Out-of-Core Surface Reconstruction via Global TGV Minimization

Nikolai Poliarnyi

Agisoft LLC, St. Petersburg, Russia

polarnick@agisoft.com

Abstract

We present an out-of-core variational approach for surface reconstruction from a set of aligned depth maps. Input depth maps are supposed to be reconstructed from regular photos or/and can be a representation of terrestrial LIDAR point clouds. Our approach is based on surface reconstruction via total generalized variation minimization (TGV) because of its strong visibility-based noise-filtering properties and GPU-friendliness. Our main contribution is an out-of-core OpenCL-accelerated adaptation of this numerical algorithm which can handle arbitrarily large real-world scenes with scale diversity.

1. Introduction

The structure from motion pipeline makes it possible to take photos of the same object/scene and then not only align and calibrate these photos, but also reconstruct an observed surface with a high amount of details. At the moment, the progress in camera sensor development opens a possibility for a regular user to take photos with a size up to hundreds of megapixels, the number which has been increasing rapidly over the past decades. Additionally, due to help of UAVs, affordable quadrocopters and automatic flight planners, it becomes possible to gradually increase the amount of pictures one can take in a short span of time. Therefore, in the area of photogrammetry, the task of being able to use all of the available data for a detailed noise-free surface reconstruction in an out-of-core fashion is necessary to make a highly detailed large scale reconstruction possible on affordable computers with limited RAM.

We present a surface reconstruction method that has strong noise-filtering properties and can take both depth maps and terrestrial LIDAR scans as an input. The whole method is implemented in an out-of-core way: the required memory usage is low even for very large datasets - we targeted the usage to be around 16 GB even for a Copenhagen city dataset with 27472 photos - see Fig. 1. Each processing stage is divided into independent parts for out-of-core guarantees, thus additionally obtaining a massive parallelism

Figure 1. Our method can handle an arbitrary large scene – even $425 \, \mathrm{km}^2$ of the Copenhagen city (this polygonal model was reconstructed from 27472 aerial photos – see supplementary for details).

property (i.e. pipeline is cluster-friendly). Calculation-heavy stages (the computation of histograms and iterative numeric scheme) are accelerated with GPUs via OpenCL API.

2. Related Work

Poisson surface reconstruction method [18] performs well in local geometry details preservation by respecting normals of input point clouds, however it normally fails to handle scale diversity and often fails to filter noise between the surface and the sensor origins. It should be noted that handling of scale diversity can be added, and noise filtering can be implemented on an early stage as a depth map fil-

tering [22]. Nevertheless, while Poisson reconstruction can be implemented in an out-of-core fashion [2] - the preceding depth map filtering approach will likely require a high amount of memory to keep all depth maps relevant for a current depth map filtering in RAM.

Graph cut-based reconstruction methods [15], [17], [14], [33] explicitly take into account visibility rays from sensors' origins to samples in depth maps, and therefore such methods have great noise filtering properties. Scale diversity is also naturally supported via Delaunay tetrahedralized space discretization. However, Delaunay tetrahedralization and minimum graph cut estimation of an irregular graph [3], [12] have high memory consumption and are computationally heavy. Because of this, the out-of-core SSR [24] method happens to be more than one order of magnitude slower than our method.

Local fusion methods [5], [20], [19] including FSSR [9], [10] are well suited for parallelization and scalability [20]. However on the other hand due to their local nature, they have weak hole filling properties and can not filter strong depth map noise in difficult cases like the basin of the fountain in the Citywall dataset as shown in [27].

Photoconsistent mesh refinement-based methods [29], [21] are fast due to GPU acceleration but are not able to change the topology of an input mesh, and thus they heavily depend on the quality of an initial model reconstruction.

Total variation minimization-based methods [32], [13], [25] are shown to have great noise filtering properties due to visibility constraints, and can easily be GPU-accelerated [31]. Additionally, as shown in [27], [28], a variation minimization scheme can be implemented in a compact and scale diversity-aware way by the use of a balanced octree, but even with such compact space representation, the peak memory consumption becomes critical for a large scale scene reconstruction task.

Our TGV-functional formulation follows [25], it was adapted for 3D space in a way, discussed in [32]. We use a 2:1 balanced octree similar to [27] for 3D space representation. In contrast with their method, our framework has strict peak memory guarantees and is much faster thanks to GPU acceleration in the most time-consuming stages (as shown in [27] – its bottlenecks were in the computation of the histograms (17%) and the energy minimization (80%) stages, so we implemented both of them on GPU).

3. Algorithm Overview

To begin with, we would like to discuss the chosen functional minimization with a focus on noise robustness, scale diversity awareness and GPU-friendliness, while not taking memory requirements into account. Later, we will show how to adapt this minimization scheme for an out-of-core fashion in Section 4.
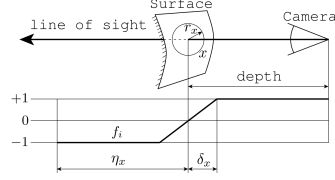

Figure 2. Generalization of a range field like in [32]. f_i equal to $+1$ on each ray between the camera and a depth map sample and then fades away to -1 right under the surface sample.
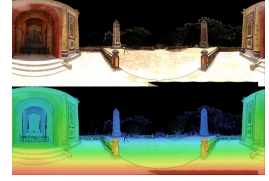

Figure 3. An example of a terrestrial LIDAR scan from the Tomb of Tu Duc dataset (RGB colors and depth from the sensor).

3.1. Distance Fields as Input Data

We prefer to be able to support different kinds of range image data as an input, such as:

- Depth maps built with stereo methods like SGM [16] from regular terrestrial photos or aerial UAV photos;

- RGB-D cameras, which are essentially the same as previously mentioned depth maps;

- Terrestrial LIDAR point clouds with a known sensor origin.

It means that we need to generalize over all these types of data and work with an abstraction of a range image that can be used in the functional formulation. All this data can naturally be formulated in a way, described in [32], as a distance field f_i, which is equal to $+1$ on each ray between the camera and a depth map sample and then fades away to -1 right under the surface sample – see Fig. 2. The only difference for our case is that we want to work with scenes with diverse scales, so δ (the width of a relevant near-surface region) and η (the width of an occluded region behind the surface) should be adaptive to the radius r_x of a material point x. Thus, in all our experiments we use $\delta_x = 6 \cdot r_x$ and $\eta_x = 3 \cdot \delta_x = 18 \cdot r_x$ (smaller values lead to holes in thin surfaces, larger values lead to 'bubbliness' - excessive thickness of thin surfaces).

Both depth maps from stereo photos and RGB-D cameras can be represented in such way naturally. The only non-trivial question is how to estimate material point radii r_x for each pixel in a depth map. For each depth map's pixel, we are estimating the distance in 3D space to the neighboring pixels and take half of that distance as the sample point's radius r_x.

To represent terrestrial LIDAR point clouds as range images, we rely on the fact that the structure of such point clouds is very similar to the structure of pictures, taken with 360-degree cameras (see Fig. 3). Because of that, we treat them just like depth maps of 360-degree cameras with the only difference that LIDAR data is nearly noise-free, and

thus we can rely on such data with more confidence (i.e. with a weaker regularization term) – see Section 3.4 below.

3.2. Functional Formulation

In our task, given multiple distance fields f_i, we want to find such an indicator field u (where $u = 0$ corresponds to a reconstructed isosurface, $u = +1$ – to the exterior of the object, and $u = -1$ – to the interior of the object) that will closely represent these distance fields in some way. One of the ways to formulate what would be a good field u is to introduce some energy functional. The less energy the functional produces – the better the indicator field is.

Total variation (TV) regularization force term for u with an L^1 data fidelity term between u and f_i is one such energy functional named TVL^1 [32] and is defined as:

$$\min_u \left\{ \int_\Omega \left(|\nabla u| + \lambda \sum_i |u - f_i| \right) dx \right\}. \tag{1}$$

Note that while the TV term prevents the surface from having discontinuities, there is no term that would force a regularity of surface normals to tend the reconstruction to piecewise polynomial functions of an arbitrary order (see details in [25]). Such term was introduced as a part of the TGV energy functional via an additional vector field v in [4] and it was adapted to 2.5D reconstruction in [25]:

$$\min_{u,v} \left\{ \int_\Omega \left(\alpha_1 |\nabla u - v| + \alpha_0 |\mathcal{E}(v)| + \sum_i |u - f_i| \right) dx \right\}, \tag{2}$$

where $\mathcal{E}(v)$ denotes the symmetric gradient operator

$$\mathcal{E}(v) = \frac{\nabla v + \nabla v^T}{2}. \tag{3}$$

In order to minimize this TGV functional, we use the primal-dual method [25]. Also, like in [32], we have implemented primal-dual iterations over a coarse-to-fine scheme with the execution of iterations accelerated on GPU for a faster convergence. We have found that 200 iterations are enough for convergence on each level of the scheme.

3.3. Space Discretization

To minimize TGV w.r.t. u, we need to choose a space discretization. A regular grid [32] does not correspond to scale diversity and will potentially lead to high memory consumption. In our approach, we use an adaptive octree. Let r_{root} be the radius of the root cube of the octree. For each point sample x with the center in p_x and the radius r_x, an octree should have a cube c containing p_x. This cube

should be on such an octree depth d_c that the following inequality holds:

$$0.75 \cdot r_x \leq \frac{r_{root}}{2^{d_c}} < 1.5 \cdot r_x. \tag{4}$$

In the implementation of the primal-dual method of minimization, we need pointers from each octree cube to its neighboring cubes in order to access neighbors' u, v values and dual variables p, q. In an adaptive octree, any cube can have any number of neighbors due to adaptive subdivision and scene scale diversity. However, we aim to execute the iterative scheme on GPU, meaning that in order to achieve good performance we need to limit the number of neighboring voxels in some way, resulting in a limited number of references needed to store per each voxel. We utilize the approach, discussed in[27], which results in 2:1 balancing of the adaptive octree, leading to the point when each cube has only 4 or less neighbors over each face.

As will be discussed later in Section 3.4, it is important to know what average radius of point samples S_c corresponds to each octree cube c creation. Therefore, for each octree cube we also store its density, which we also call the cube's radius r_c, defined as

$$r_c = \frac{\sum_{x \in S_c} r_x}{|S_c|}. \tag{5}$$

3.4. Distance Fields to Histograms

We transform all distance fields into histograms in our octree like in [32]. This allows us to run iterations with compact histograms [31] (thanks to fixed size per voxel) instead of large (due to high overlap) depthmaps [32] in memory. The main difference is that we want the algorithm to be aware of scale diversity and to implement the minimization framework in the coarse-to-fine scheme. Because of this, it is impossible to ignore how big or small a voxel projection is in a depth map: if a projected voxel is large (for example, on the coarser levels) and is covered with many depth map pixels (i.e. intersected with many distance field rays), we need to account for all of them. This problem is very similar to the texture aliasing problem in computer graphics, which can be solved with texture mipmaps [30]. Similarly, we have used depth map pyramids by building mipmaps for each depth map. Therefore, when we project a voxel to the depth map, we choose an appropriate depth map level of details, and only then we estimate a histogram bin of a current voxel to which the current depth map will contribute, see the listing in Algorithm 1.

Note that such voxel projection into the pyramid of a depth map is very unstable and changes heavily with any change of the size of a working region bounding box, which happens because the natural voxel's radius in the octree is
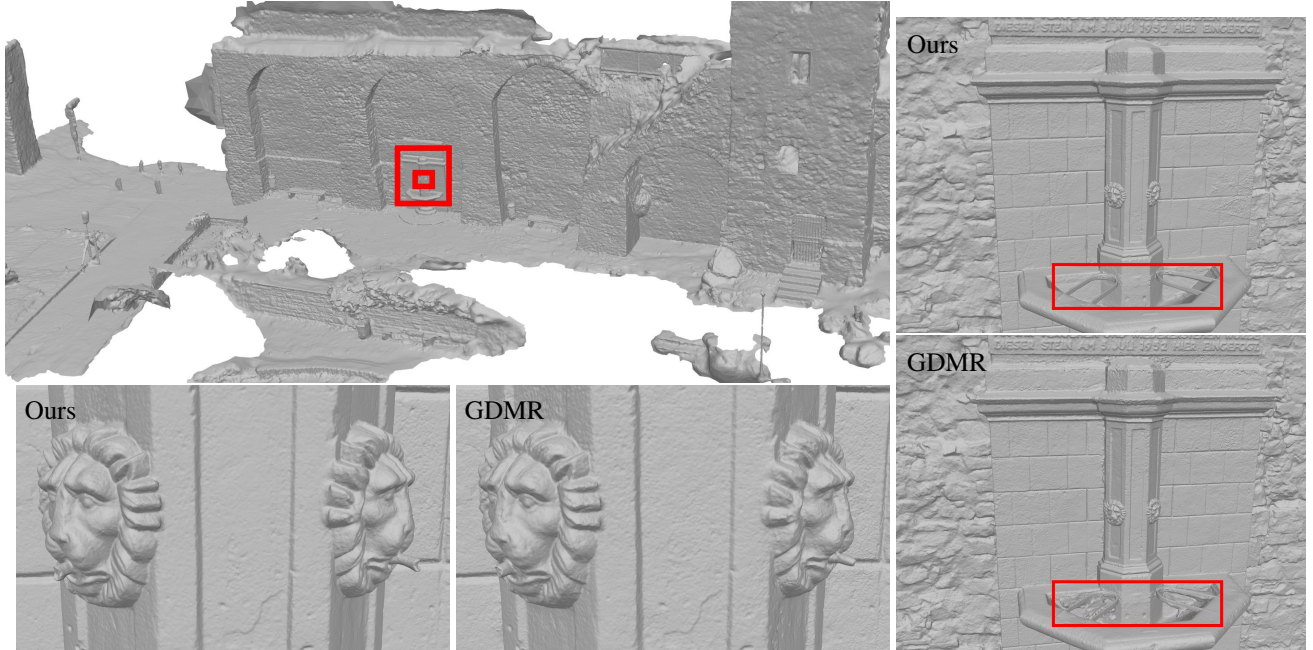
Figure 4. Results on the Citywall dataset and comparison with GDMR [27] results. The results are comparable. Note that due to strong visibility-based noise-filtering properties our method led to the cleaner basin of the fountain.

equal to $\frac{r_{root}}{2^d}$, where d is the voxel's depth and r_{root} is the radius of the octree's root voxel which depends on the size of a whole working region. To be invariant to the selection of the working region size, for each cube, in addition to its center, we store its density value, which is equal to the average radius r_c of point samples that it represents, see Eq. 5.

These details lead to local and stable progressive isosurface refinement from coarse to fine levels - see Fig. 5.

4. Out-of-Core Adaptation

Our main contribution is an out-of-core adaptation of the TGV minimization scheme on a 2:1 balanced octree. Consequently, we implement each stage of the algorithm in an out-of-core way, where the stages are:

4.1 Build a linear octree from all cubes discussed above in Space discretization (Section 3.3)

4.2 Balance the octree, so that each cube has a limited number of neighbors

4.3 Build an indexed treetop to run primal-dual iterations independently on each treetop leaf's part

4.4 Save distance fields' votes to voxels' histogram bins over the balanced octree (GPU-accelerated)

4.5 Coarse-to-fine functional minimization over each part of the balanced octree (GPU-accelerated)

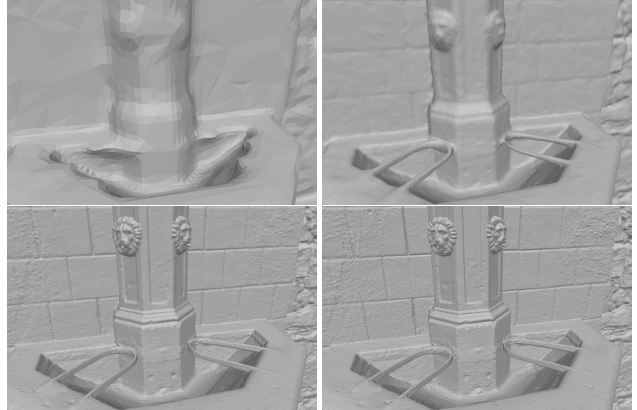**4.6** Surface extraction via the marching cubes algorithm

Figure 5. Citywall dataset. Isosurface progression from coarse to fine levels: 11, 13, 15, 17. Due to cube radius robustness described in Section 3.4, we see local progressive isosurface changes from level to level. Also, note the progressive refinement of topology, which can not be achieved in photoconsistent refinement methods [29], [21] because they refine polygonal surfaces directly (with a 1-to-4 triangle subdivision and vertex movement). Instead, we refine the indicator field, so the topology of our implicit isosurface corresponding to zero indicator value changes together with the indicator field.

4.1. From Distance Fields to Octree

For each distance field, we estimate each sample's radius r_x as half of the distance to its neighbors in this distance field. Then for each sample, we spawn an octree cube containing this sample at an appropriate to r_x depth d, as formulated in

Algorithm 1 Pseudocode of the estimation of distance field contribution to voxel histograms.

```
procedure ADD_TO_VOXEL_HISTOGRAMS(depthmap_pyramid, voxel)
    mipmap_level, pixel ← depthmap_pyramid.project(voxel.center, voxel.radius)
    depthmap ← depthmap_pyramid.get(mipmap_level)
    depth ← depthmap.get(pixel)
    if depth = None then
        return        ▷ If the distance field doesn't have ray in such direction – it doesn't contribute to such voxel anything.
    end if
    r_x ← voxel.radius                                                          ▷ See Section 3.1 and Fig. 2:
    δ_x ← 6 · r_x                                                    ▷ δ_x – width of the relevant near-surface
    η_x ← 3 · δ_x                                                   ▷ η_x – width of the occluded region.
    distance ← depthmap_pyramid.distance_to(voxel.center)
    a ← depth - distance              ▷ a equals to zero if the voxel is exactly at the observed surface level.
    if a < −η_x then  ▷ If the voxel is further then occluded region behind the surface observed with ray then we do not
        return        ▷ observe such voxel from current depth map, i.e. it does not contribute to the voxel's histograms.
    end if
    depthmap_vote ← 1               ▷ Or depthmap_vote ← 5 in case of noise-free terrestrial LIDAR input.
    a ← max(−1.0, min(1.0, a/δ_x))                                  ▷ Clamp to the indicator range.
    bin ← floor(((a + 1.0)/2.0) · 8.0)                             ▷ We are using 8 bins following [31].
    voxel.histograms[bin] ← voxel.histograms[bin] + depthmap_vote
end procedure
```

Eq. 4. All cubes are encoded with 96-bit 3D Morton codes [23] and are saved to a single file per each distance field.

Afterwards, we need to merge all these files containing cubes (i.e. Morton codes). We encoded our cubes with Morton codes, which introduce a Z-curve order over them – see a Z-curve example on a balanced 2D quadtree on Fig. 7. Thus, cubes merging into a single linear octree can be done with an out-of-core k-way merge sort.

4.2. Octree Balancing

To limit the number of neighbors for each cube, we need to balance the obtained octree. A linear octree can be arbitrarily large because it describes the whole scene. Out-of-core octree balancing is described in [26]. Balancing also relies on the Morton code ordering – we only need to load a part of the sorted linear octree, balance that part independently from others, and save the balanced part to a separate file. Later, we only need to merge all balanced parts, which can be accomplished like in the previous stage of linear octree merging – via an out-of-core k-way merge sort.

4.3. Octree Treetop

At this moment, we need to have some high-level scene representation to be able to compute the histograms and run iterations for TGV minimization over the balanced octree part by part. In fact, such subdivision into parts will make it possible for each of the next stages, including the final polygonal surface extraction, to be split into OpenCL's $workItem$-like independent parts (i.e. with massive parallelism, which is useful for cluster-acceleration).
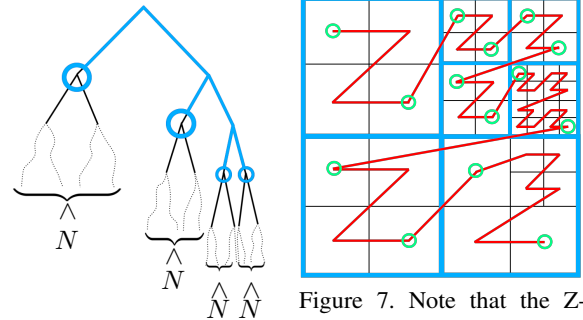

Figure 6. Blue nodes are the treetop leaves with less than $N_{cubesPerTreetopLeaf}$ cubes under each subtree.

Figure 7. Note that the Z-curve enters and leaves cubes (green circles) from each treetop leaf (blue boxes) exactly once.

Let us calculate how many descendants each intermediate cube of the octree has on deeper octree levels. Consider a treetop – a rooted subtree that contains the minimum number of octree cubes in the leaves, with the restriction that each leaf cube contains less than $N_{cubesPerTreetopLeaf}$ descendants in the original tree, see Fig. 6. In all experiments we used $N_{cubesPerTreetopLeaf} = 2^{24}$ because it is small enough to guarantee that each subsequent step will fit in 16 GB of RAM, but at the same time limits the number of leaves in a treetop to just a couple of thousands even on the largest datasets.

Due to out-of-core constraints, we can not estimate a global treetop by loading the whole octree into the memory. Therefore, we build independent treetops for all linear balanced octree parts, and then merge those treetops into a
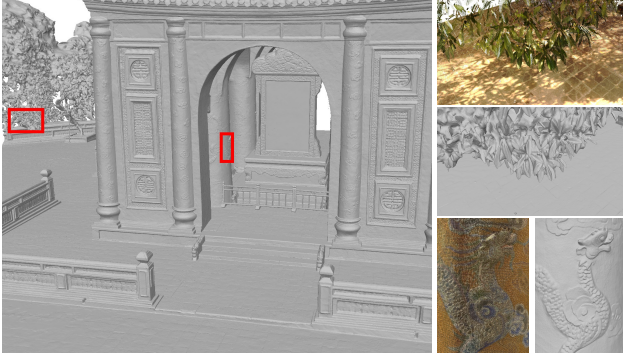
Figure 8. Tomb of Tu Duc LIDAR dataset. To the right – two closeups with colored LIDAR point clouds and resulting models.

Figure 9. To avoid having visible seams we update the indicator for all cubes inside the current leaf's border (set A) while the indicator for neighboring cubes outside of the border (set B) is frozen.

global one. At this stage, we can easily save indices of all covered relevant cubes for each treetop leaf. Moreover, these indices are consecutive due to Z-curve ordering of Morton codes – see Fig. 7. Hence, we only need to save two indices with each treetop leaf –indices of the first and the last relevant cubes from the linear balanced octree. This gives us an ability to load cubes relevant for current treetop leaf from balanced octree in IO-friendly consecutive way. In addition, we have strong guarantees that the number of such cubes is limited by $N_{cubesPerTreetopLeaf}$.

4.4. Computation of the histograms

Now we have the scene representation, provided by the balanced linear octree and its indexed treetop. As the next part of our method, we need to add votes of all distance fields to all relevant cubes in the octree.

Let us process all treetop leafs one by one and estimate relevant distance fields for each leaf, which is achieved simply by checking each distance field frustum for an intersection with the treetop leaf cube volume. Then, we can just load all relevant distance fields for each treatop leaf one by one and add their votes to all descendants of the current treetop leaf, like shown in the listing in Algorithm 1.

Note that at any moment during the computation of the histograms the memory contains no more than $N_{cubesPerTreetopLeaf}$ octree cubes and only a single distance field.

4.5. Functional Minimization

Now we need to iteratively minimize the TGV functional from Eq. 2. As shown in [32], it is highly beneficial to use a coarse-to-fine scheme (especially in regions with lack of data) for convergence speed. As we will see in this subsection – the scheme also helps to not introduce any seams between processed parts.

Suppose that we have already minimized the functional over the whole octree up to depth $level$. Now we want to execute primal-dual iterations at the depth $level + 1$ in an
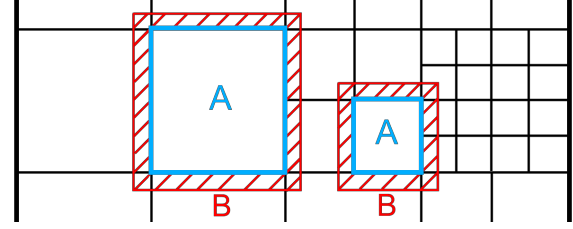
out-of-core way while not producing any seams between the parts. Like in the previous stage during the computation of the histograms, we process treetop leaves one by one. Let us load a treetop leaf's cubes in a set A and their neighbors in a set B. Now, we can iterate a numeric scheme like in [25] over the cubes from A with the only difference on the treetop leaf's border – we want our neighbors' indicator values u in cubes from B to be equal to indicator values of their parenting cubes, which were estimated on the previous level thanks to the coarse-to-fine scheme. I.e. we update the indicator for all cubes inside the current leaf's border (set A) while the indicator for neighboring cubes outside of the border (set B) is frozen - see two examples in Fig. 9.

By following this routine, at any given time we process only the cubes from a treetop leaf and their neighbors, and thus our memory consumption is bounded by their number. We do not face any misalignments on the surface next to treetop leaf borders due to explicit border constraints and the fact that the surface from one level to the next does not move far, but just progressively becomes more detailed, see Fig. 5 and details of the computation of the histograms in Section 3.4.

We notice that not so many cubes appear on the coarsest levels, meaning that each separate treetop leaf normally contains very few cubes. Therefore, we find it beneficial for performance to process multiple treetop leaves at once on the coarsest levels (w.r.t. leaves' total number of cubes on the current level).

4.6. Marching Cubes

As the last part, after estimating the indicator value u for all cubes of the octree, we need to extract a polygonal iso-surface corresponding to an indicator value $u = 0$. For that purpose, we can perform the marching cubes algorithm on per-leaf basis, using the same out-of-core tree partition as used before.

Marching cubes in a part of balanced octree is trivial: for each octree cube, we extract 3D-points between indicator values of different sign (i.e. points from iso-surface corresponding to zero indicator value), and then build their triangulation via dynamic programming by minimizing the total surface area, similar to [1].
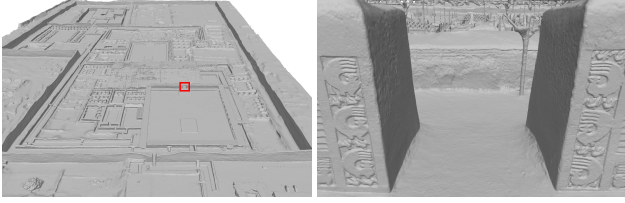
Figure 10. Palacio Tschudi dataset and a closeup of an entrance.

Table 1. Breakdown of Breisach dataset processing: 2111 photos, 2642 million cubes from input depth maps, 4 hours 20 minutes of processing on a computer with an 8-core CPU and a GeForce GTX 1080 GPU with the peak RAM usage 10.07 GB.

Processing stage	Time	Time in %
Linear octree + merge	30 + 11 min	11% + 4%
Balance octree + merge	7 + 11 min	3% + 4%
Index treetop	8 min	3%
Histograms (GPU)	49 min	19%
Primal-dual method (GPU)	88 min	34%
Marching cubes	59 min	22%

Note that neighboring surface parts have seamlessly matching borders, because both parts have the same indicator value u across the border due to stability of progressive refinement discussed in the previous subsection.

Finally, it is important to note that the number of triangle faces will be extra-large for any large dataset. We follow each octree part marching cubes with QSlim-based [11] decimation, which we modified with a strict border constraint, namely that no border edge (i.e. a triangle edge lying on a treetop leaf's cube face) should be ever collapsed. This way we achieve strict guarantees of a seamless surface between neighboring treetop leaves.

5. Results

We evaluated our method on an affordable computer with an 8-core CPU and a GeForce GTX 1080 GPU on five large datasets: Citywall[1] [10] and Breisach[2] [27] – two datasets from previous papers with high scale diversity, Copenhagen[3] [8] – large-scale aerial photos of the city (this dataset was additionally evaluated on a small cluster too), Palacio Tschudi[4] [7] and Tomb of Tu Duc[5] [6] (42 noise-free terrestrial LIDAR scans) – two large public datasets collected by CyArk and distributed by Open Heritage 3D.

The summary for these datasets presented in Table 2.

For photo-based datasets, we executed the *structure from motion* pipeline to estimate depth maps with SGM [16]

---

[1] https://www.gcc.tu-darmstadt.de/home/proj/mve/
[2] https://lmb.informatik.uni-freiburg.de/people/ummenhof/multiscalefusion/
[3] https://download.kortforsyningen.dk/content/skraafoto
[4] https://doi.org/10.26301/4h29-7e80
[5] https://doi.org/10.26301/n06n-qa49

method and evaluated our algorithm by using these depth maps as input. Note that to speed up the estimation of depthmaps, we downscaled original photos for some datasets – see Table 2. For the Tomb of Tu Duc LIDAR dataset, we converted each input scan into a 360-camera's depth map and used histogram votes with an increased weight – see the listing in Algorithm 1. Processing breakdowns for other datasets including the Copenhagen city dataset (evaluated twice – on an affordable computer and on a small cluster) with reconstruction results for many different city scenes are provided in the supplementary.

We ensured that the results are detailed and clean (see Fig. 4, 8, 10, 11) for all datasets, and that our method's peak memory usage was between 10 GB and 17 GB. Comparison with previous work [27], [24], presented in Table 3, shows that our method has significantly lower peak memory usage and is notably faster. To ensure that this speedup was not at a cost of quality, we compared our results with [27] in Fig. 4 and Fig. 11 (referred results were obtained with the software that their authors had used [6], we used the same depthmaps for quality comparison).

6. Conclusions

In this work, we present an out-of-core method for surface reconstruction from depth maps and terrestrial LIDAR scans. Our results have shown that the algorithm specifics do not increase the running time; instead, thanks to GPU-acceleration our implementation has proven to be much faster than previously published results on the datasets that we have used for testing. We have also shown that the quality of results is comparable to an in-core reconstruction method GDMR [27]. Note that an out-of-core balanced octree with treetop indexing is a rather general concept that can be used as a framework for different methods in a similar way as we used it with the TGV minimization method.

One of the main contributions of our work is an out-of-core framework for fast and detailed surface reconstruction. Our method is available as part of commercial software.

---

[6] We used pointfusion 0.2.0, publicly available at http://lmb.informatik.uni-freiburg.de/people/ummenhof/multiscalefusion
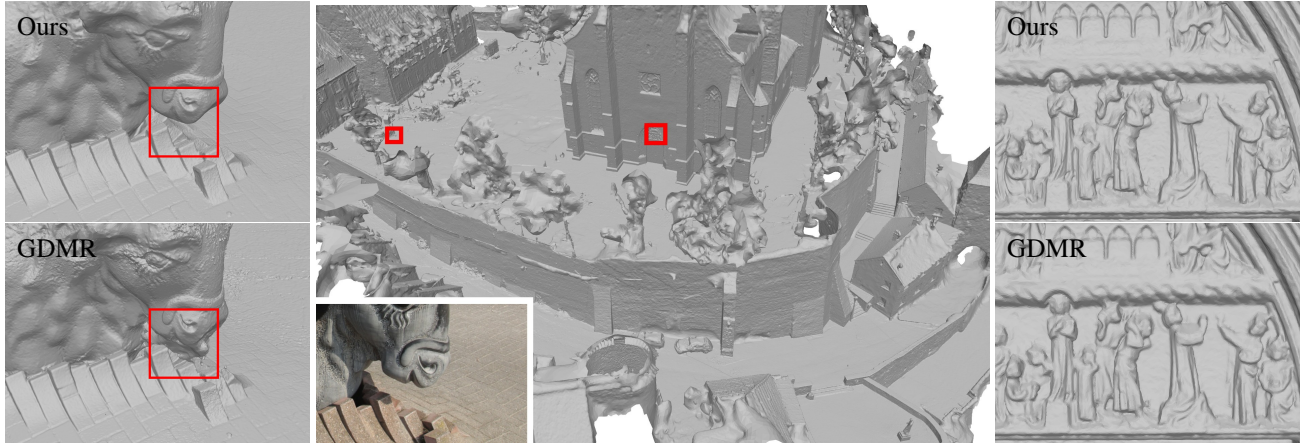
Figure 11. Results on the Breisach dataset and comparison with GDMR [27] results. The results have comparable quality. Note that thanks to strong visibility-based noise-filtering properties our method led to a cleaner space under the bull's nose.

Table 2. List of presented datasets. For depthmaps estimation speedup, we downscaled original photos for some datasets before running SGM-based [16] depthmap reconstruction. For example, original 2111 photos in Breisach dataset had the resolution of 5184x3456, and we downscaled them with x2 factor, down to 2592x1728 pixels. Note that the 'Initial cubes' column can be interpreted as 'the number of non-empty depth pixels in depthmaps', because each initial cube (before merging and octree balancing) corresponds to one sample from a depthmap.

Dataset name	Images resolution after downscale (and downscale factor)	Input data	Initial cubes	Merged and balanced cubes	Faces after marching cubes	Decimated faces	Peak RAM (GB)	Processing time
Citywall [10]	2000x1500 (x1)	564 depth maps	1205 mil	404 mil	135 mil	15 mil	13.17	63 min
Breisach [27]	2592x1728 (x2)	2111 depth maps	2642 mil	1457 mil	558 mil	57 mil	10.07	260 min
Tomb of Tu Duc (LIDAR) [6]	8000x4000 (x1)	42 LIDAR scans	661 mil	1304 mil	672 mil	48 mil	10.05	160 min
Palacio Tschudi [7]	1840x1228 (37%) 1500x1000 (63%) (x4)	13703 depth maps	16 billion	6 billion	3159 mil	243 mil	16.75	1213 min
Copenha-gen city [8]	3368x2168 (26%) 2575x1925 (74%) (x4)	27472 depth maps	28 billion	24 billion	7490 mil	267 mil	13.35	1758 min

Table 3. Comparison with the previous results - GDMR [27] and SSR 128K [24]. Note that SSR had 8.9 GB per-thread peak memory, and finished the reconstruction in 58.3 hours using 32 threads, so total peak memory could be estimated as about 32*8.9=285 GB. Also note that GDMR results for Breisach dataset were taken from the original paper, but because the authors did not mention memory and time results for the Citywall dataset – we provide our results of GDMR evaluation on a computer with an 8-core CPU starting from 1205 million input points.

Dataset name	Input data	GDMR Peak RAM	GDMR time	**Our** Peak RAM	**Our** time	SSR Peak RAM	SSR time
Citywall	564 depth maps	75 GB	19 h	13.17 GB	63 min	32*8.9 GB	58 h
Breisach	2111 depth maps	64 GB	76 h	10.07 GB	260 min	N/A	N/A

References

[1] Jules Bloomenthal. *Polygonization of implicit surfaces*. Citeseer, 1988. 6

[2] Matthew Bolitho, Michael Kazhdan, Randal Burns, and Hugues Hoppe. Multilevel streaming for out-of-core surface reconstruction. pages 69–78, 2007. 2

[3] Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE transactions on pattern analysis and machine intelligence*, 26(9):1124–1137, 2004. 2

[4] Kristian Bredies, Karl Kunisch, and Thomas Pock. Total generalized variation. *SIAM Journal on Imaging Sciences*, 3(3):492–526, 2010. 3

[5] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 303–312, 1996. 2

[6] CyArk. Complex of hu monuments - tomb of tu duc, vietnam, 2019. 7, 8

[7] CyArk. Palacio tschudi - chan chan, peru, 2020. 7, 8

[8] Danish Agency for Data Supply and Efficiency. Skraafoto of copenhagen, 2019. 7, 8

[9] Simon Fuhrmann and Michael Goesele. Floating scale surface reconstruction. *ACM Transactions on Graphics (ToG)*, 33(4):1–11, 2014. 2

[10] Simon Fuhrmann, Fabian Langguth, and Michael Goesele. Mve-a multi-view reconstruction environment. In *GCH*, pages 11–18, 2014. 2, 7, 8

[11] Michael Garland and Paul S Heckbert. Simplifying surfaces with color and texture using quadric error metrics. In *Proceedings Visualization'98 (Cat. No. 98CB36276)*, pages 263–269. IEEE, 1998. 7

[12] Andrew V Goldberg, Sagi Hed, Haim Kaplan, Pushmeet Kohli, Robert E Tarjan, and Renato F Werneck. Faster and more dynamic maximum flow by incremental breadth-first search. In *Algorithms-ESA 2015*, pages 619–630. Springer, 2015. 2

[13] Gottfried Graber, Thomas Pock, and Horst Bischof. Online 3d reconstruction using convex optimization. pages 708–711, 2011. 2

[14] Jiali Han and Shuhan Shen. Scalable point cloud meshing for image-based large-scale 3d modeling. *Visual Computing for Industry, Biomedicine, and Art*, 2(1):1–9, 2019. 2

[15] Vu Hoang Hiep, Renaud Keriven, Patrick Labatut, and Jean-Philippe Pons. Towards high-resolution large-scale multi-view stereo. pages 1430–1437, 2009. 2

[16] Heiko Hirschmuller. Stereo processing by semiglobal matching and mutual information. *IEEE Transactions on pattern analysis and machine intelligence*, 30(2):328–341, 2007. 2, 7, 8

[17] Michal Jancosek and Tomas Pajdla. Exploiting visibility information in surface reconstruction to preserve weakly supported surfaces. *International scholarly research notices*, 2014, 2014. 2

[18] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. 7, 2006. 1

[19] Andreas Kuhn, Heiko Hirschmüller, Daniel Scharstein, and Helmut Mayer. A tv prior for high-quality scalable multi-view stereo reconstruction. *International Journal of Computer Vision*, 124(1):2–17, 2017. 2

[20] Andreas Kuhn and Helmut Mayer. Incremental division of very large point clouds for scalable 3d surface reconstruction. pages 10–18, 2015. 2

[21] Shiwei Li, Sing Yu Siu, Tian Fang, and Long Quan. Efficient multi-view surface refinement with adaptive resolution control. pages 349–364, 2016. 2, 4

[22] Paul Merrell, Amir Akbarzadeh, Liang Wang, Philippos Mordohai, Jan-Michael Frahm, Ruigang Yang, David Nistér, and Marc Pollefeys. Real-time visibility-based fusion of depth maps. pages 1–8, 2007. 2

[23] Guy M Morton. A computer oriented geodetic data base and a new technique in file sequencing. 1966. 5

[24] Christian Mostegel, Rudolf Prettenthaler, Friedrich Fraundorfer, and Horst Bischof. Scalable surface reconstruction from point clouds with extreme scale and density diversity. pages 904–913, 2017. 2, 7, 8

[25] Thomas Pock, Lukas Zebedin, and Horst Bischof. Tgv-fusion. pages 245–258, 2011. 2, 3, 6

[26] Tiankai Tu and David R Ohallaron. Balance refinement of massive linear octree. 2004. 5

[27] Benjamin Ummenhofer and Thomas Brox. Global, dense multiscale reconstruction for a billion points. pages 1341–1349, 2015. 2, 3, 4, 7, 8

[28] Benjamin Ummenhofer and Thomas Brox. Global, dense multiscale reconstruction for a billion points. *International Journal of Computer Vision*, pages 1–13, 2017. 2

[29] Hoang-Hiep Vu, Patrick Labatut, Jean-Philippe Pons, and Renaud Keriven. High accuracy and visibility-consistent dense multiview stereo. *IEEE transactions on pattern analysis and machine intelligence*, 34(5):889–901, 2011. 2, 4

[30] Lance Williams. Pyramidal parametrics. In *Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 1–11, 1983. 3

[31] Christopher Zach. Fast and high quality fusion of depth maps. 1(2), 2008. 2, 3, 5

[32] Christopher Zach, Thomas Pock, and Horst Bischof. A globally optimal algorithm for robust tv-l1 range image integration. pages 1–8, 2007. 2, 3, 6

[33] Yang Zhou, Shuhan Shen, and Zhanyi Hu. Detail preserved surface reconstruction from point cloud. *Sensors*, 19(6):1278, 2019. 2