SurfelMeshing: Online Surfel-Based Mesh Reconstruction

Thomas Schöps, Torsten Sattler, and Marc Pollefeys

Abstract—We address the problem of mesh reconstruction from live RGB-D video, assuming a calibrated camera and poses provided externally (*e.g.*, by a SLAM system). In contrast to most existing approaches, we do not fuse depth measurements in a volume but in a dense surfel cloud. We asynchronously (re)triangulate the smoothed surfels to reconstruct a surface mesh. This novel approach enables to maintain a dense surface representation of the scene during SLAM which can quickly adapt to loop closures. This is possible by deforming the surfel cloud and asynchronously remeshing the surface where necessary. The surfel-based representation also naturally supports strongly varying scan resolution. In particular, it reconstructs colors at the input camera's resolution. Moreover, in contrast to many volumetric approaches, ours can reconstruct thin objects since objects do not need to enclose a volume. We demonstrate our approach in a number of experiments, showing that it produces reconstructions that are competitive with the state-of-the-art, and we discuss its advantages and limitations. The algorithm (excluding loop closure functionality) is available as open source at https://github.com/puzzlepaint/surfelmeshing.

surfel 重建支持多尺度

Index Terms—3D Modeling and Scene Reconstruction, RGB-D SLAM, Real-Time Dense Mapping, Applications of RGB-D Vision, Depth Fusion, Loop Closure, Surfels.

1 Introduction

The availability of fast programmable GPUs has enabled handling the massive amount of RGB-D data generated by depth sensors such as the Kinect and stereo algorithms in real-time. This has led to dense Simultaneous Localization and Mapping (SLAM) systems [1], [2], online 3D scene perception approaches [3], [4], and applications such as Augmented Reality [5], [6] building on top of SLAM and dense mapping.

A central question in the context of SLAM and online dense mapping is the choice of scene representation. On the one side, applications such as collision detection in physics simulations [7], occlusion handling in AR applications [5], [8], and path planning for autonomous navigation [9], [10], [11] benefit from *continuous surface* representations, *e.g.*, meshes. On the other side, scene representations need to be *flexible* to handle spatial deformations caused by loop closure events in SLAM. It is important that they can be *updated efficiently* if new data becomes available to allow for real-time processing. Ideally, a scene representation should also be *adaptive* in terms of resolution. That is, it should be able to model both large structures and small or thin objects without too much memory overhead.

通过自适应调节分辨率实现小物体重建

The predominant scene representation for RGB-D SLAM and real-time dense 3D reconstruction is voxel-based [12]. Each voxel in the volume stores the truncated signed distance to the closest surface, which can be updated very efficiently in parallel. A continuous surface representation in the form of a mesh can then be extracted from this volume with the Marching Cubes algorithm [13]. However, volume-based methods are not very flexible and handling loop closures during online operation is expensive since accurate compensation can imply changing the whole volume. At the same time, the resolution of the voxel volume is often fixed in practice for efficiency reasons [2], [14], [15], limiting the adaptiveness of the representation (*c.f.* Fig. 2, left). In addition, thin objects cannot be reconstructed if they are small or thin with respect to the voxel size.

指出TSDF的缺点是分辨率固定

Another popular approach is to represent the scene with a set of points or *surfels* [16], [17], *i.e.*, oriented discs (*c.f.* Fig. 2, middle right). The points or surfels are optimized during SLAM. This scene representation is flexible and coordinates can be updated very efficiently. It is also highly adaptive as measurements at a higher resolution lead to denser point resp. surfel clouds, and it easily handles thin objects. The main drawback of this representation is its discrete nature, which can be resolved by meshing. However, global methods such as Delaunay triangulation [18], [19], [20] or Poisson reconstruction [21] are too computationally expensive to run in real-time on the dense point clouds generated by RGB-D sensors. In contrast, efficient local meshing methods [22], [23] are very sensitive to noise, leading to both noisy surface reconstructions and holes in the meshes.

指出重建面临的问题

In this paper, we show that online meshing of reconstructed dense surfel clouds is possible through denoising and adaptive remeshing (*c.f.* Fig. 1). Our algorithm is a novel combination of surfel reconstruction [17], [24] (Sec. 3.1) with a fast point set triangulation method [22], [23] (Sec. 3.3). The key step for enabling local triangulation is a novel surfel denoising scheme (Sec. 3.2) to handle the noisy input data (*c.f.* Fig. 7). For efficiency, flexibility, and adaptability, we introduce a remeshing algorithm (Sec. 3.4) that recomputes the mesh locally if required. In contrast to volume-based methods, our approach easily adapts to loop closures and leads to higher resolution models (*c.f.* Fig. 2) while automatically handling thin objects (*c.f.* Fig. 14). In

- *The authors are with the Department of Computer Science, ETH Zurich, Switzerland.*
  E-mail: firstname.lastname@inf.ethz.ch
- *Marc Pollefeys is additionally with Microsoft, Zurich.*

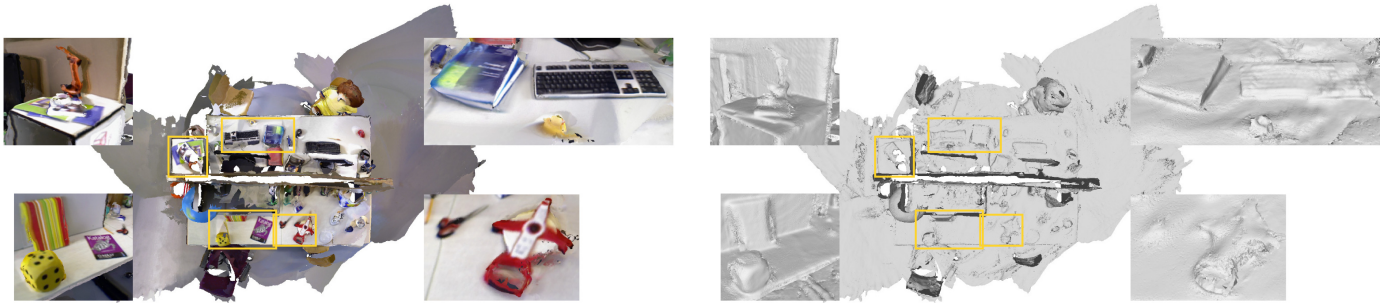


Fig. 1. Triangle mesh reconstructed with our method with loop closures handled during reconstruction, colored (left) and shaded.

contrast to point resp. surfel cloud-based approaches, our method reconstructs high-resolution meshes during online operation. To the best of our knowledge, our is the first approach to provide a continuous surface reconstruction while being flexible, adaptive, and efficient to update.

2 Related Work

This section discusses related work in the area of online triangle mesh reconstruction. We focus on methods that create consistent scene models. We exclude offline methods as the focus of our work is on (real-time) online operation.

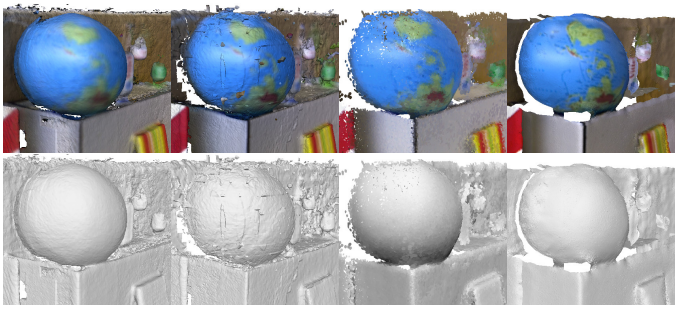
Voxel volume-based methods build on truncated signed distance function (TSDF) fusion [12]. Popularized by KinectFusion [1], many following works focused on scaling this approach to larger scenes [25], [26], adding multi-resolution capability [27], [28], or improving efficiency [29], [30]. Mesh extraction typically runs at a lower frame rate compared to TSDF fusion. In contrast to our method, these works do not handle loop closures during reconstruction.

再次强调了Kinect fusion类方法的缺陷

Kintinuous [2] uses a volume that moves with the camera. Scene parts leaving this volume are triangulated [23], and loops are closed by deforming the mesh. Compared to our method, [2] does not merge meshes after loop closures, resulting in multiple (potentially inconsistent) meshes per surface. Kähler *et al*. [14] use many small subvolumes that move independently to handle loop closures, combining overlapping subvolumes by weighted averaging during rendering. Yet, subvolumes are only aligned rigidly, which can lead to inconsistencies. BundleFusion [15] addresses loop closures by de- and re-integrating old RGB-D frames when they move, enabling it to correct all drift. It is computationally expensive, requiring two GPUs for real-time operation, and does not scale well as it has to keep all frames in memory. Using keyframes reduces the computational demand [31], [32]. However, data can be lost if it cannot be integrated into a keyframe and the reconstruction quality can degrade if data is fused with keyframes early. Our method is more efficient than [15] while still using all the data. In addition, it handles varying scan resolution naturally and can reconstruct thin objects.

再k! p¹Qω, e‡e¹lÕv„O RØ

Delaunay tetrahedralization of a point cloud is a way to discretize 3D space. Tetrahedra can be classified as 'inside' or 'outside' of objects and the surface can be extracted as the interface between these classes. [18] uses this approach while being able to incrementally add points to the model.



InfiniTAM [29] FastFusion [27] ElasticFusion [17] Ours

Fig. 2. Left to right: voxel-based meshes by InfiniTAM [29] and FastFusion [27] (each with default voxel size settings), surfel splat rendering by ElasticFusion [17], our surfel-based mesh. The same camera trajectory is used with all approaches. Notice the sharper colors produced by our method.

[19], [20] extend [18] to also allow for moving existing points. For performance reasons, these methods are typically only applied to sparse (SLAM) point clouds. Only [20] runs in real-time in this setting. In contrast to our method, these methods thus cannot handle dense surfel clouds in real-time.

Surfel-based methods represent the scene as a set of surfels [16]. MRSMap [33] stores multi-resolution surfel maps in octrees. It handles loop closures via pose graph optimization and generates a consistent map only afterwards. Weise *et al*. [34] present a system for scanning small objects. Loop closures are handled in real-time by deforming the surfel cloud using a sparse deformation graph. Another line of work [24], [35], [36] addresses high-quality surfel-based tracking and mapping, but does not address the loop closure problem. ElasticFusion [17] extends [24] with real-time loop closure handling similar to [34]. Gao and Tedrake [37] present a surfel-based approach for reconstructing dynamic objects, but note the lack of real-time mesh reconstruction as a drawback. Yan *et al*. [38] propose a probabilistic surfel map representation for SLAM. They reconstruct a mesh from deformed keyframe depth maps as a post-processing step. Each depth map is handled independently, potentially resulting in multiple meshes per surface. In contrast to these previous works, our approach is able to construct meshes during online operation.

这些都是简单叠加吗，为什么存在多个mesh?

Meshing approaches. Bodenmüller [39] presents a streaming mesh reconstruction method for point clouds, but does

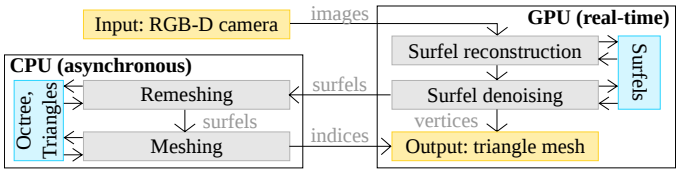


Fig. 3. Data flow overview for our algorithm.

not address noisy consumer depth cameras or loop closures. Schertler *et al*. [40] incrementally reconstruct high-quality meshes from point clouds. In contrast to our approach, their method is not suitable for real-time use on high-frequency input such as RGB-D video. Ladický *et al*. [41] learn to convert point clouds to SDF volumes for meshing using Marching Cubes. Using the same GPU as used in this work, their approach handles about 300,000 points in real-time. For comparison, each of our input depth map already contains this number of points. In contrast to ours, [41] uses a fixed voxel size, limiting the mesh resolution. Zienkiewicz *et al*. [42] fit a pre-defined triangular mesh surface model to the observations. They explicitly handle varying observation scale but only demonstrate their method on 2.5D height maps. In contrast, our approach handles both varying scale and general scenes.

3 Surfel-Based Mesh Reconstruction

The input to our algorithm is an RGB-D video stream from a calibrated camera and camera poses provided by a SLAM system such as [17]. Our algorithm consists of four main parts (*c.f*. Fig. 3): surfel reconstruction (adapted from [17], [24]), surfel denoising (new contribution), meshing (adapted from [22], [23]), and remeshing (new contribution). Surfel reconstruction and denoising reconstruct a surfel cloud at the frame rate of the video input. Meshing and remeshing create and maintain a triangulation of this surfel cloud. These components run asynchronously in the background (similar to asynchronous mesh extraction in volume-based methods [30]). The output is a mesh whose vertices correspond to the surfels (updated at frame rate) and whose topology is determined by the triangulation (updated at a lower rate). We describe each component in detail below.

3.1 Surfel Reconstruction

Our algorithm triangulates a surfel cloud. To reconstruct this cloud, we slightly adapt [17], [24]. The following outlines the relevant parts of this approach including our modifications. Sec. 3.2 presents our extensions for surfel denoising.

From the input stream of RGB-D images, we construct and maintain an unordered set of surfels covering the visible surfaces. Once a new frame becomes available, a data association step decides for each new depth measurement, *i.e.*, each pixel having a depth value, whether it creates a new surfel or is used to refine existing surfels. In the event of a loop closure, the surfel cloud is deformed to align matching surface parts.

Surfel representation. Each surfel s comprises a 3D position $\mathbf{p}_s$, a normal $\mathbf{n}_s$, an RGB color $\mathbf{c}_s$, a confidence score σ_s, a radius r_s used for neighbor search, a creation timestamp $t_{s,0}$, and a latest update timestamp t_s.

Data association. Similar to [24], we project the surfels into each new RGB-D image to determine which depth measurements should be associated with existing surfels. [24] creates a super-sampled index map rendering of all surfels, which is however still limited in resolution. We improve upon this by always taking surfel indices directly from the result of the projection computation, thus avoiding to store surfel indices in a map with limited space. The projection is repeated in each step where these indices are required.

For data association, each surfel is tested against the pixel it projects to, as well as the neighboring pixel that is closest to the projected sub-pixel position, and can potentially be associated to both. We use a simple model of the measurement uncertainty which defines the uncertainty range of a measurement with depth z as the depth interval $[(1-\gamma)z, (1+\gamma)z]$, with $\gamma = 0.05$. Both the sensor depth uncertainty and small pose uncertainties should be accounted for here. We found that this simple model worked well in the scenarios tested, however, more sophisticated models (which ideally tightly model the uncertainty of poses and 3D structure estimated by the underlying SLAM system) could be easily substituted.

Based on this model, we classify each surfel to be either *conflicting* with a depth measurement it projects to, to be *occluded* by it, or *supported* by it. A surfel *conflicts* with the measurement if it projects in front of the measurement uncertainty range, thus violating the constraint that the space between the measurement and the camera must be free. A surfel is *occluded* by the measurement if it is not conflicting, and either projects behind the uncertainty depth range, or its normal points away from the camera or differs too strongly from the measurement normal. The remaining surfels are *supported by* the measurement.

Measurement integration. As in [24], we create a new surfel for each pixel not associated with any conflicting or supported surfel: We initialize the surfel position $\mathbf{p}_s$ to the un-projected position of the pixel and estimate its normal $\mathbf{n}_s$ via finite differences in the depth image. The pixel color is used as the surfel color $\mathbf{c}_s$, the confidence σ_s is set to 1, and both timestamps are set to the current timestamp. In our method, the surfel radius r_s represents the expected maximum distance to neighboring surfels on the same surface. That is, r_s is the maximum distance of the surfel to all surfels it should later be connected to during meshing. For a depth measurement at pixel (x, y), we compute r_s as

$$r_s = 1.5 \cdot \max_{u,v \in \{-1,0,1\}} ||p(x,y) - p(x+u, y+v)||_2 \ , \quad (1)$$

where $p(x, y)$ is the 3D point corresponding to pixel (x, y). Intuitively, we chose the radius r_s to contain the direct neighbors in image space, providing us with an estimate how far away neighboring surfels can be in 3D space. To avoid noisy radii, we only use pixels for which all neighbors in the 8-neighborhood have valid depth measurements.

Differing from [24], we integrate a depth measurement into *all* surfels that are supported by it to obtain an as-smooth-as-possible reconstruction. Integration of a measurement into a surfel is performed as in [24]: We compute

累加式

the weighted averages of $\mathbf{p}_s$, $\mathbf{n}_s$, and $\mathbf{c}_s$ with the measured values $\mathbf{p}_m$, $\mathbf{n}_m$, and $\mathbf{c}_m$, and update the timestamp t_s and confidence σ_s:

$$\text{With } f(\mathbf{v}_s, \mathbf{v}_m) = \frac{\sigma_s \mathbf{v}_s + w \mathbf{v}_m}{\sigma_s + w} : \begin{array}{l} \mathbf{p}_s := f(\mathbf{p}_s, \mathbf{p}_m), \\ \hat{\mathbf{n}}_s := f(\mathbf{n}_s, \mathbf{n}_m), \\ \mathbf{c}_s := f(\mathbf{c}_s, \mathbf{c}_m). \end{array} \quad (2)$$

$$\mathbf{n}_s := \frac{\hat{\mathbf{n}}_s}{|\mathbf{n}_s|} \quad (3) \qquad \sigma_s := \min\{\sigma_s + w, \sigma_{\max}\} \quad (4)$$

The weight w is computed as $\frac{1}{|S_m|}$, with S_m being the set of all surfels supported by the measurement. We clamp the surfel confidence at a low maximum value $\sigma_{\max} = 5$ such that the weight of new measurements stays high. This allows existing surfaces and new surfels within them to quickly converge to similar attribute values. If computing Eq. 1 on the current image leads to a smaller radius, we update the radius r_s to this smaller value. Nearby surfels having very similar attributes are merged as in [24].

We decrease the confidence of conflicting surfels by one. Once the confidence of a conflicting surfel reaches zero, it is replaced with the new measurement it conflicts with, as described above for the creation of new surfels.

Loop closure handling. We use the loop closure handling procedure from [17]: We use an active window and only consider *active* surfels whose last update timestamp t_s is not too old for data assignment and integration. In the event of a loop closure, the surfel cloud is deformed based on the surfel timestamps and a new position and normal is computed for each surfel. Surfels outside the active window that are consistent with active surfels after the loop closure are re-activated. We refer to [17] for details. Regarding meshing, surfel movements due to loop closures are handled exactly like surfel movements due to integrating new measurements, with the exception of a small performance optimization mentioned in Sec. 3.4.

3.2 Surfel Denoising

There are various sources of noise such as depth camera noise, camera calibration, and pose inaccuracies. Thus, independently reconstructed surfels are usually noisy, despite the averaging used in measurement integration. This in particular applies to new surfels, causing problems as triangulation is sensitive to noise. We thus introduce a regularization step. Furthermore, discontinuities are likely to arise at the boundaries of the integrated depth images. We address them with a blending step. This section presents these two contributions.

Regularization. We extend the surfel structure with a denoised position $\bar{\mathbf{p}}_s$ (initialized to $\mathbf{p}_s$), indices N_s of four neighboring surfels, and temporary storage for accumulating cost gradients. Our proposed regularization works across transitive neighbors: each surfel's denoised position depends on the denoised positions of its neighbors, which again depend on the neighbors' neighbors, and so on. Thus, we do not require N_s to contain the closest neighbors. This enables us to use a simple neighborhood selection scheme suitable for GPU processing: During data association (Sec. 3.1), we create an index image which for each pixel references one random supported surfel (to keep computation time low). Due to the transitivity, this random choice is not significant. Afterwards, for each active supported surfel s, we examine the 4-neighborhood of the pixel it projects to. We update N_s to contain the up to four closest neighbors from both the surfel's existing neighbors N_s and the supported surfels assigned to the pixels in the 4-neighborhood. Neighbors farther away from $\mathbf{p}_s$ than $2 \cdot r_s$ are not considered.

Our denoising approach then optimizes a cost C, which contains a residual for each surfel s in the surfel cloud S:

$$C(S) = \sum_{s \in S} r_{\text{data}}(s) + w_{\text{reg}} r_{\text{reg}}(s) \ . \quad (5)$$

The cost consists of a data term r_{data} and regularizer r_{reg}. We use a weighting factor w_{reg} to weigh the two terms. It depends on the amount of noise caused by uncertainties in the camera measurements and calibration, pose estimates, *etc.*, and we empirically set it to 10 to prioritize smoothness. The data term keeps the denoised surfel position $\bar{\mathbf{p}}_s$ close to the position $\mathbf{p}_s$ where it was measured:

$$r_{\text{data}}(s) = ||\bar{\mathbf{p}}_s - \mathbf{p}_s||_2^2 \ . \quad (6)$$

The regularizer moves the surfel close to its neighbors N_s, measured along the surfel's normal direction $\mathbf{n}_s$:

$$r_{\text{reg}}(s) = \frac{1}{|N_s|} \sum_{n \in N_s} \left(\mathbf{n}_s^T (\bar{\mathbf{p}}_n - \bar{\mathbf{p}}_s) \right)^2 \ . \quad (7)$$

Notice that this formulation is different from basic Laplacian smoothing in that it applies a clearly defined smoothing strength (controlled by w_{reg}) to the surfels upon convergence, and it also takes the surfel normals into account.

After each iteration of surfel reconstruction, we run one iteration of gradient descent to minimize the cost C. We choose the descent step length individually for each surfel s as follows, which is empirically normalized to a relative length of 0.5 to obtain good convergence:

$$0.5 \cdot \left(1 + w_{\text{reg}} + \sum_{i \in S | s \in N_i} \frac{w_{\text{reg}}}{|N_i|} \right)^{-1} \ . \quad (8)$$

This step size normalizes the surfel's gradient length by the total residual weight it receives. The data term contributes weight 1, the regularization term contributes weight w_{reg}, and the third term in the sum arises from the surfel acting as a neighbor n in the regularization term of other surfels.

For performance reasons, we use an active window for denoising: We only update surfels whose latest update timestamp is within the last 30 frames. Other surfels usually do not move significantly anymore since they have been unobserved for a while and thus the optimization has nearly converged.

After an optimization iteration, we update the vertices of the output mesh by assigning the new surfel positions to them. For surfels replaced by a conflicting measurement since the last triangulation iteration, we remove all incident triangles until the next triangulation update to avoid artifacts.

We also use the surfel neighbors for smoother deformations on loop closures: Instead of directly adding the

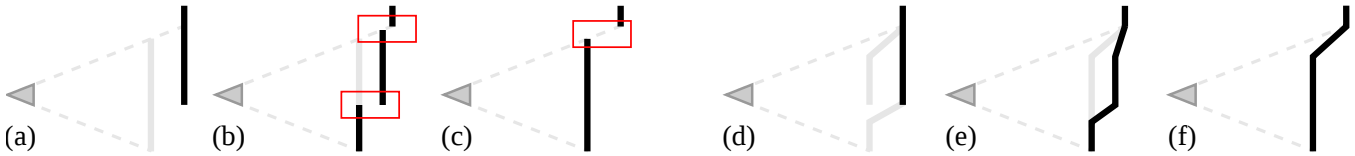


Fig. 4. Illustration of raw depth map integration, (a) - (c), vs. observation boundary blending, (d) - (f). (a) The camera observes an existing surface (black) in a different position (gray), *e.g.*, due to slight pose drift. (b) After integrating one new observation of the gray surface, discontinuities (red) are created. (c) After integrating many new observations, there is still a discontinuity, causing a hole in the mesh. (d) Observation hallucinated by Alg. 1 based on (a). (e) After integrating one new observation of the gray surface, no discontinuities are created. (f) After integrating many new observations, blending still avoids discontinuities. In contrast, using a lower integration weight for boundary regions would lead to situation (c).

computed position offset $\delta\mathbf{p}_s$ to $\bar{\mathbf{p}}_s$ and $\mathbf{p}_s$, we first average it among the neighbors for 100 iterations (chosen to be large while not causing a too strong performance hit): $\delta\mathbf{p}_s := \frac{1}{|N_s|}\sum_{n\in N_s}\delta\mathbf{p}_n$. This yields more consistent offsets among neighbors, thus existing surfaces are less likely to rip apart.

Observation boundary blending. Boundaries between the observed and currently unobserved parts of the scene are likely to cause surface discontinuities. The observed parts will be updated to match any depth bias which is currently present (*e.g.* due to drift) while the unobserved parts remain constant (*c.f.* Fig. 4 (a) - (c)). This is a general issue arising from noisy input and affects all 3D scene representations: For volume-based methods it creates step artifacts in the reconstructed surface. For surfel-based methods, it additionally creates holes in the reconstruction. Thus, we address this to reduce undesirable hole artifacts: We hallucinate a smooth transition from the measured depth to the surfel depth at observation boundaries (determined from the data association result from Sec. 3.1). This is similar to reducing the integration weight at boundaries, but in contrast to that avoids drift if the biased observations occur over many frames (*c.f.* Fig. 4 (d) - (f) for an illustration).

The exact procedure for boundary blending is specified in Alg. 1. This image-based algorithm first detects observation boundaries in the depth image by finding pixels which both have a depth measurement and an associated supported surfel, as well as at least one neighbor pixel which lacks at least one of these. These pixels are used as seed pixels: The difference between the measurement depth and the supported surfel depth at these pixels is computed and dilated to suitable neighbor pixels (which are on the side of the boundary that will be corrected) for a number of iterations. Each pixel which receives a propagated depth difference is then updated: We hallucinate a change to the pixel's depth to reduce the depth difference, while linearly decreasing this effect the farther a pixel is away from the observation boundary.

Algorithm 1 Observation boundary blending

$D(p)$: depth of pixel p; is set to 0 if there is no depth measurement there
SD(p) : average supported surfel depth of pixel p; is set to 0 if there is no supported surfel at this pixel
$I_d, I_s, \delta_d, \delta_s$: temporary image-sized buffers
$i_{\text{count}} = 10$: iteration count
procedure BLEND
  # Initialize I_d, I_s to -1
  for all pixels p **do** $I_d(p) := -1;\ I_s(p) := -1$
  # For all pixels with depth and surfel(s) ...
  for all pixels p with $D(p) \neq 0$ and SD$(p) \neq 0$ **do**
    d := SD$(p) - D(p)$ # Surfel-vs-measurement delta
    for all pixels q in p's 8-neighborhood **do**
      # At boundary of measurement area?
      if $I_d(p) = -1$ and $D(q) = 0$ **then**
        # Start propagation (1st case)
        $\delta_d(p) :=$ d; $I_d(p) := 0;\ D(p) :=$ SD(p)
      # At boundary of surfel area?
      if $I_s(p) = -1$ and SD$(q) = 0$ **then**
        # Start propagation (2nd case)
        $\delta_s(p) :=$ d; $I_s(p) := 0$
  for $i \in [1, i_{\text{count}} - 1]$ **do** # Perform blending iterations
    # For all pixels with depth ...
    for all pixels p with $D(p) \neq 0$ **do**
      # Propagate among pixels with surfel(s)?
      if SD$(p) \neq 0$ and $I_d(p) = -1$ **then**
        UPDATE(i, p, δ_d, I_d) # Update (1st case)
      # Propagate among pixels without surfels?
      if SD$(p) = 0$ and $I_s(p) = -1$ **then**
        UPDATE(i, p, δ_s, I_s) # Update (2nd case)
function UPDATE(i, p, δ, I)
  # Average deltas of previous iteration ...
  sum := 0; count := 0
  for all pixels q in p's 8-neighborhood **do**
    if $I(q) = i - 1$ **then** sum += $\delta(q)$; count += 1
  # Apply the (weighted) averaged delta?
  if count > 0 **then**
    $I(p) := i;\ \delta(p) := \frac{\text{sum}}{\text{count}};\ \ D(p)$ += $(1 - \frac{i}{i_{\text{count}}})\frac{\text{sum}}{\text{count}}$

3.3 Meshing

We require a very fast, scale-independent (re)meshing algorithm to maintain a triangulation of millions of surfels during reconstruction. For meshing, we adapt [22], [23] with minor modifications. We briefly present this approach with our modifications here.

Spatial access. The meshing algorithm needs to quickly and accurately find all other surfels within a surfel's radius. Thus, a spatial access structure is needed. For our algorithm, this structure must also efficiently adapt to moved surfels. In contrast to [22], [23], which assume that the 3D surfel positions are static, we thus use a compressed octree [43] which we update lazily: A moved surfel is only propagated up in

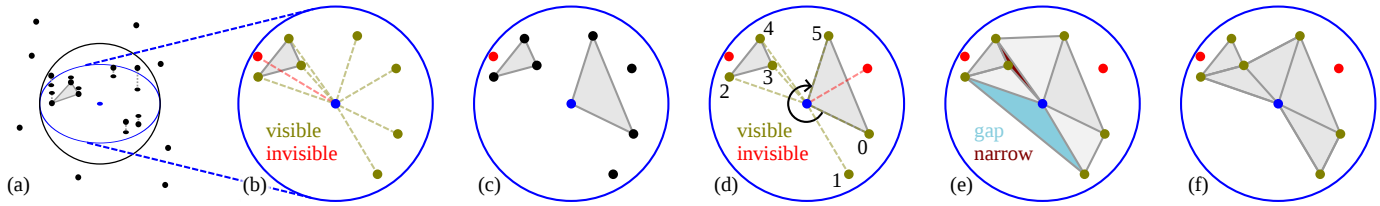


Fig. 5. Overview of triangulation for the center surfel (following [22]): (a) Neighbor search and projection onto the tangent plane of the surfel, shown in (b) with visibility in the plane. (c) After initial triangle creation. (d) Updated visibility and neighbor ordering. (e) Gap and narrow triangle classification. (f) Result after gap and narrow triangle removal.

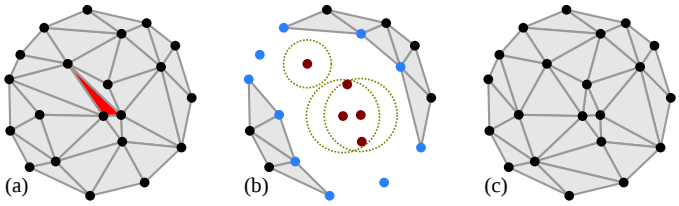


Fig. 6. Overview of the remeshing process: (a) Invalid triangles (red) are identified (*c.f.* Sec. 3.4). (b) All triangles connected to surfels (red) within the neighbor search radii (yellow) of the triangles' corner vertices are deleted. Affected surfels (red and blue) are scheduled for remeshing. (c) Holes are filled by the meshing algorithm (Sec. 3.3 / Fig. 5).

the tree to the first node which contains its new position. We only propagate the surfel downwards when a search traverses that node, creating new nodes if required. Propagation stops if a leaf node or a node not traversed by the search is reached. The octree provides scale-independence with an asymptotic search time of $\mathcal{O}(n)$. While this worst-case hardly occurs in practice, voxel hashing [26] might be a faster alternative if the scan resolution is fixed and known.

Triangulation. Similar to [22], each surfel is assigned a triangulation state: *free* if the surfel has no incident triangles, *front* (which corresponds to *fringe* and *boundary* in [22]) if it lies on the current boundary of the mesh, or *completed* if it lies within the mesh. All new surfels are initially marked *free*. The algorithm then greedily iterates over all new surfels and all *free* and *front* surfels which moved since the last triangulation iteration. We also iterate over all surfels queued for (re)meshing (*c.f.* Sec. 3.4). For each such surfel, its neighborhood is locally triangulated.

For the current surfel s in this loop, the algorithm from [22] first finds all neighboring surfels as candidates for triangulation. In our implementation, we use the surfel's radius r_s as search radius (*c.f.* Fig. 5(a)). r_s is defined as a scaled 3D distance to the surfels that are neighbors to s under the highest resolution under which s was observed (*c.f.* Eq. 1). Thus, surfels farther away than r_s can be ignored safely. Furthermore, if s is on the mesh boundary and its boundary neighbors are further away than r_s, we extend the search radius as necessary to include those neighbors, up to $2 \cdot r_s$. This helps in meshing surfaces observed under slanted angles without having to resample these surfaces as done in [23]. If this limit is exceeded, triangulation for this surfel is aborted.

All neighbor surfels are projected onto the tangent plane defined by the surfel's normal $\mathbf{n}_s$ (*c.f.* Fig. 5(a,b)). Neighboring surfels which are invisible from s as seen in this 2D projection (*c.f.* [22] for details and Fig. 5(b,d) for an illustration), or whose normal differs too much from $\mathbf{n}_s$, are discarded. If s is *free*, the algorithm from [23] first attempts to create an initial triangle with s as one of its vertices (*c.f.* Fig. 5(c)). The remaining visible neighbors are sorted according to their angle to s in the 2D projection (*c.f.* Fig. 5(d)). Spaces between adjacent neighbors are classified as *gap* resp. *narrow* if the angle between them is considered too large resp. small for triangulation (*c.f.* Fig. 5(e)). Unless this leads to holes, neighbors which would form narrow triangles are discarded to avoid degenerate triangles [22]. The exact thresholds are not critical to our algorithm. We remove the *gap*' classification where this closes a hole in the mesh. Finally, new triangles are added to fill the space between s and the remaining neighbors, excluding *gaps* (*c.f.* Fig. 5(f)). After a meshing iteration finishes, we update the triangle indices of our mesh to the new triangulation.

3.4 Remeshing

Remeshing is performed on the existing mesh (with vertex positions updated from the current surfel positions) before each meshing iteration. Fig. 6 shows an overview: We identify invalid triangles and re-create the surface locally there. The basic idea behind our remeshing approach is to use the algorithm from Sec. 3.3 to also update the 3D mesh. Thus, our definition of invalid triangles is based on the criteria used to generate triangles.

We define a triangle to be *invalid* if it is impossible for the meshing algorithm to create it given the current surfel cloud. A triangle can become invalid when, due to new input, surfels move or new surfels are created. However, most triangles usually stay valid if the surfels do not move too much. We derive the following criteria to identify valid triangles from the algorithm from Sec. 3.3: 1) For a valid triangle, at least one of its vertex surfels (named s in the following) contains all triangle vertices in its neighbor search radius and has a similar normal to them (otherwise the triangle vertices could not be found during neighbor search). 2) The triangle normal must be within $90°$ of surfel s' normal $\mathbf{n}_s$ (otherwise the triangle would be upside down in s' tangent plane). 3) In s' tangent plane, no other neighbor surfel projects into the triangle and the triangle does not intersect another triangle (otherwise vertex visibility would prevent the triangle's creation).

Since gaps can be filled for hole closing and some narrow triangles cannot be removed, these are not criteria for invalid triangles. For efficiency reasons, we modify the

criteria as follows: a) We allow triangles to become 1.5 times longer than the maximum extended neighbor search radius in order to allow for more surfel movement before remeshing. b) We do not fully test for criterion 3) to avoid performing many (slow) spatial searches. As a frequent special case of this criterion, we address the case of newly created surfels next to existing triangles: we find all existing surfels within the neighbor search radius of new surfels and delete all triangles connected to them. However, we do not test for conflicts arising for existing surfels or triangles. We found that this test is not strictly required, as affected regions typically are remeshed anyways due to violating other criteria.

In our remeshing algorithm, we test all triangles whose vertices have moved since the last test (excluding inactive surfels moved by loop closures, which we only consider once they are active again to improve performance). We delete detected invalid triangles, and also all other triangles connected to surfels within the neighbor search radii of their corner surfels (*c.f.* Fig. 6(b)). All affected surfels are scheduled for meshing in the following meshing iteration, *i.e.*, the meshing algorithm in Sec. 3.3 also fills the holes created by this step. This is possible since the remaining mesh can be used as initial state for meshing, and new triangles are added at mesh boundaries such as holes.

4 Evaluation

We implemented surfel reconstruction and denoising on a GPU using CUDA 8.0. Meshing and remeshing run in parallel on the CPU. When a meshing iteration is expected to finish (based on the previous iteration's duration), we transfer the current surfel cloud from the GPU to the CPU. We tested our method on a PC with an Intel Core i7 6700K and an MSI Geforce GTX 1080 Gaming X 8G. Camera poses and loop closures are computed via a state-of-the-art SLAM system [17]. We mainly evaluate our approach on the TUM RGB-D benchmark [44] and the ICL-NUIM dataset [45], but also use the pre-registered version of the CoRBS dataset [46]. Note that due to the nature of the topic, our approach is best seen in motion in the supplementary video (available at https://youtu.be/ouspbzHk5L0).

Depth image preprocessing. We drop pixels with a depth larger than $3m$ since they tend to be very noisy [17]. We run a bilateral filter with depth-dependent σ_z parameter ($\sigma_{xy} = 3px, \sigma_z = 0.05 \cdot z$) on the images to mitigate quantization. We filter outliers by projecting each pixel into the 4 previous and 4 following frames, only keeping pixels that project to valid depth measurements within 2% of the projected depth in all these frames. We remove all depth pixels within $2px$ of missing depth measurements to reduce the impact of foreground fattening. We compute normals via finite differences. Pixels whose normal differs by more than $85°$ from the direction to the camera are dropped.

4.1 Ablation Study

The impact of denoising. In a first experiment, we show that the two denoising / smoothing steps (regularization and boundary blending) introduced in Sec. 3.2 are necessary when meshing surfel point clouds.

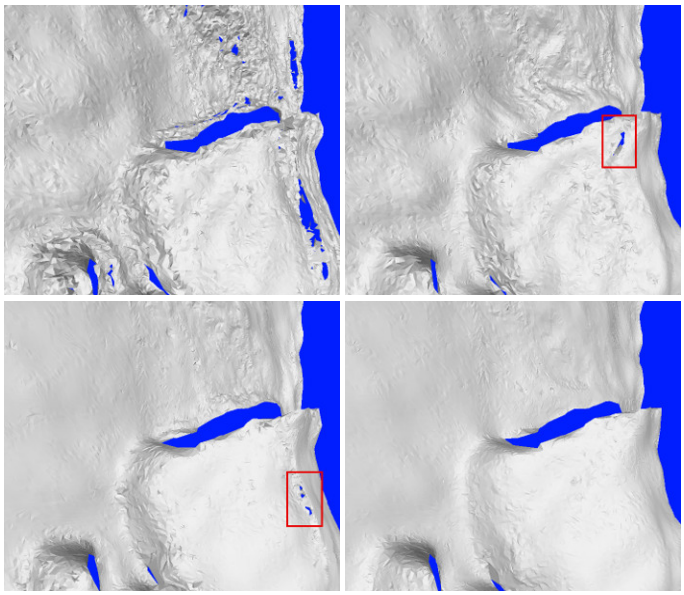


Fig. 7. Comparison between no smoothing (top left), observation boundary blending only (top right), regularization only (bottom left), and both (bottom right). Both types of smoothing contribute to a hole-free triangulation and improve the surface quality.

Fig. 7 shows that both proposed smoothing steps contribute to a hole-free mesh. In this example, both steps are required to get a closed surface. We also compare our smoothing against standard Laplacian smoothing. Laplacian smoothing is defined by the update equation $\mathbf{p}_s := \mathbf{p}_s + \lambda/|N_s| \sum_{n \in N_s} (\mathbf{p}_n - \mathbf{p}_s)$. Notice that Laplacian smoothing does not contain a data term and is thus prone to shrinking surfaces towards the mean of all data points. We use the same vertex neighbor set for N_s which we use for our regularization approach, and choose $\lambda = 1$. Fig. 8 shows that in contrast to standard Laplacian smoothing, our approach avoids object shrinkage and does not over-smooth when many iterations are applied. Furthermore, only our blending step is able to remove a scanning boundary artefact.

As a quantitative experiment, Tab. 1 evaluates a number of mesh quality metrics with and without our denoising steps on the reconstructions of three datasets. Both denoising steps consistently yield smoother surfaces on these datasets when enabled. In addition, they reduce the number of free and boundary surfels, indicating less noise and holes. Furthermore, meshes are manifold in more places and there are fewer self-intersections. This clearly shows that our proposed denoising stages lead to better surface reconstructions.

The impact of remeshing. Remeshing is required for updating existing surfaces. To be more effective than meshing from scratch, it must be faster while providing comparable results. We show the latter by comparing the mesh quality of the reconstructions generated by our proposed approach with remeshing (lines with "remesh" in Tab. 1) with an approach that meshes the final surfel clouds from scratch (lines without "remesh" in Tab. 1). Most metrics are similar, with the meshing from scratch yielding results that are manifold in slightly more places and that have slightly less self-intersections. The latter is to be expected, since we do not

TABLE 1
Mesh quality of our method (with and without regularization (reg), blending (bld), and incremental remeshing (remesh)) and of volume-based methods. We evaluate the amount of unreferenced vertices (free), the amount of vertices on a mesh boundary (bdry), the average minimum triangle angle (angle), the amount of vertices having a locally manifold (boundary or non-boundary) triangle neighborhood with consistent orientation (manif), the amount of triangles having a self-intersection with the mesh (intsc), and the mean curvature (crv) in $\frac{0.01}{m}$.

			fr1/desk [44]						fr1/xyz [44]						fr3/office [44]					
reg	bld	remesh	free	bdry	angle	manif	intsc	crv	free	bdry	angle	manif	intsc	crv	free	bdry	angle	manif	intsc	crv
O	O	O	0.8%	4.8%	31.4°	98.6%	0.4%	2.0	0.4%	3.6%	32.2°	99.3%	0.2%	1.3	0.5%	2.8%	**32.9°**	99.2%	0.3%	1.4
O	O	X	0.8%	4.8%	**31.5°**	98.4%	0.6%	2.0	0.5%	3.5%	**32.5°**	99.1%	0.2%	1.3	0.5%	2.7%	32.7°	99.0%	0.5%	1.4
O	X	O	0.3%	2.8%	30.9°	99.4%	0.4%	1.1	**0.1%**	2.0%	32.0°	99.8%	**0.1%**	0.6	**0.1%**	1.6%	32.5°	99.7%	0.2%	0.7
O	X	X	0.3%	2.8%	31.3°	99.2%	0.5%	1.1	**0.1%**	1.9%	**32.5°**	99.8%	**0.1%**	0.6	0.2%	1.5%	32.6°	99.6%	0.2%	0.7
X	O	O	0.7%	3.6%	29.1°	99.0%	0.3%	1.0	0.2%	2.3%	31.0°	99.7%	**0.1%**	0.6	0.3%	2.0%	31.1°	99.6%	0.2%	0.7
X	O	X	0.7%	3.4%	29.5°	98.9%	0.4%	1.0	0.2%	2.1%	31.7°	99.6%	0.2%	0.6	0.3%	1.7%	31.3°	99.5%	0.2%	0.7
X	X	O	**0.2%**	2.4%	29.5°	**99.6%**	**0.2%**	**0.6**	**0.1%**	1.9%	31.4°	**99.9%**	**0.1%**	**0.3**	**0.1%**	1.5%	31.7°	**99.8%**	**0.1%**	**0.5**
X	**X**	**X**	**0.2%**	2.3%	30.0°	99.5%	0.3%	**0.6**	**0.1%**	1.7%	32.0°	99.8%	**0.1%**	**0.3**	**0.1%**	1.3%	32.0°	99.7%	0.2%	**0.5**
InfiniTAM [29]			**0.0%**	9.7%	32.5°	**100%**	**0.0%**	7.4	**0.0%**	11.1%	33.7°	**100%**	**0.0%**	3.9	**0.0%**	5.5%	**35.3°**	**100%**	**0.0%**	2.6
FastFusion [27]			**0.0%**	17.7%	**33.8°**	98.4%	1.0%	8.4	**0.0%**	15.6%	**34.1°**	99.1%	0.6%	7.2	**0.0%**	18.3%	34.8°	97.9%	1.7%	6.6

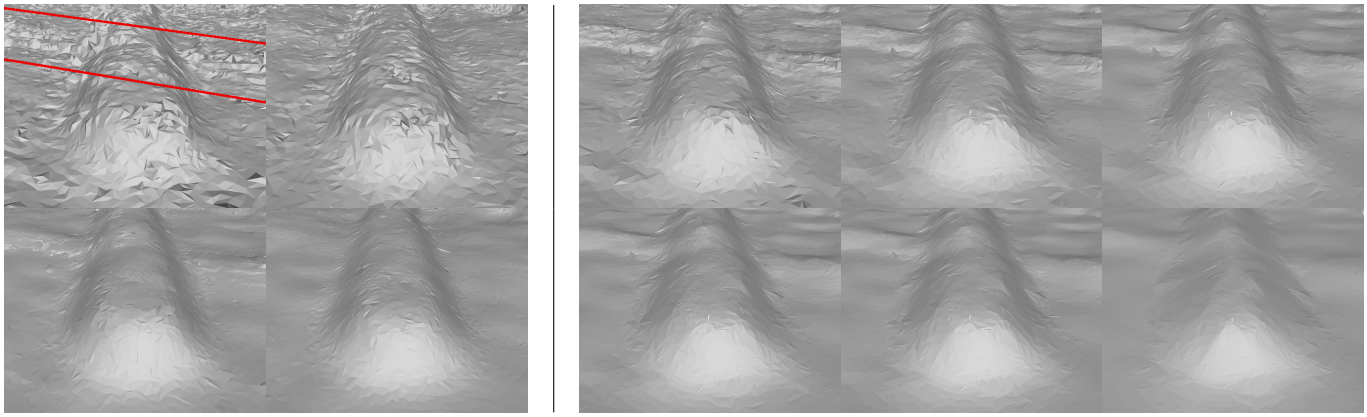


Fig. 8. Comparison between our denoising (with regularization, shown in the fully converged state) and standard Laplacian Smoothing. **Left block:** no smoothing (top left), observation boundary blending only (top right), regularization only (bottom left), and both (bottom right). **Right block:** Top row: one, two, three iterations of Laplacian smoothing only. Bottom row: four, five, twenty iterations of Laplacian smoothing only. Out of all smoothing methods, only boundary blending is able to remove the horizontal scanning artefact caused by small pose errors (marked in red in the top left image). Laplacian smoothing shrinks the object and continues to smooth it with every iteration, while ours converges to a good solution due to our data term.

test for this in the remeshing algorithm. Notice that in our implementation of [22], [23], we favor completeness over manifoldness. Still, the overall differences between remeshing and meshing from scratch are very small. Meshing the surfel cloud from scratch is much slower than our partial remeshing, as shown in Fig. 9. For example, meshing the final surfel cloud in this example from scratch takes about 5.6 seconds. Fig. 9 also shows that the remeshing time of the volumetric FastFusion approach from [27] (purple line) is similar to ours.

Overall, our experiments clearly demonstrate that our online remeshing approach achieves comparable results as performing meshing from scratch at faster run-times and better scalability, since the performance mostly depends on the region affected by remeshing instead of the size of the whole reconstruction.

4.2 Comparison with the State-of-the-Art

Mesh quality. Tab. 1 also compares the mesh quality of our approach with two volume-based approaches, InfiniTAM [29] and FastFusion [27]. We use source code provided by the authors for comparison. Both approaches use the Marching Cubes algorithm [13] for obtaining a triangle mesh from their volumetric representations. We merge vertices having the same position before evaluation. By design, Marching Cubes produces clean meshes wrt. unreferenced vertices, manifoldness and self-intersections (the meshing of [27] seems to be affected by bugs though). However, judging by the minimum triangle angles, our method creates triangles of comparable quality. Our approach creates less boundary vertices and our meshes exhibit a significantly lower curvature, indicating smoother reconstructions. At the same time, our approach is flexible enough to adapt to loop closure events (see also the supplementary video), which cannot be handled by [27], [29].

Accuracy and completeness. We compare our reconstructions to other state-of-the-art methods in a quantitative evaluation. For this experiment, we use the living room sequences of the synthetic ICL-NUIM dataset [45], which provides depth maps with simulated noise.

We first compare against ElasticFusion [17] to show that we improve upon it. The kt0 and kt2 sequences trigger loop closures. We leave out kt3 since the open source code of [17] failed to estimate a reasonable trajectory. We align the reconstructions to the ground truth model with point-to-plane ICP. We compute accuracy and completeness as the amount of reconstruction resp. ground truth points which are closer to the ground truth resp. reconstruction than an evaluation threshold [47], [48]. We also evaluate mean curvature as a measure of smoothness. Results are given

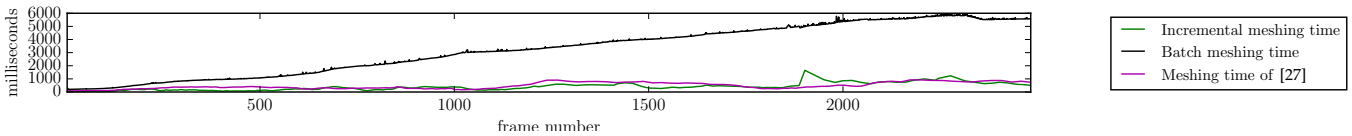


Fig. 9. Performance of our incremental meshing (sum of synchronization, remeshing and meshing times per frame) on the dataset from Fig. 1 compared to: 1) Batch-meshing the surfel reconstruction at each frame from scratch, and 2) [27]'s meshing performance.

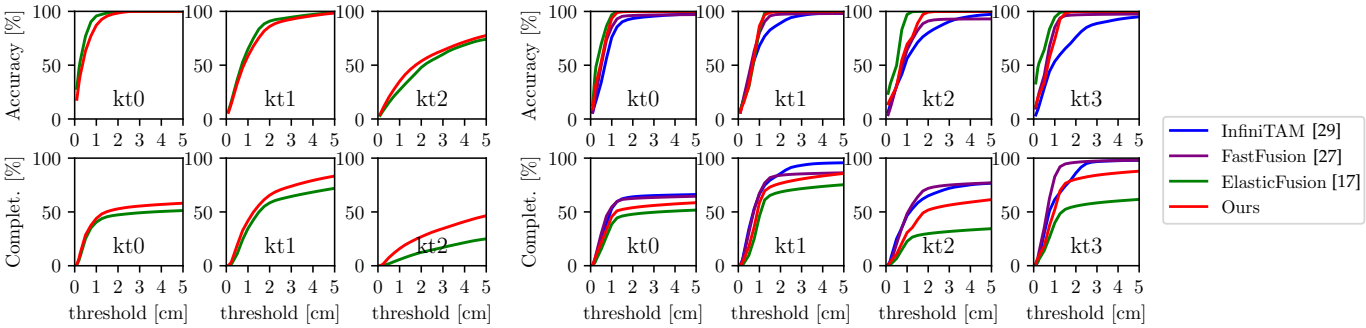


Fig. 10. Accuracy and completeness from Tab. 2, plotted for varying evaluation threshold. Left: trajectory with loop closures, right: ground truth trajectory.

TABLE 2
Accuracy [%], completeness [%], and mean curvature [$\frac{0.01}{m}$] results for ICL-NUIM [45] sequences. Evaluation threshold $1cm$. Our method generally yields higher completeness than ElasticFusion [17] and higher accuracy than InfiniTAM [29] and [27]. Since [29] and [27] cannot handle loop closures, there are no results for these cases.

		with loop closures			ground truth trajectory			
	Method	**kt0**	**kt1**	**kt2**	**kt0**	**kt1**	**kt2**	**kt3**
Accur.	InfiniTAM [29]	-	-	-	75.9	68.1	56.3	53.5
	FastFusion [27]	-	-	-	85.5	80.1	64.2	85.3
	ElasticFusion [17]	**95.8**	**64.9**	26.9	**96.9**	83.2	**97.2**	**94.6**
	SurfelMeshing (Ours)	87.2	58.5	**35.0**	93.5	**86.4**	69.5	74.0
Compl.	InfiniTAM [29]	-	-	-	53.2	66.6	46.6	61.6
	FastFusion [27]	-	-	-	**54.7**	**67.3**	**48.4**	**82.8**
	ElasticFusion [17]	40.6	34.6	5.8	38.6	46.0	22.7	39.8
	SurfelMeshing (Ours)	**44.0**	**41.8**	**15.9**	45.6	58.6	30.8	52.1
Curvat.	InfiniTAM [29]	-	-	-	2.48	2.23	4.71	3.69
	FastFusion [27]	-	-	-	0.99	1.47	1.68	1.33
	ElasticFusion [17]	0.86	0.39	0.39	0.24	0.43	0.34	0.47
	SurfelMeshing (Ours)	**0.22**	**0.15**	**0.30**	**0.15**	**0.17**	**0.18**	**0.32**

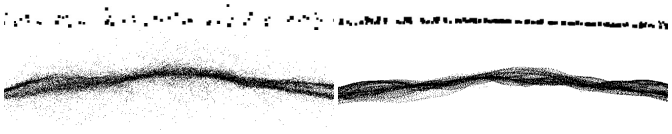


Fig. 11. Cut through surfaces from ElasticFusion (top left) and [36] (bottom left) compared to corresponding surfels in our approach (right). Our method generates smoother surfaces due to our denoising steps (Sec. 3.2).

in Tab. 2 (left) for an evaluation threshold of $1cm$, and are plotted for different thresholds in Fig. 10. ElasticFusion uses more aggressive outlier filtering and thus gives partly more accurate but always less complete results than ours. The plots show that this does not depend on the choice of evaluation threshold. Our results are always smoother than ElasicFusion's due to our denoising. This is also shown qualitatively in Fig. 11.

Next, we compare to the voxel-based methods [27] and InfiniTAM [29]. Comparing against methods that handle loop closures, *e.g.*, [14], is difficult without including the respective SLAM systems in the comparison. For this experiment, we therefore use the ground truth trajectories and disable loop closure handling for all methods. Results are given in Tab. 2 (right) and plotted in Fig. 10. While the voxel-based methods can rely on the TSDF for outlier filtering and obtain more complete results, ours achieves higher accuracy almost throughout. The plots show that these results are again mostly constant for varying the evaluation threshold within a reasonable range. Furthermore, ours has several advantages due to not using a volume (*c.f.* Sec. 4.3).

4.3 Qualitative Results

We perform qualitative experiments on real datasets, mainly using datasets from [44] captured with Kinect v1 cameras. In addition to Fig. 1, Fig. 12 qualitatively shows reconstructions for different scenes, with statistics of the datasets and reconstructions. All estimated camera trajectories for these datasets include loop closures.

Adaptivity to varying scan resolution. Since surfels are created to match the input image resolution, our approach reconstructs the scene's colors at this resolution without texture mapping. This is illustrated by Fig. 13 (left), which shows the mesh being refined as the camera moves closer. Voxel-based methods require a very high volume resolution to match this capability. Thus, they typically reconstruct blurrier texture, as shown in Fig. 2. This figure also shows that our approach leads to improved sharpness compared to the surfel-based ElasticFusion method [17]. This is due to denoising (*c.f.* Fig. 11) and meshing, which improves the appearance over individual splat rendering. In our experience, transitions between areas of different resolution do not cause triangulation issues (*c.f.* Fig. 13 (right)).

Reconstructing thin objects. Another benefit of our method is its ability to reconstruct thin objects. This is possible

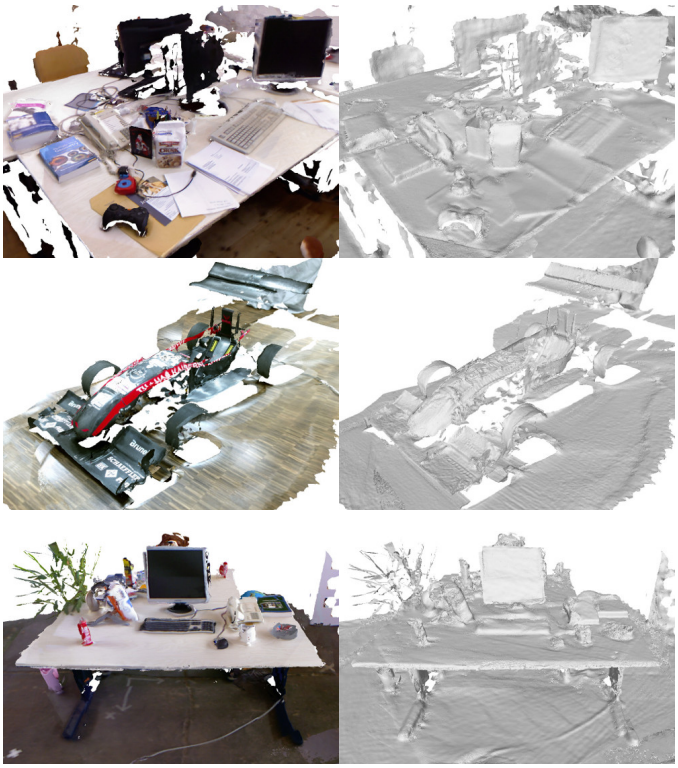


Fig. 12. Qualitative results for our method (top to bottom; M = million):
fr1/desk [44]: 573 frames, 1.2M surfels, 2.4M triangles.
R3 [46]: 2151 frames, 1.9M surfels, 3.7M triangles.
fr2/desk [44]: 2893 frames, 1.3M surfels, 2.6M triangles.

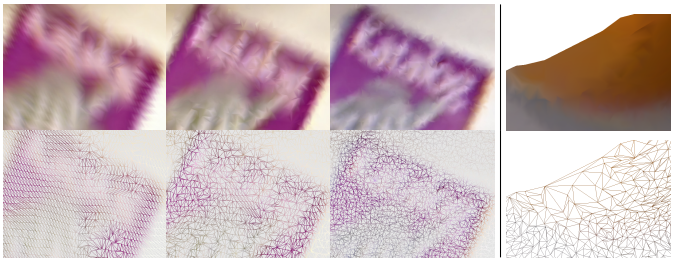


Fig. 13. **Left:** As the camera moves closer (left to right), the mesh (bottom) is refined and thus the colors (top) become more detailed. **Right:** The algorithm naturally supports triangulations with differing surfel resolution.

since it does not require surfaces to enclose a volume, and surfels with opposing normals do not interfere. Thin objects are very hard to reconstruct correctly with volume-based methods, as their space discretization must be able to represent the object and the camera pose must be accurate enough to retain its volume. We demonstrate this on the example of a flyer in Fig. 14.

Impact of the input resolution. We downsample the input images to test the effect of using lower resolution input. The original image size is 640×480 and the downsampled sizes are 320×240 and 160×120. Since our outlier filtering (*c.f.* paragraph on depth image preprocessing) depends on the image resolution, we slightly adapt it for the lower resolutions: We do not remove image pixels which are within 2 pixels of a pixel without depth value. This helps avoid removing too much geometry next to depth discontinuities

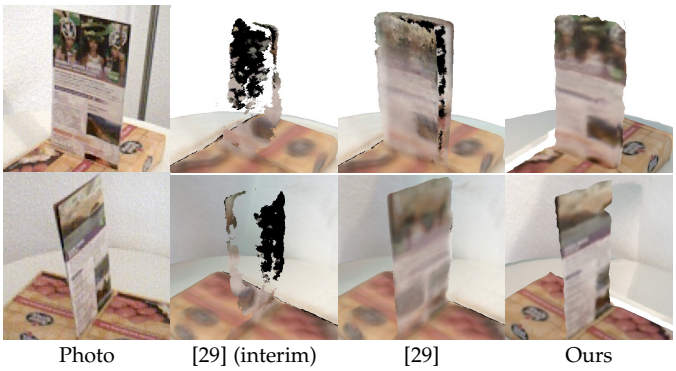


Fig. 14. Thin object reconstruction. Top: front side, bottom: back side. While ours reconstructs both sides successfully, InfiniTAM [29] (using a voxel size of $1mm$) deletes one side before reconstructing the other (see the interim step shown) and finally reconstructs a mixture of the texture of both sides.

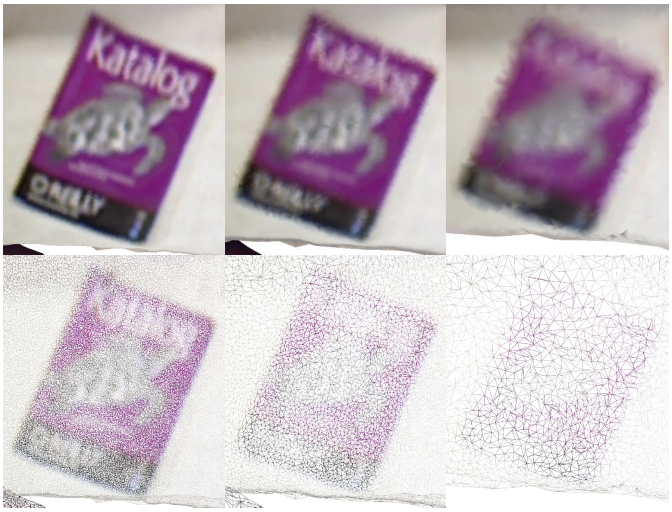


Fig. 15. From left to right: reconstruction with full, half, and quarter resolution input. With smaller input images, the mesh and color resolution decrease.

at lower resolutions. However, the gradient and radius calculation still require all neighbor pixels to be valid. Due to the resulting filtering, this leads us to remove more surface area at low resolutions compared to the full resolution.

Fig. 15 shows a close-up view on how the mesh resolution, and thus the colors, change with the input resolution. As expected, using a lower input resolution leads to both a lower mesh resolution and a lower texture resolution.

Fig. 16 shows the changes in mesh geometry on a larger scale. We note that the level of detail of the mesh degrades gracefully with the resolution of the RGB-D input images. However, lower resolution input leads to larger holes in the mesh, which we believe is partly due to the filtering when computing gradients and radii (mentioned above).

4.4 Performance

Fig. 17 analyzes the performance of our approach on the dataset from Fig. 1 depending on the image size. We optimized our (re)meshing implementation, but did not optimize surfel reconstruction and denoising, which run in real-time for almost all frames using the original image resolu-

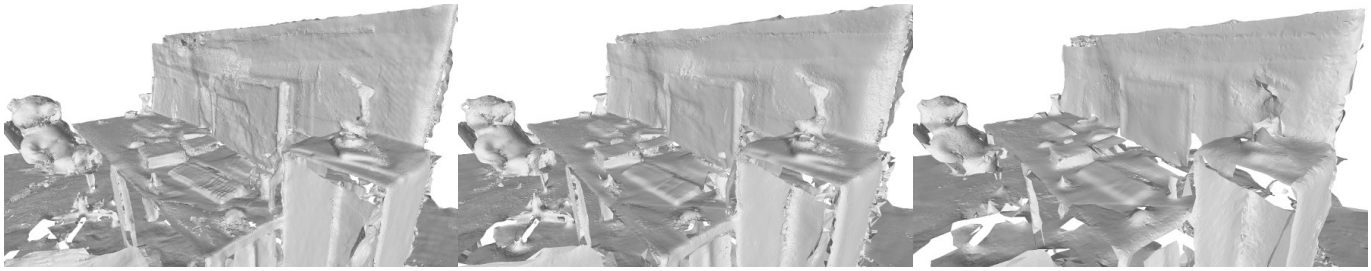


Fig. 16. From left to right: reconstruction with full, half, and quarter resolution input. The geometry becomes less detailed as the resolution of the input images decreases. The mesh also becomes smoother due to stronger averaging of the sensor noise in the downsampling step and due to the denoising affecting larger areas, as well as the fact that small pose errors matter less at lower resolution.

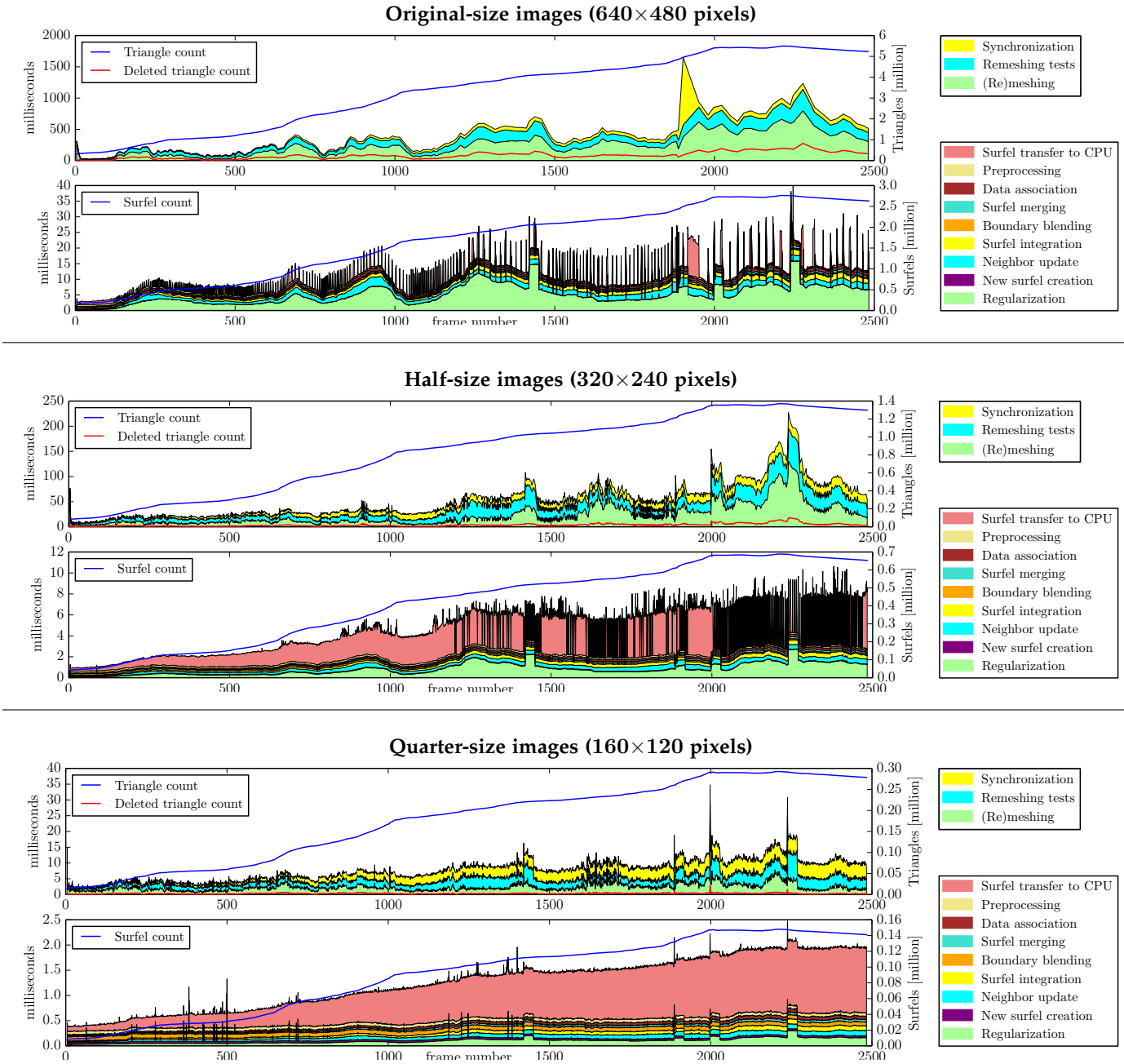


Fig. 17. Performance of (re)meshing (top of each plot pair) and surfel reconstruction (bottom of each plot pair) on the fr3/long_office_household dataset [44], for full input size (top), half size (middle), and quarter size (bottom). The discontinuities in the surfel denoising performance are caused by re-activation of old surfels due to loop closures. These loop closures are also responsible for the increase in (re)meshing time by the end of the sequence. Please notice the different scales.

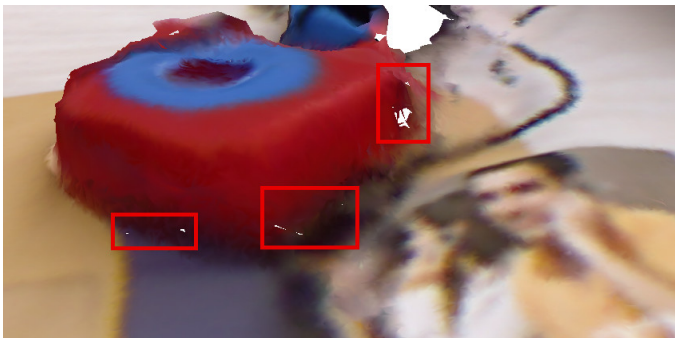


Fig. 18. Example of tiny holes (highlighted in red) in a close-up of the reconstruction of fr1/desk [44] which is also shown in Fig. 12. The holes are created since the reconstructed surfels are too noisy for meshing in these areas. While they are usually tiny, they can be quite noticeable if the mesh is rendered without anti-aliasing.

tion of 640×480. Due to the strong smoothing during loop closures, they cause short disruptions. For example, handling a loop closure for 3.1 million surfels takes ca. $680ms$. Since loop closures are disruptive events in any case, we did not perceive this as an issue. In addition, the number of smoothing iterations could be reduced for higher speed. On average, remeshing deletes 6% of the scene's triangles per iteration, and the average time per iteration is $212ms$.

5 Discussion & Conclusion

We presented the first surfel-based approach for live mesh reconstruction from RGB-D video. Compared to surfel reconstruction only, our approach creates a triangle mesh, which is useful for many applications. Compared to volumetric methods, ours in particular handles loop closures with little effort, reconstructs high-resolution colors due to its ability to adapt the resolution of the 3D mesh to the input, and can reconstruct thin objects. The approach can therefore provide a dense surface representation during SLAM. To the best of our knowledge, ours is the first method providing all these advantages while enabling online 3D reconstruction.

The proposed approach also comes with some limitations. The Marching Cubes algorithm used by voxel-based methods creates manifold meshes. In contrast, our meshes are not guaranteed to be manifold. As shown in Fig. 18, our meshes can also contain holes if the smoothing is insufficient. Future work could aim to resolve this by using temporary volumes for meshing, similar to [41], which might however also remove the ability to reconstruct thin objects. However, as our evaluations show, non-manifoldness does not occur often. In addition, improvements to depth cameras would also be expected to lead to less smoothing and filtering requirements, and thus less holes.

Finally, for loop closures inducing large deformations, the approach we adopted [17] may rip apart surfaces. In particular, if a surface is observed for a longer time, close-by surfels may be assigned to unrelated deformation nodes. We believe that replacing the node assignment with a similar method from [34], which takes into account which nodes have been observed together, may help mitigating this.

Acknowledgments

Thomas Schöps was supported by a Google PhD Fellowship.

References

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

[2] T. Whelan, M. Kaess, H. Johannsson, M. Fallon, J. Leonard, and J. McDonald, "Real-time large scale dense RGB-D SLAM with volumetric fusion," *IJRR*, 2014.

[3] T. Schöps, T. Sattler, C. Häne, and M. Pollefeys, "Large-scale outdoor 3D reconstruction on a mobile device," *CVIU*, vol. 157, 2017.

[4] C. Häne, L. Heng, G. H. Lee, F. Fraundorfer, P. Furgale, T. Sattler, and M. Pollefeys, "3D visual perception for self-driving cars using a multi-camera system: Calibration, mapping, localization, and obstacle detection," *Image and Vision Computing*, 2017.

[5] R. A. Newcombe, S. J. Lovegrove, and A. Davison, "DTAM: Dense tracking and mapping in real-time," in *ICCV*, 2011.

[6] T. Schöps, M. R. Oswald, P. Speciale, S. Yang, and M. Pollefeys, "Real-time view correction for mobile devices," *TVCG*, vol. 23, 2017.

[7] T. Schöps, J. Engel, and D. Cremers, "Semi-dense visual odometry for AR on a smartphone," in *ISMAR*, 2014.

[8] C. Arth, M. Klopschitz, G. Reitmayr, and D. Schmalstieg, "Real-time self-localization from panoramic images on mobile devices," in *ISMAR*, 2011.

[9] A. Breitenmoser and R. Siegwart, "Surface reconstruction and path planning for industrial inspection with a climbing robot," in *Applied Robotics for the Power Industry*. IEEE, 2012, pp. 22–27.

[10] S. Garrido, M. Malfaz, and D. Blanco, "Application of the fast marching method for outdoor motion planning in robotics," *Robotics and Autonomous Systems*, vol. 61, no. 2, pp. 106–114, 2013.

[11] A. Bircher, K. Alexis, M. Burri, P. Oettershagen, S. Omari, T. Mantel, and R. Siegwart, "Structural inspection path planning via iterative viewpoint resampling with application to aerial robotics," in *ICRA*, 2015.

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

[13] 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.

[14] O. Kähler, V. A. Prisacariu, and D. W. Murray, "Real-time large-scale dense 3D reconstruction with loop closure," in *ECCV*, 2016.

[15] A. Dai, M. Nießner, M. Zollhöfer, S. Izadi, and C. Theobalt, "BundleFusion: Real-time globally consistent 3D reconstruction using on-the-fly surface re-integration," *ACM Transactions on Graphics (TOG)*, vol. 36, no. 3, p. 24, 2017.

[16] H. Pfister, M. Zwicker, J. Van Baar, and M. Gross, "Surfels: Surface elements as rendering primitives," in *Annual Conference on Computer Graphics and Interactive Techniques*, 2000.

[17] T. Whelan, S. Leutenegger, R. Salas-Moreno, B. Glocker, and A. Davison, "ElasticFusion: Dense SLAM without a pose graph," in *RSS*, 2015.

[18] V. Litvinov, M. Lhuillier, and F. Aubière, "Incremental solid modeling from sparse and omnidirectional structure-from-motion data." in *BMVC*, 2013.

[19] A. Romanoni and M. Matteucci, "Efficient moving point handling for incremental 3D manifold reconstruction," in *ICIAP*, 2015.

[20] E. Piazza, A. Romanoni, and M. Matteucci, "Real-time CPU-based large-scale 3D mesh reconstruction," in *RA-L*, 2018.

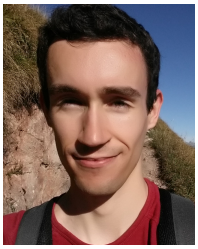
[21] M. Kazhdan, M. Bolitho, and H. Hoppe, "Poisson surface reconstruction," in *SGP*, 2006.

[22] M. Gopi and S. Krishnan, "A fast and efficient projection-based approach for surface reconstruction," in *High Performance Computer Graphics, Multimedia and Visualization*, 2000.

[23] Z. C. Marton, R. B. Rusu, and M. Beetz, "On fast surface reconstruction methods for large and noisy point clouds," in *ICRA*, 2009.

[24] M. Keller, D. Lefloch, M. Lambers, S. Izadi, T. Weyrich, and A. Kolb, "Real-time 3D reconstruction in dynamic scenes using point-based fusion," in *3DV*, 2013.

[25] J. Chen, D. Bautembach, and S. Izadi, "Scalable real-time volumetric surface reconstruction," *ACM Transactions on Graphics (TOG)*, vol. 32, no. 4, p. 113, 2013.
[26] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger, "Real-time 3D reconstruction at scale using voxel hashing," in *SIGGRAPH Asia*, 2013.
[27] F. Steinbrücker, J. Sturm, and D. Cremers, "Volumetric 3D mapping in real-time on a CPU," in *ICRA*, 2014.
[28] O. Kähler, V. A. Prisacariu, J. Valentin, and D. Murray, "Hierarchical voxel block hashing for efficient integration of depth images," *RA-L*, vol. 1, no. 1, pp. 192 – 197, 2016.
[29] O. Kähler, V. A. Prisacariu, C. Y. Ren, X. Sun, P. H. S. Torr, and D. W. Murray, "Very high frame rate volumetric integration of depth images on mobile device," in *ISMAR*, 2015.
[30] M. Klingensmith, I. Dryanovski, S. Srinivasa, and J. Xiao, "Chisel: Real time large scale 3D reconstruction onboard a mobile device using spatially hashed signed distance fields." in *RSS*, 2015.
[31] R. Maier, R. Schaller, and D. Cremers, "Efficient online surface correction for real-time large-scale 3D reconstruction," in *BMVC*, 2017.
[32] L. Han and L. Fang, "FlashFusion: Real-time globally consistent dense 3D reconstruction using CPU computing," in *RSS*, 2018.
[33] J. Stückler and S. Behnke, "Multi-resolution surfel maps for efficient dense 3D modeling and tracking," *Journal of Visual Communication and Image Representation*, vol. 25, no. 1, pp. 137–147, 2014.
[34] T. Weise, T. Wismer, B. Leibe, and L. Van Gool, "Online loop closure for real-time interactive 3D scanning," *CVIU*, vol. 115, no. 5, pp. 635–648, 2011.
[35] D. Lefloch, T. Weyrich, and A. Kolb, "Anisotropic point-based fusion," in *International Conference on Information Fusion*, July 2015.
[36] D. Lefloch, M. Kluge, H. Sarbolandi, T. Weyrich, and A. Kolb, "Comprehensive use of curvature for robust and accurate online surface reconstruction," *PAMI*, vol. 39, pp. 2349–2365, Dec. 2017.
[37] W. Gao and R. Tedrake, "SurfelWarp: Efficient non-volumetric single view dynamic reconstruction," in *RSS*, 2018.
[38] Z. Yan, M. Ye, and L. Ren, "Dense visual SLAM with probabilistic surfel map," *TVCG*, vol. 23, no. 11, pp. 2389–2398, 2017.
[39] T. Bodenmüller, "Streaming surface reconstruction from real time 3D measurements," Ph.D. dissertation, TUM, 2009.
[40] N. Schertler, M. Tarini, W. Jakob, M. Kazhdan, S. Gumhold, and D. Panozzo, "Field-aligned online surface reconstruction," *ACM Transactions on Graphics (TOG)*, vol. 36, no. 4, p. 77, 2017.
[41] L. Ladicky, O. Saurer, S. Jeong, F. Maninchedda, and M. Pollefeys, "From point clouds to mesh using regression," in *ICCV*, 2017.
[42] J. Zienkiewicz, A. Tsiotsios, A. Davison, and S. Leutenegger, "Monocular, real-time surface reconstruction using dynamic level of detail," in *3DV*, 2016.
[43] K. L. Clarkson, "Fast algorithms for the all nearest neighbors problem," in *Annual Symp. on Foundations of Computer Science*, 1983.
[44] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers, "A benchmark for the evaluation of RGB-D SLAM systems," in *IROS*, 2012.
[45] A. Handa, T. Whelan, J. McDonald, and A. Davison, "A benchmark for RGB-D visual odometry, 3D reconstruction and SLAM," in *ICRA*, 2014.
[46] O. Wasenmüller, M. Meyer, and D. Stricker, "CoRBS: Comprehensive RGB-D benchmark for SLAM using Kinect v2," in *WACV*, 2016.
[47] A. Knapitsch, J. Park, Q.-Y. Zhou, and V. Koltun, "Tanks and temples: Benchmarking large-scale scene reconstruction," *ACM Transactions on Graphics (TOG)*, vol. 36, no. 4, 2017.
[48] T. Schöps, J. L. Schönberger, S. Galliani, T. Sattler, K. Schindler, M. Pollefeys, and A. Geiger, "A multi-view stereo benchmark with high-resolution images and multi-camera videos," in *CVPR*, 2017.



Thomas Schöps received the MS degree in Informatics from TU Munich in 2014. He is currently working towards the PhD degree in the Computer Vision and Geometry Group at ETH Zurich under the supervision of Marc Pollefeys. His research interests are dense 3D reconstruction and SLAM.

Torsten Sattler received his PhD degree from RWTH Aachen University in 2014. He joined ETH Zurich as a postdoc in the Computer Vision and Geometry Group and is currently a senior researcher. From 2016 to 2018, he was Marc Pollefeys' deputy, effectively managing the group during Marc Pollefeys' sabbatical. His research interests center around visual localization and 3D mapping and include local features, camera pose estimation, SLAM, dense 3D reconstruction, image retrieval, and semantic scene understanding. He has organized workshops and tutorials on these topics at ECCV, ICCV, and CVPR.

Marc Pollefeys is a Professor of Computer Science at ETH Zurich and Director of Science at Microsoft working on HoloLens and Mixed Reality. He is best known for his work in 3D computer vision, having been the first to develop a software pipeline to automatically turn photographs into 3D models, but also works on robotics, graphics and machine learning problems. Other noteworthy projects he worked on with collaborators at UNC Chapel Hill and ETH Zurich are real-time 3D scanning with mobile devices, a real-time pipeline for 3D reconstruction of cities from vehicle mounted-cameras, camera-based self-driving cars and the first fully autonomous vision-based drone. Most recently his academic research has focused on combining 3D reconstruction with semantic scene understanding. He received a master of science in electrical engineering and a PhD in computer vision from the KU Leuven in Belgium in 1994 and 1999 respectively. He became an assistant professor at the University of North Carolina in Chapel Hill in 2002 and joined ETH Zurich as a full professor in 2007.