An Efficient Volumetric Mesh Representation for Real-time Scene Reconstruction using Spatial Hashing

Wei Dong, Jieqi Shi, Weijie Tang, Xin Wang, and Hongbin Zha

***Abstract*— Mesh plays an indispensable role in dense real-time reconstruction essential in robotics. Efforts have been made to maintain flexible data structures for 3D data fusion, yet an efficient incremental framework specifically designed for online mesh storage and manipulation is missing. We propose a novel framework to compactly generate, update, and refine mesh for scene reconstruction upon a volumetric representation. Maintaining a spatial-hashed field of cubes, we distribute vertices with continuous value on discrete edges that support $O(1)$ vertex accessing and forbid memory redundancy. By introducing Hamming distance in mesh refinement, we further improve the mesh quality regarding the triangle type consistency with a low cost. Lock-based and lock-free operations were applied to avoid thread conflicts in GPU parallel computation. Experiments demonstrate that the mesh memory consumption is significantly reduced while the running speed is kept in the online reconstruction process.**

I. Introduction

Due to the appearance of light-weight, consumer level depth sensors such as Kinect and Structure Sensor, on-the-fly dense reconstruction of ordinary scenes has been a popular topic. In the field of robotics, real-time dense geometric acquisition enables informative environment perception and serves as a valuable cue for localization and navigation. Besides, dense 3D models portrait scenes and produce insightful visualizations.

When we refer to 3D reconstruction, it is inevitable to consider the geometric representation. In the context of real-time reconstruction using consumer level sensors, the data structures that are robust to noise and suitable for data fusion are preferred. Therefore volumetric scalar fields (*e.g.* signed distance field) have gained their reputation for the ability to easily integrate noisy data at various viewpoints; point-based methods are also appreciated for their elegance in math using filtering techniques. Mesh, as a widely-used classical 3D representation, however, is not paid much attention to for its loose organization of vertex arrays and their indices interpreted as triangles.

Although in many cases not as suitable as other methods for real-time data fusion, mesh owns various advantages. Composed by triangles, it is highly efficient for rendering, acting as the default structure on most graphics hardwares and industrial softwares. Besides, it is a reasonable simplification and sampling of the continuous 3D surfaces that can provide control points especially useful for deformation estimation, essential in real-time dynamic reconstruction [12].

All authors are with the Key Laboratory of Machine Perception, School of EECS, Peking University, Beijing, China. {w.dong, jaycee_sjq, twjjwt, xinwang_cis}@pku.edu.cn, zha@cis.pku.edu.cn

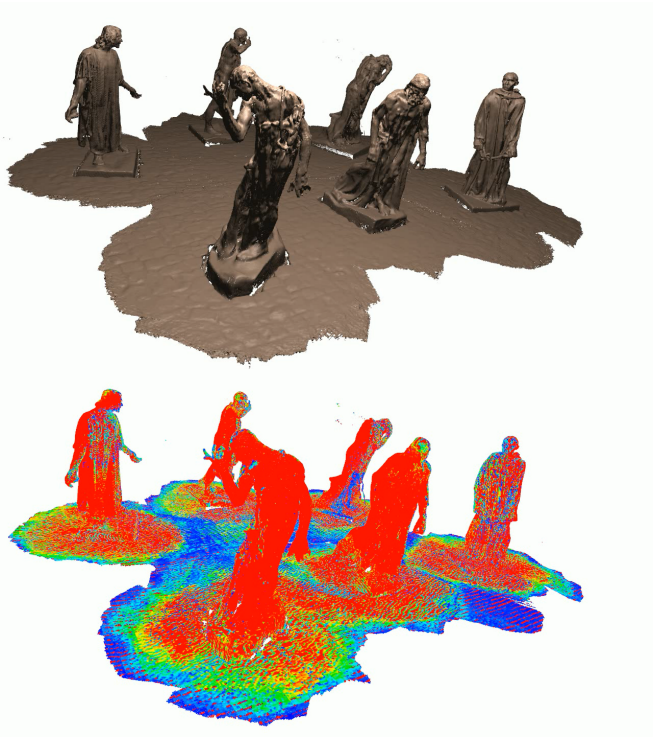


Fig. 1: *Top*, final reconstructed mesh of scene *burghers* (cube resolution 8mm) rendered with Phone shading. *Bottom*, visualization of the duration of each vertex maintained in memory; the warmer the color, the longer the duration, indicating a stronger temporal consistency.

Topology is reserved viewing the connectivity of vertices, hence 3D segmentation over mesh is also desirable to provide high-level understanding of the scene during data fusion [19].

In view of this, mesh is extracted in many incremental reconstruction frameworks where it is indispensable. These implementations are, however, usually either functionally separated as utilities [12], [17], [19], [24], or adopting loose mesh storage strategies, not fully taking the advantage of compact spatial representations [5], [7]. This would impair the neatness of a reconstruction pipeline, possibly cutting off relations between mesh and latent data; duplicate vertices are prone to be allocated, losing the mutual connections between triangles. Besides, an additional data structure such as KD-Tree or octree is required if spatial vertex querying is needed, which is not uncommon in neighbor searching and resampling that usually takes place in object deformation and segmentation.

Preprint submitted to 2018 IEEE International Conference on Robotics and Automation. Received September 12, 2017.

In this paper we design a spatial-hashing based data structure for mesh generation and update, whose advantage is two-fold: first, it is highly efficient in 3D space for mesh vertex storage and access, as proved in the voxel version [7], [12]; second, triangulation will be compact, linking the mature volumetric data fusion techniques [2] and mesh extraction functions available on volumes [10]. This framework significantly reduces the mesh redundancy, guarantees the spatial and temporal consistency of vertices, and keeps the running speed. Apart from high-quality online reconstruction, as Fig.1 depicts, it might also be regarded as an on-the-shelf solution for mesh object deformation and segmentation, due to its efficiency in unit accessing. We also reveal the deficiency in the mesh created by real-time volumetric scene reconstruction systems, and provide a simple yet effective local regularization method that improves the consistency of mesh triangle shapes.

II. Related Work

3D data representation for online fusion. Real-time dense 3D reconstruction of ordinary scenes requires data fusion, which is aimed at integrating data acquired at different viewpoints with possible overlaps, and reducing noise. To meet such demand, many representations have been proposed. A fairly popular strategy is to divide the world volumetrically, and analyze per-voxel local geometric information. Curless and Levoy [1] introduce the signed distance field (SDF) to describe the Euclidean distance from each voxel center to its closest surface. Newcombe *et al.* [2] adopt a truncated version of signed distance field (TSDF) and implement a real-time application on GPU which incrementally fuses depth data captured by a Kinect. Since it manages the spatial volume with a 3D plain array, its working space is limited due to memory constraints. Zeng *et al.* [3] utilize an octree to replace the plain array, reducing the memory consumption to some extent. Similarly, Steinbrücker *et al.* [5] propose an octree-based structure that is able to run in real-time on a CPU. Whelan *et al.* [4], [17] instead maintain a moving volume of active area, and generate mesh when a region is streamed out. Nießner *et al.* [6] use a 2-level cascade voxel hashing strategy to manage voxels that is highly efficient for GPU. The method is extended to CPU by Klingensmith *et al.* [7], and is further optimized by Kahler *et al.* [23]. Other than volumetric approaches, there are also point-based [22] and surfel-based [8] methods to perform data integration. Marton *et al.* [18] demonstrate an adaptive mesh generation method by directly re-sampling over point clouds, but the underlying KD-tree is not efficient enough to support real-time processing. Zienkiewicz *et al.* [9] fuse data into mesh with non-local optimizations; presented as a 2.5D height map, occlusions can hardly be handled.

Real-time rendering. Regarding the underneath representation of 3D data type, *i.e.* , volume, point cloud, and mesh, several approaches have been raised to reveal the underlying 3D surfaces so as to render and visualize. Rendering mesh is trivial, as the modern graphics pipelines are mostly designed for triangles. For volumetric data, there are mainly two options available: generate mesh at the isosurfaces from the volumetric field with methods such as Marching Cubes (MC) [10] and fall back to the regular triangle rendering [5], [7], or directly trace each ray from the pixels of a virtual camera to find the intrinsic physical properties, *i.e.* , the surfaces laying on the zero-crossing set [2], [6], [23], which is in principle the same as surface determination in MC. Point-based rendering have also been proposed for dense visualization of point clouds [11], usually based on splatting. Due to the architecture of modern graphics hardwares, the techniques other than mesh rendering are relatively more expensive and less compatible, therefore a conversion into mesh is preferred in various systems.

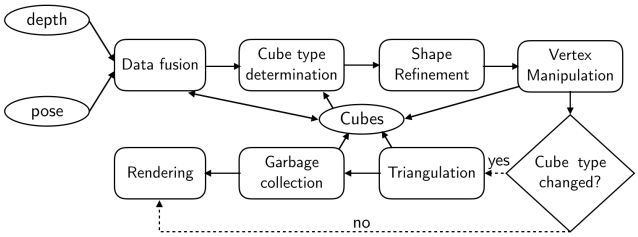


Fig. 2: System pipeline.

Mesh generation from volumes. The cornerstone of mesh generation from volumetric data is laid by Lorensen and Cline [10] with MC. This simple algorithm that can run in parallel has been widely used up to now with various refinements [21], [28], however mostly designed for static data. In the real-time reconstruction systems, MC is usually implemented in its original form with minor adaptations to the data structure of the volumetric scalar field. Steinbrücker *et al.* [5] manage the 3D space with an octree and store mesh in each node with $8\times8\times8$ voxels, where complicated border situations are decided and a recursive search through the tree is processed. Klingensmith *et al.* [7] follow [6] and divide the space into spatial-hashed bricks, each holding a batch of voxels (*e.g.* , $8\times8\times8$). Only bricks in the sensor's viewing frustum will be operated for mesh generation, where a vector of mesh triangles are loosely maintained per brick. These triangles are not connected even with shared vertices; the incremental meshing for each frame can be described as an entire new mesh generation in local areas, where no temporal continuity is reserved.

III. System Overview

Our system extends the prevalent volumetric dense reconstruction pipeline in [6], [7], illustrated in Fig.2. The system is fed by a stream of depth images acquired by a hand-held sensor such as Kinect, along with the sensor's poses assumed known. In our work, the poses come from ground truth; RGB-D version of ORB-SLAM2 [13] can also be utilized as a black box for pose estimation. At each timestamp, the depth image is fused into the maintained volumetric *cube* field by changing TSDF distributions at the *corner* of each cube, to be discussed in §IV; afterwards, a local mesh generation or update is performed based on

Preprint submitted to 2018 IEEE International Conference on Robotics and Automation. Received September 12, 2017.

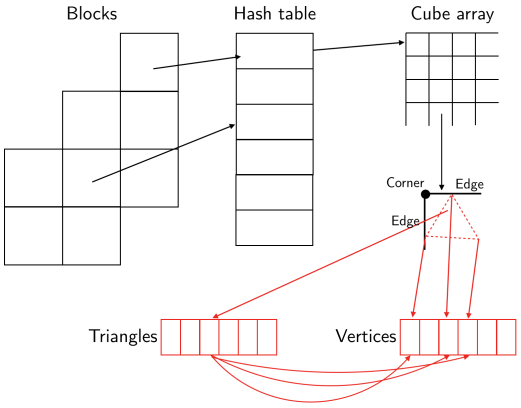


Fig. 3: 2 level spatial management. Space is first coarsely divided into blocks around surfaces, managed by a hash table. Blocks are then further split into an array of cubes. Each cube holds 1 corner, 3 edges, and up to 5 triangles, which are allocated on the memory heap. Visualized in 2D for simplicity.

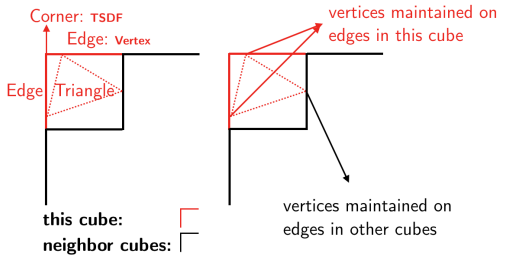


Fig. 4: Cube level data structure. Only 1 corner and 3 edges are maintained per cube, while others can be accessed at adjacent cubes. Visualized in 2D for simplicity.

MC [10], possibly accompanied by local refinements; finally mesh is reformatted for traditional triangle rendering instead of ray-casting. Our algorithms are specifically designed for parallel running on a GPU, but can be easily moved to CPU. In the following sections, the stages of the pipeline will be discussed one-by-one, except for mesh refinement and rendering. The former requires a detailed observation, therefore is separated, while the latter is too trivial to be discussed.

Fig.3 shows a 2-level cascade structure to manage spatial information following [6]. At first, it splits the space into large *blocks* as the basic unit for spatial hashing; each block is further divided into many (*e.g.* $8 \times 8 \times 8$) small *cubes* to hold local geometric information. This strategy constraints the size of hash table and therefore avoids hash collision to some extend, meanwhile guarantees the resolution of geometric information.

In [5], [7], [23] scalar geometric values, *i.e.* TSDF, are stored at the fine-scale *voxel* level, while triangles are coarsely managed in the *block* level in hierarchy. Instead, we carefully maintain a *cube* structure to hold both triangles with their vertices and TSDF values, shown in Fig.4; all the mesh manipulations are performed at the fine scale.

IV. Volumetric Triangle Representation

Terms are first introduced in this section. The space is split into *blocks* allocated only around object surfaces. A block is further divided into *cubes*, typically owning 8 *corners* and 12 *edges*; considering the overlap, however, only 1 corner and 3 edges need to be stored in average.

In detail, we align each cube to the xyz axises, maintain the corner at $c = (x_0, y_0, z_0)$, and preserve the edges $e_x = (l, 0, 0)$, $e_y = (0, l, 0)$, and $e_z = (0, 0, l)$ that start from c, where l is the cube's side length. TSDF $d(c)$ is incrementally updated on c for data fusion, and vertices v_x, v_y, v_z intersected on axises, if existing, are stored on the correspondent edges with limited local degree of freedom ensured by MC. This binds the continuous vertex position to a discrete edge coordinate, making it possible for vertices to be directly accessed in $O(1)$ with a hash table visiting plus a local indexing; vertex sharing between adjacent cubes becomes especially simple via edge indexing. A cube also holds up to 5 triangles that connect the vertices on edges, which might come from a nearby cube; cube type in MC is recorded as a supplement to indicate the number and shape of triangles, both previously and currently, to be discussed in §V-B.

In terms of memory efficiency, edges and triangles in a cube are stored in pointer arrays of the size 3 and 5, while the pointed data are managed on the memory heap. This can be further optimized by saving only 1 pointer each for edges and triangles, where pointer arrays are also dynamically managed on the memory heap. A vertex stores position and normal, and reserves the space for color. In addition, we introduce a reference count to determine whether recycle is needed, referring to §V-E. A triangle holds 3 pointers to index its vertices.

V. Mesh Generation, Update, and Refinement

A. Block Collection and Data Fusion

When a depth image $\mathcal{D}_i : \mathbb{R}^2 \to \mathbb{R}$ along with a sensor pose (from sensor to world) ${}^w_s T_i = [{}^w_s R_i \mid {}^w_s t_i] \in SE(3)$ is received at the timestamp i, we first find each valid pixel $p \in \Omega_i \subset \mathbb{R}^2$, where Ω_i is the set of valid pixels in $\mathcal{D}_i$, and form a ray:

$$r = {}^w_s t_i + \lambda {}^w_s R_i \mathcal{D}_i(p) K^{-1} \tilde{p}, \tag{1}$$

where $\tilde{p}$ is the 3D homogeneous coordinate, K is the intrinsic matrix of the sensor, λ is the length parameter along the ray, and $\mathcal{D}_i(p)$ reads the depth value at p. In a certain range around the scanned point along r, *i.e.* $\lambda \in [1 - \delta, 1 + \delta]$, blocks are collected and will be allocated if have not been so. Therefore only the blocks affected by new observations will be processed.

After collection, every corner of cubes $c \in \mathbb{R}^3$ inside the gathered blocks are projected to the depth image to find the approximately closest scanned point, and truncated distance

Preprint submitted to 2018 IEEE International Conference on Robotics and Automation. Received September 12, 2017.

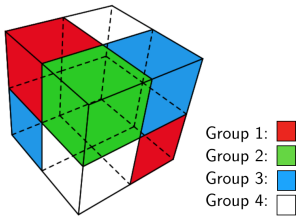


Fig. 5: The grouping of cubes. Cubes at opposite poles are gathered with no overlapping edge hence no shared vertex.

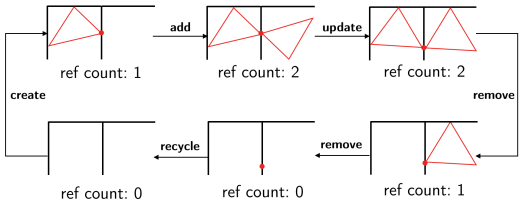


Fig. 6: Change of a vertex's reference count according to triangle insertion and deletion. Recycling will be triggered when a vertex is not referenced anymore.

is computed accordingly:

$$\tilde{d}_i(c) = \phi(\mathcal{D}_i(K_s^w T_i^{-1} c) - ({}_s^w T_i^{-1} c).z), \tag{2}$$

where $\phi(\cdot)$ is the truncation function and $({}_s^w T_i^{-1} c).z$ is the depth of c in the camera coordinate system; $\tilde{d}_i(c)$ is then integrated into stored $d(c)$, details discussed in [2]. TSDF value inside the sensor's viewing frustum around surfaces will be updated, being the basis of mesh generation.

B. Cube Type Determination

MC [10] is utilized in the generation of mesh in the following sections. In MC, a table $\mathcal{T} : \{0,1\}^8 \rightarrow \{0,1\}^{12}$ is precomputed to indicate the triangle distributions, *i.e.* the number of triangles and on which edges do their vertices lie. Each bit of t denotes whether the scalar value at the related corner (in our case, $d(c)$) is below an isovalue (in our case, 0); each bit of $\mathcal{T}(t)$ indicates the existence of a vertex on the corresponding edge, 3 in the current cube and 9 in adjacent cubes. In most situations, the access of scalar value at corners is as trivial as visiting an adjacent value in a plain array. There exists border cases that the neighbor cube providing shared corner is not in the same block, where an additional $O(1)$ spatial hash table lookup is required. The current cube type t_i is computed and stored along with the previous cube type t_{i-1} to provide a cue for temporal consistency.

C. Vertex Initialization and Update

Having determined $\mathcal{T}(t)$, linear interpolation of endpoints' positions of an edge whose indicator bit is 1 will be computed in order to decide the position of the vertex it binds. The assignment is lazy: vertices are initialized only when first used; otherwise an update is sufficient.

The most elaborate part of this method different from the original version lies in vertex sharing in the neighbor cubes. In a serial implementation, *e.g.* loop based CPU version, this is trivial once we choose the correct loop order. This is however, absolutely non-trivial when the program runs on a GPU where thousands of stream-processors are working simultaneously and vertices are determined in parallel. If no care is taken of, memory leak will be severe, causing 2 to 3 times of additional memory consumption; unexpected results may also take place. We attempt two solutions to guarantee the correctness of sharing:

Lock-free version. A typical method to avoid conflict between threads is to utilize the reduction method with a divide-and-conquer strategy [27]. In its original form to sum up an array of numbers, the array is divided into two non-overlapping parts and summed up in each part; the process is iteratively operated until the array is not separable.

Inspired by this manipulation, we divide the 3D array of cubes inside a block into several groups in which no overlap exists. Illustrated in Fig.5, we divide a $2 \times 2 \times 2$ cube into 4 parts, which can also be extended to a wider region. As no edges are shared, this process is lock-free and can run in parallel.

Lock-based version. Lock is another traditional solution for resource sharing. A pure mutex-based operation will be inappropriate, however, as thousands of threads querying mutexes will easily lead to severe deterioration of performance. Instead, we adopt an atomic operation under such circumstances. In this implementation, only the first thread who atomically acquires a vertex will have the privilege to allocate and assign it. Since the interpolation ratio of an edge's endpoints are already determined in the data fusion stage, the correctness of the vertex's position will hold for the other threads.

D. Triangulation

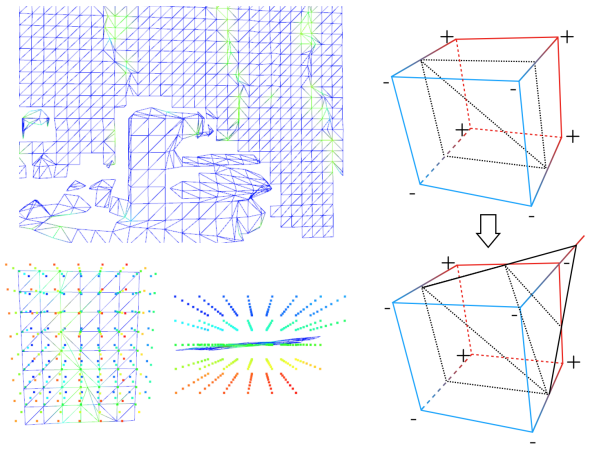
Up to this stage, we have determined the vertices of triangles to be processed. To reduce the cost of triangle allocation and assignment, we compare t_i and t_{i-1}: if they coincide, common in the incremental process, the list of triangles and their vertices will remain unchanged inside the cube, keeping a temporal consistency; otherwise the previous triangles will be cleared and new ones will be created.

E. Garbage Collection

The shared vertices are referenced by and only by triangles. In order to manage memory correctly, we use the reference counting technique. When a new triangle is created, the reference count of its related vertices will be increased by 1; when a destroying operation takes place, a symmetry decrease operation will be processed, as Fig.6 illustrates. The vertices with a 0 reference count will be regarded as garbage and recycled, waiting for a new allocation.

Aside from a recycling indicator, the reference count can also be regarded as the degree of a vertex in topology, which might be an important property in mesh analysis.

Preprint submitted to 2018 IEEE International Conference on Robotics and Automation. Received September 12, 2017.



(a) (b)

Fig. 7: Illustration of 'cracks'. *(a), Top*, a general look into the *copyroom* scene where a printer is placed in front of a wall. Color of mesh represents the confidence of TSDF values around vertices, green accounts for more unsteadiness. Note the irregular green triangles appear in pattern. *Left bottom*, a closer view of a specific block, where TSDF values at corners are also visualized in color. The warmer the color, the more positive the value; green is approximately 0. *Right bottom*, upper view of the block with mesh. Triangles whose vertices are around zero-value TSDF corners are prone to be irregular. *(b)*, abruptly changed triangles in *dotted lines* emerge due to disturbance of SDF value at one corner.

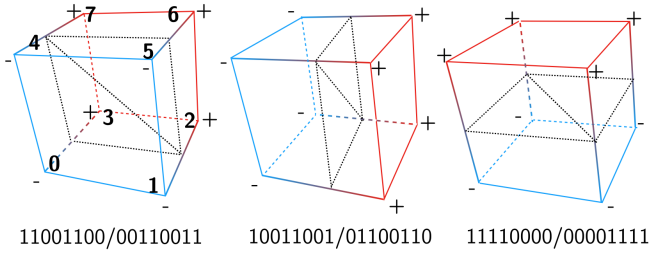


Fig. 8: Regular and most frequently created triangle shapes in ordinary scenes. The order of 8-bits is presented in the first cube.

F. Mesh Refinement

A problem of mesh extracted from volumetric fields during online data fusion can be found in the illustrated figures in literature [5], [7], and also appears in our system. It is an interesting phenomenon depicted in Fig.7a, where irregular triangles appear in an observable pattern. After a careful analysis, we find it is the limited resolution and the principle of MC that leads to the deficiency. In most cases, a real world plane will go through the middle of a cube, see Fig.8; the corners of the cube at two sides separated by the plane will hold TSDFs who are dominated by a series of $\tilde{d}_i(c)$ (see §V-A) of the same sign. In such a case, the cube type t can be determined with confidence, producing two triangles that is neat enough to represent the crossing plane of a cube. However, when a plane in the world coordinate system is not strictly parallel to the axises of cubes, it is highly possible to intersect cubes at corners, as illustrated in Fig.7. In such cases, positive and negative $\tilde{d}_i(c)$ are distributed evenly at c; a small disturbance would lead to the flip of sign of $d(c)$, hence the bit-array t will be directly affected, resulting in an abrupt change of the output $\mathcal{T}(t)$ and its correspondent triangle distribution shown in Fig.7b. This event would repeat itself along the plane every time such an intersection occurs.

Attene *et al.* [26] provided a comprehensive review of available mesh repair techniques in the application perspective, yet a satisfying on-the-fly solution does not appear regarding online mesh generation. Dzitsuik *et al.* [14] have came up with the idea of fitting planes to increase the smoothness in an incremental fashion. This method runs efficiently when the scene is smooth with many planes, but might face performance problems in a complex environment with a high resolution according to our experiments. In the context, we introduce an simple yet effective local method to reformat the triangle shapes.

Intuitively, in the disturbed cube in Fig.7b, if we extend the affected triangle edges, they would approximately intersect outside the cube (may not exactly intersect, but fairly close given a small TSDF's absolute value), forming a large triangle bounded by the *solid lines*. Assuming a smooth transition of TSDF in the local area, the extrapolated vertex would coincide with the vertex held by its correspondent neighbor. Therefore geometrically it is reasonable to eliminate the small fragments and maintain one large triangle instead. A tricky implementation is adopted to achieve the target: we simply set the type of the disturbed cube to the undisturbed type and run MC. The interpolation in MC would, with the same equation, serve as extrapolation given the same sign of TSDF of two adjacent corners.

The following operation detects the disturbance: given the 8-bit vector t_{i-1} and t_i denoting the previous and currently estimated cube type, if

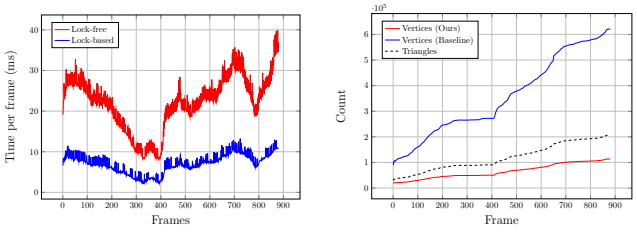
$$d_H(t_i, t_{i-1}) \leq 3, \tag{3}$$

$$d_H(t_i, t_{rj}) \leq 3, \exists j \in \{1 \cdots 6\}, \tag{4}$$

$$|d(c_k)| < \epsilon, \forall k \mid t_{i,k} \oplus t_{rj,k} = 1, \tag{5}$$

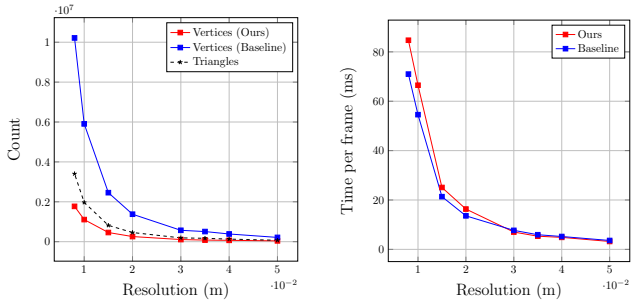
are satisfied, where d_H denotes the Hamming distance, $t_{i,k}$ is the kth bit of t_i, $d(c_k)$ reads the TSDF value at the kth corner, $\oplus$ is the xor sign, ϵ is a preset threshold proportional to cube size, and $t_{r1\ldots6} = \{11001100, 00110011, 10011001, 01100110, 11110000, 00001111\}$ holds the 'regular cube' types shown in Fig.8, then we assume it is the disturbances at the kth corners in Eq.5 that flip the sign, leading to irregularity. Under such circumstances, $t_i = t_{rj}$ is applied before MC.

This approach is robust: Hamming distance between each pair of regular type vector $t_{rj} \in t_{r\{1\ldots6\}}$ is either 4 (perpendicular) or 8 (parallel with all sign reversed), therefore choosing 3 as a discriminating value in Eq.3-5, the triangles will adhere to the closest regular type in Fig.8; the sign



(a) (b)

Fig. 9: Experiment results of an incremental reconstruction on *office2* sequence with a 3cm cube resolution. *(a)*, running time comparison between *lock-based* and *lock-free*. *(b)*, mesh memory consumption.



(a) (b)

Fig. 10: Comparison of our method and baseline. *(a)*, mesh memory consumption. *(b)*, average running time. Both experiments are conducted on the *office2* dataset with a series of resolution configurations.

reversion will also be strongly limited by the temporal constraint t_{i-1} and the tolerant threshold ϵ.

VI. Experiments

We test our method on various RGB-D datasets, including ICL-NUIM [15], TUM [16], and dataset provided by Zhou and Koltun [20], where depth images with registered poses are all provided. The experiments are conducted on a laptop with an Intel Core i7-6700HQ CPU, and an Nvidia GTX 1070M graphics card. We take advantage of the core components including GPU hash table and data fusion from the open-source code provided by [6], and implement the meshing pipeline entirely. The code is written in C++, with CUDA 8.0 for parallel computation and OpenGL 3.3 for rendering. In CUDA, each *block* is assigned to a stream processor, and each *cube* is manipulated by a thread. In all configurations, a *block* contains $8 \times 8 \times 8$ *cubes*. The generated mesh is directly compressed and copied in GPU memory from the CUDA context to OpenGL for real-time feedback.

A. Lock-based and Lock-free Comparison

In §V-C we have discussed two possible solutions for parallel vertex sharing. To determine which approach to adopt, we evaluate both and draw the conclusion that the *lock-free* implementation, although theoretically achievable, is not better than the *lock-based* version.

Fig.9a illustrates the result of running time for meshing stage of two methods. The time of *lock-free* is 2 to 3 times of the *lock-based* version; it seems that the avoidance of thread conflicts cannot compensate for the expenses of group-level serial launches. This trend also holds for higher cube resolutions. In view of this, we choose the *lock-based* version in following experiments. The idea of grouping cubes in *lock-free* might be utilized in a multi-threaded CPU version where the order of loop is critical.

B. Memory and Running Time Results

In this section, we compare our incremental meshing method with the previous work proposed in [7], for which we also implemented a parallel version as the baseline. For simplicity, all the mesh is stored in a global array instead of arrays allocated per block in [7]. Without loss of fairness, we generate mesh only for blocks in frustum to test running speed, and generate mesh over all blocks to test memory usage.

Memory. Experiments are first conducted to show reduction of mesh memory consumption. Fig.9b shows a typical trend of mesh accumulation during the sequential online reconstruction. The growing of vertex count is significantly constrained in our method, compared to baseline. Fig.10a illustrates the number of vertices with different resolution selections. When the precision is fairly high, the gain will be considerable. In Table I we list the memory consumption for several datasets.

It seems strange at the first glance that the vertices are even fewer than triangles. Consider a mesh that looks like the regular part in Fig.7: when we take an area of $w \times h$, the number of triangles will be $2wh$, while the number of vertices will be $(w+1)(h+1)$, hence in the infinite case the triangles will be $\lim_{w,h\to\infty} 2wh/[(w+1)(h+1)] = 2$ times of the vertices. Therefore, any vertex count that is greater than half of triangle count will be reasonable.

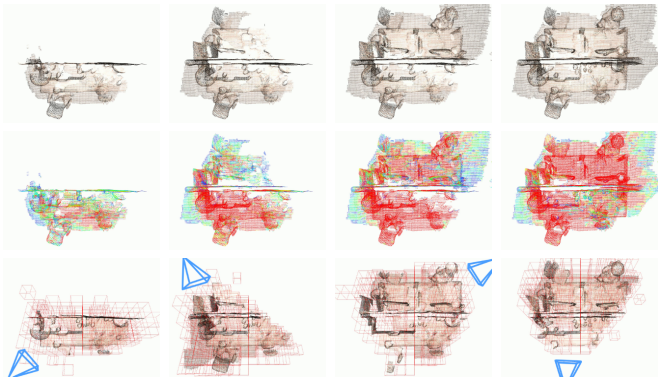
At current, although a reduction of vertex count is apparent, the memory cost in total (at 8mm cube resolution, 50000 blocks, 1.8M vertices, and 3.5M triangles, which is enough for all our test scenes) is in fact increased to about 1.6GB, since the data structure of a cube is not fully optimized, storing 56 bytes per unit. If it were minimized to 24 bytes by optimization discussed in §IV, the total memory including the cubes and the mesh they hold will be reduced to around 700MB, approximately the same as the memory of a TSDF field plus the non-optimized mesh [7]. With a similar total memory cost, our method holds much more geometric information such as connectivity, and supports $O(1)$ vertex accessing.

Time. While introducing additional computations, we manage to maintain the running speed of meshing stage (refinement included) in general. For relatively low cube resolution, *i.e.* 3cm, typically suitable for dense reconstruction used in most real-time robotics applications, the running time is slightly faster than the baseline, as shown in Fig.10b

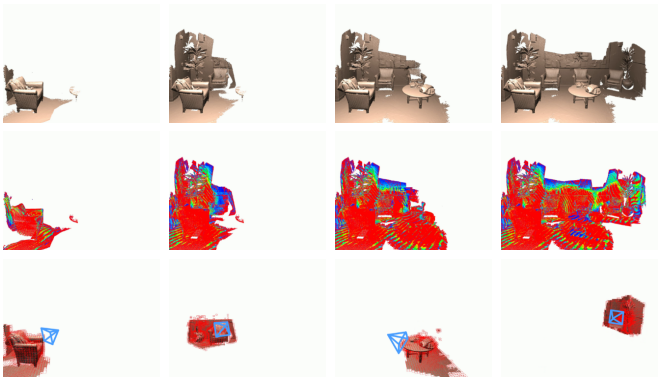
Preprint submitted to 2018 IEEE International Conference on Robotics and Automation. Received September 12, 2017.

Dataset	Frames	Time (ms)				Memory (vertex count)	
		Ours		Baseline		Ours	Baseline
		Meshing	All	Meshing	All		
ICL/lv1	965	**6.06**	7.20 (8.35)	6.55	7.48	**85634**	450903
ICL/lv2	880	**6.89**	7.85 (9.19)	7.10	8.05	**105406**	583986
ICL/office1	965	**5.36**	6.42 (7.71)	5.88	6.78	**103574**	577011
ICL/office2	880	**7.09**	8.09 (9.39)	7.70	8.54	**115885**	619629
TUM/household	2486	10.94	12.21 (12.90)	**9.61**	10.83	**64198**	327729
Zhou/copyroom	5490	**3.71**	4.85 (5.68)	3.95	4.94	**85699**	446775
Zhou/lounge	3000	**4.03**	5.05 (5.85)	4.08	5.05	**62144**	323562
Zhou/burghers	11230	**3.67**	4.67 (5.48)	3.76	4.72	**99976**	532152

TABLE I: Average running time and total vertex consumption comparison of our method and baseline, at the resolution of 3cm. In implementation, our method requires an additional compressing operation before copying data to the rendering pipeline, while this step is ignored in the baseline due to our simplified implementation. Therefore total running time of our method is displayed both without and with compressing stage, the latter in brackets.

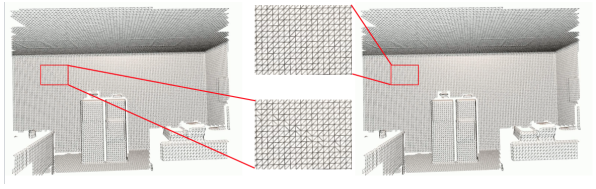


(a)

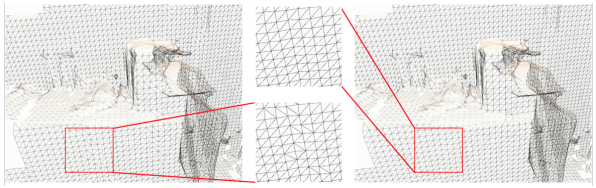


(b)

Fig. 11: Incrementally reconstructed mesh. *(a)*, *household* with cube resolution 3cm and max scanning range 2.5m. *(b)*, *lounge* with cube resolution 8mm and max scanning range 1.6m (20 frames with heavy motion blur were manually filtered out). Each 3 rows from top to down: global mesh; visualized duration of vertices, where warmer color indicates a longer sustained time; locally updated mesh in frustum, where red bounding boxes represent blocks and blue pyramids denote viewing frustums (line width exaggerated for easier recognition). Best viewed in color and enlarged ($\geq 300\%$).



(a)



(b)

Fig. 12: Mesh before (left) and after (right) refinement. *(a)*, simulated *office1* dataset. *(b)*, real world *copyroom* dataset. Besides areas zoomed in, similar refinements appear in the entire scene. The images are slightly degenerated due to compression.

and Table I. It turns out that our method becomes slower than baseline in the very dense case, *i.e.* 8mm. This might be improved by dealing border cubes per block specifically, where many redundant hash queries are processed. In spite of this, our data structure serves as a trade-off between mesh accessing and generation, potentially saves a large amount of time to access, modify, and analyze mesh, which are highly possible to be required in further operations such as deformation and segmentation.

C. Qualitative Results

Since the mesh generated by our method is in theory (also in implementation) identical to the baseline method in geometric appearance, we do not focus on comparing mesh quality with baseline. Instead, we conduct experiments in two aspects, incremental reconstruction and refinement.

Incremental reconstruction. We process two sequences and render the global mesh against the newly modified mesh in sensor's frustum per frame. In addition, we visualize the existing duration of vertices, see Fig.11. In *household*

Preprint submitted to 2018 IEEE International Conference on Robotics and Automation. Received September 12, 2017.

(Fig.11a) where sensor is generally far from the scene objects and motion blurs appear frequently, we accept a large scanning range to fuse in more valid data and a coarse cube resolution to resist noise. When a loop closure emerges, most previous vertices are preserved, as shown in the color map. In *lounge* (Fig.11b) where sensor are close to the objects and depth images are carefully captured, we run the program with a small scanning range and a high resolution. Most blocks are ignored during mesh generation, saving a large amount of time, while the mesh representing the whole scene remains consistent.

Refinement. Results with and without mesh refinement stage are compared both in the simulated dataset *office2* (Fig.12a) with perfect sensor poses and depth images and the real-world dataset *copyroom* (Fig.12b). The irregular 'cracks' in the scenes are significantly reduced, leading to consistent triangle shapes and smooth planes. In the incremental reconstruction, the type of triangles sometimes suffer instability due to frequently flipped SDF signs around, which could be further ameliorated by emphasizing temporal constraints.

VII. Conclusions and Future Work

We propose a novel mesh representation with spatial hashed cube units that supports memory-efficient vertex sharing and time-efficient $O(1)$ accessing. Equipped with parallel algorithms, the data structure achieves considerable performance in the task of real-time scene reconstruction; additional refinement further improves the quality of mesh.

In the future, we plan to optimize the project by a more careful border case manipulation and a further simplification of cube unit structure. More sophisticated spatial hashing structure might be introduced as proposed in [25]. We intend to open source the code when the project is refined enough as an useful tool for both reconstruction and mesh-based deformation and segmentation. In the research viewpoint, on the other hand, we are working on building an entirely mesh-based pipeline that directly involve 3D data and priors with local optimization process where TSDF is not explicitly needed.

References

[1] B. Curless and M. Levoy, "A Volumetric Method for Building Complex Models from Range Images", in ACM SIGGRAPH, 1996.

[2] R. Newcombe, A. Davison, S. Izadi, P. Kohli, O. Hilliges, J. Shotton, D. Molyneaux, S. Hodges, D. Kim, and A. Fitzgibbon, "KinectFusion: Real-time Dense Surface Mapping and Tracking", in ISMAR, 2011.

[3] M. Zeng, F. Zhao, J. Zheng, and X. Liu, "Octree-based Fusion for Real-time 3D Reconstruction", Graphical Models, vol 3, pp 126-136, 2013.

[4] T. Whelan, M. Kaess, H. Johannsson, M. Fallon, J. Leonard, and J. McDonald."Real-time Large-scale Dense RGBD SLAM with Volumetric Fusion", IJRR, vol 34, pp 598-626, 2015.

[5] F.Steinbrucker, J. Sturm, and D. Cremers. "Volumetric 3D Mapping in Real-time on a CPU", in ICRA, 2014.

[6] M. Niessner, M. Zollhofer, S. Izadi, and M. Stamminger."Real-time 3D Reconstruction at Scale Using Voxel Hashing", ACM TOG, vol 6, pp 169, 2013.

[7] M. Klingensmith, I. Dryanovski, S. Srinivasa, and J. Xiao. "CHISEL: Real Time Large Scale 3D Reconstruction On-board a Mobile Device using Spatially-Hashed Signed Distance Fields", in RSS, 2015.

[8] K. Kolev, P. Tanskanen, P. Speciale, and M. Pollefeys, "Turning Mobile Phones into 3D Scanners", in CVPR, 2014.

[9] J. Zienkiewicz, A. Tsiotsios, A. Davison, and S. Leutenegger. "Monocular Real-time Surface Reconstruction using Dynamic Level of Detail", in 3DV, 2016.

[10] W. Lorensen and H. Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm", in ACM SIGGRAPH, 1987.

[11] M. Botsch and L. Kobbelt. "High-quality Point-based Rendering on Modern GPUs", in PCCGA, 2003

[12] R. Newcombe, D. Fox, and S. Seitz."DynamicFusion: Reconstruction and Tracking of Non-rigid Scenes in Real-time", in CVPR, 2015.

[13] R. Mur-Artal and J. D. Tardos."ORB-SLAM2: an Open-source SLAM System for Monocular, Stereo and RGB-D Cameras", arXiv preprint, 2016.

[14] M. Dzitsiuk, J. Sturm, R. Maier, L. Ma, and D. Cremers."De-noising, Stabilizing and Completing 3D Reconstructions On-the-go using Plane Priors", in ICRA, 2017.

[15] A. Handa, T. Whelan, J. McDonald, and A. Davison. "A Benchmark for RGB-D Visual Odometry, 3D Reconstruction and SLAM", in ICRA, 2014.

[16] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers. "A Benchmark for the Evaluation of RGB-D SLAM Systems", in IROS, 2012.

[17] T. Whelan, M. Kaess, M. Fallon, H. Johannsson, J. Leonard, and J. McDonald. "Kintinuous: Spatially Extended Kinect-Fusion", Technical report, 2012.

[18] Z. C. Marton, R. B. Rusu, and M. Beetz. "On Fast Surface Reconstruction Methods for Large and Noisy Point Clouds", in ICRA, 2009.

[19] R. Salas-Moreno, R. Newcombe, H. Strasdat, P. Kelly, and A. Davison."SLAM++: Simultaneous Localisation and Mapping at the Level of Objects", in CVPR, 2013.

[20] Q. Zhou and V. Koltun."Dense Scene Reconstruction with Points of Interest". ACM TOG, vol 4, pp 112, 2013.

[21] T. S. Newman and H. Yi."A Survey of the Marching Cubes Algorithm", Computers and Graphics, vol 5, pp 854-879, 2006.

[22] M. Keller, D. Lefloch, M. Lambers, T. Weyrich, and A. Kolb."Real-time 3D Reconstruction in Dynamic Scenes using Point-based Fusion", in 3DV, 2013.

[23] O. Khler, V. Prisacariu, C. Ren, X. Sun, P. Torr, and D. Murray."Very High Frame Rate Volumetric Integration of Depth Images on Mobile Devices", in ISMAR, 2015.

[24] V. Prisacariu ,O. Khler, S. Golodetz, M. Sapienza, T. Cavallari, P. Torr, and D. Murray. "InfiniTAM v3: A Framework for Large-Scale 3D Reconstruction with Loop Closure", arXiv e-prints, 2017.

[25] O. Kahler, V. Prisacariu, J. Valentin, and D. Murray. "Hierarchical Voxel Block Hashing for Efficient Integration of Depth Images", IEEE RA-L, vol 1, pp 192-197, 2016.

[26] M. Attene, M. Campen, and L. Kobbelt. "Polygon Mesh Repairing: An Application Perspective." ACM CSUR, vol 2, pp 15, 2013.

[27] M. Harris."Optimizing CUDA", in SC07: High Performance Computing With CUDA, 2007.

[28] T. Zirr."Simplistic Marching Cubes Optimization", online resource, http://alphanew.net/index.php?section=articles&site=marchoptim.

Preprint submitted to 2018 IEEE International Conference on Robotics and Automation. Received September 12, 2017.