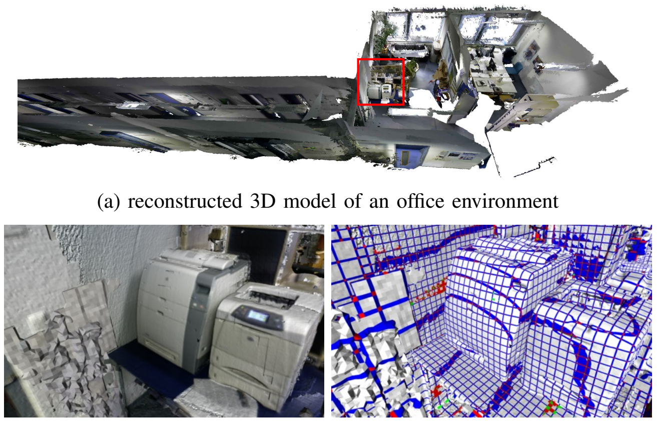
Volumetric 3D Mapping in Real-Time on a CPU

Frank Steinbrücker, Jürgen Sturm, and Daniel Cremers

Abstract— **In this paper we propose a novel volumetric multi-resolution mapping system for RGB-D images that runs on a standard CPU in real-time. Our approach generates a textured triangle mesh from a signed distance function that it continuously updates as new RGB-D images arrive.**

We propose to use an octree as the primary data structure which allows us to represent the scene at multiple scales. Furthermore, it allows us to grow the reconstruction volume dynamically. As most space is either free or unknown, we allocate and update only those voxels that are located in a narrow band around the observed surface. In contrast to a regular grid, this approach saves enormous amounts of memory and computation time. The major challenge is to generate and maintain a consistent triangle mesh, as neighboring cells in the octree are more difficult to find and may have different resolutions. To remedy this, we present in this paper a novel algorithm that keeps track of these dependencies, and efficiently updates corresponding parts of the triangle mesh. In our experiments, we demonstrate the real-time capability on a large set of RGB-D sequences. As our approach does not require a GPU, it is well suited for applications on mobile or flying robots with limited computational resources.


(a) reconstructed 3D model of an office environment

(b) shaded close-up view (c) underlying bricks

Fig. 1. Our approach enables 3D reconstruction from RGB-D data in real-time on a CPU. We fuse the input images into a multi-resolution octree data structure that stores both the SDF and the triangle mesh. The top view shows a visualization of an office area. The bottom images show a close up of the printer area (red square), where the resolution varies adaptively between 5 mm and 2 cm.

I. Introduction

Reconstructing the geometry and texture of the world in real-time from a sequence of images remains one of the key-challenges in computer vision and robotics. For example, architects would greatly benefit from a wearable 3D scanning device that generates and visualizes a 3D model in real-time. Similarly, a robot navigating through an unknown environment benefits from an up-to-date 3D map to support obstacle avoidance, path planning, and autonomous exploration. This problem is known as Simultaneous Localization and Mapping (SLAM), where both the camera poses and the map have to be estimated at the same time. In this paper, we set the focus on the mapping task, which means that we assume that the camera poses are known.

Low-cost depth sensors such as the Microsoft Kinect have recently led to strong boost in this domain, because they help to overcome the scale ambiguity of monocular systems and, in contrast to photometric stereo systems, provide a quality depth map independent of the scene texture. While the first mapping approaches relied on classical feature matching [9], [7], recent results indicate that dense methods lead to more accurate reconstructions. In particular, the seminal KinectFusion paper [14] demonstrated that instantaneous dense 3D reconstructions with a hand-held sensor were possible by estimating a signed distance function (SDF) [5] on a GPU. However, when the SDF is represented as a regular grid, the memory consumption and the computation time grows cubically with the resolution, so that reconstruction resolutions seldom exceed 512^3 voxels. It has recently been shown that the memory consumption can be greatly reduced by using an adaptive representation such as octrees [21], [16]. However, due to the complexity of the data structures, these methods were not yet able to generate a triangle mesh in real-time.

The key insight behind our work is that a band-limited octree does not only reduce the memory consumption, but also greatly speeds up data fusion as significantly fewer voxels need to be updated. In particular, we found in our work that high resolution data fusion and meshing is feasible in real-time on a CPU, which enables 3D mapping on resource-limited mobile platforms without GPUs such as quadrotors. We prefer mesh extraction over a raycasting visualization because it enables us to compute view-independent visualization updates on a mobile platform, while the user is able to view the mesh on a base-station from every desired virtual camera pose. With a raycasting visualization, the mobile platform would have to render a new image for every camera pose and transmit this image to the base-station.

The contributions of this paper are:

1) an octree data structure that supports volumetric multi-resolution 3D mapping and meshing,
2) a speeded-up data fusion algorithm that runs at a resolution of up to 5 mm in real-time ($>$ 45 Hz on

All authors are with the Department of Computer Science, Technical University of Munich, Germany. This work has partially been supported by the DFG under contract number FO 180/17-1 in the Mapping on Demand (MOD) project. {steinbrf, sturmju, cremers}@in.tum.de

average) on a single CPU core,

3) a multi-resolution, incremental meshing algorithm that runs asynchronously on a second CPU core and outputs an up-to-date mesh at approximately 1 Hz.

In a large set of experiments, we evaluated the run-time behavior and memory consumption of our algorithm on real data. A video illustrating our approach is available at http://youtu.be/7s9JePSln-M.

II. Related Work

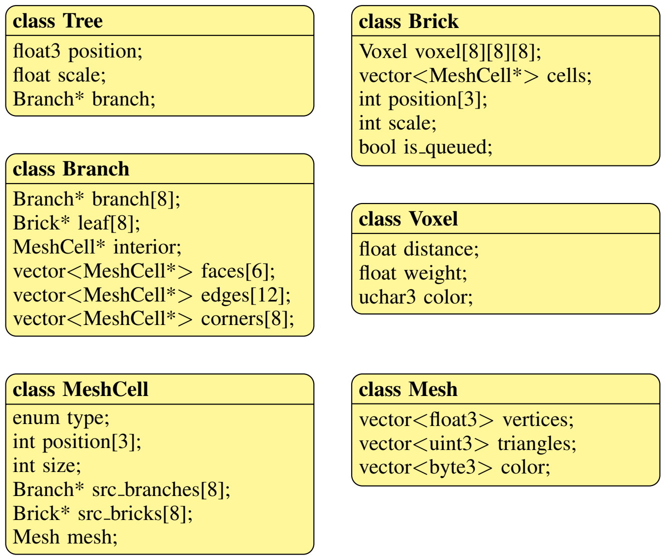
In this paper, we focus on the problem of mapping with known poses. This means that we assume that camera poses have already been determined by a visual odometry, Structure-from-Motion (SfM) or SLAM algorithm [3], [6], [12], [1], [11].

Previous approaches to the mapping problem can be categorized roughly into two classes: When the map is represented as a sparse set of visual features [3], [1], [9], [7] the map cannot easily be used for robot navigation, as it is difficult to distinguish free space from occupied space. Furthermore, removing outliers is hard and the semantics of regions with no data is unclear. To reduce memory consumption, point clouds can be often downsampled into surface patches or voxels [10], [20].

Alternatively, the map can be represented volumetrically, for example in the form of an occupancy grid map or a signed distance function (SDF). The advantage of this representations is that they inherently distinguish between known and unknown as well as free and occupied space. The disadvantage is a high memory demand and computational effort. As most computations can be parallelized and executed by a GPU, Newcombe et al. [14] demonstrated with KinectFusion that camera tracking and geometry fusion is possible in real-time.

To overcome the problem of memory limitation, several authors proposed shifting reconstruction volumes [15], [18], where older parts of the SDF are converted into a triangle mesh using Marching Cubes [13]. The difficulty then is that triangle meshes are hard to update after a loop-closure [19], which may lead to self-intersections and other inconsistencies. Therefore, it is desirable to represent the geometry as an SDF while more data is added, as it allows for arbitrary topology changes.

When the SDF is internally represented as an octree [20], the memory consumption can be greatly reduced by allocating only those cells that are located suffcently close to the surface. Fuhrmann et al. [8] proposed to store data not only at the leaves but at all levels of the tree. As a result, this representation allows for adaptive spatial resolutions and thus better reconstructions in regions with more data. This approach has also been used in computer graphics for efficient rendering of static data [4]. To the best of our knowledge, all currently existing implementations capable of real-time processing rely heavily on a GPU for parallelization [21], [16], [2], which renders these approaches unsuitable on resource constrained platforms such as quadrotors or smart phones. Furthermore, all of these works perform the final mesh extraction in a time-consuming offline step on the CPU, so that they not suited for live visualizations.



```
class Tree
float3 position;
float scale;
Branch* branch;

class Branch
Branch* branch[8];
Brick* leaf[8];
MeshCell* interior;
vector<MeshCell*> faces[6];
vector<MeshCell*> edges[12];
vector<MeshCell*> corners[8];

class MeshCell
enum type;
int position[3];
int size;
Branch* src_branches[8];
Brick* src_bricks[8];
Mesh mesh;

class Brick
Voxel voxel[8][8][8];
vector<MeshCell*> cells;
int position[3];
int scale;
bool is_queued;

class Voxel
float distance;
float weight;
uchar3 color;

class Mesh
vector<float3> vertices;
vector<uint3> triangles;
vector<byte3> color;
```

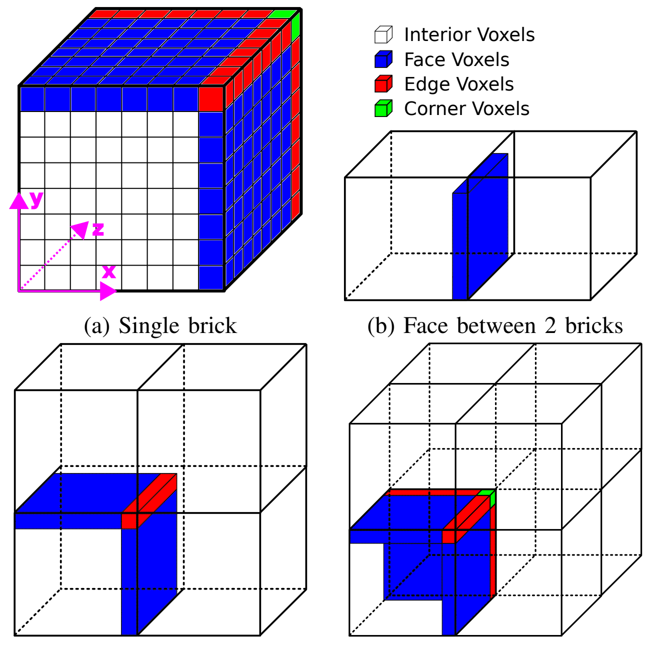
Fig. 2. Proposed data structures for the octree. We represent the 3D model internally as an octree. Every branch refers to up to 8 subbranches and 8 bricks. A brick contains 8^3 voxels. A mesh cell stores the triangle mesh of a particular region. Cross references (e.g., from a branch to its mesh cells and vice versa) enable fast traversal during data fusion and meshing.

This work is inspired by our recent finding [16] that data fusion in an octree runs extremely fast on a GPU (>200 Hz), so that real-time processing on CPU comes back into reach. Furthermore, as the number of updated voxels per RGB-D image is limited, we found likewise that meshing can be performed incrementally in real-time. Our resulting algorithm is capable of real-time mapping while requiring only two cores of a standard CPU. Our method fuses incoming RGB-D images in real-time at 45 Hz and outputs up-to-date triangle meshes at approximately 1 Hz at 5 mm resolution at the finest level.

III. Multi-Resolution Data Fusion in an Octree

We represent the geometry using a signed distance function (SDF), that provides any point the signed distance to the closest surface. As most space is either free or unknown, we represent the SDF in an octree, where only cells close to the surface are actually allocated. As different parts of the scene will be observed at different distances, we save geometry information at different levels in the tree (and thus at different resolutions). Instead of saving only a single SDF value at every node of the tree, we store the geometry in small cubic volumes, called "bricks", consisting of 8^3 voxels. A single voxel stores the truncated signed distance, the weight, and the color. Figure 2 shows a diagram of all relevant data structures of our approach.

Figure 3a shows a visualization of a single brick. Choosing a multiple of 8 for the size of the brick in every dimension, for the rest of the paper denoted by m, makes it easy to use Single-Instruction-Multiple-Data (SIMD) instructions such as SSE and AVX, supported by the vast majority of



(a) Single brick (b) Face between 2 bricks (c) Edge between 4 bricks (d) Corner between 8 bricks

Fig. 3. Upper boundary of a brick. For meshing, the blue face voxels need at least one other brick, the red edge voxels need at least three other bricks, and the green corner voxels need seven other bricks.

modern CPUs. While all bricks have the same size, they have different scales, depending on their depth in the tree. This accounts for the fact that geometry that is only seen from cameras far away can be stored in less detail than geometry seen from a camera up close. We denote the scale of a brick by $s_l \in \mathbb{N}$. Additionally, we store an integer offset $\mathbf{o}_l \in \mathbb{N}^3$ for every brick for its position in the tree. The tree itself is axis-aligned and has itself a floating-point position $\mathbf{o}_t \in \mathbb{R}^3$ and a scale $s_t \in \mathbb{R}_+$.

To fuse the geometry, we follow the approach of Levoy and Curless [5] that we adopted for our data structure as follows: First, we iterate over the pixels in the image, compute the 3D point for every pixel, and look-up its corresponding brick in the tree. We put all the bricks that contain a point of the current depth map into a queue. Second, we iterate over this queue and update the distance values in every brick. In the next two sections we will explain how we optimize these two steps for serial and SIMD processing on the CPU.

A. Traversing of the Tree for all Points of the Depth Map

We assume a pinhole camera with rectangular image plane $\Omega \subset \mathbb{R}^2$, scaled focal lengths $f_x, f_y \in \mathbb{R}_+$ and a central point $c = (c_x, c_y)^\top \in \Omega$. At time t, the camera records an RGB image $\mathcal{I}_t^c : \Omega \to \mathbb{R}_+^3$, and a depth image $\mathcal{Z}_t : \Omega \to \mathbb{R}_+$. Further, we assume that we know the camera translation $T_t \in \mathbb{R}^3$ and orientation $R_t \in \mathrm{SO}(3)$. Correspondingly, a pixel $(x, y)^\top$ on the image plane is related to the 3D point $\mathbf{p}_w \in \mathbb{R}^3$ (in world coordinates) by the formula

$$\mathbf{p}_w = R_t \cdot \begin{pmatrix} (x - c_x)\frac{\mathcal{Z}_t(x,y)}{f_x} \\ (y - c_y)\frac{\mathcal{Z}_t(x,y)}{f_y} \\ \mathcal{Z}_t(x,y) \end{pmatrix} + T_t \qquad (1)$$

Depending on the value of $\mathcal{Z}_t$, we also assign an (integer) scale $s_l := \exp_2 \lfloor \log_2 \max\{\mathcal{Z}_t, 1\} \rfloor$ to the point. Given the 3D world coordinates of a pixel and the desired scale of the brick, we look-up the corresponding brick in the tree and place it in a queue. Note that we update the brick at scale s_l and all (existing) coarser bricks above. The update weight for each voxel depends on the distance to the surface [5] and on the scale of the brick. Furthermore, in our implementation, the size of the truncation band grows with the scale of the brick, so that the updated band always contains the same number of voxels independent of the scale.

While this look-up step has a complexity of $O(\log n)$ operation (for a tree containing n bricks), we found that this step is costly enough that minimizing the amount of look-ups in the tree leads to a significant performance increase. Therefore, checking beforehand whether two or more points lie in the same brick can prevent us from performing unnecessary tree traversals. A fast heuristic is to check whether a point lies in the same brick with its left or upper neighbor in the image, and traverse the tree only if it does not. We evaluate the performance gain of this method in Section V-A.

B. Updating the Distance Function in Every Brick

Algorithm 1 shows a naïve approach to updating the SDF in the voxels [5]. To optimize this algorithm, we (1) make use of the serial order of iterations to reuse precomputed values and (2) get rid of branches in the code (i.e., all "if"-statements) to enable SIMD computations.

We first discuss the naïve algorithm. The individual steps are:

- an unordered - possibly parallel - iteration over all voxels in the brick volume (1),
- the transformation of the voxel position into camera coordinates (2),
- a check whether the transformed point lies in front of the camera and projects into the image domain (3),
- a check whether the depth image at the corresponding pixel position contains a valid depth value (5),
- the computation of the truncated distance value (6,7), and the computation of the incremental weight $w(\Delta_D)$ as well as the update of the voxel distance, weight and color values (8 to 10).

Of course, several of the values can be buffered in a trivial way. We omitted these steps here for sake of better readability, but used them in the naïve reference implementation used for runtime evaluation in Section V-A.

The SIMD-optimized algorithm is given in Algorithm 2. The first change is that we removed the computationally cumbersome point transformation: Instead of performing at least 9 floating-point multiplications and another 9 additions - not taking into account the transformation from brick- to tree-coordinates, we make use of the serialized processing order and reduce the transformation to a mere 3 additions in every iteration in lines 17 to 21.

The second step of optimization gets rid of the branches in lines 3 and 5 of Algorithm 1, and encodes them in a casted multiplication or bitwise **and** in line 12 of Algorithm

Algorithm 1 Naïve SDF update algorithm

```
1:  for $\mathbf{p}_l \in \{0, ..., m-1\}^3$ do
2:      $\mathbf{p}_c \leftarrow R_t^\top(\mathbf{o}_t + (\mathbf{o}_l + \mathbf{p}_l s_l)s_t) - R_t^\top T$
3:      if $[\mathbf{p}_c]_z > 0$ and $\pi(\mathbf{p}_c) \in \Omega$ then
4:          $z \leftarrow \mathcal{Z}_t(\pi(\mathbf{p}_c))$
5:          if $valid(z)$ then
6:              $\Delta_D \leftarrow |\mathbf{p}_c| \left(1 - \frac{z}{[\mathbf{p}_c]_z}\right)$
7:              $\Delta_D^\Phi \leftarrow \max\{\min\{\Delta_D, \Phi\}, -\Phi\}$
8:              $W(\mathbf{p}_l, t) \leftarrow w(\Delta_D) + W(\mathbf{p}_l, t-1)$
9:              $D(\mathbf{p}_l, t) \leftarrow \frac{D(\mathbf{p}_l,t-1)W(\mathbf{p}_l,t-1)+\Delta_D^\Phi w(\Delta_D)}{w(\Delta_D)+W(\mathbf{p}_l,t-1)}$
10:             $C(\mathbf{p}_l, t) \leftarrow \frac{C(\mathbf{p}_l,t-1)W(\mathbf{p}_l,t-1)+\mathcal{I}_t^c w(\Delta_D)}{w(\Delta_D)+W(\mathbf{p}_l,t-1)}$
11:         end if
12:     end if
13: end for
```

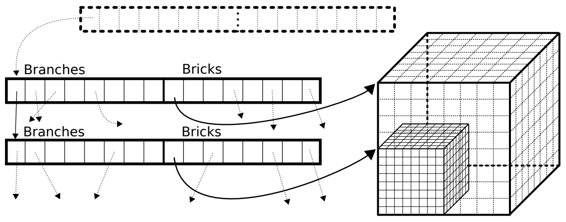
Algorithm 2 Serialized and SIMD-capable SDF update algorithm

```
1:  $\mathbf{p}_c^z \leftarrow s_t R_t^\top \mathbf{o}_t - R_t^\top T$
2:  for $z = 0$ to $m - 1$ do
3:      $\mathbf{p}_c^y \leftarrow \mathbf{p}_c^z$
4:      for $y = 0$ to $m - 1$ do
5:          $\mathbf{p}_c^x \leftarrow \mathbf{p}_c^y$
6:          for $x = 0$ to $\frac{m}{n_{\text{SIMD}}} - 1$ do
7:              for $k \in \{0, ..., n_{\text{SIMD}} - 1\}$ do
8:                  $\mathbf{p}_c \leftarrow \mathbf{p}_c^x + R[:, 1] k s_l s_t$
9:                  $z \leftarrow \mathcal{Z}_t(\pi_\Omega(\mathbf{p}_c))$
10:                 $\Delta_D \leftarrow |\mathbf{p}_c| \left(1 - \frac{z}{[\mathbf{p}_c]_z}\right)$
11:                 $\Delta_D^\Phi \leftarrow \max\{\min\{\Delta_D, \Phi\}, -\Phi\}$
12:                 $w_M \leftarrow w(\Delta_D) \cdot ([\mathbf{p}_c]_z > 0 \land \pi(\mathbf{p}_c) \in \Omega)$
13:                 $W(\mathbf{p}_l, t) \leftarrow w_M + W(\mathbf{p}_l, t-1)$
14:                 $D(\mathbf{p}_l, t) \leftarrow \frac{D(\mathbf{p}_l,t-1)W(\mathbf{p}_l,t-1)+\Delta_D^\Phi w_M}{w_M+W(\mathbf{p}_l,t-1)}$
15:                 $C(\mathbf{p}_l, t) \leftarrow \frac{C(\mathbf{p}_l,t-1)W(\mathbf{p}_l,t-1)+\mathcal{I}_t^c w_M}{w_M+W(\mathbf{p}_l,t-1)}$
16:             end for
17:             $\mathbf{p}_c^x \leftarrow \mathbf{p}_c^x + R[:, 1] s_l s_t n_{\text{SIMD}}$
18:         end for
19:         $\mathbf{p}_c^y \leftarrow \mathbf{p}_c^y + R[:, 2] s_l s_t$
20:     end for
21:     $\mathbf{p}_c^z \leftarrow \mathbf{p}_c^z + R[:, 3] s_l s_t$
22: end for
```

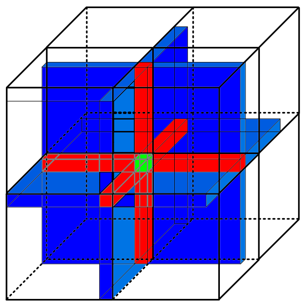
2. To avoid invalid memory access, we additionally have to clip the projected points to the image plane, denoted by the projection $\pi_\Omega(\mathbf{p}_c)$. The iteration over x is now split into a serial outer iteration and a possibly parallel inner SIMD iteration.

IV. Real-Time Meshing

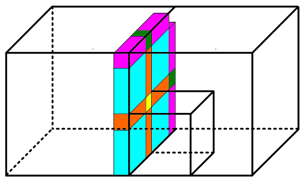
The geometry representation as an SDF is already useful for several robotic tasks such as path planning or obstacle avoidance. However, for visualization, map transmission, and multi-robot coordination, a map representation in form of a triangle mesh is highly beneficial, because it has a much lower memory footprint. In our implementation we used the Marching Cubes algorithm for zero-level extraction [13].



(a) Two bricks at the same position on different levels in the tree



(b) Inner boundary



(c) Meshes between branches

Fig. 4. Mesh extraction from a SDF stored on multiple scales. (a) Two bricks and their memory representation in the tree. (b)+(c) Boundary meshes created when a branch is subdivided.

While the Marching Cubes algorithm is simple to apply on a regular grid, it is more difficult to apply on our brick data structure as finding neighboring voxels at the boundary is more involved.

A. Marching Cubes

For meshing the voxel at position $(x, y, z)^\top \in \mathbb{N}^3$, we have to know the SDF values of this voxel as well as its 7 higher neighbor voxels $(x+1, y, z)^\top$, $(x, y+1, z)^\top$, $(x, y, z+1)^\top$, $(x+1, y+1, z)^\top$, $(x+1, y, z+1)^\top$, $(x, y+1, z+1)^\top$, and $(x+1, y+1, z+1)^\top$. For all lower voxels $v \in \{0, ..., m-2\}^3$ of the brick (depicted in white in Figure 3a), computing the mesh between v and its 7 higher neighbors is simple. However, for the voxels located on the three higher faces of the brick cube (depicted in blue), the voxels of at least one adjacent brick need to be known, for the voxels on the three high edges (depicted in red), we need the voxels of at least three other bricks and for the high corner voxel (depicted in green), we need the voxels of seven other bricks.

A naïve approach for the mesh extraction of all bricks would be to traverse the tree for all higher neigbors of a brick and compute the mesh between the voxels afterwards. However, this would require all bricks to have the same scale, which is not the case in our model, as one or more smaller bricks may lie inside a larger brick. In those parts where detailed geometry information from smaller bricks is present, a higher-resolution mesh should be computed, while the larger-scaled brick provides the rest of the mesh on a coarser resolution. An example of this case is shown in Figure 4a: While the smaller brick provides SDF values on a finer grid, the remaining $\frac{7}{8}$ of the branch has to be filled by the coarser-sampled SDF values of the larger brick.

Therefore, we abandon the notion of associating a mesh with a brick, and rather associate meshes with branches in

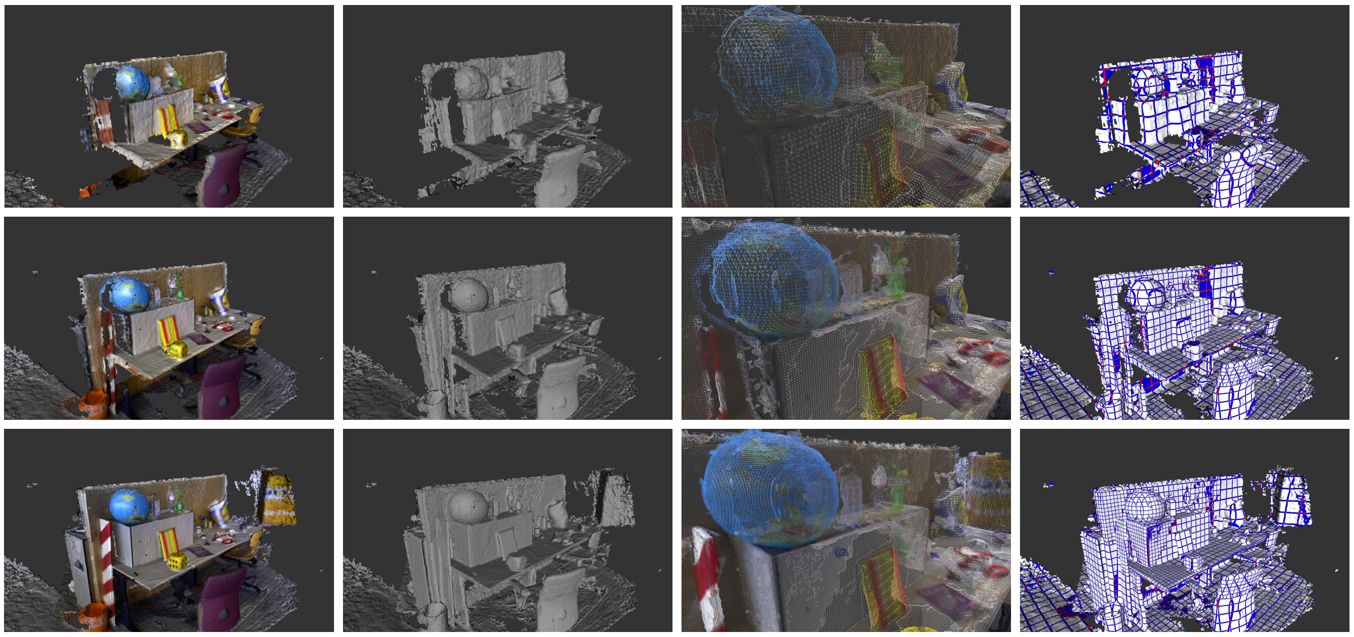


Fig. 5. Geometry evolution of the fr3/long_office_household sequence. Top to bottom: Result after the first 50, 170, and 300 frames. Left to right: Geometry and texture, geometry only, textured wireframe, and visualization of different mesh parts as in Figure 3. The level of detail in the geometry increases as the camera records more close-up images.

the tree (see again the data structures defined in Figure 2): A mesh cell represents either an interior mesh of one branch, a face mesh between two branches, an edge mesh between four branches, or a corner mesh between eight branches. It also contains the mesh generated for this region. We will discuss the reason for distinguishing between these four types of mesh cells in Section IV-B. A brick needs to be associated with the mesh cells belonging to the entire subtree below the branch on which it is located. Since we want to find all those mesh cells associated with a brick in $O(1)$ time, a brick stores references to all related mesh cells. To speed up traversal between mesh cells, bricks, and branches, a mesh cell stores references to all branches and bricks it belongs to.

Figure 4b illustrates the problem where a branch with a brick has at least one subbranch with a smaller brick. This spawns a large number of new mesh cells corresponding to faces (blue), edges (red) and corners (green) between the sub-branches: In particular, we have to compute the mesh of the interior of all existing subbranches as well as the interior of the non-existing subbranches with the SDF information of the larger brick. Additionally, we compute meshes for the 12 faces between the subbranches, 4 in each x-, y-, and z-direction, visualized in blue color in Figure 4b. We also compute meshes for 6 edges, 2 in each direction, between 4 subbranches respectively, visualized in red, and finally we compute a mesh for the middle corner of all 8 subbranches, visualized in green.

The last problem remaining is the fact that the mesh cells corresponding to these faces, edges, and the corner in Figure 4b need to be computed recursively as well, if one of their associated branches is subdivided. An example of this is demonstrated in Figure 4c: The subdivision of the branch on the right subdivides the face between the left and right branch. The face itself is divided into 4 new faces, and 4 edges as well as a middle corner are created between them. the edges encasing the face are divided into two edges and one corner each.

This leaves us with 4 nested recursive algorithms:

- If a branch contains a subbranch, the algorithm computing the mesh of the branch interior recurses into the subbranches and spawns the meshing of the faces, edges, and middle corner as described in the last paragraph and visualized in Figure 4b.
- If any of the 2 branches associated with a face contains a subbranch, the algorithm computing the mesh of the face recurses into the 4 subfaces and spawns the computation of the 4 edges and a middle corner between them. This is visualized in cyan, orange and yellow in Figure 4c.
- If any of the 4 branches associated with an edge contains a subbranch, the algorithm computing the mesh of the edge recurses into the 2 subedges and spawns the computation of the middle corner between them. This is visualized in pink and dark green in Figure 4c.
- If any of the 8 branches associated with a corner contains a subbranch, the algorithm recurses to the smaller size and the existing subbranches are passed to the recursive call.

As noted earlier, our octree data structure may contain overlapping bricks at different resolutions. For meshing, we only use the brick at the highest resolution. If the brick has missing values, for example, because only part of the brick was visible at the highest resolution, we fall back to a lower resolution brick for interpolation. In the future, it would be interesting to apply regularization over multiple scales prior to meshing to optimally exploit the available information.

When this algorithm is applied to the root of the tree, it produces a mesh of the entire geometry. While this can be desirable in some cases, in an online setting it has the disadvantage that the entire tree has to be traversed for the meshing of the geometry, which constitutes an $O(n)$ operation for every meshing step. For real-time capability, however, we need a complexity of $O(1)$.

The basic insight here is that a single depth image will only change the SDF in a small region of the tree in a reasonably large scene. Note that this number is typically even constant, as the sensor has a limited range and thus only a limited number of bricks can possibly get updated.

B. Reducing the Runtime Complexity for Incremental Meshing

To efficiently iterate only those mesh cells that need re-meshing, we keep track of the neighborhood of every branch in the tree when we allocate new branches and bricks.

Every time we subdivide a branch $\mathcal{B}$, we delete its interior mesh cell and add 8 new interior mesh cells for the children of $\mathcal{B}$. Analogously to Figure 4b, we add mesh cells for faces, edges and the middle corner in the interior of $\mathcal{B}$ as well.

For every outer face, edge and corner associated with $\mathcal{B}$ we look-up the other branches in the mesh cell structure. Concerning faces and edges, if $\mathcal{B}$ was larger than at least one other neighboring branch, it will be associated with an array of multiple face and edge cells, stemming from an earlier subdivision of a neighbor.

An example of this case is shown in Figure 4c. Let $\mathcal{B}$ be the large left branch. When we divide $\mathcal{B}$, we would pass each of the 4 face cells to one of the subbranches, pass each of the 4 orange edge cells to 2 subbranches of $\mathcal{B}$, and pass the yellow corner cell to 4 subbranches.

Whenever we add a new brick to the tree, we associate it with all the mesh cells of the branch and subbranches it lies on. For every brick queue containing references to the bricks that have been newly created or changed by the fusion of a depth map, we can now easily create a mesh cell queue that contains references to all the mesh cells that need to be updated.

In general, Marching Cubes takes more time than the data fusion, because it has to take into account neighborhood relations of the voxels. Therefore, we decided to run the meshing in a second thread parallel to the data fusion. Whenever the data fusion thread updates a brick, it pushes the affected mesh cells into a queue. We use a binary flag to indicate that a mesh cell has already been added to the queue to prevent adding the same cell twice. When a mesh cell has been processed, its "is_queued" flag is reset and it is removed from the queue.

In this way, the runtime complexity for the meshing is reduced from linear time $O(n)$ in the number of bricks to linear time in the number of bricks in the queue. Since the latter is bounded by a constant for typical camera trajectories, we end up with $O(1)$ complexity for the meshing operations, rendering our method real-time capable. For example, a quadrotor can incrementally stream the triangle mesh to a basestation, requiring a constant bandwidth. However, it should be noted that the total mesh grows with $O(n)$, and thus if the triangle mesh is visualized, for example, in OpenGL, it has to be copied, which has again a complexity of $O(n)$ (although with a small factor). The discrepancy between those two cases is visualized in Figure 6c.

Dataset	Bricks	Traversal Time [ms]		SDF Update Time [ms]		
		Naïve	Brick Check	Naïve	S.T	S.T. + SSE
fr1/360	905	22.2	14.0	22.9	18.9	7.3
fr1/desk	988	19.3	11.8	24.7	20.3	8.2
fr1/desk2	1058	20.6	12.8	26.4	21.7	8.8
fr1/plant	1121	19.6	12.8	28.0	23.0	9.1
fr1/room	1069	23.1	14.2	26.8	22.1	8.7
fr1/rpy	1046	21.7	13.6	26.1	21.9	8.5
fr1/teddy	1167	19.1	12.6	29.0	24.0	9.3
fr1/xyz	1054	18.7	11.5	26.8	22.0	8.2
fr2/desk	1327	20.7	13.1	33.7	27.7	10.8
fr3/office	1276	22.0	13.4	32.2	26.5	10.2
Average	1101	20.7	12.9	27.6	22.8	8.9

TABLE I

QUANTITATIVE EVALUATION OF THE PERFORMANCE GAINS OF VARIOUS OPTIMIZATIONS. THE VALUES ARE AVERAGED OVER ALL FRAMES OF THE SEQUENCE. THE OPTIMIZATION OF THE TREE TRAVERSAL YIELDS AN AVERAGE SPEEDUP OF 37%, THE OPTIMIZATION OF THE SDF UPDATE YIELDS AN AVERAGE SPEEDUP OF 67%.

V. EXPERIMENTAL EVALUATION

The goal of our experiments was to (1) evaluate the run-times of data fusion and meshing, (2) evaluate the asymptotic behavior of brick creation and brick updates, and (3) demonstrate the refinement of the geometry and texture in our multi-resolution representation as more images arrive.

A. Quantitative Evaluation of Performance Gains

To evaluate the run-time and assess the performance gains of our optimizations, we ran our method on several sequences of the TUM RGB-D benchmark [17]. The benchmark contains sequences with different camera trajectories in different scenes and provides time-synchronized camera poses obtained from a motion capture system.

We set the maximum resolution to 5 mm, and set the truncation/band size Φ to twice the voxel scale in every brick. We do not need to specify the reconstruction volume, as our implementation supports infinite grow of the tree by inserting additional branches at the top. For most sequences, the final tree had a depth of 10, yielding a volume of 8192^3 voxels on the fines resolution.

We measured the timings on a Laptop-CPU of type Intel Core i7-2720QM with 2.2 GHz and 8 GB RAM. For all sequences, we measured the time of the first thread (running on a single CPU core) to traverse the tree to allocate and queue branches, bricks, and mesh cells in the tree, as well as the time to update the SDF in the voxels of the queued bricks. Table I shows the results. On average, 1101 bricks get updated per RGB-D frame. For traversing and queuing bricks, the naïve algorithm required an average of 20.7 ms, while our optimized version required 12.9 ms, corresponding to a speedup of 37%. Updating the SDF took another 27.6 ms for the naïve implementation, but only 8.9 ms after optimization using the serialized transform and the SSE-SIMD instructions. This corresponds to a speedup of 67%. In total, the optimized fusion algorithm requires on average 21.8 ms per RGB-D frame, corresponding to an average processing speed of 45 Hz.

To study the sequential behavior of our algorithm in more detail, we also evaluated the computational load over time. Figure 6 shows the result for the fr1/teddy sequence.

At the top, we plotted the total time needed per RGB-D frame for the traversal and the SDF update. As can be seen, the processing time stays below 33 ms for almost all frames.

Beneath we show how the number of newly created branches, bricks, and mesh cells varies over time, depending on the camera motion and the amount of newly discovered geometry. A peak in these values is due to the fact, that the camera visits “unknown territory” at this time. In contrast, the number of updated bricks per RGB-D frame remains more or less constant around 1,100, which is closely related to the computation time for that frame.

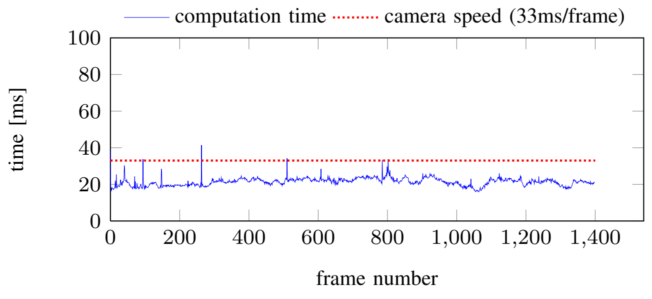
At the bottom of Figure 6 we show the processing time of the meshing queue, and thus the latency at which the updated mesh becomes available. The latency varies between 0.5 s to 1.5 s, while final mesh merging (for visualization in OpenGL) grows monotonically to 0.1 s after 1,400 frames. The final mesh consists of 3314765 triangles on 2786330 vertices.

B. *Qualitative Reconstruction Results*

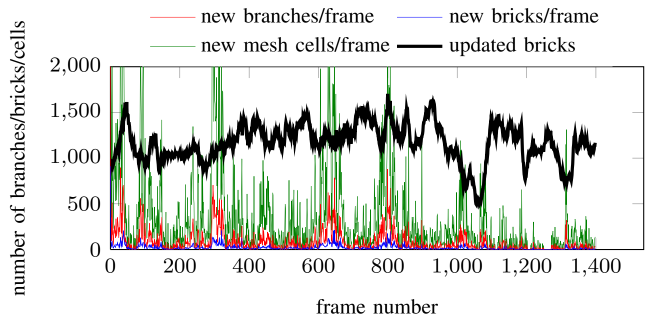
Figure 1 and Figure 5 present 3D reconstructions obtained with our method. Figure 1 demonstrates that, due to the sparse multi-level representation of the geometry, we are able to reconstruct large scenes while preserving fine details. The reconstructed environment spans approximately 25m x 10m, consists of 154273 bricks and consumes approximately 1.4 GB in memory. Figure 5 shows a sequence of the reconstructed geometry on the fr3/office sequence [17] after integrating 50, 170, and 300 frames. As can be seen, the level of detail increases significantly from column to column while the camera passes the geometry at close range.

The right column of Figure 5 and Figure 1c show a visualization of the different mesh cells, colored the same way as in Figure 3: White corresponds to interior mesh cells, blue to face cells, red to edge cells, and green to corner cells. At the same time, this coloring provides an intuitive visualization of the underlying bricks stored at multiple resolutions.

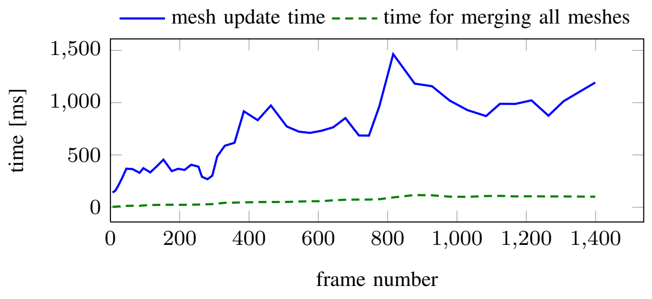
Furthermore, we provide a video demonstrating our approach on the fr3/office sequence at http://youtu.be/7s9JePSln-M. The video was recorded in real-time using a screen capture tool and thus illustrates



(a) Computation time for data fusion per frame



(b) Number of branch/brick/cell updates per frame



(c) Computation time (latency) for incremental meshing

Fig. 6. Evaluation of the runtime on the fr1/teddy sequence. (a) computation time for the geometry fusion of every frame, compared to the camera frame-rate. (b) amount of updated bricks for every frame as well as the number of newly created branches, bricks, and mesh cells. (c) Evaluation of the runtime for meshing (corresponding to display latency).

the real-time capability of our approach. While the geometry fusion runs at approximately 45 Hz, the mesh is updated at approximately 1 Hz. Similar to the figures, the video shows the incremental meshing in different visualizations, as color mesh cells, as a wire frame, and as a shaded 3D model.

VI. Conclusion and Outlook on Future Work

In this paper, we presented a novel approach that enables multi-resolution online geometry fusion on a standard CPU in real-time. We proposed an efficient octree data structure that allows for fast SDF updates and incremental meshing. In our experiments, we demonstrated that our system is capable of performing the geometry fusion easily in real-time, rendering it practical to use it for volumetric mapping on a resource-constrained platform such as a quadrotor. In the future, we plan to implement camera tracking based on the computed map, with the goal to assist or replace an external

SLAM system. Furthermore, it would be interesting to use the voxel neighborhood information provided by the mesh cells to perform efficient regularization of the SDF. We plan to make our code publicly available soon.

References

[1] S. Agarwal, N. Snavely, I. Simon, S. Seitz, and R.Szeliski. Building rome in a day. In *ICCV*, 2009.

[2] J. Chen, D. Bautembach, and S. Izadi. Scalable real-time volumetric surface reconstruction. In *SIGGRAPH*, 2013.

[3] A. Chiuso, P. Favaro, H. Jin, and S. Soatto. 3-D motion and structure from 2-D motion causally integrated over time: Implementation. In *ECCV*, 2000.

[4] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *SIGGRAPH*, 2009.

[5] B. Curless and M. Levoy. A volumetric method for building complex models from range images. In *SIGGRAPH*, 1996.

[6] A. Davison. Real-time simultaneous localisation and mapping with a single camera. In *Proc. of the IEEE Intl. Conf. on Computer Vision (ICCV)*, 2003.

[7] F. Endres, J. Hess, N. Engelhard, J. Sturm, D. Cremers, and W. Burgard. An evaluation of the RGB-D SLAM system. 2012.

[8] S. Fuhrmann and M. Goesele. Fusion of depth maps with multiple scales. *ACM Trans. Graph.*, 30(6):148, 2011.

[9] P. Henry, M. Krainin, E. Herbst, X. Ren, and D. Fox. RGB-D mapping: Using depth cameras for dense 3D modeling of indoor environments. 2010.

[10] P. Henry, M. Krainin, E. Herbst, X. Ren, and D. Fox. RGB-D mapping: Using depth cameras for dense 3D modeling of indoor environments. In *Intl. Symp. on Experimental Robotics (ISER)*, 2010.

[11] C. Kerl, J. Sturm, and D. Cremers. Dense visual slam for rgb-d cameras. In *Proc. of the Int. Conf. on Intelligent Robot Systems (IROS)*, 2013.

[12] G. Klein and D. Murray. Parallel tracking and mapping for small AR workspaces. In *IEEE and ACM Intl. Symp. on Mixed and Augmented Reality (ISMAR)*, 2007.

[13] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics*, 21(4):163–169, 1987.

[14] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, and A. Fitzgibbon. KinectFusion: Real-time dense surface mapping and tracking. In *ISMAR*, 2011.

[15] H. Roth and M. Vona. Moving volume KinectFusion. In *BMVC*, 2012.

[16] F. Steinbrücker, C. Kerl, J. Sturm, and D. Cremers. Large-scale multi-resolution surface reconstruction from RGB-D sequences. In *ICCV*, 2013.

[17] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers. A benchmark for the evaluation of RGB-D SLAM systems. In *IROS*, 2012.

[18] T. Whelan, H. Johannsson, M. Kaess, J. Leonard, and J. McDonald. Robust real-time visual odometry for dense RGB-D mapping. In *ICRA*, Karlsruhe, Germany, 2013.

[19] T. Whelan, M. Kaess, J. Leonard, and J. McDonald. Deformation-based loop closure for large scale dense RGB-D SLAM. In *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems, IROS*, Tokyo, Japan, Nov 2013. Accepted. To appear.

[20] K. Wurm, A. Hornung, M. Bennewitz, C. Stachniss, and W. Burgard. OctoMap: A probabilistic, flexible, and compact 3D map representation for robotic systems. In *Workshop on Best Practice in 3D Perception and Modeling for Mobile Manipulation at ICRA*, 2010.

[21] M. Zeng, F. Zhao, J. Zheng, and X. Liu. A Memory-Efficient KinectFusion using Octree. In *Computational Visual Media*, volume 7633 of *Lecture Notes in Computer Science*, pages 234–241. Springer Berlin Heidelberg, 2012.