf{R}_k}$  of region  $\mathbf{R}_k$  to Sync-required.
5   Add  $\mathbf{T}_j^G$  to region  $\mathbf{R}_k$  and triangles list of vertices  $\mathbf{V}_\alpha, \mathbf{V}_\beta, \mathbf{V}_\gamma$ 
6 Function Erase_triangle( $\mathbf{T}_j$ ):
7   Get vertex indices  $\{\alpha, \beta, \gamma\} = \text{Pts\_id}(\mathbf{T}_j)$ 
8   Remove  $\mathbf{T}_j$  from triangles list of vertices  $\mathbf{V}_\alpha, \mathbf{V}_\beta, \mathbf{V}_\gamma$ .
9   Find the region  $\mathbf{R}_k$  with Center( $\mathbf{T}_j$ ) via (7).
10  Set the status flag  $f_{\mathbf{R}_k}$  of region  $\mathbf{R}_k$  to Sync-required.
11  Remove  $\mathbf{T}_j$  from region  $\mathbf{R}_k$ .
12  Delete triangle  $\mathbf{T}_j$  from memory.
13 foreach  $\mathbf{T}_j \in \mathcal{T}_i^{\text{Add}}$  do
14   Add_triangle( $\mathbf{T}_j$ )
15 foreach  $\mathbf{T}_j \in \mathcal{T}_i^{\text{Erase}}$  do
16   Erase_triangle( $\mathbf{T}_j$ )

```

vertices and regions). The detailed processes of *push* step are shown in Algorithm 4.

F. Parallelism

To further improve the real-time performance, we implement our algorithms with parallelism for better utilization of the computation power of a multi-core CPU. In ImMesh, we have two major parallelisms as follows:

The first parallelism is implemented between the *localization* module and the *meshing* module. Except for the point cloud registration in *localization* module, which needs to operate the mesh vertices as the meshing operation, the remaining processes of *localization* module are parallelized with the *meshing* module. More specifically, once our meshing processes start, the *localization* module is allowed to process the new incoming LiDAR scans for estimation of the pose of LiDAR. However, the subsequent point cloud registration step is only allowed to be executed after the end of the current meshing process.

The second parallelism is implemented among the voxel-wise meshing operation of each *activated* voxel. The voxel-wise meshing operations of different voxels are independent; thus, no conflicted operations exist on the same set of data.

G. The full meshing algorithm

To sum up, our full meshing processes are shown in Algorithm 5.

VII. BROADCASTER

In ImMesh, the *broadcaster* module publishes our state estimation results (i.e., odometry) and mapping results (i.e., newly registered point cloud and triangle mesh) to other applications. Additionally, if a depth image is needed, the *broadcaster* module will rasterize the triangle meshes into a depth image.

Algorithm 5: The full meshing process of each update of LiDAR scan

Input : The set of voxels $\mathcal{O} = \{\mathcal{O}_1, \mathcal{O}_2, \dots, \mathcal{O}_m\}$ that activated in Section V-B

Start : The triangle facets that need to be added $\mathcal{T}^{\text{Add}} = \{\text{null}\}$, and to be erased $\mathcal{T}^{\text{Erase}} = \{\text{null}\}$ in this update .

```

1 foreach  $\mathcal{O}_i \in \mathcal{O}$  do in parallel
2   Retrieve vertices  $\mathcal{V}_i$  with Algorithm 1.
3   Reconstruct the triangle facets  $\mathcal{T}_i$  with  $\mathcal{V}_i$  (Section VI-D2),
4   Performing voxel-wise mesh pull (Algorithm 2) to get  $\mathcal{T}_i^{\text{Pull}}$ . ▷ // Mesh pull
5   Performing voxel-wise mesh commit (Algorithm 3) to get the triangle facets that need to be added  $\mathcal{T}_i^{\text{Add}}$  and erased  $\mathcal{T}_i^{\text{Erase}}$ . ▷ // Mesh commit
6    $\mathcal{T}^{\text{Add}} = \mathcal{T}^{\text{Add}} \cup \mathcal{T}_i^{\text{Add}}$ ,  $\mathcal{T}^{\text{Erase}} = \mathcal{T}^{\text{Erase}} \cup \mathcal{T}_i^{\text{Erase}}$ 
   /* === Mesh push start === */
7 foreach  $\mathcal{T}_j \in \mathcal{T}^{\text{Add}}$  do
8   Add_triangle( $\mathcal{T}_j$ ) ▷ // In Algorithm 4
9 foreach  $\mathcal{T}_j \in \mathcal{T}^{\text{Erase}}$  do
10  Erase_triangle( $\mathcal{T}_j$ ) ▷ // In Algorithm 4
   /* === Mesh push end === */
11 foreach  $\mathcal{O}_i \in \mathcal{O}$  do
12  Reset status flag  $f_{\mathcal{O}_i}$  of  $\mathcal{O}_i$  as deactivated.
```

A. Broadcast of triangle facets

Since the triangle facets are stored in regions in an unstructured way, they can not be directly applied for broadcast. To resolve this problem, our *broadcaster* module maintains a background thread that asynchronously copies the triangle facets from each *sync-required* region (set as *sync-required* after the triangle facets are updated in Algorithm 4) to a structured array for broadcasting. Then, these *sync-required* regions are marked as *synced* after the copying. Finally, The *broadcaster* module publishes the newest triangle facets to other applications.

B. Rasterization of depth image

Some robotic applications, such as autonomous navigation [68] and exploration [69] tasks, require dense accurate depth images for obstacle avoidance. To meet the requirements of these scenarios, the broadcaster module utilizes the triangle facets from Section VII-A to rasterize a depth image at any customized resolution and FoV, based on the fast implementation of *OpenGL* [58].

Besides depth image rasterization, the mesh obtained by our meshing module can reinforce the raw LiDAR point cloud measurements by increasing the resolution and enlarging the FoV. In detail, with the projection matrix and estimated pose used for rasterizing the depth image, the 3D points are obtained (i.e., unproject) from each pixel of the depth image. The unprojected 3D points would have higher resolution and larger FoV than the raw LiDAR measurement scan (see our Application-1 in Section VIII-D).

VIII. EXPERIMENTS AND RESULTS

In this paper, we conduct the experiments by evaluating our meshing ability, especially on the runtime performance and accuracy in reconstructing the triangle mesh.



Fig. 6: (a) shows our handheld device for data collection and online mesh reconstruction. (b) shows a snapshot of our *accompanying video* [70] (starting at 00:09) of Experiment-1, with three time-aligned views of different sources including a screen-recorded view (in red), a camera preview (in yellow), and a third-person view (in blue).

A. Experiment-1: ImMesh for immediate mesh reconstruction

In this experiment, we verify the overall performance of ImMesh toward real-time simultaneous localization and meshing with live video demonstrations. As shown in Fig. 6(b), we record the entire process of our data collection at the campus of the University of Hong Kong (HKU), deploying the ImMesh for simultaneously estimating the sensor pose and reconstructing the triangle mesh on the fly. The *accompanying video* [70] (starting at 00:09) demonstration of this experiment is available on YouTube.

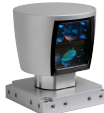



1) *Experiment setup*: Our handheld device for data collection is shown in Fig. 6(a), which includes a mini-computer (equipped with an *Intel i9-10900* CPU and 64 GB RAM), a *Livox avia* 3D LiDAR (FoV: $70.4^\circ \times 77.2^\circ$), and an RGB camera for previewing. In this experiment video, three time-aligned views of different sources are presented, including: 1) a screen-recorded view that shows the estimated pose and online reconstructed triangles mesh of ImMesh. 2) a camera preview that records the video stream of the front-facing camera. 3) a third-person view that records the whole process of this experiment.

2) *Result and analysis*: As presented in the video, benefiting from the accurate uncertainty models of the LiDAR point and plane that account for both LiDAR measurement noise and sensor pose estimation errors in our *localization* module, ImMesh is able to provide the 6 DoF pose estimation of high accuracy in real-time. Without any additional processing (i.e., loop detection), all of these two trials can close the loop itself after traveling 957 m and 391 m, respectively. In addition, with the efficient architecture design and careful engineering implementation on our *meshing* module, the triangle mesh of the surrounding environment is incrementally reconstructed on the fly. With the live preview of real-time meshing, it informs users whether the data collection is sufficient enough for any part of the scene. This important function could lower the revisit chances and facilitate the collection process. Immediately after the data collection, the dense accurate triangle mesh of this scene would be available for analysis. Due to this reason, our system is named as the **Immediately Meshing** (ImMesh).

B. Experiment-2: Extensive evaluation of ImMesh on public datasets with various types of LiDAR in different scenes

With all the modules delicately designed for efficiency, both the *localization* and *meshing* modules easily achieve real-time performances on a standard multi-core CPU. In this

TABLE I: The specifications of LiDARs in four datasets

Dataset	Kitti	NCLT	NTU VIRAL	R ³ LIVE
LiDAR				
	Velodyne HDL-64E	Velodyne HDL-32E	Ouster OS1-16 Gen1	Livox Avia
Scanning mechanism	Mechanical, spinning 64-line	Mechanical, spinning 32-line	Mechanical, spinning 16-line	Solid-state, Risley prism
Field of View (Horizontal ^o × Vertical ^o)	360.0 ^o × 26.8 ^o	360.0 ^o × 41.3 ^o	360.0 ^o × 33.2 ^o	70.4 ^o × 77.2 ^o
Points per second ^[1]	1,333,312	695,000	327,680	240,000
Price (U.S. Dollar)	\$ 75,000	\$ 8,800	\$ 3,500	\$ 1,599

¹ Only show the point rate of single-return mode.

TABLE II: This table shows the detailed information (e.g., length, duration, scenarios) of each testing sequence, the time consumption of ImMesh in processing a LiDAR scan, and the number of vertices and facets of each reconstructed mesh in Experiment-2. Our *accompanying video [70]* (starting at 05:21) that visualizes the online mesh reconstruction process with sequence Kitti_00 is available on YouTube.

Sequece	Traveling length (m)	Durations (s)	LiDAR frames	Meshing mean/Std (ms)	Localization mean/Std (ms)	Number of vertices (m)	Number of facets(m)	Scenarios
Kitti_00	3,724.2	456	4,541	32.1 / 12.0	49.0 / 11.7	3.33	7.70	Urban city
Kitti_01	2,453.2	146	1,101	34.5 / 10.5	51.1 / 18.5	2.03	4.05	High way
Kitti_02	5,058.9	509	4,661	33.5 / 7.0	36.2 / 9.5	4.39	10.03	Residential
Kitti_03	560.9	88	801	28 / 7.1	49.0 / 12.2	0.73	1.55	Countryside; Road
Kitti_04	393.6	27	271	30.1 / 9.4	42.4 / 12.9	0.41	0.85	Urban city; Road
Kitti_05	2,205.6	303	2,761	29.6 / 8.2	38.7 / 11.5	2.17	4.95	Residential
Kitti_06	1,232.9	123	1,101	23.1 / 5.6	56.9 / 9.7	0.89	1.89	Urban city
Kitti_07	2,453.2	114	1,101	20.7 / 7.4	31.3 / 8.6	0.76	1.71	Urban city
Kitti_08	3,222.8	441	4,071	32.4 / 7.8	45.7 / 17.7	3.56	7.94	Urban city
Kitti_09	1,705.1	171	1,591	34.5 / 7.5	43.1 / 19.2	1.83	4.12	Countryside; Road
Kitti_10	919.5	132	1,201	23.4 / 6.9	30.9 / 11.9	0.94	2.10	Residential
NCLT 2012-01-15	7,499.8	6739	66,889	26.3 / 14.1	21.3 / 9.8	9.66	26.61	Campus; Indoor
NCLT 2012-04-29	3,183.1	2598	25,819	25.4 / 13.9	19.1 / 5.4	4.82	13.43	Campus
NCLT 2012-06-15	4,085.9	3310	32,954	24.5 / 14.4	22.3 / 7.7	6.36	17.47	Campus
NCLT 2013-01-10	1,132.3	1024	10,212	20.2 / 12.5	19.3 / 6.5	2.02	5.50	Campus
NCLT 2013-04-05	4,523.6	4167	41,651	20.6 / 13.8	26.8 / 11.7	9.58	23.98	Campus
NTU VIRAL eee_01	265.3	398	3,987	11.2 / 6.7	14.5 / 3.4	0.60	1.38	Aerial; Outdoor
NTU VIRAL nya_01	200.6	396	3,949	9.4 / 5.3	10.2 / 1.7	0.54	1.24	Aerial; Indoor
NTU VIRAL rtp_01	449.6	482	4,615	12.1 / 8.5	10.9 / 2.6	0.72	2.03	Aerial; Outdoor
NTU VIRAL sbs_01	222.1	354	3,542	11.4 / 8.0	17.2 / 3.2	0.47	1.15	Aerial; Outdoor
NTU VIRAL tnp_01	319.4	583	5,795	6.3 / 3.7	8.8 / 1.2	0.16	0.41	Aerial; Indoor
R ³ LIVE hku_campus_00	190.6	202	2,022	12.0 / 7.3	11.5 / 3.2	0.58	1.24	Campus
R ³ LIVE hku_campus_01	374.6	304	3,043	20.4 / 12.6	17.2 / 6.9	1.32	2.86	Campus
R ³ LIVE hku_campus_02	354.3	323	3,236	13.5 / 6.4	11.9 / 2.8	0.87	1.91	Campus
R ³ LIVE hku_campus_03	181.2	173	1,737	12.2 / 5.7	11.3 / 2.9	0.55	1.13	Campus
R ³ LIVE hku_main_building	1,036.9	1170	11,703	16.9 / 14.3	12.5 / 8.0	3.03	6.80	Indoor; Outdoor
R ³ LIVE hku_park_00	247.3	228	2,285	30.1 / 15.9	12.6 / 3.7	0.92	2.38	Cluttered field
R ³ LIVE hku_park_01	401.8	351	3,520	31.5 / 12.2	12.6 / 3.9	1.67	3.96	Cluttered field
R ³ LIVE hkust_campus_00	1,317.2	1073	10,732	26.0 / 12.8	18.0 / 7.6	4.92	11.25	Campus
R ³ LIVE hkust_campus_01	1,524.3	1162	11,629	27.1 / 13.9	16.8 / 6.7	5.35	12.64	Campus
R ³ LIVE hkust_campus_02	2,112.2	1618	4,787	26.7 / 14.5	20.3 / 6.1	1.99	4.65	Campus
R ³ LIVE hkust_campus_03	503.8	478	16,181	33.6 / 13.3	21.0 / 5.3	7.67	18.25	Campus

TABLE III: Two ImMesh configurations for two types of LiDARs (i.e., mechanical and solid-state LiDAR).

	Minimum point distance ξ (m)	Size of region S_R (m)	Size of voxel S_O (m)
Mechanical LiDAR	0.15	15.0	0.60
Solid-state LiDAR	0.10	10.0	0.40

experiment, we evaluate the average time consumption on four public datasets with the computation platform listed in Section VIII-A1.

The four datasets we chose are: Kitti dataset [71], NCLT dataset [72], NTU VIRAL dataset [73] and R³LIVE dataset [74]. They are collected in different scenarios ranging from structured urban buildings to field-cluttered complex environ-

TABLE IV: The average/maximum time of *meshing* and *localization* module for processing each LiDAR scan in four datasets.

	Kitti mean/max	NCLT mean/max	NTU VIRAL mean/max	R ³ LIVE mean/max
Meshing (ms)	31.3 / 34.5	24.2 / 25.4	9.8 / 17.2	25.3 / 33.6
Localization (ms)	42.2 / 56.9	22.3 / 26.8	11.9 / 17.2	16.6 / 21.0

ments (see TABLE V), using various types of LiDARs that include mechanical spinning LiDAR of different channels and solid-state LiDAR of small FoV (see TABLE I).

1) *Experiment setup*: ImMesh is robust to its parameter values, which requires minimal user-adjustable parameters to achieve good results without extensive parameter tuning. We

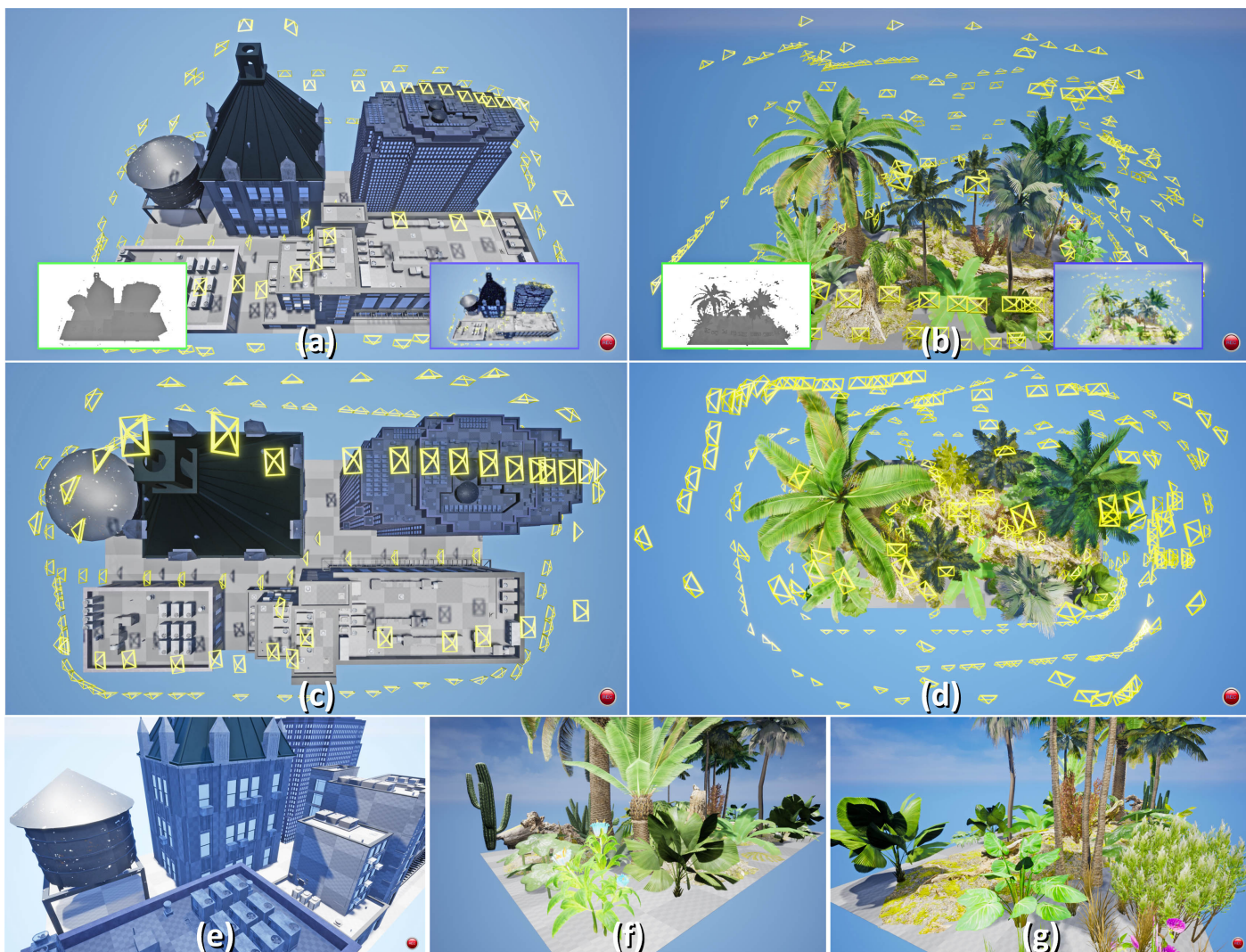


Fig. 7: The screenshots of our *Microsoft AirSim* simulator used for generating synthetic data. (a, c, and e) show the “Urban city” environments, while (b, d, f, and g) depict the “Cluttered field” environments. The yellow frustums in (a~d) represent the poses of LiDAR sensor used to capture synthetic data. These frustums are set as invisible during data generation, as shown in (e-g). The images within the green and blue boxes in (a and d) respectively show the produced depth and RGB images.

benchmark ImMesh in four datasets with only two sets of configurations. The two configurations are reasonably required for adapting two classes of LiDARs (i.e., mechanical and solid-state LiDAR), as shown in TABLE III. Since the 3D points sampled by a solid-state LiDAR are distributed in a small sensor FoV, the accumulated point cloud of solid-state LiDAR usually has a higher density. Therefore, we set the minimum point distance and voxel size for solid-state LiDAR 1.5 times smaller than those for mechanical LiDAR, as shown in TABLE III. We maintained the same configuration for the other setups except for some necessary adjustments to match the hardware setup.

2) *Result and analysis*: TABLE V shows the detailed information (e.g., length, duration, scene) of each sequence, the average time consumption of our *localization* and *meshing* module in processing a LiDAR scan, and the number of vertices and facets of each reconstructed mesh. From Table V, it is seen that the average cost-time of both *localization* and *meshing* modules are closely related to the density of the input LiDAR scan. To be detailed, the LiDAR of a higher

channel has a much higher point sampling rate (see Table I) which causes more data to be processed in each update of a LiDAR frame (e.g., more points in a voxel and more voxels activated in each frame). Besides, the processing time varies among different scenarios for the same set of datasets. The sequences sampled in a high-way or field environment (e.g., Kitti_01, Kitti_09) usually have a longer LiDAR sampling range, leading to more points per frame to be processed. Thanks to the efficient data structures (e.g., ikd-Tree, hash tables) and parallelism strategy, which allows us to perform the state estimation and incremental mesh reconstruction simultaneously, the time consumption of large-scale datasets is bounded in an acceptable value (≤ 35 ms for meshing, ≤ 49 ms for localization).

The average and maximum time consumption of ImMesh in the four datasets are shown in TABLE IV, reflecting that our system satisfies the real-time requirement even with different types of LiDARs and scenarios. Notice that the LiDAR frame rate are 10 Hz for all datasets, and our *meshing*

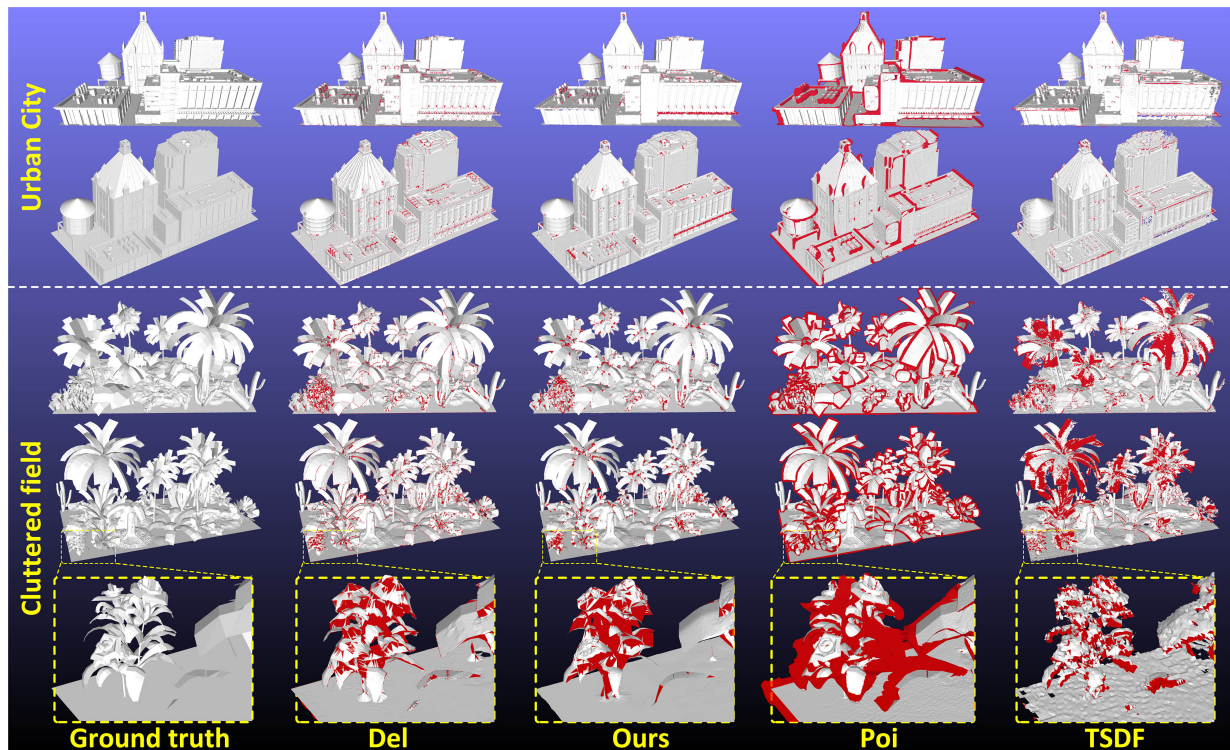


Fig. 8: The qualitative comparison of ground truth and four evaluated methods, which are tested with the depth images resolution of 640×480 . The facets colored in red represent surfaces that have been incorrectly reconstructed, with 80% of their sampling points not lying on the ground truth surface (i.e., the distances between these points and the nearest ground truth surface are larger than 5 cm).

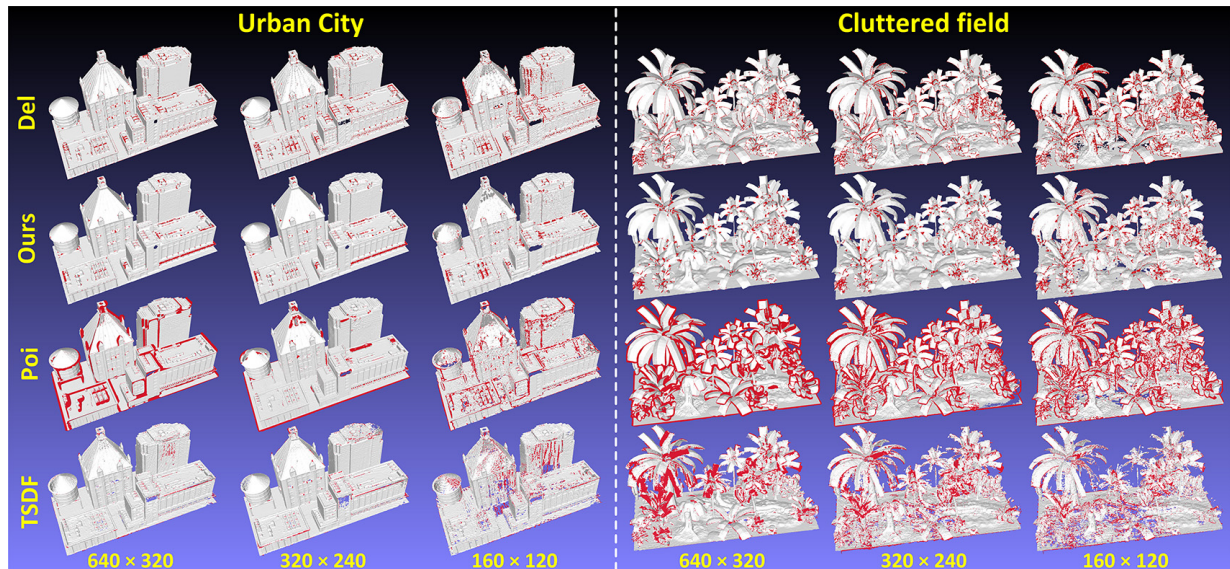


Fig. 9: The qualitative comparisons of four methods that evaluated with depth images of different resolutions. The facets colored in red represent surfaces that have been incorrectly reconstructed, with 80% of their sampling points not lying on the ground truth surface.

and *localization* modules run in parallel (see Section VI-F).

C. Experiment-3: Quantitative evaluation of ImMesh

In this experiment, we use both real-world and synthetic data to conduct the quantitative evaluations of ImMesh, by comparing it against existing reconstruction methods.

1) *Preparation of large-scale, real-world data*: We conducted a quantitative evaluation using large-scale real-world

LiDAR data collected from the *Complex Urban Dataset* [75]. This dataset provides a high-quality set of ground truth LiDAR poses and ground truth point clouds, which enables a comprehensive assessment of our proposed method and existing baselines. The detailed traveling length and the number of LiDAR frames of tested sequences are shown in TABLE V.

2) *Preparation of synthetic data*: To further evaluate the performance of all the methods under diverse scenarios, with varying levels of clutteredness, we generated synthetic data using the *Microsoft AirSim* simulator [4]. The screenshots of

TABLE V: The quantitative evaluation result with real-world data from *Complex Urban Dataset*. The \uparrow denotes larger is better while \downarrow indicates lower is better.

Sequence	Traveling length (Km)	Number of LiDAR frames	Method	Cost time \downarrow (hour:min:sec)	Fairness		Correctness				
					Max-Min angle ($^\circ$) \downarrow	C2SE \downarrow	Completeness (m) \downarrow	Accuracy (m) \downarrow	Recall (%) \uparrow	Precision (%) \uparrow	F-score \uparrow
Urban01	11.72	13846	Poi	09:58:30	60.1014	0.9760	0.0632	0.0724	0.8554	0.7563	0.8028
			ImMesh (ours)	00:05:39	56.2941	0.8630	0.0404	0.0568	0.9477	0.8260	0.8827
Urban02	4.20	8961	Poi	05:49:36	59.9695	0.9739	0.0792	0.0822	0.8818	0.7261	0.7964
			ImMesh (ours)	00:03:01	57.3564	0.8605	0.0392	0.0556	0.9623	0.8398	0.8968
Urban03	3.06	9091	Poi	04:55:04	60.1614	0.9770	0.0070	0.0059	0.8871	0.7754	0.8275
			ImMesh (ours)	00:03:07	57.4131	0.8628	0.0398	0.0564	0.9597	0.8359	0.8935

TABLE VI: The quantitative evaluation result with synthetic data generated with *Microsoft AirSim* simulator.

Method	Scenario	Resolution	Cost time (min:sec) \downarrow	Fairness		Correctness					
				Max-Min angle ($^\circ$) \downarrow	C2SE \downarrow	Completeness (m) \downarrow	Accuracy (m) \downarrow	Recall (%) \uparrow	Precision (%) \uparrow	F-score \uparrow	
ImMesh (ours)	Del	Urban city	640 \times 480	17:51	48.5148	0.7825	0.0883	0.0341	0.7976	0.7976	0.7976
	ImMesh (ours)	Urban city	640 \times 480	00:31	48.0909	0.7843	0.1002	0.0265	0.7290	0.8525	0.7859
	Poi	Urban city	640 \times 480	15:42	53.4110	0.8664	0.1094	0.2244	0.7367	0.7798	0.7576
	TSDF	Urban city	640 \times 480	00:25	64.7606	1.0530	0.1506	0.0361	0.5859	0.8665	0.6991
ImMesh (ours)	Del	Urban city	320 \times 240	07:22	49.1746	0.7960	0.0928	0.0515	0.7685	0.7470	0.7576
	ImMesh (ours)	Urban city	320 \times 240	00:23	52.9000	0.8235	0.1002	0.0265	0.7290	0.7821	0.7546
	Poi	Urban city	320 \times 240	04:42	52.7233	0.8566	0.1216	0.0788	0.6845	0.6904	0.6875
	TSDF	Urban city	320 \times 240	00:25	64.4962	1.0474	0.1544	0.0655	0.4994	0.7397	0.5962
ImMesh (ours)	Del	Urban city	160 \times 120	02:02	49.8635	0.8098	0.1186	0.0914	0.6822	0.5574	0.6135
	ImMesh (ours)	Urban city	160 \times 120	00:19	54.4587	0.8466	0.1341	0.0834	0.5493	0.5914	0.5696
	Poi	Urban city	160 \times 120	01:03	54.6500	0.9052	0.1849	0.1453	0.4777	0.6159	0.5381
	TSDF	Urban city	160 \times 120	00:24	65.1098	1.0564	0.2802	0.2508	0.3352	0.4799	0.3947
ImMesh (ours)	Del	Cluttered field	640 \times 480	21:14	56.6578	0.8304	0.2767	0.0496	0.7489	0.7036	0.7255
	ImMesh (ours)	Cluttered field	640 \times 480	00:33	57.1687	0.8558	0.2953	0.0519	0.7027	0.7404	0.7211
	Poi	Cluttered field	640 \times 480	24:31	59.5750	0.9649	0.3052	0.3960	0.6981	0.7009	0.6995
	TSDF	Cluttered field	640 \times 480	00:24	65.4224	1.0882	0.4130	0.4270	0.4837	0.4936	0.4886
ImMesh (ours)	Del	Cluttered field	320 \times 240	07:38	57.9700	0.8526	0.2919	0.0882	0.5416	0.7198	0.6181
	ImMesh (ours)	Cluttered field	320 \times 240	00:25	57.6159	0.8603	0.3105	0.0784	0.6146	0.6404	0.6272
	Poi	Cluttered field	320 \times 240	10:28	59.4470	0.9722	0.3620	0.3395	0.5630	0.5506	0.5567
	TSDF	Cluttered field	320 \times 240	00:23	65.7206	1.0892	0.5268	0.4784	0.2114	0.2567	0.2319
ImMesh (ours)	Del	Cluttered field	160 \times 120	01:56	59.4879	0.8785	0.3438	0.1781	0.3947	0.5696	0.4663
	ImMesh (ours)	Cluttered field	160 \times 120	00:21	60.1208	0.8970	0.3512	0.1694	0.4200	0.4863	0.4507
	Poi	Cluttered field	160 \times 120	01:07	59.1544	0.9744	0.3541	0.4164	0.3775	0.3820	0.3797
	TSDF	Cluttered field	160 \times 120	00:23	65.1832	1.0815	0.5561	0.3681	0.2099	0.2944	0.2451

our simulating scenarios are presented in Fig. 7, where we prepared two typical environments: “Urban city” (Fig. 7(a, c, and e)) and “Cluttered field” (Fig. 7(b, d, f, and g)), both of which have dimensions of 20 m \times 10 m \times 8 m. The “Urban city” environment consists of structured objects, such as buildings, towers, and water tanks, providing a realistic representation of an urban setting. On the other hand, the “Cluttered field” environment incorporates a diverse range of plants, including trees, flowers, grasses, and other vegetation, creating a more complex and cluttered scenario.

To simulate point clouds collected by a real LiDAR, we unproject the 3D points from the depth image. The depth images are obtained by querying the *AirSim*’s API, specifically the images shown within the green box in Fig. 7(a and b). The depth image has a field of view (FoV) of 120 $^\circ$ \times 80 $^\circ$. We manually positioned the poses, represented by the yellow frustums in Fig. 7(a ~ d), to ensure that the generated point cloud covers most of the surfaces in the scene. Additionally, we simulate LiDAR data with different point cloud densities by generating data using three different sets of depth image resolutions: 640 \times 480, 320 \times 240, and 160 \times 120, as shown in TABLE VI.

3) *Experiment setup*: In this experiment, we performed a comprehensive evaluation of meshing ability among our work

and existing mesh reconstruction baselines, which includes a TSDF-based method implemented by *Point cloud library (PCL)* [54] with GPU acceleration, Delaunay triangulation and graph cut based method implemented by *OpenMVS* [76], and the official implementation of Poisson surface reconstruction [19, 20].

We conducted the evaluation of these methods on a desktop PC equipped with an *Intel i7-9700K* CPU, 64Gb RAM, and an *Nvidia 2080 Ti* GPU with 12Gb of graphics memory. We fed online reconstruction method *ImMesh* and TSDF-based (*TSDF*) methods with LiDAR points frame by frame. To mitigate the impact of pose estimation errors on meshing results, we disabled the pose estimation module and provided the ground truth poses to the online mesh reconstruction methods *ImMesh* and *TSDF*. For the offline mesh reconstruction methods, namely Delaunay triangulation (*Del*) and Poisson surface reconstruction (*Poi*), we fed them with the accumulated point cloud from all frames. Additionally, to address the issue of uneven point cloud density, which can result in errors when calculating normals for *Poi*, and to prevent *Del* from reconstructing small facets that could bias accuracy calculations. We leverage a voxel grid filter with a leaf size of 1.0 cm \times 1.0 cm \times 1.0 cm to downsample the accumulated point cloud before providing it as input to both

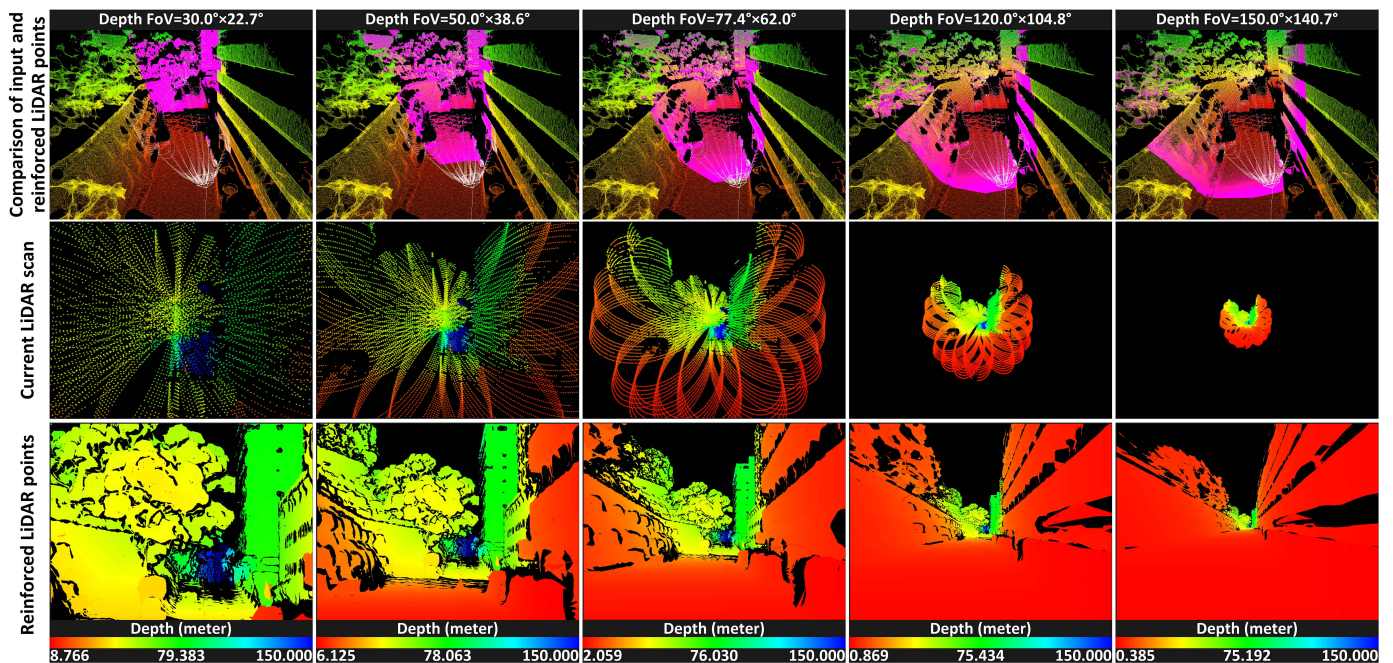


Fig. 10: The first row of images shows the comparisons between a raw LiDAR frame (colored in white) and our reinforced points (colored in magenta) under different sets of rasterizing FoV. The second and third rows of images show the comparisons of raw and reinforced points after projection on the current sensor frame. For more detailed visualizations of this process, please refer to our *accompanying video* [70] (starting at 08:19) on YouTube.

Poi and *Del*.

Due to the limited graphics memory (12Gb for *Nvidia 2080 Ti*), we set the *TSDF* cell size as 0.2m such that *TSDF* can utilize the GPU acceleration while preserving satisfying precision in the mesh reconstruction. For our *ImMesh*, the parameter configuration for solid-state LiDAR is used, as shown in TABLE III. For *Poi*, we set the octree level as 12 and removed large hulls by deleting facets with one of their edges longer than 15.0 cm. For other configurations of all methods, we set them as their default configuration. It is noted that other than *TSDF* using GPUs for acceleration, the rest methods, *Del*, *Poi*, and ours, use the CPU only. We compare the efficiency of four methods by evaluating their time consumption in reconstructing the mesh. For online methods (i.e., *TSDF* and ours), we accumulate the processing time of all frames, while for offline methods (i.e., *Poi* and *Del*), we count the total time in processing the offline data. The results of their time consumption are listed in TABLE V and TABLE VI.

4) *Evaluation of fairness*: In this experiment, we employ the triangle fairness criteria to evaluate the quality of reconstructed triangle facets. This evaluation involves analyzing the average error of the maximum and minimum interior angles of the triangles (as utilized in work [77]), which we refer to as the *Max-Min angle* in TABLE V and TABLE VI. Additionally, we consider the average ratio of the circumradius to the shortest edge length (referred to as *C2SE* in Tables) as used in works [78, 79]. A lower value for both the *Max-Min angle* and *C2SE* indicates higher mesh quality, as it signifies that the triangle facets are closer to being equilateral.

In the evaluation with large-scale, real-world data, the results for *Del* and *TSDF* methods were not available due to specific limitations: 1) For *Del*, we encountered difficulties when running it with the *Complex Urban Dataset*. Despite

multiple attempts, the *Del* method either crashed midway or failed to produce any result after running for over three days. 2) As for *TSDF*, allocating the voxels requires a massive amount of graphics memory. This exceeds the capabilities of our hardware platforms, particularly for sequences in Table V with a traveling length of over 3 kilometers.

As indicated by the fairness metrics listed in TABLE V and VI, we can conclude that leveraging Delaunay triangulation eliminates the formation of sliver triangles. The *Del* method demonstrates the best results in this regard. Following that is *ImMesh*, which utilizes Delaunay triangulation for meshing the point set after dimension reduction through projection. On the other hand, the meshes reconstructed by the *Poi* and *TSDF* methods, which employ the marching cubes algorithm, exhibit inferior results. This is due to the inherent limitation of the marching cubes algorithm [23], which generates sliver triangles when a facet is positioned closely and nearly parallel to the edges of the cube.

5) *Evaluation of correctness*: For the quantitative evaluation of the methods' correctness in reconstructing the mesh, we utilized 3D geometry metrics as employed in works *NeuralRecon* [80] and *Atlas* [81]. These metrics encompass the following measurements: *accuracy*, *completeness*, *precision*, *recall*, and *F-score*. The calculations for these metrics are as follows:

$$\text{Accuracy: } \text{mean}_{\mathbf{p} \in \mathcal{P}} (\min_{\mathbf{p}^* \in \mathcal{P}^*} \|\mathbf{p} - \mathbf{p}^*\|)$$

$$\text{Completeness: } \text{mean}_{\mathbf{p}^* \in \mathcal{P}^*} (\min_{\mathbf{p} \in \mathcal{P}} \|\mathbf{p} - \mathbf{p}^*\|)$$

$$\text{Precision: } \text{mean}_{\mathbf{p} \in \mathcal{P}} (\min_{\mathbf{p}^* \in \mathcal{P}^*} \|\mathbf{p} - \mathbf{p}^*\| < 0.05)$$

$$\text{Recall: } \text{mean}_{\mathbf{p}^* \in \mathcal{P}^*} (\min_{\mathbf{p} \in \mathcal{P}} \|\mathbf{p} - \mathbf{p}^*\| < 0.05)$$

$$\text{F-score: } \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

where \mathcal{P} refers to the point cloud obtained by uniformly sampling the reconstructed mesh generated by the method under evaluation. This point cloud is sampled at a spatial resolution of 0.01 m. On the other hand, \mathcal{P}^* represents the downsampled ground truth point cloud. It is also downsampled at a spatial resolution of 0.01 m.

The quantitative evaluation results for metrics such as *accuracy*, *completeness*, *precision*, *recall*, and *F-score* are provided in TABLE VI. We can observe that *Del* achieves the highest overall correctness in constructing the mesh of the scene. Following that, *ImMesh* demonstrates slightly lower precision. Then, *Poi* exhibits even lower mesh correctness, and *TSDF* shows the lowest correctness among all methods.

The qualitative comparison results of the four benchmarked methods evaluated with synthetic data are presented in Fig. 8 and Fig. 9. In these figures, the red facets represent incorrectly reconstructed surfaces with 80% of their sampling points not lying on a ground truth surface (i.e., the distances between these points and the nearest ground truth surface are larger than 5 cm). Among the evaluated methods, *Del* and *ImMesh* exhibit comparable results in reconstructing the mesh of scenes well. In contrast, *Poi* exhibits lower mesh correctness due to the presence of unwanted facets at the sharp edges of the models, as indicated in the fourth column of Fig. 8. The *TSDF* method shows the lowest results with the appearance of holes on the reconstructed surface, as observed in the roofs of buildings and the leaves of trees shown in the fifth column of Fig. 8.

When reconstructing complex and small objects in the scene, such as the flower in the “Cluttered field” environment, as depicted in the RGB image shown in the bottom-left corner of Fig. 7(f) and the corresponding mesh models displayed in the fifth row of Fig. 8. *Del*, *TSDF* and *ImMesh* fail to recover the details of surface well. This limitation arises from different factors for each method: *Del* requires a large number of camera-to-point correspondences to extract intricate surface details, which may pose challenges when dealing with complex and tiny objects. *TSDF* and *ImMesh* are constrained by the fixed voxel size, which can not reconstruct the details of surfaces whose size is smaller than voxel. What is worth mentioning is that we found *Poi* can recover the details of the flower’s petals well. This is achieved through the use of a scalable resolution based on an octree structure, which allows *Poi* to adapt its resolution for reconstructing small and intricate surfaces.

In addition, as observed in Fig. 9 and with the metrics listed in Table VI, we can see that as the point cloud becomes sparser (due to lower resolution depth images), the correctness of the reconstruction methods decreases accordingly. However, both *Del* and *ImMesh* demonstrate stronger robustness in resiliently handling the drop in point cloud density. On the other hand, the meshes reconstructed by *Poi* and *TSDF* exhibit discontinuities and contain more holes and gaps when compared to the results of *Del* and *ImMesh*.

Lastly, in the evaluation with real-world data from *Complex Urban Dataset* [75], we discovered that the mesh reconstructed by *Poi* also exhibits unwanted facets appearing at the edges of objects such as buildings and trees. These undesirable facets, as indicated by the red facets in Fig. 8 and Fig. 9

for *Poi*, have a negative impact on the overall correctness of the reconstruction. As a result, *Poi* performs inferiorly across all evaluated correctness metrics when compared to *ImMesh*, as shown in TABLE V.

6) *Evaluation of runtime performance*: According to the *cost time* listed in TABLE V, it is clear that *ImMesh* demonstrates a significant advantage in terms of runtime performance when evaluated with large-scale sequences. The execution time of *ImMesh* is only 0.93% ~ 1.06% of that of *Poi*.

TABLE VI displays the average time consumption of the four benchmarked methods when evaluated with synthetic data. The online methods, *ImMesh* and *TSDF*, exhibit similar runtime performance. In contrast, the offline methods (*Del* and *Poi*) consume significantly more time, ranging from 5 to 40 times longer than the online methods (*TSDF* and *ImMesh*). Notably, *TSDF* achieves comparable runtime performance to our method with the assistance of an *Nvidia 2080 Ti* GPU, highlighting the high computational efficiency of our *ImMesh* framework compared to the other three methods.

7) *Summary*: Based on the results and analysis regarding runtime performance, fairness, and correctness, we have reached the following conclusions for Experiment-3: 1) For offline applications, which only care about quality and neglect time consumption, *Del* is the best choice, and our *ImMesh* is the second best one. 2) For real-time applications, our work *ImMesh* is the best choice. Even though *TSDF* with GPU acceleration can run in real-time, its meshing correctness is much lower than *ImMesh*.

D. Application-1: LiDAR point cloud reinforcement

Benefiting from *ImMesh*’s real-time ability to reconstruct the triangle mesh on the fly, depth images can be rasterized from the reconstructed facets online in the current sensor frame. By unprojecting the 3D points from the depth image, point clouds of a regular pattern can be retrieved with wider FoV and denser distribution than the original input LiDAR scan. We termed this process as LiDAR point reinforcement.

In this experiment, we demonstrate the LiDAR point cloud reinforcement with a solid-state LiDAR *Livox Avia* with FoV of $70.4^\circ \times 77.2^\circ$. The comparisons between the original points of a LiDAR frame (colored in white) and after our reinforcement (colored in magenta) with different sets of rasterization FoV are shown in Fig. 10. As the white points shown in the first row of Fig. 10, the input LiDAR scan is sparse with an irregular scanning pattern. After the reinforcement, the resultant 3D points colored in magenta are distributed in a regular pattern, with a higher density and wider FoV (as the rasterization FoV is bigger than LiDAR’s). To better understand their differences, we present the comparisons of depth images after projection, as shown in the second and third rows of Fig. 10.

E. Application-2: Rapid, lossless texture reconstruction

In this application, we show how *ImMesh* can be applied in applications of lossless texture reconstruction for rapid field surveying. As shown in Fig. 11(b1~b3), we mounted a *Livox*

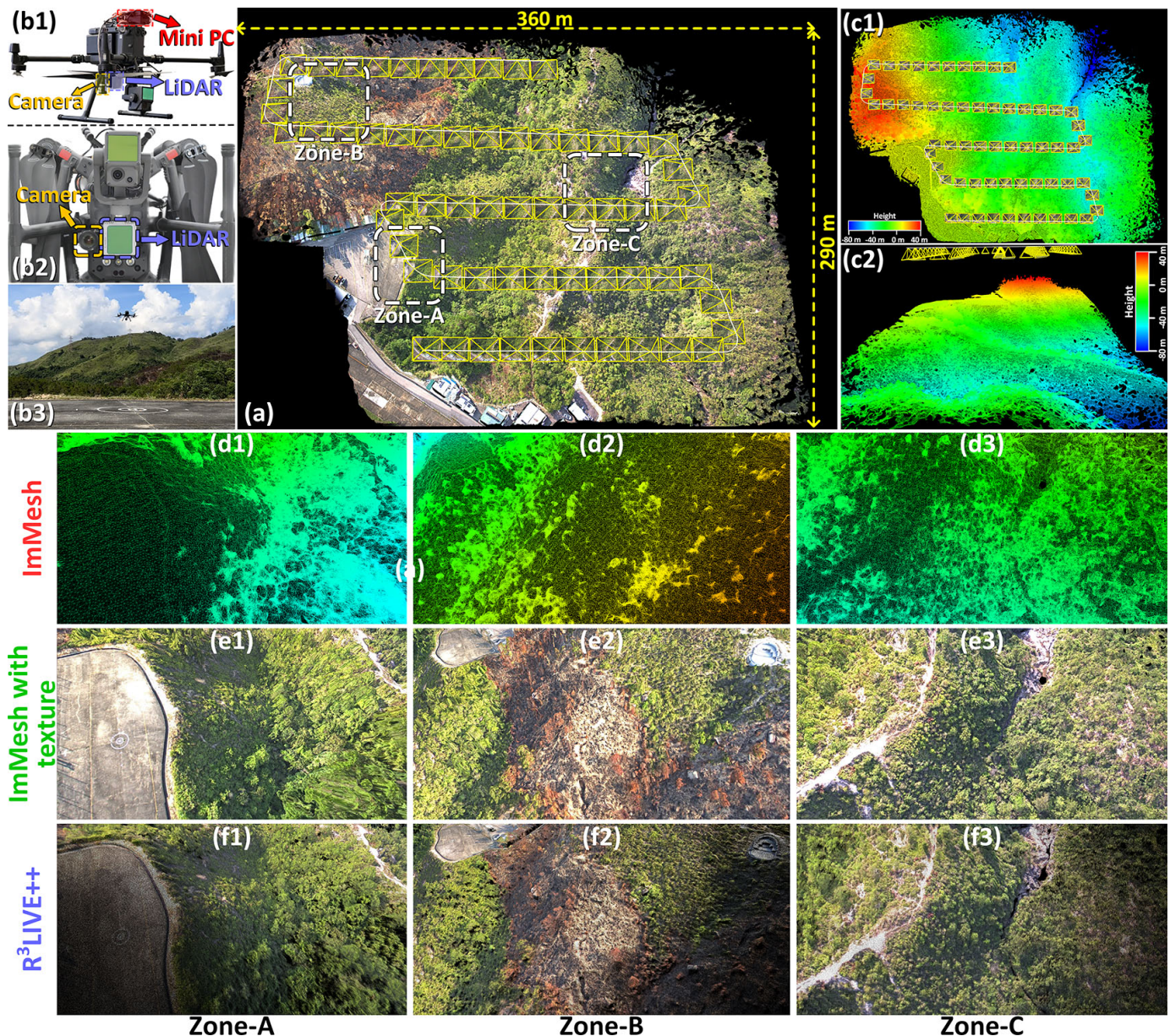


Fig. 11: (b1~b3) show our UAV platform for data collection. (a) show the bird view of our lossless texture reconstruction result. (c1 and c2) show the altitude of this map by coloring the facets in their height w.r.t. the take-off point (i.e., the ground plane in Zone-A). The qualitative comparison of mapping results in Zone-A, B, and C of ImMesh, ImMesh after texturing, and R³LIVE++ are shown in (d~f). To see the detailed reconstruction process of the scene, please refer to our *accompanying video* [70] (starting at 10:22) on YouTube.

avia LiDAR and a Hikvision CA-050-11UC global shutter RGB camera on a DJI M300 drone platform.

We collected the data in a mountain field by taking off from Zone-A (see Fig. 11(a)) and flying in a “s”-like pattern trajectory with a traveling distance of 975 m. We leveraged ImMesh for reconstructing the mesh from the collected LiDAR data and used R³LIVE++ [82] for estimating the camera’s poses (as the yellow frustum shown in Fig. 11(a, c1 and c2)). We textured each facet of the reconstructed mesh by the RGB image captured by the nearest camera frame with the estimated camera pose from R³LIVE++. Benefiting from the high efficiency of ImMesh and R³LIVE++, the total time of reconstructing the RGB textured mesh from this sequence of duration 325 s cost only 686 s, with 328 s for ImMesh, 330 s for R³LIVE++, and 28 s for texturing. Fig. 11(a) shows a bird

view of our mesh after texturing, with the close-up views of textured mesh in Zone-A, B, and C shown in Fig. 11(e1, e2, and e3), respectively. In Fig. 11(c1 and c2), we show the altitude of this map by coloring the facets in their height w.r.t. the take-off point (i.e., the ground plane in Zone-A).

As shown by the close-up views in the bottom three rows of Fig. 11, the reconstructed mesh (d1~d3) from our ImMesh after texturing (e1~e3) successfully preserves the map textures when comparing with the RGB-colored point cloud reconstructed by R³LIVE++ (f1~f3). Due to the limited point cloud density, the RGB-colored point cloud by R³LIVE++ is unable to reconstruct the scene losslessly. Compared to existing counterparts (e.g., 3D reconstruction from photogrammetry [13, 26]) that reconstructs a scene from captured images (and RTK measurements), our system shows significant advantages:

1) It is a reliable solution that does not require GPS measurement. 2) It is a rapid reconstruction method that costs only 2~3 times the data sampling time for reconstructing a scene. 3) It preserves a geometry structure of high accuracy that is reconstructed from LiDAR's measurements. The *accompanying video* [70] (*starting at 10:22*) that records the full process of this lossless texture reconstruction is available on our YouTube, and an additional trial is shown in our Supplementary Material [83].

Notice that in Fig. 11, the presence of isolated mesh facets is a result of missing scanning data, while the blurry texture artifacts are caused by the large viewing angle of the facets and textured images, both can be addressed through proper data collection processes.

IX. CONCLUSIONS AND FUTURE WORK

A. Conclusions

In this work, we proposed a novel meshing framework termed ImMesh for achieving the goal of simultaneous localization and meshing in real-time. The real-time incremental meshing nature of our system, even in large-scale scenes, makes it one of a kind. The *localization* module in ImMesh represents the surrounding environment in a probabilistic representation, estimating the sensor pose in real-time by leveraging an iterated Kalman filter to maximize the posterior probability. The *meshing* module directly utilizes the spatially-downsampled registered LiDAR points as mesh vertices and reconstructs the triangle facets in a novel incremental manner in real-time. To be detailed, our *meshing* module first retrieves all voxels that contain newly appended vertices. Then, the voxel-wise 3D meshing problem is converted into a 2D one by performing dimension reduction for efficient meshing. Finally, the triangle facets are incrementally reconstructed with *pull*, *commit*, and *push* steps.

Our system is evaluated by real experiments. First, we verified the overall performance by presenting live video demonstrations of how the mesh is immediately reconstructed in the process of data collection. Then we extensively tested ImMesh with four public datasets collected by four different LiDAR sensors in various scenes, which confirmed the real-time ability of our system. Lastly, we benchmarked the meshing performance of ImMesh in Experiment-3 by comparing it against existing meshing baselines. The results show that ImMesh achieves high meshing accuracy while keeping the best runtime performance among all methods.

Applications of our system were demonstrated. We first show how ImMesh can be applied for LiDAR point cloud reinforcement, which generates reinforced points in a regular pattern with denser density and wider FoV than raw LiDAR scans. In Application-2, we combined our works ImMesh and R³LIVE++ to achieve the goal of lossless texture reconstruction of scenes. Finally, we make our code publicly available on our GitHub: github.com/hku-mars/ImMesh.

B. Limitations and future works

One major limitation of our work is its lack of scalability in spatial resolution. Specifically, when dealing with large planar

surfaces, ImMesh tends to inefficiently reconstruct the mesh with numerous small facets due to the fixed vertex density. Conversely, for tiny objects smaller than the size of a voxel, ImMesh struggles to accurately reconstruct their surfaces, as mentioned in our quantitative evaluation results in Section VIII-C5. To address this limitation, our future work will focus on developing an adaptive resolution meshing strategy.

The second limitation is that our system does not currently implement any loop correction mechanism, resulting in potential gradual drift due to accumulated localization errors at revisited places. This potentially leads to inconsistent reconstructed results if revisit occurs. In our future work, we plan to address this limitation by integrating our recent works [84, 85] on loop detection based on LiDAR point clouds. This loop detection mechanism will allow us to detect loops online and apply loop corrections to reduce drift and improve the consistency of the reconstructed results.

Furthermore, we have noticed that a number of works appeared in the literature recently, which utilize the reconstructed mesh for improving the localization accuracy of both visual-slam (e.g., [86]) and LiDAR-slam system (e.g., [87]–[89]). Motivated by these works, our future work would improve our localization accuracy by utilizing our online reconstructed mesh.

Lastly, when realizing the goal of lossless texture reconstruction of scenes, we combined ImMesh and R³LIVE at the system level as presented in our Application-2 (in Section VIII-E). Our future would couple ImMesh with R³LIVE more tightly to improve the overall efficiency.

X. ACKNOWLEDGEMENTS

The authors would like to thank DJI Co., Ltd² for providing devices and research funds.

REFERENCES

- [1] S. Mystakidis, "Metaverse," *Encyclopedia*, vol. 2, no. 1, pp. 486–497, 2022.
- [2] Y. Wang, Z. Su, N. Zhang, R. Xing, D. Liu, T. H. Luan, and X. Shen, "A survey on metaverse: Fundamentals, security, and privacy," *IEEE Communications Surveys & Tutorials*, 2022.
- [3] P. Ciproso, I. A. C. Giglioli, M. A. Raya, and G. Riva, "The past, present, and future of virtual and augmented reality research: a network and cluster analysis of the literature," *Frontiers in psychology*, p. 2086, 2018.
- [4] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in *Field and Service Robotics: Results of the 11th International Conference*. Springer, 2018, pp. 621–635.
- [5] Y. Song, S. Naji, E. Kaufmann, A. Loquercio, and D. Scaramuzza, "Flightmare: A flexible quadrotor simulator," in *Conference on Robot Learning*. PMLR, 2021, pp. 1147–1157.
- [6] S. Laine and T. Karras, "High-performance software rasterization on gpus," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, 2011, pp. 79–88.
- [7] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-time rendering*. AK Peters/crc Press, 2019.
- [8] J. Arvo, *Graphics gems II*. Elsevier, 2013.
- [9] P. Jiménez, F. Thomas, and C. Torras, "3d collision detection: a survey," *Computers & Graphics*, vol. 25, no. 2, pp. 269–285, 2001.
- [10] C. Ericson, *Real-time collision detection*. Crc Press, 2004.
- [11] R. Featherstone, *Rigid body dynamics algorithms*. Springer, 2014.

²<https://www.dji.com>

- [12] D. Baraff, "An introduction to physically based modeling: rigid body simulation i—unconstrained rigid body dynamics," *SIGGRAPH course notes*, vol. 82, 1997.
- [13] J. L. Schonberger and J.-M. Frahm, "Structure-from-motion revisited," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 4104–4113.
- [14] F. Kong, X. Liu, B. Tang, J. Lin, Y. Ren, Y. Cai, F. Zhu, N. Chen, and F. Zhang, "Marsim: A light-weight point-realistic simulator for lidar-based uavs," *arXiv preprint arXiv:2211.10716*, 2022.
- [15] W. Wang, D. Zhu, X. Wang, Y. Hu, Y. Qiu, C. Wang, Y. Hu, A. Kapoor, and S. Scherer, "Tartanair: A dataset to push the limits of visual slam," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2020, pp. 4909–4916.
- [16] D. S. SolidWorks, "Solidworks®," *Version Solidworks*, vol. 1, 2005.
- [17] B. O. Community, "Blender—a 3d modelling and rendering package," *Blender Foundation*, 2018.
- [18] C. Yuan, W. Xu, X. Liu, X. Hong, and F. Zhang, "Efficient and probabilistic adaptive voxel mapping for accurate online lidar odometry," *IEEE Robotics and Automation Letters*, vol. 7, no. 3, pp. 8518–8525, 2022.
- [19] M. Kazhdan, M. Bolitho, and H. Hoppe, "Poisson surface reconstruction," in *Proceedings of the fourth Eurographics symposium on Geometry processing*, vol. 7, 2006.
- [20] M. Kazhdan and H. Hoppe, "Screened poisson surface reconstruction," *ACM Transactions on Graphics (ToG)*, vol. 32, no. 3, pp. 1–13, 2013.
- [21] J. Wilhelms and A. Van Gelder, "Octrees for faster isosurface generation," *ACM Transactions on Graphics (TOG)*, vol. 11, no. 3, pp. 201–227, 1992.
- [22] R. Shekhar, E. Fayyad, R. Yagel, and J. F. Cornhill, "Octree-based decimation of marching cubes surfaces," in *Proceedings of Seventh Annual IEEE Visualization'96*. IEEE, 1996, pp. 335–342.
- [23] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," *ACM siggraph computer graphics*, vol. 21, no. 4, pp. 163–169, 1987.
- [24] M. Kazhdan, M. Chuang, S. Rusinkiewicz, and H. Hoppe, "Poisson surface reconstruction with envelope constraints," in *Computer graphics forum*, vol. 39, no. 5. Wiley Online Library, 2020, pp. 173–182.
- [25] P. Labatut, J.-P. Pons, and R. Keriven, "Efficient multi-view reconstruction of large-scale scenes using interest points, delaunay triangulation and graph cuts," in *2007 IEEE 11th international conference on computer vision*. IEEE, 2007, pp. 1–8.
- [26] V. Litvinov and M. Lhuillier, "Incremental solid modeling from sparse and omnidirectional structure-from-motion data," in *British Machine Vision Conference*, 2013.
- [27] M. Jancosek and T. Pajdla, "Exploiting visibility information in surface reconstruction to preserve weakly supported surfaces," *International scholarly research notices*, vol. 2014, 2014.
- [28] F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva, and G. Taubin, "The ball-pivoting algorithm for surface reconstruction," *IEEE transactions on visualization and computer graphics*, vol. 5, no. 4, pp. 349–359, 1999.
- [29] J. Cao, A. Tagliasacchi, M. Olson, H. Zhang, and Z. Su, "Point cloud skeletons via laplacian based contraction," in *2010 Shape Modeling International Conference*. IEEE, 2010, pp. 187–197.
- [30] R. Wang, J. Peethambaran, and D. Chen, "Lidar point clouds to 3-d urban models : a review," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 11, no. 2, pp. 606–627, 2018.
- [31] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon, "Kinectfusion: Real-time dense surface mapping and tracking," in *2011 10th IEEE international symposium on mixed and augmented reality*. Ieee, 2011, pp. 127–136.
- [32] J. Chen, D. Bautembach, and S. Izadi, "Scalable real-time volumetric surface reconstruction," *ACM Transactions on Graphics (ToG)*, vol. 32, no. 4, pp. 1–16, 2013.
- [33] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger, "Real-time 3d reconstruction at scale using voxel hashing," *ACM Transactions on Graphics (ToG)*, vol. 32, no. 6, pp. 1–11, 2013.
- [34] O. Kähler, V. Prisacariu, J. Valentin, and D. Murray, "Hierarchical voxel block hashing for efficient integration of depth images," *IEEE Robotics and Automation Letters*, vol. 1, no. 1, pp. 192–197, 2015.
- [35] E. Vespa, N. Nikolov, M. Grimm, L. Nardi, P. H. J. Kelly, and S. Leutenegger, "Efficient octree-based volumetric SLAM supporting signed-distance and occupancy mapping," *IEEE Robotics and Automation Letters*, vol. 3, no. 2, pp. 1144–1151, Apr. 2018.
- [36] O. Kähler, V. A. Prisacariu, C. Y. Ren, X. Sun, P. Torr, and D. Murray, "Very high frame rate volumetric integration of depth images on mobile devices," *IEEE transactions on visualization and computer graphics*, vol. 21, no. 11, pp. 1241–1250, 2015.
- [37] M. Klingensmith, I. Dryanovski, S. S. Srinivasa, and J. Xiao, "Chisel: Real time large scale 3d reconstruction onboard a mobile device using spatially hashed signed distance fields," in *Robotics: science and systems*, vol. 4, no. 1. Citeseer, 2015.
- [38] H. Oleynikova, Z. Taylor, M. Fehr, R. Siegwart, and J. Nieto, "Voxblox: Incremental 3d euclidean signed distance fields for on-board mav planning," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 1366–1373.
- [39] D. Lefloch, M. Kluge, H. Sarbolandi, T. Weyrich, and A. Kolb, "Comprehensive use of curvature for robust and accurate online surface reconstruction," *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 12, pp. 2349–2365, 2017.
- [40] D. Lefloch, T. Weyrich, and A. Kolb, "Anisotropic point-based fusion," in *2015 18th International Conference on Information Fusion (Fusion)*. IEEE, 2015, pp. 2121–2128.
- [41] T. Weise, T. Wismer, B. Leibe, and L. Van Gool, "In-hand scanning with online loop closure," in *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops*. IEEE, 2009, pp. 1630–1637.
- [42] S. Rusinkiewicz, O. Hall-Holt, and M. Levoy, "Real-time 3d model acquisition," *ACM Transactions on Graphics (TOG)*, vol. 21, no. 3, pp. 438–446, 2002.
- [43] M. Habbecke and L. Kobbelt, "A surface-growing approach to multi-view stereo reconstruction," in *2007 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2007, pp. 1–8.
- [44] T. Bodenmueller, "Streaming surface reconstruction from real time 3d measurements," Ph.D. dissertation, Technische Universität München, 2009.
- [45] T. Whelan, S. Leutenegger, R. Salas-Moreno, B. Glocker, and A. Davison, "Elasticfusion: Dense slam without a pose graph." *Robotics: Science and Systems*, 2015.
- [46] T. Whelan, R. F. Salas-Moreno, B. Glocker, A. J. Davison, and S. Leutenegger, "Elasticfusion: Real-time dense slam and light source estimation," *The International Journal of Robotics Research*, vol. 35, no. 14, pp. 1697–1716, 2016.
- [47] W. Gao and R. Tedrake, "Surfelwarp: Efficient non-volumetric single view dynamic reconstruction," *arXiv preprint arXiv:1904.13073*, 2019.
- [48] T. Schöps, T. Sattler, and M. Pollefeys, "Surfelmeshing: Online surfel-based mesh reconstruction," *IEEE transactions on pattern analysis and machine intelligence*, vol. 42, no. 10, pp. 2494–2507, 2019.
- [49] M. Teschner, B. Heidelberger, M. Müller, D. Pomerantes, and M. H. Gross, "Optimized spatial hashing for collision detection of deformable objects." in *Vmv*, vol. 3, 2003, pp. 47–54.
- [50] C++ std::unordered_map: https://cplusplus.com/reference/unordered_map/unordered_map/.
- [51] ISO, *ISO/IEC 14882:1998: Programming languages – C++*, Sep. 1998.
- [52] W. Xu, Y. Cai, D. He, J. Lin, and F. Zhang, "Fast-lio2: Fast direct lidar-inertial odometry," *IEEE Transactions on Robotics*, 2022.
- [53] Y. Cai, W. Xu, and F. Zhang, "ikd-tree: An incremental kd tree for robotic applications," *arXiv preprint arXiv:2102.10808*, 2021.
- [54] R. B. Rusu and S. Cousins, "3d is here: Point cloud library (pcl)," in *2011 IEEE international conference on robotics and automation*. IEEE, 2011, pp. 1–4.
- [55] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration." *VISAPP (1)*, vol. 2, no. 331–340, p. 2, 2009.
- [56] R. Stevens, *Computer Graphics Dictionary*, ser. ADVANCES IN COMPUTER GRAPHICS AND GAME DEVELOPMENT SERIES. Charles River Media, 2002. [Online]. Available: <https://books.google.com.hk/books?id=XqJcMi1Pi0C>
- [57] W. Kahan, "Miscalculating area and angles of a needle-like triangle," *University of California, Berkeley*, vol. 94720, 1776.
- [58] M. Woo, J. Neider, T. Davis, and D. Shreiner, *OpenGL programming guide: the official guide to learning OpenGL*. Addison-Wesley Longman Publishing Co., Inc., 1999.
- [59] F. Evans, S. Skiena, and A. Varshney, "Optimizing triangle strips for fast rendering," in *Proceedings of Seventh Annual IEEE Visualization'96*. IEEE, 1996, pp. 319–326.
- [60] D. Hearm, M. P. Baker, and M. P. Baker, *Computer graphics with OpenGL*. Pearson Prentice Hall Upper Saddle River, NJ., 2004, vol. 3.
- [61] J. Loeliger and M. McCullough, *Version Control with Git: Powerful tools and techniques for collaborative software development*. " O'Reilly Media, Inc.", 2012.

- [62] K. R. Castleman, *Digital image processing*. Prentice Hall Press, 1996.
- [63] A. Fabri and S. Pion, "Cgal: The computational geometry algorithms library," in *Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems*, 2009, pp. 538–539.
- [64] C. D. Toth, J. O'Rourke, and J. E. Goodman, *Handbook of discrete and computational geometry*. CRC press, 2017.
- [65] A. Rosinol, M. Abate, Y. Chang, and L. Carlone, "Kimera: an open-source library for real-time metric-semantic localization and mapping," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020, pp. 1689–1696.
- [66] D. Attali, J.-D. Boissonnat, and A. Lieutier, "Complexity of the delaunay triangulation of points on surfaces the smooth case," in *Proceedings of the nineteenth annual symposium on Computational Geometry*, 2003, pp. 201–210.
- [67] "Face culling in opengl." [Online]. Available: https://www.khronos.org/opengl/wiki/Face_Culling
- [68] B. Zhou, J. Pan, F. Gao, and S. Shen, "Raptor: Robust and perception-aware trajectory replanning for quadrotor fast flight," *IEEE Transactions on Robotics*, vol. 37, no. 6, pp. 1992–2009, 2021.
- [69] B. Zhou, Y. Zhang, X. Chen, and S. Shen, "Fuel: Fast uav exploration using incremental frontier structure and hierarchical planning," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 779–786, 2021.
- [70] J. Lin, C. Yuan, Y. Cai, H. Li, Y. Ren, Y. Zou, X. Hong, and F. Zhang, "Accompanying video for immesh," 2023. [Online]. Available: <https://youtu.be/pzT2fMwz428>
- [71] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *2012 IEEE conference on computer vision and pattern recognition*. IEEE, 2012, pp. 3354–3361.
- [72] N. Carlevaris-Bianco, A. K. Ushani, and R. M. Eustice, "University of michigan north campus long-term vision and lidar dataset," *The International Journal of Robotics Research*, vol. 35, no. 9, pp. 1023–1035, 2016.
- [73] T.-M. Nguyen, S. Yuan, M. Cao, Y. Lyu, T. H. Nguyen, and L. Xie, "Ntu viral: A visual-inertial-ranging-lidar dataset, from an aerial vehicle viewpoint," *The International Journal of Robotics Research*, vol. 41, no. 3, pp. 270–280, 2022.
- [74] J. Lin and F. Zhang, "R³live: A robust, real-time, rgb-colored, lidar-inertial-visual tightly-coupled state estimation and mapping package," in *2022 International Conference on Robotics and Automation (ICRA)*. IEEE, 2022, pp. 10 672–10 678.
- [75] J. Jeong, Y. Cho, Y.-S. Shin, H. Roh, and A. Kim, "Complex urban dataset with multi-level sensors from highly diverse urban environments," *The International Journal of Robotics Research*, vol. 38, no. 6, pp. 642–657, 2019.
- [76] D. Cernea, "OpenMVS: Multi-view stereo reconstruction library," 2020. [Online]. Available: <https://cdseacave.github.io/openMVS>
- [77] C. L. Lawson, "Software for c1 surface interpolation," in *Mathematical software*. Elsevier, 1977, pp. 161–194.
- [78] J. R. Shewchuk, *Delaunay refinement mesh generation*. Carnegie Mellon University, 1997.
- [79] X.-Y. Li, "Generating well-shaped d-dimensional delaunay meshes," *Theoretical Computer Science*, vol. 296, no. 1, pp. 145–165, 2003.
- [80] J. Sun, Y. Xie, L. Chen, X. Zhou, and H. Bao, "Neuralrecon: Real-time coherent 3d reconstruction from monocular video," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 15 598–15 607.
- [81] Z. Murez, T. Van As, J. Bartolozzi, A. Sinha, V. Badrinarayanan, and A. Rabinovich, "Atlas: End-to-end 3d scene reconstruction from posed images," in *Computer Vision—ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part VII 16*. Springer, 2020, pp. 414–431.
- [82] J. Lin and F. Zhang, "R³live++: A robust, real-time, radiance reconstruction package with a tightly-coupled lidar-inertial-visual state estimator," *arXiv preprint arXiv:2209.03666*, 2022.
- [83] J. Lin, C. Yuan, Y. Cai, H. Li, Y. Ren, Y. Zou, X. Hong, and F. Zhang, "Supplementary material for immesh," 2023. [Online]. Available: https://github.com/hku-mars/ImMesh/blob/main/supply/Supplementary_material.pdf
- [84] C. Yuan, J. Lin, Z. Zou, X. Hong, and F. Zhang, "Std: Stable triangle descriptor for 3d place recognition," *arXiv preprint arXiv:2209.12435*, 2022.
- [85] J. Lin and F. Zhang, "A fast, complete, point cloud based loop closure for lidar odometry and mapping," *arXiv preprint arXiv:1909.11811*, 2019.
- [86] V. Panek, Z. Kukulova, and T. Sattler, "Meshloc: Mesh-based visual localization," in *European Conference on Computer Vision*. Springer, 2022, pp. 589–609.
- [87] I. Vizzo, X. Chen, N. Chebrolu, J. Behley, and C. Stachniss, "Poisson surface reconstruction for lidar odometry and mapping," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021, pp. 5624–5630.
- [88] M. Dreher, H. Blum, R. Siegwart, and A. Gawel, "Global localization in meshes," in *ISARC. Proceedings of the International Symposium on Automation and Robotics in Construction*, vol. 38. IAARC Publications, 2021, pp. 747–754.
- [89] M. Oelsch, M. Karimi, and E. Steinbach, "R-loam: Improving lidar odometry and mapping with point-to-mesh features of a known 3d reference object," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 2068–2075, 2021.

Supplementary Material: An additional trial of our lossless texture reconstruction based on ImMesh

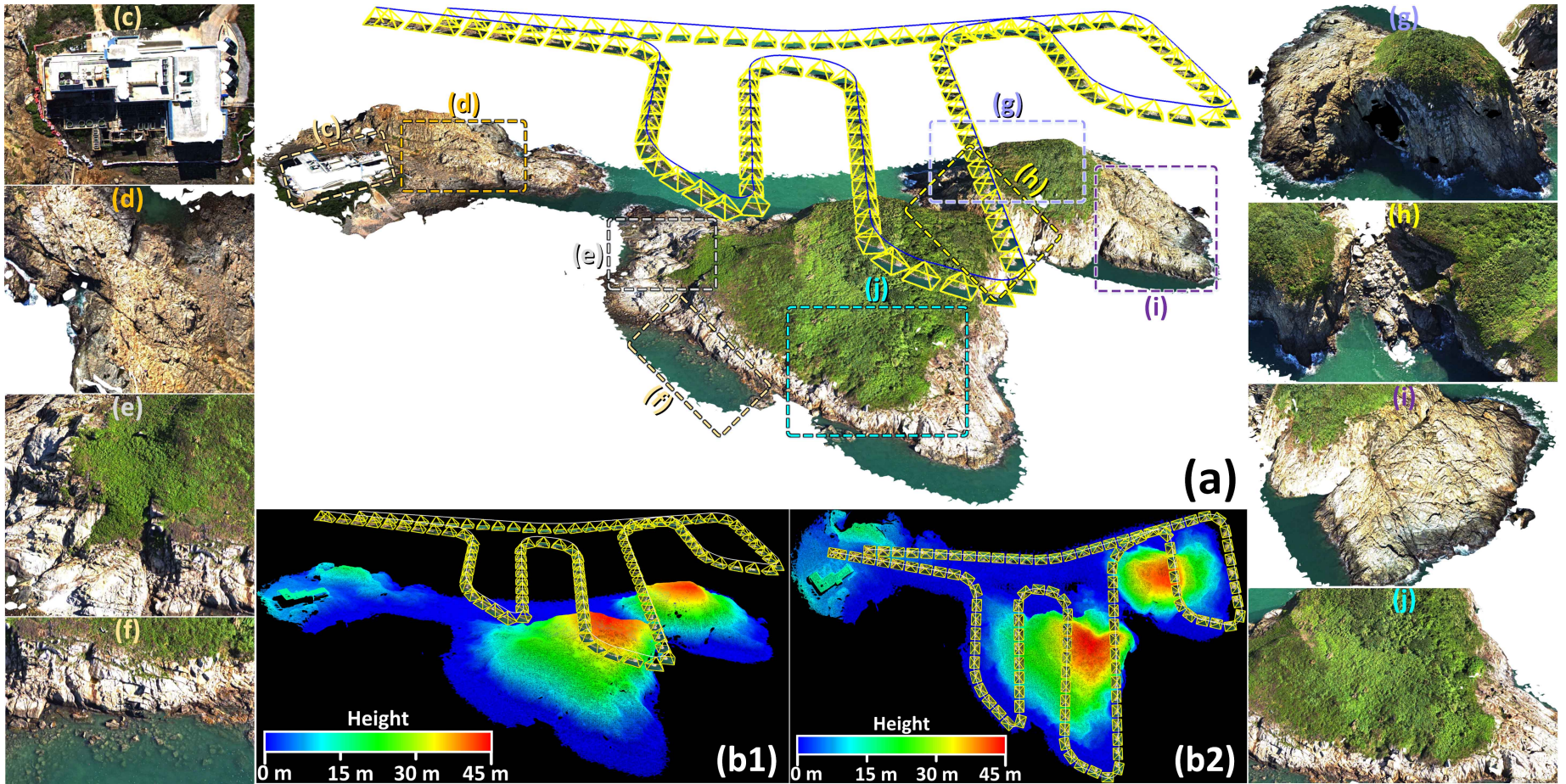


Fig. 1: Results of an additional trial test. In this trial, we collected the data by flying over islands in an “B”-like trajectory, as shown by the blue path in (a). (b1) and (b2) show the side view and bird view of our reconstructed triangle mesh, where the mesh is colored by their altitude w.r.t. the sea level. (a) show the overview of our lossless texture reconstruction result, where we use the estimated camera poses (the yellow frustums) of $R^3LIVE++$ for texturing the mesh with the collected images. The entire texture reconstruction of this 578 s sequence only costs 1210 s (on *Intel i9-10900*), with 583 s for ImMesh, 587 s for $R^3LIVE++$, and 40 s for texturing. To see the detailed reconstruction process of the scene, please refer to our video on YouTube: youtu.be/pzT2fMwz428?t=892.