Large-Scale Multi-Resolution Surface Reconstruction from RGB-D Sequences

Frank Steinbrücker, Christian Kerl, Jürgen Sturm, and Daniel Cremers
Technical University of Munich
Boltzmannstrasse 3, 85748 Garching
*{steinbrf,kerl,sturmju,cremers}@in.tum.de

Abstract

We propose a method to generate highly detailed, textured 3D models of large environments from RGB-D sequences. Our system runs in real-time on a standard desktop PC with a state-of-the-art graphics card. To reduce the memory consumption, we fuse the acquired depth maps and colors in a multi-scale octree representation of a signed distance function. To estimate the camera poses, we construct a pose graph and use dense image alignment to determine the relative pose between pairs of frames. We add edges between nodes when we detect loop-closures and optimize the pose graph to correct for long-term drift. Our implementation is highly parallelized on graphics hardware to achieve real-time performance. More specifically, we can reconstruct, store, and continuously update a colored 3D model of an entire corridor of nine rooms at high levels of detail in real-time on a single GPU with 2.5GB.

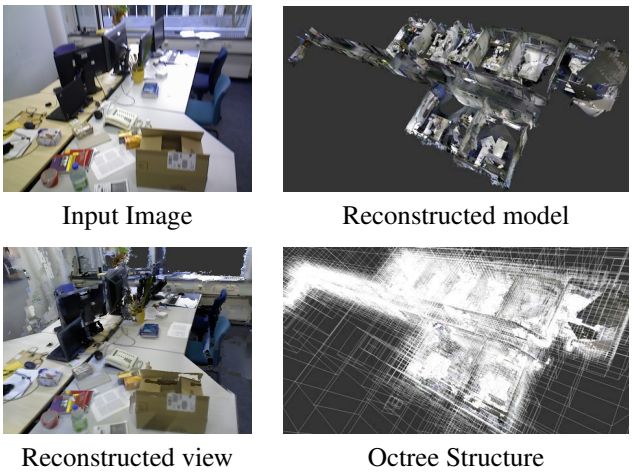


Figure 1: Reconstruction of an office floor consisting of 9 rooms over an area of 45m×12m×3.4m.

1. Introduction

Reconstructing the geometry and texture of the world from a sequence of images is among the fascinating challenges in computer vision. Going beyond the classical problem known as Simultaneous Localization and Mapping (SLAM) or Structure-from-Motion (SFM), we want to estimate the camera poses, the scene geometry and the scene-texture.

While impressive progress in this domain has been achieved over the last decade [2, 1, 9, 5], many of these approaches are based on visual keypoints that are reconstructed in 3D, which typically leads to sparse reconstructions in form of 3D point clouds. More recent methods based on depth images such as KinectFusion [12] aim at dense reconstruction using 3D voxel grids, which however requires a multiple in terms of memory and computational complexity. Various methods have been proposed to overcome this limitation, for example, by using rolling reconstruction volumes [14, 19] or octree data structures [7, 21]. However, in contrast to our approach, all of the above works are either not real-time [7], lack texture estimation [21], or do not support revisiting already tesselated volumes [14, 20]. Our approach integrates all of these features in a single system running at more than 15 Hz.

To estimate the camera poses, classical structure-from-motion approaches match visual features across multiple images and optimize the camera poses to minimize the re-projection errors. In contrast, recently upcoming dense methods aim at aligning the full image by minimizing the photometric error over all pixels [3, 13, 15, 18], thereby exploiting the available image information better than feature-based methods. It is important to note that approaches that track the camera pose with respect to the reconstructed model (such as all existing KinectFusion-based methods [12, 21, 20]) are inherently prone to drift. In contrast, we integrate dense image alignment in a SLAM framework to effectively reduce drift while keeping the advantages of dense image registration.

Our key contributions are:

*This work has partially been supported by the DFG under contract number FO 180/17-1 in the Mapping on Demand (MOD) project.

1. a fast, sparse, multi-resolution tree structure for geometry and texture reconstruction of large-scale scenes,

2. a SLAM system based on a dense image alignment to estimate a drift-free camera trajectory, with superior performance to several state-of-the-art methods.

An example of a reconstructed model of a large office floor is given in Figure 1: As it can be seen, the resulting model is globally consistent and contains fine details, while it still fits completely in the limited memory of a state-of-the-art GPU.

This paper is organized as follows: In Section 2, we describe how we achieve memory-efficient surface reconstruction using octrees on the GPU. In Section 3, we present how we extend dense tracking to compensate for drift by the detection of loop closures. In Section 4, we present an evaluation of our approach on various datasets, and close with a conclusion in Section 5.

2. Multi-Resolution Surface Reconstruction

In this section, we describe our approach to memory-efficient surface reconstruction. First, we provide a brief introduction to implicit surface representations based on signed distance functions. Subsequently, we replace the regular grid by an adaptive octree and a narrow-band technique to significantly reduce the memory consumption.

2.1. Signed Distance Volume

Following the works of [4, 12], we store our surface implicitly as a signed distance function in a 3D volume. The volume is approximated by a finite number of voxels. At every point in the volume the function indicates how far away the closest surface is. Points in front of an object have a negative sign and points inside a positive one. The zero crossing indicates the location of the surface. When considering a geometry that is updated and changed over time, the signed distance representation has the benefit of being able to handle arbitrary changes in the surface topology, in constrast to e.g. a surface mesh.

The signed distance function is incrementally constructed from a sequence of RGB-D images and associated camera poses. Given an RGB-D image at time t with a color valued image $\mathcal{I}_t^c$ and depth map $\mathcal{Z}_t$, the camera pose $\boldsymbol{T}_t$, and the intrinsic camera parameters, we integrate it into the volume using the following procedure.

For every voxel in the volume, we compute its center point in the camera frame $\mathbf{p}_c$

$$\mathbf{p}_c = \boldsymbol{T}_t \, \mathbf{p}. \tag{1}$$

Afterwards, we determine the pixel location $\mathbf{x}$ of the voxel center $\mathbf{p}_c$ in the depth map $\mathcal{Z}_t$ using the projection function of the standard pinhole camera model, i.e., $\mathbf{x} = \pi(\mathbf{p}_c)$. The measured point is reconstructed using the inverse projection function $\pi^{-1}(\mathbf{x}, Z)$:

$$\mathbf{p}_{\text{obs}} = \pi^{-1}(\mathbf{x}, \mathcal{Z}_t(\mathbf{x})). \tag{2}$$

The value of the signed distance function at the voxel center is determind by

$$\Delta_D = \max\{\min\{\Phi, |\mathbf{p}_c| - |\mathbf{p}_{\text{obs}}|\}, -\Phi\}. \tag{3}$$

The choice of the truncation threshold Φ depends on the expected uncertainty and noise of the depth sensor. The lower Φ is, the more fine scale structures and concavities of the surface can be reconstructed. If Φ is too low however, objects might appear several times in the reconstruction due to sensor noise or pose estimation errors. We chose it to be twice the voxel scale of the grid resolution for the experiments in this paper. Furthermore, we compute a weight for the measurement Δ_D using the weight function $w(\Delta_D)$, that expresses our confidence in the distance observation according to our sensor model. Figure 2 visualizes the truncated signed distance and the weight function used in our experiments. The distance $D(\mathbf{p}, t)$ and the weight $W(\mathbf{p}, t)$ stored in the voxel are updated using the following equations:

$$W(\mathbf{p}, t) = w(\Delta_D) + W(\mathbf{p}, t-1), \tag{4}$$

$$D(\mathbf{p}, t) = \frac{D(\mathbf{p}, t-1)W(\mathbf{p}, t-1) + \Delta_D w(\Delta_D)}{w(\Delta_D) + W(\mathbf{p}, t-1)}. \tag{5}$$

Similar to the distance update we compute the new voxel color $C(\mathbf{p}, t)$ as

$$C(\mathbf{p}, t) = \frac{C(\mathbf{p}, t-1)W(\mathbf{p}, t-1) + \mathcal{I}_t^c(\mathbf{x})w(\Delta_D)}{w(\Delta_D) + W(\mathbf{p}, t-1)}. \tag{6}$$

We use a weighting function that assigns a weight of 1 to all pixels in front of the observed surface, and a linearly decreasing weight behind the surface (cf. Figure 2):

$$w(\Delta_D) = \begin{cases} 1 & \text{if } \Delta_D < \delta \\ \frac{\Phi - \Delta_D}{\Phi - \delta} & \text{if } \Delta_D \geq \delta \text{ and } \Delta_D \leq \Phi. \\ 0 & \text{if } \Delta_D > \Phi \end{cases} \tag{7}$$

For the experiments in this paper we chose the value 0.005 for δ, which is one tenth of the voxel resolution.

2.2. Sparse Representation of the Distance Values

One of the key observations leading to a sparse geometry representation is the fact that all distance values at positions $\mathbf{p}$, that have an associated weight $W(\mathbf{p}, t) = 0$, do not need to be stored explicitly. As we require online-capability for our system, it is not feasable to perform an update in every voxel for the fusion of every new depth map. We rather want to restrict the computation effort to those parts in space

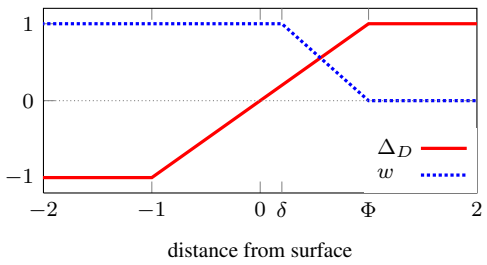


Figure 2: Visualization of the truncated signed distance function and the linear weighting function.

that are actually being updated. Therefore, we need to infer which voxels need to be updated from the camera pose and the depth map recorded at that pose. A naive narrow-band approach of a voxel array, storing the data for the voxel as well as its position, would require an iteration over all voxels to find the ones that need to be updated and therefore contradict this demand. Another naive approach of storing a full pointer-grid would save us the computation time for unnecessary voxel updates, but already a 4 byte pointer array of size 1024^3 requires 4GB of memory. For a 5mm resolution, this alone would restrict the scene to $(5\text{m})^3$, not even accounting for the actual distance, weight, and color values. To overcome both problems we propose to use an octree data structure to store the values: In a tree, the retrieval of voxels required for update is feasible in logarithmic time and the memory required for storage of the tree is linear in the amount of leaves.

Multiscale Approach For disparity-based depth sensors, the depth Z in each pixel is a reciprocal function of a measured disparity value d. In a standard pinhole camera model, this dependency is given as

$$Z = \frac{fB}{d}, \tag{8}$$

where f is the focal length and B the baseline. A common assumption is that the disparity measurements of the sensor lie in the interval

$$[d - \sigma, d + \sigma] \tag{9}$$

around the true disparity d. Substituting (8) in (9) and applying a first order Taylor expansion in σ we get the depth interval

$$\left[Z + \frac{Z^2}{fB}, Z - \frac{Z^2}{fB} \right]. \tag{10}$$

For further information on this topic we refer to [8]. As (10) shows the error of the depth measurement grows quadratically with the measured depth. Accordingly, we update the signed distance function at a coarser resolution for points far away from the camera, saving memory. To store values at a lower resolution we allocate leaves at intermediary

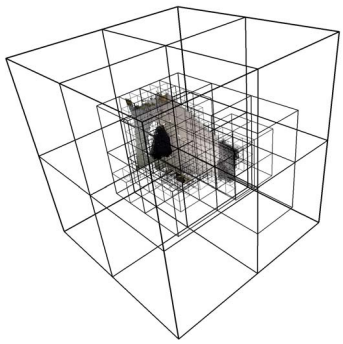


Figure 3: Example of a small octree. We store the geometry at multiple levels of resolution.

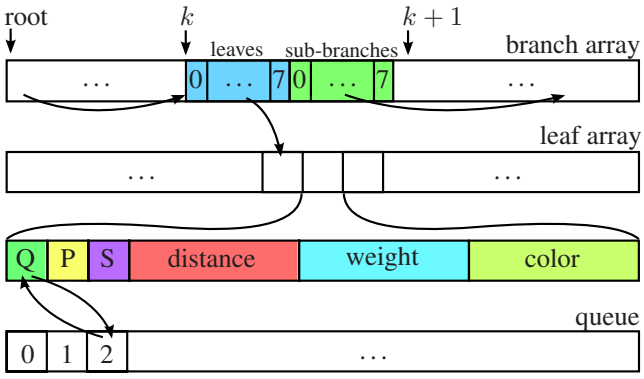


Figure 4: Octree representation in memory. We store all branches in one array. One branch comprises 8 pointers to sub-branches and 8 pointers to its leaves. All leaves are stored in a second array. For fast access during the integration of a new RGB-D image a queue maintains pointers to the leaves that have to be updated.

branches that are located higher in the tree. The estimated error interval described above is used to determine the scale, i.e., the octree level.

Structure of the Tree In an octree every branch has 8 children, either leaves or sub-branches, due to the binary sub-division in every dimension. The spatial structure is depicted in Figure 3. In our representation every intermediary branch contains not only its children, but also 8 leaves to enable storage of our multi-scale signed distance function.

Figure 4 depicts the memory layout of our octree on the GPU. We store the branches in a 4 byte pointer array. Each branch has 8 pointers for its sub-branches and 8 pointers for its leaf nodes. The branch pointers hold offsets into the same array. In contrast, the leaves are stored in a separate array. Every leaf is a brick containing 8^3 voxels. Each voxel contains its current distance, color and weight. In total, each voxel requires 14 bytes of memory. Additionally, we store some meta data for every brick comprising the position in

the volume, the scale, and a pointer into the update queue. Overall one brick occupies 7,180 bytes , 512*14 bytes for the voxels, 3*2 bytes for the position of the brick, 2 bytes for its scale, and 4 bytes for its queue index. The queue contains pointers to all leaves, which have to be updated to integrate a new RGB-D image. In the following we explain the steps to fuse a new RGB-D image into the octree.

Traversal of a Depth Map in the Tree As explained in Section 2.1, for a given depth map and camera pose, only the voxels around the surface of that depth map are assigned with positive weights and are thus updated. Therefore, only a small number of voxels, the so-called "narrow band" need to be allocated on the first access and updated later on.

For a depth map, we iterate over all pixels in parallel on the GPU. If a pixel contains a valid depth measurement, we compute its corresponding point in 3D with the given camera pose and intrinsic parameters. Depending on the measured depth, we compute the bandwidth and the leaf scale for this point. Then we intersect a box with a side length of twice the bandwidth around the point with the tree. As it would be too computationally expensive to traverse the tree from top to bottom for every leaf intersecting the box, we perform a depth-first-search on the tree. The only values that need to be stored in this search for every level in the tree are a 4 byte index of the current branch and one byte for the children remaining to be searched in that level. These 5 bytes per level comfortably fit into the shared and register memory of newer GPUs.

For every branch, we check which child branches intersect the box and allocate them if necessary. Once we reach the desired leaf level in our search, we check whether the desired leaf has already been allocated (i.e. the corresponding index in the branch array has been set) and allocate it otherwise. After the leaf has been allocated, we check whether it is already in the queue and if not, insert its leaf index at the end of the queue and update the queue index in the leaf structure.

Note that we already allocate all GPU memory during initialization to avoid memory allocation during reconstruction. Allocating new branches and leaves in the tree and adding leaves to the queue are both performed by atomic additions on three global counter variables.

Update of the Distance, Weight, and Color Values After the traversal step all new branches and leaves intersecting the band of the current surface have been allocated and all the leaves, previously allocated and newly allocated ones, have been placed in the queue.

Given the index, position, and scale of every leaf we now efficiently update the position, weight, and color values in every voxel in the queue according to equations (4), (5) and (6). This step is performed in parallel for all voxels, with one thread block per leaf in the queue. The update involves reading the leaf index and the leaf position and scale under that index from memory, which is an O(1) operation. Projecting the voxel positions into the images is done with high efficiency using the GPU texture hardware.

Growing the Tree In a large-scale setting, its not feasable to compute the bounding box of the final reconstruction beforehand. Therefore, we start with the geometry of one depth map and subsequently grow the tree, if the geometry of a new depth map exceeds the current bounding volume.

This involves adding a new root node, inserting the current root node as one of its children, shifting all branches and adding an increment to all leaf positions. This is an O(n) operation in the number of leaves, however it amortizes over the number of images because the tree will only grow an O(log n) number of times.

To summarize, we introduced in this section our dense reconstruction algorithm using a sparse, adaptive tree structure. It allows us to fuse large-scale RGB-D sequences in terms of structure and texture with high spatial resolution.

3. Dense RGB-D SLAM

The surface reconstruction approach described above requires an accurate camera pose for each RGB-D image in the sequence. Our goal is to estimate the camera motion solely from this sequence. The motion estimation algorithm comprises two main parts: We employ a dense image alignment method to find the relative transformation between two RGB-D frames. Based on this, we construct a pose graph and add loop-closure edges where we detect them. After construction, we optimize the pose graph to compensate for drift and to obtain a metrically correct camera trajectory.

3.1. Dense Alignment of RGB-D Frames

We seek to estimate the camera motion $\boldsymbol{T}^*$ between two consecutive grey valued intensity images $\mathcal{I}_1$ and $\mathcal{I}_2$ with corresponding depth maps $\mathcal{Z}_1, \mathcal{Z}_2$. Our dense motion estimation algorithm is based on the following ideas: For every scene point $\mathbf{p}$ observed from the first camera pose with an associated intensity and given the correct motion $\boldsymbol{T}^*$ we can compute its pixel location in the second intensity image. In the ideal case we can formulate the photo-consistency constraint, that the intensity measurement in the first image $\mathcal{I}_1(\mathbf{x})$ should be equal to the intensity measured at the transformed location $\mathbf{x}'$ in the second image, i.e., $\mathcal{I}_1(\mathbf{x}) \stackrel{!}{=} \mathcal{I}_2(\mathbf{x}')$. This constraint should hold for every pixel. Therefore, we can use it to compute the camera motion $\boldsymbol{T}^*$ given two intensity images and one depth map. A similar constraint can be formulated for the depth measurements. For every scene point observed from the first camera

pose, we can predict the measured depth value and pixel location in the second depth map given the correct motion $\boldsymbol{T}^*$. Using this constraint we can compute the motion $\boldsymbol{T}^*$ given two depth maps. In the following we formalize these ideas into a non-linear minimization problem to estimate the camera motion $\boldsymbol{T}^*$ from two RGB-D pairs.

Photometric and Geometric Error We define the photometric error as:

$$e_{\mathcal{I}}(\boldsymbol{T}, \mathbf{x}) = \mathcal{I}_2\big(\pi(\boldsymbol{T}\,\mathbf{p})\big) - \mathcal{I}_1(\mathbf{x}) \tag{11}$$

where $\boldsymbol{T}$ is the rigid body motion represented as 4×4 homogeneous transformation matrix, $\pi(\mathbf{p})$ is the projection function of the pinhole camera model, and $\mathbf{p}$ is the 3D point reconstructed from pixel $\mathbf{x}$ and its depth measurement $\mathcal{Z}_1(\mathbf{x})$ using the inverse projection function $\pi^{-1}(\mathbf{x}, Z)$. Similarly, the geometric error is given as:

$$e_{\mathcal{Z}}(\boldsymbol{T}, \mathbf{x}) = \mathcal{Z}_2\big(\pi(\boldsymbol{T}\,\mathbf{p})\big) - \big[\boldsymbol{T}\,\mathbf{p}\big]_Z \tag{12}$$

where $[\mathbf{p}]_Z$ extracts the Z coordinate of a point $\mathbf{p}$. The parametrization of the rigid body motion as a transformation matrix is problematic for optimization as it has 16 parameters, but only six degrees of freedom. Therefore, we use the minimal twist parametrization $\boldsymbol{\xi}$ provided by the Lie algebra $\mathfrak{se}(3)$ associated with the group of rigid body motions $\mathrm{SE}(3)$. The transformation matrix is related to the twist parameters by the matrix exponential: $\boldsymbol{T} = \exp(\hat{\boldsymbol{\xi}})$.

Non-linear Minimization We adapt the probabilistic framework proposed in [10] to combine the photometric and geometric error. This stands in contrast to [18, 19] where the terms are additively combined using a heuristically chosen weight. To this end, we assume the combined error $\boldsymbol{r} = (r_{\mathcal{I}}, r_{\mathcal{Z}})^\mathsf{T}$ to be a bivariate random variable following a t-distribution, i.e., $\boldsymbol{r} \sim \mathrm{p_t}(\boldsymbol{\mu_r}, \boldsymbol{\Sigma_r}, \nu)$. The t-distribution has mean $\boldsymbol{\mu_r}$, scale matrix $\boldsymbol{\Sigma_r}$, and ν degrees of freedom. We fix the mean to zero and the degrees of freedom to five. By minimizing the negative log-likelihood of $\mathrm{p}(\boldsymbol{\xi} \mid \boldsymbol{r})$ we obtain the following non-linear, weighted least squares problem:

$$\boldsymbol{\xi}^* = \arg\min_{\boldsymbol{\xi}} \sum_i^n w_i\, \boldsymbol{r}(\boldsymbol{\xi}, \mathbf{x}_i)^\mathsf{T}\, \boldsymbol{\Sigma_r}^{-1}\, \boldsymbol{r}(\boldsymbol{\xi}, \mathbf{x}_i) \tag{13}$$

where n is the number of pixels. The per pixel weight w_i is defined as:

$$w_i = \frac{\nu + 1}{\nu + \boldsymbol{r}_i\, \boldsymbol{\Sigma_r}^{-1}\, \boldsymbol{r}_i}. \tag{14}$$

The resulting normal equations are:

$$\begin{gathered} \boldsymbol{A}\Delta\boldsymbol{\xi} = \boldsymbol{b} \\ \sum_i^n w_i \boldsymbol{J}_i^\mathsf{T} \boldsymbol{\Sigma_r}^{-1} \boldsymbol{J}_i \Delta\boldsymbol{\xi} = -\sum_i^n w_i \boldsymbol{J}_i^\mathsf{T} \boldsymbol{\Sigma_r}^{-1} \boldsymbol{r}_i \end{gathered} \tag{15}$$

where $\boldsymbol{J}_i$ is the 2×6 Jacobian matrix containing the derivatives of the photometric and geometric error with respect to the motion parameters $\boldsymbol{\xi}$. We iteratively update and solve the normal equations for parameter increments $\Delta\boldsymbol{\xi}$. The scale matrix $\boldsymbol{\Sigma_r}$ of the error distribution and the weights are re-estimated at every iteration. Furthermore, we employ a coarse-to-fine scheme to account for a larger range of camera motions.

Parameter Uncertainty We assume the estimated parameters $\boldsymbol{\xi}$ to be normally distributed with mean $\boldsymbol{\xi}^*$ and covariance $\boldsymbol{\Sigma_\xi}$. The inverse of the $\boldsymbol{A}$ matrix in the normal equations gives an estimate of the parameter covariance, i.e., $\boldsymbol{\Sigma_\xi} = \boldsymbol{A}^{-1}$.

3.2. Keyframe-based Pose SLAM

The presented visual odometry method has inherent drift, because of a small error in every frame-to-frame match, that accumulates over time. The elimination of the drift is an important prerequisite to obtain a metrically correct reconstruction. Therefore, we embed our dense visual odometry method into a SLAM system.

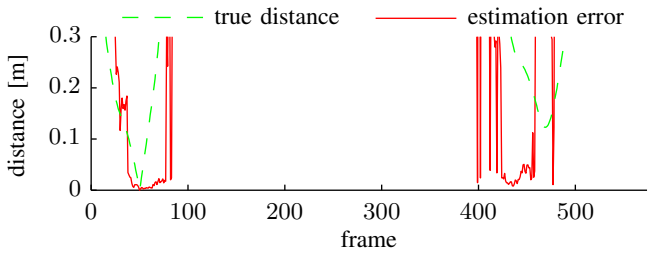
To eliminate local drift, we match the latest frame against a keyframe instead of the previous frame. As long as the camera stays close enough to the keyframe, no drift is accumulated. From all keyframes we incrementally build a map of the scene. Whenever a new keyframe is added to the map, we search for loop closures to previously added keyframes. The loop closures provide additional constraints, which allow to correct the accumulated drift. In the following we describe a consistent measure to select new keyframes and detect loop closures. Afterwards, we describe our map representation and global optimization.

Keyframe Selection Different strategies for keyframe selection exist. Common approaches use a threshold on a distance measure, e.g., the number of frames or translational distance between the frames. In contrast, we want a measure taking into account how well the motion between the current keyframe and latest frame could be estimated. The naive approach of comparing the final error values (cf. (13)) is not applicable, because the scale matrix $\boldsymbol{\Sigma_r}$ varies between different frame pairs.

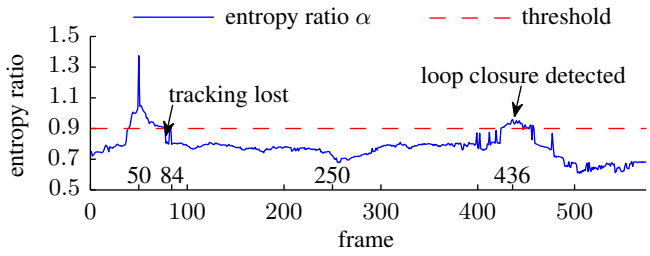
In contrast, we found a relationship between the entropy of the parameter distribution $H(\boldsymbol{\xi})$ and the trajectory error. The differential entropy of a random variable $\boldsymbol{x}$ having a multivariate normal distribution with m dimensions and covariance $\boldsymbol{\Sigma}$ is defined as:

$$\begin{aligned} H(\boldsymbol{x}) &= 0.5\, m\, \big(1 + \ln(2\,\pi)\big) + 0.5\, \ln(|\boldsymbol{\Sigma}|) \\ H(\boldsymbol{x}) &\propto \ln(|\boldsymbol{\Sigma}|). \end{aligned} \tag{16}$$

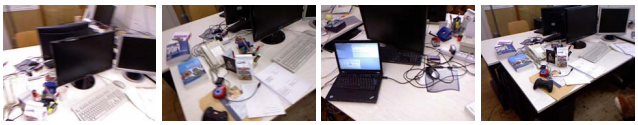
Dropping the constant terms it is proportional to the natural logarithm of the determinant of the covariance matrix. The



(a) error of the estimate w.r.t. frame 50



(b) uncertainty of the estimate w.r.t. frame 50



(c) Frame 50 (d) Frame 84 (e) Frame 250 (f) Frame 436

Figure 5: Frame 50 of the fr1/desk dataset matched against all frames of the dataset. Plot (a) shows the groundtruth distance of each frame to frame 50 and the error in the estimate. Note that the upper part of the plot above 0.3m has been cut off. Plot (b) displays the entropy ratio α. High entropy ratio values coincide with low error in the estimate. The second peak in the entropy ratio indicates a detected loop closure.

entropy of the estimated motion parameters is then $H(\boldsymbol{\xi}) \propto \ln(|\boldsymbol{\Sigma}_{\boldsymbol{\xi}}|)$. As the parameter entropy varies between different scenes, a direct thresholding is not applicable. Therefore, we use the ratio α between the parameter entropy of the first estimate towards the k^{th} keyframe $\boldsymbol{\xi}_{k:k+1}$ and the current one $\boldsymbol{\xi}_{k:k+l}$, i.e.,

$$\alpha = \frac{H(\boldsymbol{\xi}_{k:k+1})}{H(\boldsymbol{\xi}_{k:k+j})}. \tag{17}$$

The reasoning is that the first frame matched against a keyframe is closest and the parameter estimate is therefore the most accurate. Figure 5b shows a plot of the entropy ratio for frame 50 of the fr1/desk sequence matched against all other frames and Figure 5a displays the translational error in the estimate compared to the groundtruth. Its clearly visible, that high values of entropy ratio coincide with small errors in the estimate.

Loop Closure Detection The entropy ratio criterion can also be applied to detect loop closures. As Figure 5b shows the entropy ratio rises again around frame 436 and the error in the estimate drops as the camera returns to the previously visited region. To compute the entropy ratio with a loop closure candidate we do not use the parameter entropy of the first match, but the average entropy of all successful matches against the keyframe, i.e., in (17) instead of $H(\boldsymbol{\xi}_{k:k+1})$ we use:

$$H_{\text{avg}} = \frac{1}{l}\sum_{j=1}^{l} H(\boldsymbol{\xi}_{k:k+j}). \tag{18}$$

The argument here is similar, that the frames matched against the keyframe were closest and the estimates have high accuracy. Furthermore, our keyframe selection criterion ensures that there are no outliers in the averaging. As this criterion requires actual parameter estimation linear search over all existing keyframes becomes quickly intractable. Therefore, we limit the number of loop closure candidates by only considering keyframes in a certain distance to the new keyframe. Afterwards, we first estimate the motion parameters on a coarse scale and check the entropy ratio criterion. If this first test succeeds, we estimate the final parameters and test again. In case this test is also successful we add the loop closure constraint to the map.

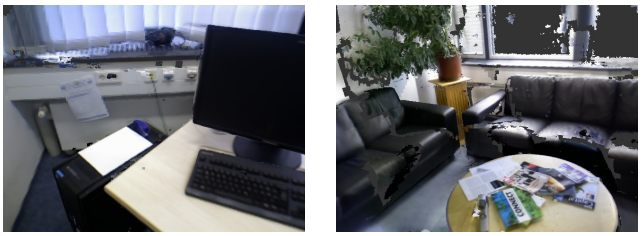
Map Representation and Optimization We represent the map as a graph. The vertices represent camera poses and the edges correspond to relative transformations between two RGB-D images estimated by our dense visual odometry algorithm. Every edge is weighted with the estimated motion covariance $\boldsymbol{\Sigma}_{\boldsymbol{\xi}}$. The camera poses are optimized by minimizing the squared error in the graph. The corrections are distributed according to the weights of the edges. After estimating the pose of the last RGB-D image when processing a dataset, we search for additional loop closures over all keyframes and optimize the final graph with a larger number of iterations than during online tracking. For the implementation of the graph structure and optimization we use the g2o framework [11].

In this section, we described a SLAM system based on a dense visual odometry method, which outputs a metrically correct trajectory. The optimized camera poses are the input to the proposed surface reconstruction algorithm.

4. Results

We evaluated the performance of our dense visual SLAM system on the TUM RGB-D benchmark [17]. Furthermore, we compare our results to results obtained with several state-of-the-art systems. These include the RGB-D SLAM system [6], Multi-Resolution Surfel Maps (MRSMap) [16], and the open-source implementation of KinectFusion (KinFu) [12] included in the pointcloud library (PCL).

Input image



Reconstructed view

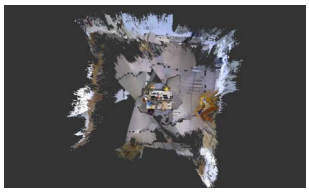
Novel view

Figure 6: Sample reconstructions of two different sequences (left and right columns). Top row: Example input image. Middle row: Visualization of the 3D model from the same viewpoint. Bottom row: Visualization from a different viewpoint.

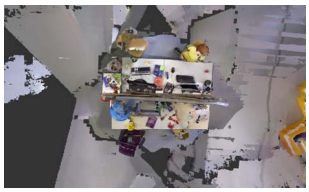
This set of experiments was conducted on a PC with Intel Core i7-2600 CPU with 3.40GHz and 16GB RAM. Table 1 shows the results. As an evaluation metric we use the root mean square error (RMSE) of the absolute trajectory error (ATE). In 9 out of 10 datasets our method outperforms the existing state-of-the-art methods. Especially on complex trajectories like fr1/room and fr1/teddy, our approach demonstrates a significant improvement compared to existing feature-based systems.

The whole visual SLAM system runs on the CPU. The frontend and backend run in separate threads. The dense visual odometry runs at 25Hz on a single CPU core. The runtime of the backend depends on the number of keyframes and loop closure constraints, but typical average timings for the insertion of a keyframe are between 100ms to 200ms.

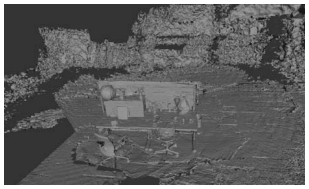
Next to the evaluation on the benchmark datasets, we recorded an office scene of approximately 45m×12m×3.4m consisting of 24076 images. We estimated the camera poses and reconstructed the scene with a voxel resolution of 5mm. As we use a multi-scale



Entire reconstruction



Top view



Flat-shaded side view

Flat-shaded colored closeup

Figure 7: Geometry reconstructions of the fr3/office dataset.

approach, not all the geometry is stored at all resolutions. For example, at a resolution of 5mm, we only consider depth measurements up to 1m distance of the camera. Figure 1 shows a view of the entire reconstructed geometry and the tree structure it is embedded in. The ceiling of the offices has been removed in a postprocessing step for a better view of the office interior from this position. We used the marching cubes algorithm to create a mesh at the 0-isolevel of the signed distance function for visualization. Each brick is reconstructed at finest resolution and voxels with zero weight are filled from coarser levels, if available. Figure 6 shows some comparisons of RGB images at different camera positions on the trajectory. It shows that we are able to reconstruct fine details in the scene where they are available, even though the entire scene fits into approximately 2.5GB GPU memory. The fusion was performed on an NVIDIA GTX680 GPU at an average of 220 frames per second (without post-processing such as marching cubes and visualization).

5. Conclusion

In this paper, we presented a method for the reconstruction of large scenes at high detail from an RGB-D sequence in real-time on a standard desktop PC. We construct a pose graph using dense image alignment that we optimize to eliminate for drift and to establish a metrically correct camera trajectory. Subsequently, we fuse both the depth and the color values of all frames into a multi-scale oct-tree structure that allows us to obtain high accuracies while keeping the memory usage low. In extensive experiments, we demonstrated that our approach outperforms existing feature-based methods on publicly available benchmark sequences. Furthermore, we presented reconstructions of large scenes showing both the preservation of fine

Dataset	Images	Processing Time				Memory	Absolute Trajectory Error (RMSE)			
		Acq.	SLAM	Geo. Fusion	Total Speed		Ours	RGB-D SLAM	MRSMap	KinFu
fr1/xyz	792	26s	44s	2.46s	16.7Hz	65MB	**0.013m**	0.014m	0.013m	0.026m
fr1/rpy	694	23s	41s	2.94s	15.7Hz	113MB	**0.021m**	0.026m	0.027m	0.133m
fr1/desk	573	19s	30s	1.89s	17.8Hz	135MB	**0.021m**	0.023m	0.043m	0.057m
fr1/desk2	620	20s	38s	2.54s	21.9Hz	186MB	**0.027m**	0.043m	0.049m	0.420m
fr1/room	1352	45s	72s	4.96s	17.5Hz	453MB	**0.054m**	0.084m	0.069m	0.313m
fr1/360	744	24s	72s	2.79s	9.6Hz	333MB	0.073m	0.079m	**0.069m**	0.913m
fr1/teddy	1401	46s	111s	4.78s	11.9Hz	248MB	**0.036m**	0.076m	0.039m	0.154m
fr1/plant	1126	37s	62s	4.15s	16.8Hz	221MB	**0.021m**	0.091m	0.026m	0.598m
fr2/desk	2893	96s	101s	12.0s	25.5Hz	178MB	**0.019m**	-	0.052m	-
fr3/office	2488	82s	106s	10.8s	21.0Hz	297MB	**0.030m**	-	-	0.064m
average	1286	41s	67s	4.93s	17.4Hz	223MB	**0.026m**	0.054m	0.043m	0.297m

Table 1: Quantitative evaluation of our system. From left to right: Number of images in the dataset, acquisition time of the dataset, processing time for SLAM, geometry fusion time of the dataset, average time for fusing one depth map, memory requirements for reconstruction, absolute trajectory error comparison of our method against the state-of-the-art.

details and the global consistency. In the future, we plan to integrate photoconsistency constraints in the geometric reconstruction to complement the depth measurements. With this work, we hope to contribute to the development of a mobile 3D scanner that can be used to acquire accurate 3D models of entire buildings.

References

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

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

[3] A. Comport, E. Malis, and P. Rives. Accurate Quadri-focal Tracking for Robust 3D Visual Odometry. In *ICRA*, 2007.

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

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

[6] N. Engelhard, F. Endres, J. Hess, J. Sturm, and W. Burgard. Real-time 3D visual SLAM with a hand-held camera. In *RGB-D Workshop on 3D Perception in ERF*, 2011.

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

[8] D. Gallup, J. Frahm, P. Mordohai, and M. Pollefeys. Variable baseline/resolution stereo. 2008.

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

[10] C. Kerl, J. Sturm, and D. Cremers. Robust odometry estimation for RGB-D cameras. In *ICRA*, 2013.

[11] R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard. g2o: A general framework for graph optimization. In *ICRA*, 2011.

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

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

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

[15] F. Steinbrücker, J. Sturm, and D. Cremers. Real-time visual odometry from dense RGB-D images. In *Workshop on Live Dense Reconstruction with Moving Cameras at ICCV*, 2011.

[16] J. Stückler and S. Behnke. Integrating depth and color cues for dense multi-resolution scene mapping using rgb-d cameras. In *MFI*, 2012.

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

[18] T. Tykkälä, C. Audras, and A. Comport. Direct iterative closest point for real-time visual odometry. In *Workshop on Computer Vision in Vehicle Technology at ICCV*, 2011.

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

[20] T. Whelan, M. Kaess, M. Fallon, H. Johannsson, J. Leonard, and J. McDonald. Kintinuous: Spatially extended KinectFusion. In *RSS Workshop on RGB-D: Advanced Reasoning with Depth Cameras*, 2012.

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