Parametric Surface Representation with Bump Image for Dense 3D Modeling Using an RBG-D Camera

Diego Thomas[1,2] · **Akihiro Sugimoto**[1]

Received: 23 September 2015 / Accepted: 1 November 2016 / Published online: 19 November 2016

Abstract When constructing a dense 3D model of an indoor static scene from a sequence of RGB-D images, the choice of the 3D representation (e.g. 3D mesh, cloud of points or implicit function) is of crucial importance. In the last few years, the volumetric truncated signed distance function (TSDF) and its extensions have become popular in the community and largely used for the task of dense 3D modelling using RGB-D sensors. However, as this representation is voxel based, it offers few possibilities for manipulating and/or editing the constructed 3D model, which limits its applicability. In particular, the amount of data required to maintain the volumetric TSDF rapidly becomes huge which limits possibilities for portability. Moreover, simplifications (such as mesh extraction and surface simplification) significantly reduce the accuracy of the 3D model (especially in the color space), and editing the 3D model is difficult. We propose a novel compact, flexible and accurate 3D surface representation based on parametric surface patches augmented by geometric and color texture images. Simple parametric shapes such as planes are roughly fitted to the input depth images, and the deviations of the 3D measurements to the fitted parametric surfaces are fused into a geometric texture image (called the Bump image). A confidence and color texture image are also built. Our 3D scene representation is accurate yet memory efficient. Moreover, updating or editing the 3D model becomes trivial since it is reduced to manipulating 2D images. Our experimental results demonstrate the advantages of our proposed 3D representation through a concrete indoor scene reconstruction application.

Keywords 3D modeling · Consumer depth camera · Bump image · Real-time · Parametric surface

Communicated by S.-C. Zhu.

✉ Diego Thomas
diegot.thomas@gmail.com

Akihiro Sugimoto
sugimoto@nii.ac.jp

1 National Institute of Informatics, Tokyo, Japan

2 Present Address: Kyushu University, Fukuoka, Japan

1 Introduction

The construction of fidelity 3D models from RGB-D measurements is of wide interest for the computer vision community, with various potential applications. For example, 3D models of real scenes can be used in augmented and virtual reality for serious games or remote communication. With recent efforts on developing inexpensive depth sensors such as the Microsoft Kinect camera or the Asus Xtion Pro camera (called RGB-D cameras), capturing depth information in indoor environment becomes an easy task. This new set-up opens new possibility for 3D modeling, and several softwares have been already proposed to realize live 3D reconstruction using RGB-D cameras (Davison et al. 2007; Neibner et al. 2013; Newcombe et al. 2011; Weise et al. 2009). The output of a 3D modeling software is a mathematical representation of the 3D surfaces of a target scene. As a consequence the choice of the 3D scene representation is of crucial importance and should be made carefully depending on the type of sensor used (e.g., cheap depth sensor with video frame rate and noisy depth images or expensive laser scanner with low frame rate but accurate depth images) and depending on the type of scene to reconstruct (e.g., static man-made indoor scene or dynamic outdoor scene). This choice will impact on what tools can be used and what options are enabled.

In general, the 3D modeling process using RGB-D cameras can be divided into 4 steps: (1) RGB-D measurement acquisition at the video frame rate; (2) camera tracking (Lowe 1999; Segal et al. 2009); (3) integration (or fusion) of aligned RGB-D measurements (Chen et al. 2013; Neibner et al. 2013; Newcombe et al. 2011; Weise et al. 2009) and (4) 3D surface reconstruction (3D textured mesh generation for example) (Kazhdan et al. 2006; Pfister et al. 2000). The objective is then to carry out these tasks with both computational efficiency and high accuracy. Computational speed is important because we capture 30 frames per second, so each image needs to be processed quickly in order to process the full sequence in a reasonable amount of time. Accuracy of the output 3D model is also important; in particular when we want it to be usable in simulations or visualisation tasks.

Much work has been proposed that dwells on improving accuracy of the generated 3D models, but it is only recent that a few work addressed the problem of compactness of the 3D representation (Chen et al. 2013; Henry et al. 2013; Neibner et al. 2013). Compactness of the generated 3D models is of outmost importance for scalability (when building large 3D environments) and also for remote user interaction (when the reconstructed 3D model needs to be lively streamed through the internet). Most of state-of-the-art methods employ implicit volumetric models to represent a 3D scene or objects. The main advantage of volumetric models is accuracy and easiness for incremental update. The drawback of volumetric models, however, is that they require large amount of memory and that they are not well suited for manipulation or edition. On the other hand, parametric surface models like splines or subdivision surfaces are compact representations that have been widely used in the computer graphics community when manipulating 3D scenes, but they are not used for 3D modeling using an RGB-D camera and deserve further investigation.

In this paper, we introduce a new compact and flexible parametric surface representation of the static scene that is well adapted for 3D modeling using RGB-D cameras. Our proposed representation consists of a set of parametric surface patches that requires low memory use and, nevertheless, realises accurate and efficient 3D modeling from an RGB-D image sequence. The main idea is to represent a static scene as a set of parametric surface patches to each of which we attach as its attributes three images describing the local geometry, color and accumulated confidence. This representation is flexible in that we can easily build and update the 3D representation and more importantly, with the attributes, we can always recover the full 3D information of the scene or objects while reducing required memory.

The advantages of our proposed 3D representation over state-of-the-art are threefold: (1) its low memory usage enables live streaming of constructed dense 3D models through the standard network; (2) generated 3D models are segmented, which allows reasoning at the level of not points but objects in post-processing (e.g. loop closure, object selection/removal); (3) the fine 3D geometry is encoded into 2D images, and thus edition becomes easy since it is reduced to manipulating 2D images. We remark that a part of this work has been reported in Thomas and Sugimoto (2013).

2 Related Work

In general, when constructing a 3D model using an RGB-D camera, two major questions arise: (1) how to represent the 3D model? and (2) how to update the 3D model with incoming data? We discuss the state-of-the-art on these two problems in this section.

2.1 3D Representations for 3D Modeling

A critical question that all 3D modeling softwares need to carefully answer is: what 3D representation should we use during the reconstruction process? The 3D representation should be compact, accurate and easy to update with live measurements. In general two different kinds of representation are used: volumetric models and surface models.

2.1.1 Volumetric Models

Newcombe et al. (2011) proposed KinectFusion where the global 3D model is represented as a truncated signed distance function (TSDF) that is discretized into a volume covering a scene to be reconstructed. The advantage of using a volumetric representation of the scene is that dense depth images can be rendered for a given camera position, which helps the camera tracking process. Moreover, updating the volume is easy and fast thanks to the GPU. The major drawback is, on the other hand, compactness. That is, the data size required to maintain the 3D representation of a scene is voluminous. This poses problems when dealing with large scale scenes or when communicating data through the internet.

Extensions of KinectFusion to a large scene have been proposed where the volume is moved along with the tracked camera motion. Roth and Vona (2012) proposed a method to automatically translate and rotate the volume in space as the camera moves. The volume is remapped into a new one by the TSDF interpolation whenever sufficient camera movement is observed. The objective of this work is to output fine camera tracking with a local map of the environment. As a consequence, points that leave the volume are lost and the method can not generate the complete reconstructed scene. Similarly, Whelan et al. (2012) introduced Kintinuous, proposing a method to identify points that leave the KinectFusion volume and incrementally add them into a triangular mesh representation of the scene. Available implementation that can be found in the PCL (The PCL library) allows even to re-use existing data when moving the volume. However, rich geometric features in the RGB-D image sequence are required for accurate scene reconstruction. When a scene does not have rich geometric features, this method fails in tracking the camera motion, resulting in poor reconstruction.

Zeng et al. (2013) and Chen et al. (2013) proposed an octree-based fusion method to compress the volumetric TSDF representation of the scene. At the same time, Zhou and Koltun (2013) proposed to use local volumes around points of interest and Henry et al. (2013) proposed to segment the scene into planar patches and use 3D TSDF volumes around them to represent the 3D scene. Neibner et al. (2013) proposed to use a spatial hashing scheme to compress the empty space, while keeping real-time performance and state-of-the-art accuracy. By contrast, our proposed method requires only three 2D images for each parametric surface patch to model the scene, which allows more compact representation of the scene.

2.1.2 Surface Models

Weise et al. (2009) proposed to build a global model using surface elements called Surfels (Pfister et al. 2000). Using triangular meshes for a global model requires considerable efforts to guarantee the integrity and quality of the mesh topology after adding, updating, or removing any vertices. As addressed in Weise et al. (2009), however, the use of Surfels has the advantage that it can be easily updated with live measurements while keeping consistency of the global model. One drawback of this method is that the Surfel representation is relatively sparse compared with captured depth images, which degrades the accuracy of registration results. Moreover the memory space required to maintain the Surfel representation for large scenes is not small.

Hernandez et al. (2012) introduced the canonical 2D map framework for 3D face reconstruction where the aligned points of the 3D facial surface are accumulated in a cylindrical model (similar approach is used in Blanz and Vetter 2003). Though high quality face models could be obtained, the camera tracking system is significantly affected by inaccurate pose estimates when the current frame has a large pose variation against the reference frame. This is because they employ the frame-to-frame camera tracking approach and no discussion is given about how to use their cylindrical global model into a frame-to-global-model framework. Moreover, the proposed data integration method has its limitation as it is not view-centric (i.e. it does not account for the directional nature of the noise in depth images). Besides, their method is developed for human face reconstruction only.

2.2 Camera Tracking and Data Integration

The process of accurately estimating the camera extrinsic parameters (i.e. the camera pose), called the camera tracking process, is of crucial importance for any 3D reconstruction method. This process allows to align all acquired data into a global coordinate system, which is a pre-request for any data integration method. In general, two main strategies exist: the frame-to-frame strategy and the frame-to-global-model strategy.

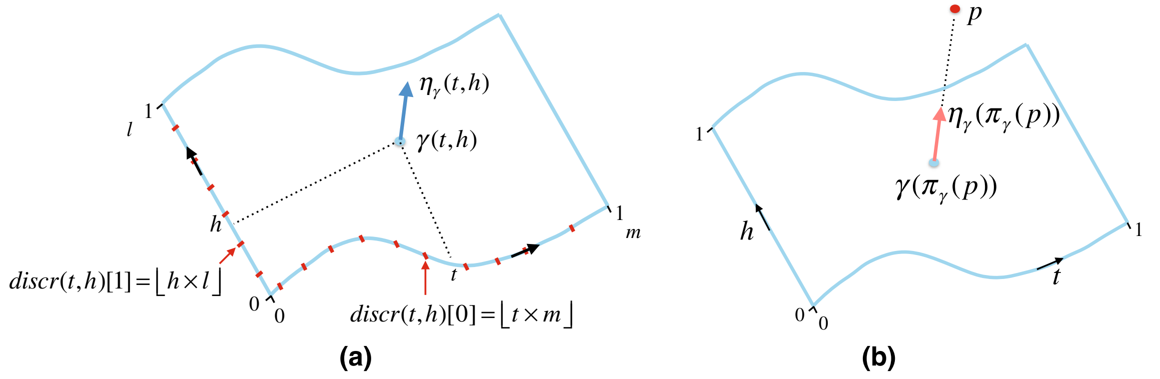
2.2.1 The Frame-to-Frame Strategy

A standard approach to camera tracking is to employ a frame-to-frame strategy (Davison et al. 2007; Henry et al. 2012; Jaeggli et al. 2003). Each incoming frame is aligned to its previous frame, then newly aligned data are integrated into a global model (using Surfels (Pfister et al. 2000) for example), which is triangulated in a final post-process (using poisson surface reconstruction (Kazhdan et al. 2006) for example). In Davison et al. (2007), a probabilistic feature-based map approach was proposed that represents at any instant a snapshot of current estimates of the state of the camera and all features of interest, and also the uncertainty in these estimates. The probabilistic state estimates of the camera and features are updated during camera motion and feature observation. When new features are observed the map is enlarged with new states and, if necessary, features can be deleted. In Henry et al. (2012), RGBD–ICP was proposed to align successive frames. RGBD–ICP combines the Iterative Closest Point (ICP) algorithm (Besl and McKay 1992) with the Scale Invariant Feature Transform (SIFT) algorithm (Lowe 1999). In this method, SIFT feature points (to which depth information is attached) are first matched together and then used to augment the standard ICP algorithm. Jaeggli et al. (2003) proposed to first align all input frames to a reference frame using the standard ICP algorithm and then to employ a multiview refinement algorithm in a post process to improve quality of camera pose estimates.

A crucial limitation of this strategy comes from the frame-to-frame error propagation, which can lead to significant errors at the end of the sequence. This becomes fatal when the trajectory of the camera is large. Note that, however, a loop closure algorithm may reduce the propagated error if a loop exists. In this strategy, only the current and previous frames are loaded on the GPU memory for camera tracking and, therefore, the GPU memory use is low and tracking is fast.

2.2.2 The Frame-to-Global-Model Strategy

Another approach to camera tracking is to use the frame-to-global-model strategy, which has been proven effective



$\eta_\gamma(t,h)$
$\gamma(t,h)$
l
h
m
t
$discr(t,h)[1]=\lfloor h\times l\rfloor$
$discr(t,h)[0]=\lfloor t\times m\rfloor$
(a)
p
$\eta_\gamma(\pi_\gamma(p))$
$\gamma(\pi_\gamma(p))$
(b)

Fig. 1 Illustration of introduced notations related to the parametric surfaces. **a** A parametric surface γ, a point $\gamma(t,h)$, on the surface at coordinates (t,h), the associated normal vector $\eta_\gamma(t,h)$ and the discretized coordinates $discr(t,h)$. **b** A 3D point **p** projects on the surface γ at coordinates $\pi_\gamma(\mathbf{p})$

with KinectFusion (Newcombe et al. 2011) and its extensions (Nguyen et al. 2012; Roth and Vona 2012; Whelan et al. 2012). In this strategy, a single high-quality global 3D model is updated along with live depth measurements. Incoming depth images are then aligned to dense high-quality predicted depth images generated from the global model. Using a global model allows to reduce the error propagation. Newcombe et al. (2011) proposed to employ a linearised version of the Generalized Iterative Closest Point (GICP) (Segal et al. 2009) (extension of ICP where the point-to-plane metric is used instead of the point-to-point metric) to align the input frame with the dense depth image predicted from the global model. Henry et al. (2013) proposed to use both photometric and geometric errors to perform dense alignment of RGB-D frames. The importance of both errors was balanced using a weighted average. Steinbrucker et al. (2013) proposed to combine photometric and geometric errors using a probabilistic framework.

In this paper, we also employ the frame-to-global-model strategy for estimating the camera trajectory.

3 Parametric 3D Scene Representation

We reason that parametric surfaces can be used to describe the 3D geometry of an indoor scene. We thus propose to represent a 3D scene as a set of parametric surface patches having attributes. To each surface patch detected in the scene, we attach as its attributes three 2D images in addition to information that identifies the surface patch.

The three images are a three-channel Bump image, an one-channel Mask image and a three-channel Color image; these three images encode geometric and color details of the scene. The Bump image encodes the local geometry around the surface patch. For each pixel, we record in the three channels the displacement of the 3D point corresponding to the pixel from the lower left-corner of the pixel. The Mask image encodes the confidence for accumulated data at a point and the Color image encodes the color of each point. The Bump image encodes the local deviation to the parametric surface patch, which allows us to accurately represent the geometry of the 3D scene while using less memory. Adding, removing or updating points is executed easily and efficiently in our representation, because we have only to manipulate 2D images.

3.1 Parametric Surfaces

Let us assume we are given a set of points $P = \{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n\}$ with $n > 0$, and a C^1 parametric surface γ that best fits P (how to determine γ is still an open question and out of the scope of this paper). Any 3D point on surface γ can be represented using parameters (t,h) with $0 \le t,h \le 1$ that run over γ (see Fig. 1a). This can be mathematically described as follows:

$$\begin{aligned}&\gamma : ([0,1])^2 \to \mathbb{R}^3\\ &(t,h) \longmapsto (x(t,h), y(t,h), z(t,h))^\top,\end{aligned}$$

where $(x(t,h), y(t,h), z(t,h))^\top$ is a 3D point on the surface γ.

Computation of normal vectors From the C^1 surface γ we derive the normal map η_γ (see Fig. 1a):

$$\begin{aligned}&\eta_\gamma : ([0,1])^2 \to SO(3)\\ &(t,h) \longmapsto norm\left(\left(\tfrac{\partial x}{\partial t}(t,h), \tfrac{\partial y}{\partial t}(t,h), \tfrac{\partial z}{\partial t}(t,h)\right)^\top \wedge\right.\\ &\qquad\left.\left(\tfrac{\partial x}{\partial h}(t,h), \tfrac{\partial y}{\partial h}(t,h), \tfrac{\partial z}{\partial h}(t,h)\right)^\top\right),\end{aligned}$$

where $norm$ is the function that normalise the input vector to the vector of norm 1, and $\wedge$ is the cross product operator.

Computation of the projected points For surface γ, we define a projector operator π_γ, such that for any $\mathbf{p} \in \mathbb{R}^3$,

$$\pi_\gamma(\mathbf{p}) = \operatorname{argmin}_{(t,h)\in([0,1])^2}(\|\mathbf{p} - \gamma(t,h)\|_2).$$

$\pi_\gamma(\mathbf{p})$ represents the parameters that correspond to the projection of $\mathbf{p}$ onto γ (see Fig. 1b).

Discretisation of the parametric surface The parametric surface γ is discretised in both dimensions along the surface into given m and l uniform segments. We define the operator $discr$ that maps real coordinates $(t, h) \in ([0, 1])^2$ that represent a point on surface γ into discrete coordinates[1] $discr(t, h) \in [\![0, m]\!] \times [\![0, l]\!]$ that represent a pixel corresponding to the point (see Fig. 1a). Namely,

$$\forall (t, h) \in ([0, 1])^2,$$
$$discr(t, h) = (\lfloor t \times m \rfloor, \lfloor h \times l \rfloor).$$

3.2 Bump Image

The Bump image, encoded as a three-channel 16 bits image, represents the local geometry of the scene around a parametric surface patch. For a given surface patch, the Bump image is obtained by identifying points around the surface patch and projecting them onto the discretized surface patch. The three values of a pixel in the Bump image encode the exact deviation from the parametric surface patch of the 3D point corresponding to the pixel. This allows us to record its exact 3D position in the global coordinate system. For the Bump image, only a few bytes are required to record the position of a 3D point regardless of the size of the scene, which makes our 3D model require low memory usage.

We describe more in detail how to obtain the Bump image and how to recover the 3D coordinates of 3D points using a simple example depicted in Fig. 2. Our objective is now to compute the Bump image $Bump$ that corresponds to the set of point P and the parametric surface γ, and to recover the set of 3D points using only the surface parameters and the computed Bump image.

For each point $\mathbf{p} \in P$, we obtain the pixel coordinates (i, j) of $\mathbf{p}$ in $Bump$ as $(i, j) = discr(\pi_\gamma(\mathbf{p}))$. We encode in the Bump image the deviation caused by the discretization process together with the signed distance of $\mathbf{p}$ from the parametric surface γ (with respect to the normal vector $\eta_\gamma(\pi_\gamma(\mathbf{p}))$) so that we can exactly recover the 3D position of $\mathbf{p}$. Namely,

$$Bump(i, j) = [\pi_\gamma(\mathbf{p})[0] \times m - i, \pi_\gamma(\mathbf{p})[1] \times l - j,$$
$$(\mathbf{p} - \gamma(\pi_\gamma(\mathbf{p}))) \cdot \eta_\gamma(\pi_\gamma(\mathbf{p}))],$$

[1] The notation $[\![a, b]\!]$ denotes the integer interval between a and b.

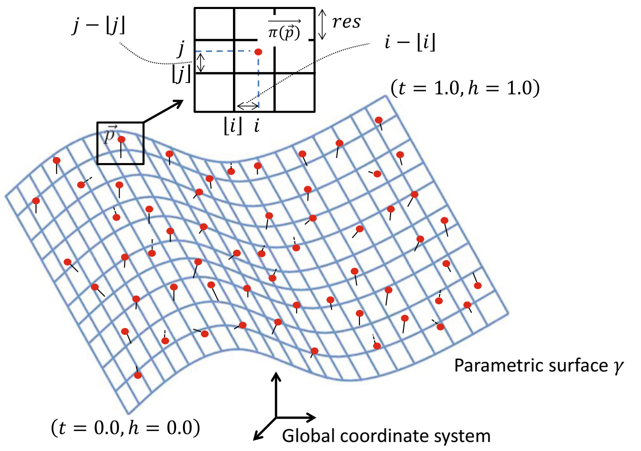


Fig. 2 A simple example for Bump image computation. A parametric surface is fitted to the input cloud of points and discretised in both dimensions to generate the 2D image attributes

where $\cdot$ is the scalar product operator. Note that in the case where two points project onto the same pixel in $Bump$, we keep in $Bump$ only the values for the point that is closest to γ.

On the other hand, computing the 3D positions of points is easy. Given a parametric surface γ and its Bump image $Bump$, it is straightforward to compute the exact position of a point corresponding to a pixel (i, j) in the global 3D coordinate system.

$$\mathbf{p}(i, j) = \boldsymbol{\gamma}\left(\frac{i + Bump(i, j)[0]}{m}, \frac{j + Bump(i, j)[1]}{l}\right)$$
$$+ Bump(i, j)[2] \times \eta_\gamma\left(\frac{i + Bump(i, j)[0]}{m}, \frac{j + Bump(i, j)[1]}{l}\right).$$

3.3 Mask and Color Images

The Mask image encodes the confidence for 3D points. This allows us to perform a running average when integrating live measurements and also to eliminate erroneous measurements. The values of the Mask image are initialized when generating the Bump image. A pixel in the Mask image is set to 1 if a point is projected onto the same pixel in the Bump image, it is set to 0 otherwise. The Mask image is then updated with live measurements as explained in Sect. 4.2. The Color image takes the RGB values of the points that are projected into the corresponding pixel in the Bump image. We can thus recover color information of 3D points as well.

3.4 Memory Consumption Analysis

One of the main advantages of our proposed 3D scene representation is its low memory consumption compared with state-of-the-art volumetric representations. In the ideal case, for both our 3D scene representation and the volumetric ones, the empty space is not recorded (using Octrees (Zeng et al. 2013), patch volumes (Henry et al. 2013) or spatial hash tables (Neibner et al. 2013) are examples on how to come close to this ideal case for the volumetric TSDF representation).

Let us consider a 3D scene composed of N points. Then, for our 3D scene representation each point is recorded in one pixel with one 8-bits mask value, three 8-bits color values and three 16-bits bump values. Thus, our representation requires in total and in the ideal case $N \times 10$ octets to build an accurate 3D model of the scene. Note that the amount of memory required to store the parameters of surface patches (i.e. the surfaces' equations and bounding boxes) is negligible compared to the amount of memory required to record all the points in the scene.

For the volumetric TSDF representation, on the other hand, a few voxels around the actual 3D position of each point are necessary to accumulate data. Considering the noise of the Kinect camera, we assume that voxels up to 5 cm in front and behind (with respect to the viewing direction) the position of each 3D point are required. With a resolution of 4 mm per voxel, this means that 20 voxels in front and behind are required for accurate reconstruction. At each voxel, a 8-bits mask value, three 8-bits color values and one 32-bits TSDF value are recorded. Therefore in total the volumetric representation requires in total and in the ideal case $N \times 328$ octets to build an accurate 3D model of the scene.

As we see from the above arguments, our proposed 3D scene representation achieves significant gain in memory consumption compared with state-of-the-art volumetric representations (more than 95% of lossless compression in this example). Note that since we employ 2D images, 2D image compression techniques such as JPEG can be employed to further reduce memory consumption.

4 3D Modeling

Our proposed 3D modeling algorithm follows the standard frame-to-global-model 3D reconstruction framework. Namely, given a current state of the global 3D model and an input RGB-D image, the algorithm first tracks the camera motion by aligning the input RGB-D image with a predicted one (rendered from the global model). The global model is then updated with new data before processing the next input RGB-D image. To successfully and efficiently perform this framework, challenging tasks are: (1) efficient rendering of our proposed 3D representation to produce predicted RGB-D images; (2) fast and accurate camera motion tracking; (3) fast and coherent update of our proposed 3D representation with aligned input RGB-D images.

We propose to render a dense and high quality predicted RGB-D image from our proposed 3D representation using OpenGL and the implicit quadrangulation given by the 2D discretization of each surface patch (see Sect. 4.1). Aligned measurements are then fused into the predicted RGB-D image before updating our proposed 3D representation with fused data. New surface patches, if any, are then detected as the camera moves through the scene.[2] The whole pipeline of our proposed method is illustrated in Fig. 3.

4.1 Camera Tracking

Accurately tracking the camera is of crucial importance for any 3D reconstruction method. As proposed in Henry et al. (2013), we also employ the GICP (Segal et al. 2009) algorithm combined with color information because it is fast with sufficient accuracy when using RGB-D sensors.

For the linearized GICP to work well, a key issue is to align incoming frames with a dense and high-quality predicted depth image. Directly projecting all the points of the global model into the current camera plane is not appropriate. This is because (1) parts of the scene that are close to the camera would produce relatively sparse depth information and (2) handling occlusions would require significant efforts. Instead, we take advantage of the natural quadrangulation given by the 2D image discretization to render a depth image using meshes.

From the Mask image of each surface patch, we identify effective quads (i.e., quads that have positive mask values at their four summits) as illustrated in Fig. 4. This straightforwardly gives us a quadrangulation for each surface patch, which can be quickly rendered into the current camera plane using a rasterising rendering pipeline. Note that for a given camera pose, only surface patches intersecting with the perspective frustum of the current camera pose are rendered, which eases computation.

Using the rendered color image is not good for the registration task. This is because small errors in camera tracking, misalignments between depth and color images and motion blur tend to blur the color image, which is fatal when optimising the alignment. To avoid this problem, we use the color image of the previous frame superimposed over our predicted depth image to align the current frame.

4.2 Updating Model with Live Measurements

With the estimated camera pose, and live RGB-D measurements, we can now update and refine our 3D scene representation. As proposed in Newcombe et al. (2011) we employ the running average in the camera plane domain to

[2] Note that in our implementation the plane detection is run every 20 frames.

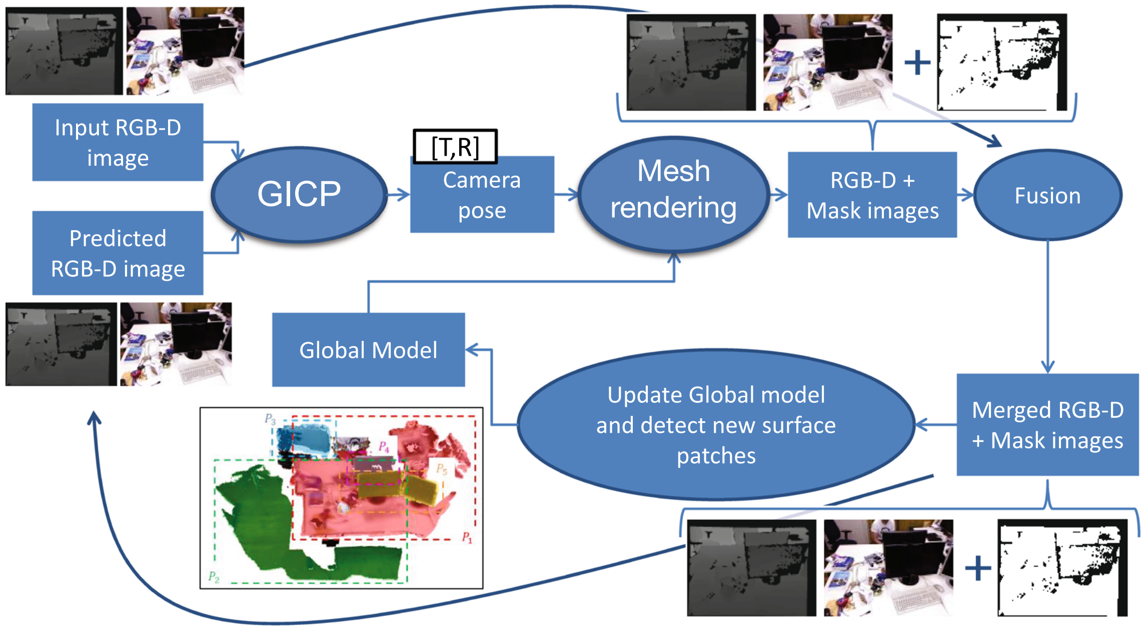


Fig. 3 The pipeline of our proposed method. We employ GICP combined with color information (Henry et al. 2013) to track the camera position from the input and predicted RGB-D images. The predicted RGB-D image is obtained by rendering the global 3D model in the current camera image plane. Measurements are fused in the camera plane and then recorded into each surface patch's attributes

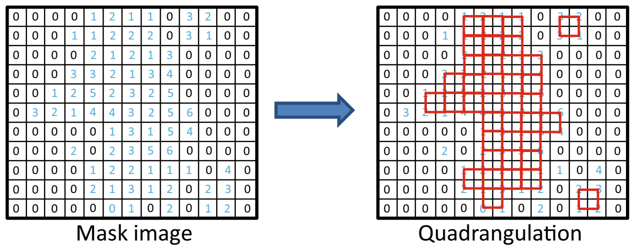


Fig. 4 Quads identification from the Mask image. A quad is identified whenever all mask values at the four summits are greater or equal to 1

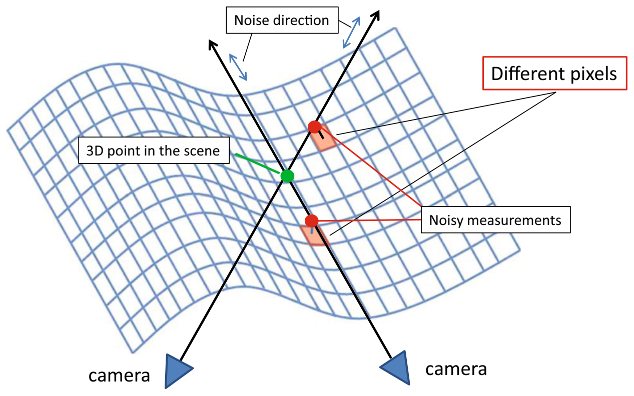


Fig. 5 Because of noise in the line of sight, noisy measurements of the same 3D point may project into different pixels in the Bump image

reduce input noise. Points lying within δ cm deviation to the global model are used to update the global model (we used $\delta = 10.0$ in Sect. 5). Note that a crucial, implicit assumption for this approach to be efficient is that all measurements that are averaged together must come from the same point on the scene. This is why registering incoming frame has to be done before integration and is a crucial process.

However, even if the registration process is successful a problem arises due to noise when integrating new depth measurements directly into the Bump image as seen in Hernandez et al. (2012). Namely, the problem is that (as shown in Figs. 5, 6) due to noise the same point viewed in two different frames may be projected into different pixel coordinates in the Bump image (Fig. 5), and also two different points of the scene may be projected into the same pixel of the Bump image (Fig. 6). This results into erroneous averaging computations. In order to avoid this problem, the integration process should be executed directly in the camera plane domain rather than in the Bump image domain. This is because noise in a depth image obtained with an RGB-D camera is mainly distributed along the viewing direction.[3]

To do so, the currently aligned depth image is merged with the predicted depth image using a rendered Mask image. Newly obtained 3D points are then projected onto different surface patches and the attributes of each surface patch are updated by replacing mask, color and bump values with the newly computed ones (if available).

[3] Radial distortion as exhibited in Zhou et al. (2013) is ignored in this work. How to properly handle this noise is left for future work (Zhou et al. 2013) is a pointer about how to handle such a noise).

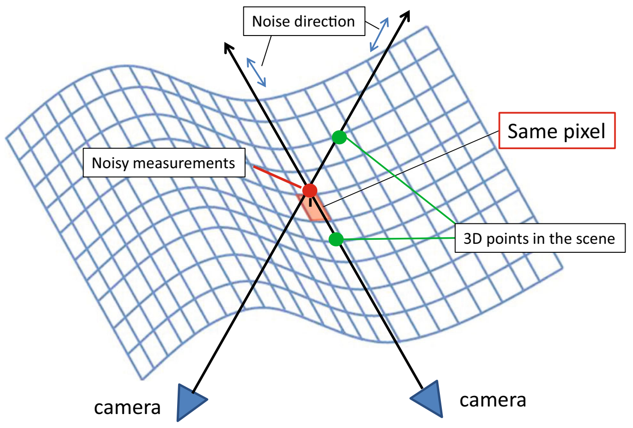


Fig. 6 Because of noise in the line of sight, noisy measurements of two different 3D points may project into the same pixel in the Bump image

Let us consider an RGB-D image (D_M, rgb_M) (where D_M is the depth image and rgb_M is the color image) rendered at the current camera pose, and a newly acquired RGB-D image (D_T, rgb_T). We also render a projected Mask image $\pi Mask$ in the same way as for (D_M, rgb_M) but by replacing the RGB values of vertices with the mask values. This allows us to perform a running average. Thereafter we merge the two RGB-D images using the Mask image and visibility constraints.

Let (i, j) be a pixel, $(D_{merged}, rgb_{merged})$ be the merged RGB-D image and $\pi Mask_{merged}$ be the merged Mask image. Then $D_{merged}(i, j)$ is computed as follows:

$$\begin{aligned}
&D_{merged}(i,j) = D_M(i,j)\\
&\text{and } \pi Mask_{merged}(i,j) = \pi Mask(i,j)\\
&\text{if } D_T(i,j) \text{ is not available}\\
&D_{merged}(i,j) = D_T(i,j)\\
&\text{and } \pi Mask_{merged}(i,j) = 1\\
&\text{if } D_M(i,j) \text{ is not available}\\
&D_{merged}(i,j) = D_T(i,j)\\
&\text{and } \pi Mask_{merged}(i,j) = 1\\
&\text{if } D_M(i,j) > D_T(i,j) + \delta \text{ (occlusion)}\\
&D_{merged}(i,j) = D_M(i,j)\\
&\text{and } \pi Mask_{merged}(i,j) = \pi Mask(i,j) - 1\\
&\text{if} D_M(i,j) < D_T(i,j) - \delta \text{ (visibility violation)}\\
&D_{merged}(i,j) = \frac{D_T(i,j) + \pi Mask(i,j) \times D_M(i,j)}{1 + \pi Mask(i,j)}\\
&\text{and } \pi Mask_{merged}(i,j) = \pi Mask(i,j) + 1 \text{ otherwise.}
\end{aligned}$$

The threshold δ is used to check visibility violations and occlusions (we used $\delta = 10.0$ in Sect. 5). Note that the color image rgb_{merged} is set to the input one (no averaging) to avoid motion blurring as much as possible (this helps camera motion tracking when using color). This process is detailed in Algorithm 1.

Algorithm 1 Merge

```
Require: Two aligned depth images $D_T$ and $D_M$, their color images
  $rbg_T$ and $rgb_M$, and a projected Mask image $\pi Mask$ corresponding
  to $D_M$.
Ensure: A merged depth image $D_{merge}$, a color image $rgb_{merge}$ and
  an updated projected Mask image $\pi Mask_{merge}$.
  for ($i \in [1 : col]$, $j \in [1 : row]$ ) ($col$ and $row$ are the number of
  columns and rows respectively in the depth images) do
    if $D_T(i, j)$ is not available then
      $D_{merge}(i, j) \leftarrow D_M(i, j)$
      $\pi Mask_{merge}(i, j) \leftarrow \pi Mask(i, j)$
    else if $D_M(i, j)$ is not available then
      $D_{merge}(i, j) \leftarrow D_T(i, j)$
      $\pi Mask_{merge}(i, j) \leftarrow 1$
    else if $D_M(i, j) > D_T(i, j) + \delta$ (occlusion) then
      $D_{merge}(i, j) \leftarrow D_T(i, j)$
      $\pi Mask_{merge}(i, j) \leftarrow 1$
    else if $D_M(i, j) < D_T(i, j) - \delta$ (visibility violation) then
      $D_{merge}(i, j) \leftarrow D_M(i, j)$
      $\pi Mask_{merge}(i, j) \leftarrow \pi Mask(i, j) - 1$
    else
      $D_{merge}(i, j) \leftarrow \frac{D_T(i,j)+D_M(i,j)\pi Mask(i,j)}{1+\pi Mask(i,j)}$
      $\pi Mask_{merge}(i, j) \leftarrow \pi Mask(i, j) + 1$
    end if
  end for
  $rgb_{merge} \leftarrow rgb_T$
  return $D_{merge}, rgb_{merge}$ and $\pi Mask_{merge}$
```

From the merged RGB-D image we can compute a new set of 3D points and record them into the attributes of each surface patch. Each point is first projected onto its corresponding surface patch. Then, for a point **p**, if the mask value at the projected pixel is smaller than that of **p**, then mask, color and bump values are all replaced by the newly computed values for the point **p**. Note that if the mask values are the same, then we record the values of the point closest to the surface patch. This process is detailed in Algorithm 2.

Overall, our method (1) aligns the input RGB-D image (D_T, rgb_T) to the predicted one (D_M, rgb_M); (2) renders our proposed 3D representation in the current camera plane to produce a new pair of predicted RGB-D and mask images $(D_M, rgb_M, \pi Mask)$; (3) merges input data (D_T, rgb_T) into the newly computed RGB-D image (D_M, rgb_M) using the mask image $\pi Mask$; (4) updates our proposed 3D model using the merged data $(D_{merge}, rgb_{merge}, \pi Mask_{merge})$ and (4) detects new planar patches from a residual RGB-D image. This last residual image is computed by removing from (D_{merge}, rgb_{merge}) all points that already belong to a surface patch. With the updated 3D scene representation, we can now proceed to the next frame. This process is detailed in Algorithm 3.

Algorithm 2 UpdateBumpImage

```
Require: A depth image $D$, a color image $rgb$ and its corresponding projected Mask image $\pi Mask$; a Bump Image $Bump$, a Color image $RGB$ and a Mask image $Mask$ corresponding to a surface patch $\gamma$.
Ensure: The updated Bump Image $Bump$, Color image $RGB$ and Mask image $Mask$.
  for $(i \in [1 : col], j \in [1 : row])$ do
    $\mathbf{p}(i, j) \leftarrow$ vertex at pixel $(i, j)$ in $D$
    $\mathbf{p}'(i, j) \leftarrow$ vertex in global coordinate system.
    $(k, q) \leftarrow discr(\pi_\gamma(\mathbf{p}'(i, j)))$
    if $Mask(k, q) < \pi Mask(i, j)$ then
      $Bump(k, q) \leftarrow [\pi_\gamma(\mathbf{p}'(i, j))[0] \times m - i, \pi_\gamma(\mathbf{p}'(i, j))[1] \times l - j, (\mathbf{p}'(i, j) - \gamma(\pi_\gamma(\mathbf{p}'(i, j)))) \cdot \eta_\gamma(\pi_\gamma(\mathbf{p}'(i, j)))]$ (Section 3.2)
      $RGB(k, q) \leftarrow rgb(i, j)$
      $Mask(k, q) \leftarrow \pi Mask(i, j)$
    end if
  end for
  return $Bump$, $RGB$ and $Mask$
```

Algorithm 3 3D modeling

```
Require: A predicted depth image $D_M$ and the color image $rgb_M$, the current camera pose $T_{pose}$, an input depth image $D_T$ and the color image $rgb_T$, a set of parametric surface patches $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_n\}$ with their attributes $\{Bump_1, Bump_2, \ldots, Bump_n\}$, $\{RGB_1, RGB_2, \ldots, RGB_n\}$ and $\{Mask_1, Mask_2, \ldots, Mask_n\}$.
Ensure: A new predicted depth image $D_{merge}$ and color image $rgb_{merge}$ and an updated set of parametric surface patches $\Gamma'$ with their updated attributes.
  $T_{curr} \leftarrow$ AlignGICP($D_T, rgb_T, D_M, rgb_M$) (Henry et al. 2013)
  $T_{pose} \leftarrow T_{pose} * T_{curr}$
  $(D_M, rgb_M, \pi Mask) \leftarrow$ RenderMesh($\Gamma, T_{pose}^{-1}$) (Section 4.1)
  $(D_{merge}, rgb_{merge}, \pi Mask_{merge}) \leftarrow$ Merge($D_T, rgb_T, D_M, rgb_M, \pi Mask$)
  for $(i \in [1 : n])$ do
    $(Bump_i, RGB_i, Mask_i) \leftarrow$ UpdateBumpImage $(D_{merge}, rgb_{merge}, \pi Mask_{merge}, \gamma_i)$
  end for
  $\Gamma' \leftarrow$ DetectAndAddNewSurface($D_{merge}, \Gamma$)
  return $D_{merge}, rgb_{merge}$ and $\Gamma'$
```

5 Indoor Scene Reconstruction

We demonstrate the advantages of using our proposed 3D parametric surface representation for the task of 3D modeling. Namely, we applied our proposed 3D representation to indoor scene reconstruction using a single hand-held RGB-D camera. The scene is assumed to be static and the objective is to quickly output an accurate 3D model of the scene.

5.1 Planar Patch Based Representation

We reason that an indoor static scene can be described using simple parametric surfaces such as planes because it is mainly composed of man-made objects (such as table, wall and storage for example), with rather simple shapes. We thus used planes as parametric surfaces and we represented an indoor scene as a set of planar patches having attributes as discussed Sect. 3. Each planar patch is identified by its plane equation

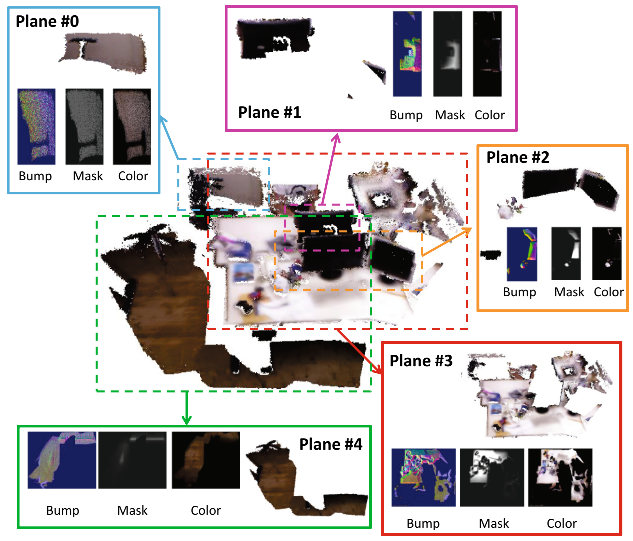


Fig. 7 The proposed 3D representation for indoor static scenes. The scene is segmented into several planar patches with attributes the Bump, Color and Mask images

and a bounding box. An example of our proposed 3D scene representation for an indoor scene is shown in Fig. 7. We discuss in the following of this section how to detect and manipulate the planar patches.

Planar patches segmentation We detect different planes in a depth image by using the gradient image of the normal image. From the gradient image, we identify contours and extract connected components in the depth image. For each connected component that represents one object, we apply RANSAC to fit a plane. Once the plane equation is computed, we identify the bounding box of all planar patches and initialise their attributes. A planar patch segmentation of a depth image is illustrated in Fig. 8.

To identify contour pixels in the depth image we apply the Canny edge detector (Canny 1986) on the normal image. This detector executes sequentially Sobel filter, Hysteresis filter and Non-max reduction. The output of the detector is a binary image with 1 in smooth regions and 0 at contour locations. We apply an erosion filter (with size 2 pixels in the experiments) to close the contours and identify connected components in the binary image using OpenCV's *floodFill* function.

The basic idea here is that each smooth region represents an object, and that each object is mostly distributed around a plane (for the case of indoor scenes). To compute the plane equation that best fit each region, we execute RANSAC on GPU for each region. Because noisy measurements can cause leaking of one object into another (often happens in practice), or because one object may be distributed along multiple planes (like a cylinder for example) we iteratively refine the segmentation as follows.

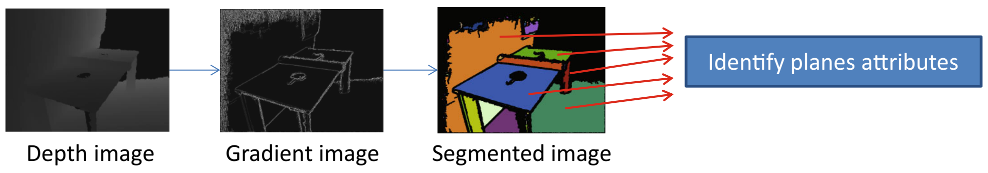


Fig. 8 Example of a planar patches segmentation of a depth image. First, contours in the depth image are identified using the gradient image. Then the depth image is segmented out by identifying connected components in the binary image corresponding to the contours (i.e. the image with 1 if the pixel is a contour, 0 otherwise). Finally, for each segment the plane equation fitted to the points in the segment is computed and the attributes are initialised

Pixels that were initially labelled in a region but do not belong to the fitted plane are unlabelled. This generates a residual mask image that is applied to the original contour image. Namely, for each pixel in the contour image, if the pixel is already labelled then the value of the contour at this pixel becomes 0. Then again, connected components identification followed by RANSAC is executed. This iteration is repeated until no more plane is detected. A plane is detected if it contains sufficient number of points (2000 points in the experiments).

Note that at each new RGB-D frame the label image is initialised using previously computed planar patches. To be more precise, the label image is first obtained by rendering each planar patch in the current camera image plane with its label as color. This significantly reduces the computational cost of segmentation at run-time since only a small part of the depth image needs to be segmented.

Resizing planar patches From a single depth image, only some parts of planes are visible in general (like the ground for example). It is not possible to know the complete size of a plane right after detecting it. Therefore, we dynamically resize each plane (i.e. its bounding box) with live measurements. To do so, we use an arbitrary resizing step s ($s = 32$ pixels in the experiments). At each iteration, we evaluate whether points are added close to the boundary (closer than a distance of s). If added, we extend the size of the bounding box by s in the corresponding direction (for example, if points are close to the left boundary, then the bounding box's size is extended in the left direction). The information for the bounding box is updated accordingly to the new one and the Mask, Color and Bump images are re-sized.

Merging two planar patches Because of noise in the depth images, one planar region may be segmented into two connected regions in the planar patch segmentation process. Then multiple planes may represent the same parts of the scene, which must be avoided. We identify such planes by computing the intersection of the bounding boxes of planes having similar parameters. If two planes have non-empty intersection between their bounding boxes, then the two planes are merged. To do so, we identify the parameters $(\mathbf{n}, d)$ of the largest plane and compute the union of the two bounding boxes of the largest plane and a plane to be merged in order to create the bounding box of the merged plane. Thereafter, points from the two planes are projected onto the merged plane with their respective color and mask values. Note that if two points project onto the same pixel in the merged plane, only the one with the maximum mask value is kept.

5.2 Experimental Results

We evaluated our algorithm using data that we captured using a Microsoft Kinect for Windows V1 sensor. All scenes were captured at 30 fps. If not stated otherwise, we used a resolution (i.e., size of one pixel) of 0.4 cm for the attribute images. We quantitatively and qualitatively evaluated the performances of our proposed method by comparing our results with those obtained using the Infinitam software (Kahler et al. 2015), which is a recent software that implements most recent techniques for 3D reconstruction using a TSDF 3D representation. We ran both our method and Infinitam on a MacBook Pro Retina with a 2.8 GHz Intel Core i7 processor and an AMD Radeon R9 M370X 2048MB Graphic card. Because the graphic card we used does not support CUDA, we implemented our proposed method using OpenCL and we ran Infinitam on the CPU (only CUDA GPU implementation was available). As a consequence we could not report processing speed of Infinitam on our dataset.[4] With our OpenCL implementation of our proposed method we processed the 640×480 RGB-D image sequences at 12 fps in average. Datasets used for the experiments shown in this paper are publicly available (https://sites.google.com/site/diegotthomas/datasets) and summarised in Table 1 (the size is given as length $\times$ width $\times$ height). Note that we ran Infinitam with the parameters as set in the code available at http://www.robots.ox.ac.uk/~victor/infinitam/download.html.

For all results we report in Table 2 the maximal amount of GPU and CPU memory usage at run-time with our method. The maximal amount of CPU memory usage corresponds to the value read from the Windows task manager toolkit. It

[4] For information, Infinitam was reported to run above 20 fps.

Table 1 Dimensions of datasets (available at https://sites.google.com/site/diegotthomas/datasets.) used to produce our experimental results

Name of dataset	(Length × width × height)	# frames
Desk	2 m × 1 m × 1 m	880
Lounge	8 m × 4.5 m × 2.5 m	3000
Library	60 m × 5 m × 2.5 m	19,900
Library- 2	40 m × 3 m × 2.5 m	11,000
Kitchen	5 m × 4 m × 2.5 m	3730
Human	4.5 m × 3 m × 2.5 m	1260
Corridor	11 m × 2.5 m × 2.5 m	1600

takes into account not only the amount of memory required to maintain the 3D model but also the memory usage of the whole application. We thus also report (in parenthesis) the amount of memory dedicated to maintain only the raw data of our 3D representation (i.e., the three Bump, Color and Mask images for each planar patch). We also report in Table 2, for each dataset, the size in the hard drive of the produced 3D model, the number of planar patches that compose it, the number of 3D points represented and the mean, minimal and maximal fps at run-time. For comparison purpose, we also show in Table 2 the CPU memory usage (read from the Windows task manager) at run-time of Infinitam (GPU was not available), the size in the hard drive of the produced 3D mesh and the number of 3D points represented. Note that in some cases, the Infinitam software pre-allocated memory for the 3D reconstruction and we had memory usage of 931 MB, though the amount of memory actually used to maintain the 3D model was probably lot less.

On the CPU, our raw data (i.e., uncompressed images) of the Bump, Color and Mask images for each planar patch are kept in memory to maintain our built 3D model. Once saved on the hard drive, the images are compressed using file format. This is why the size of the files in the hard drive is less than the reported CPU memory usage. On the GPU, at runtime, for each visible planar patch (the number of visible patches is bounded by the visual frustum) we maintain (in addition to the raw Bump, Color and Mask images) a vertex buffer object and quads indices to allow fast OpenGL rendering. This speeds up reconstruction but increases memory footprint. Note that planar patches that do not appear in the visual frustrum for a certain amount of time (100 frames in our experiments) are unloaded from the GPU on-the-fly, and the Bump, Color and Mask images are kept on the CPU to save GPU memory space. As a consequence, the amount of memory used in the GPU is independent from the overall size of he scene.[5]

We first applied our method to a small-scale scene to see the ability of our proposed method for reconstructing fine geometric details. We scanned a closed view of a desk, called Desk (see Table 1). Note that we chose a simple scene so that planar patches allow the representation of most of captured 3D points. This allows us to quantitatively evaluate reconstruction accuracy for all points obtained with the volumetric reconstruction of Infinitam. Figure 9a, b show the results obtained with our method and with Infinitam, respectively. Note that, in this experiment, we ran out method with a resolution of 0.2 cm for the attribute images. Our produced 3D model consisted of 13 planar patches having each a Bump, Color and Mask image. It took 3.05 MB of memory space and represented 534,490 3D points with color. The raw .stl file generated by Infinitam was about 18.1 MB. We loaded this model into MeshLab and merged duplicated vertices, which decreased the number of vertices from 1,112,142 to 208,843. We then exported the 3D mesh to a binary .ply file (without color), which was of size 7.15 MB. At run-time, our proposed method ran at 13.5 fps in average, with a maximum of 14.5 fps and a minimum of 13 fps.[6]

o quantitatively evaluate reconstruction accuracy, we first aligned the cloud of points obtained with our method (Fig. 9a) with that obtained with Infinitam (Fig. 9b) using the ICP algorithm provided by the PCL library (http://pointclouds.org), and then computed the Euclidean distance between each point obtained with Infinitam and its closest point obtained with our proposed method. Figure 9c illustrates distances as a heat map. From Fig. 9, we can see that the 3D model obtained by our method has almost the same accuracy with that obtained by Infinitam, while using two times less amount of memory space. Note that with our proposed 3D representation we also recorded color of the 3D scene.

To attest the ability of our proposed method for reconstructing large-scale 3D scenes we ran three experiments using one large and two very-large scale scene. First we produced results with our method and Infinitam using the data Lounge of size about 8 m × 4.5 m × 2.5 m (see Table 1) publicly available at http://www.stanford.edu/~qianyizh/projects/scenedata.html. This data represents a fairly large scale indoor 3D scene (far beyond the capacities of the original KinectFusion algorithm). The results in Fig. 10 show that our proposed method can successfully reconstruct a large-scale scene in details. The resources used to obtain the 3D model in our representation are given in Table 2.[7] Results obtained with Infinitam with this data were not good because of incorrect camera tracking. This probably happened because of fast camera motion. The

[5] The GPU memory usage at run-time depends only on the complexity of the scene (i.e., number and size of planar patches in the current visual frustrum).

[6] Memory usage at run-time was less than 150 MB on the GPU and less than 35 MB on the CPU.

[7] At run-time, the GPU memory usage never exceeded 300 MB, while the number of visible planar patches never exceeded 28 planes.

Table 2 Detailed memory usage on GPU and CPU at run time, 3D models' size, number of planar patches and number of 3D points represented, and mean, minimal and maximal fps at run-time

		GPU	CPU	# patch	Size	# points	fps		
		Max (MB)	Max (MB)		(MB)	After unification	Mean	Min	Max
Desk	Our method	150	149 (35)	13	**3.05**	534,490	13.5	13	14.5
	Infinitam	–	931	–	7.15	208,843	–	–	–
Lounge	Our method	300	512 (388)	204	**34.1**	4,634,811	13.5	12.5	16
	Infinitam	–	429	–	165.4	4,859,909	–	–	–
Library	Our method	447	2340 (2180)	972	**256.1**	34,502,106	11.3	8.5	14.5
	Infinitam	–	Failed	–	Failed	Failed	–	–	–
Library- 2	Our method	311	1178 (881)	365	**98.8**	17,661,018	13.5	10	17.5
	Infinitam	–	Failed	–	Failed	Failed	–	–	–
Kitchen	Our method	220	341 (133)	60	**12.2**	2,456,257	13	10.5	16
	Infinitam	–	261	–	97	2,739,421	–	–	–
Human	Our method	160	236 (76)	51	**7.71**	1,233,470	12.5	11	16.5
	Infinitam	–	931	–	97	2,830,081	–	–	–
Corridor	Our method	250	265 (140)	38	**14.5**	3,114,343	14	12	16
	Infinitam	–	931	–	161	4,724,034	–	–	–

Bold values indicate that the 3D models built with our proposed method have low memory footprint

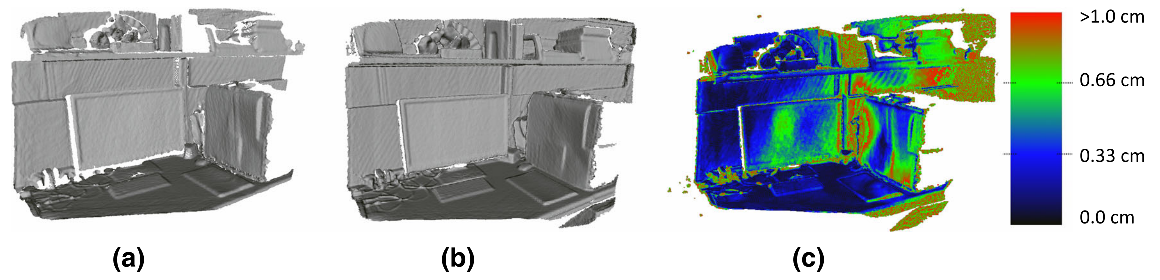


Fig. 9 Results obtained with data Desk. The results obtained with our proposed method were about the same accuracy as those obtained with Infinitam, while using two times less amount of memory. **a** Our method. **b** Infinitam. **c** Error image

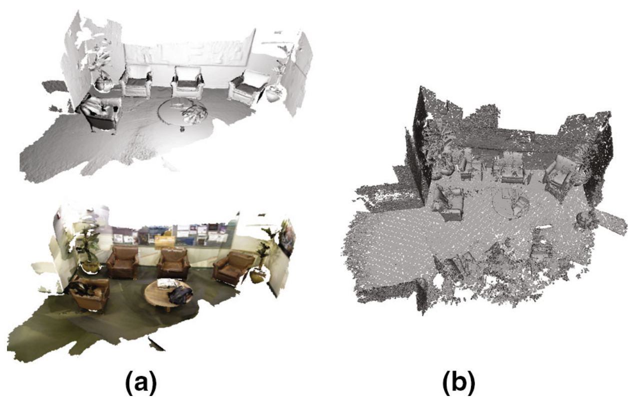


Fig. 10 Results obtained with data Lounge from 3D Scene Dataset with our method and Infinitam. **a** Our method. **b** Infinitam

3D mesh generated by Infinitam contained about 4,859,909 3D points after unifying duplicated vertices. It took about 165.4 MB of memory space after recording it as a binary .ply file. Note that our 3D representation also have color while color was not available in the 3D mesh built by Infinitam.

Second, we captured two large scale scenes using the Kinect for Windows V1 sensor. The sequences were captured in a library and results obtained with our method are shown in Figs. 11 and 12. The two datasets are called Library and Library- 2, whose dimensions are given in Table 1. As we can see from Figs. 11 and 12, though reconstruction results are satisfactory locally, globally results are not good. This is because of the accumulation of drift errors that inevitably occur when tracking camera movements. In particular sequences taken when walking through desks were challenging for the camera tracking because of few salient features (texture was quite uniform and the shape of the desks with chairs present several symmetries). This resulted in quite dramatic errors in camera pose estimation at large scale and poor quality of the global 3D reconstruction. To overcome this problem a loop closure

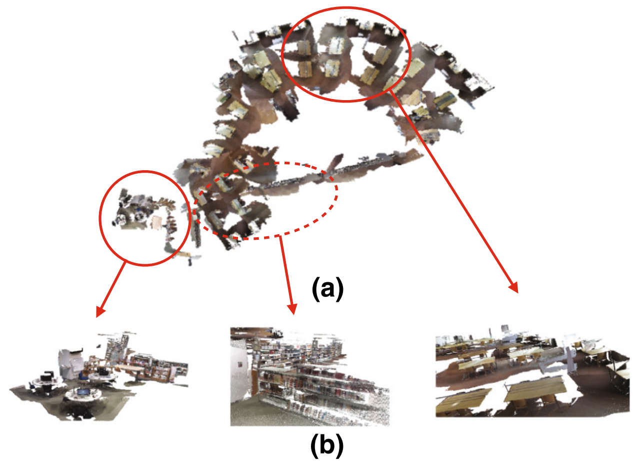


Fig. 11 Results obtained with data LIBRARY with our method. **a** Result obtained with our method. **b** Zoom 1, zoom 2, zoom 3

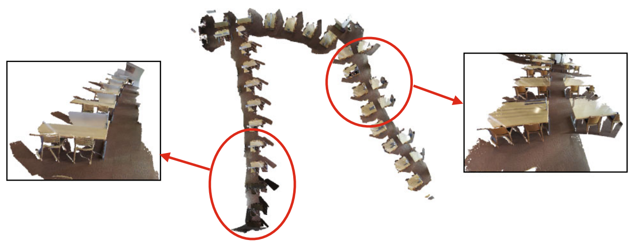


Fig. 12 Results obtained with data LIBRARY- 2 with our method. Result obtained with our method

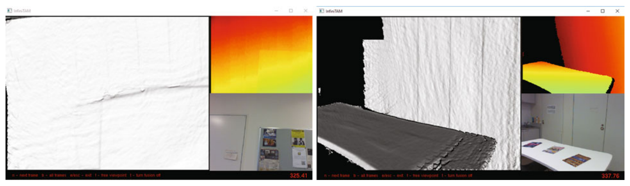


Fig. 13 Two input RGB-D images of the KITCHEN data-set and current 3D reconstruction captured from the Infinitam software

algorithm should be used to re-position all planar patches together in a consistent manner, which is left for future work.

The resources used to build the 3D models in our representation with datasets LIBRARY and LIBRARY- 2 are given in Table 2.[8] From these results, we can confirm that our proposed representation is promising to reconstruct very-large scale 3D scenes, because the GPU memory usage at run time is, as expected, low and independent from the size of the scene, and our method can also process long sequences of RGB-D images in real-time. We note that the Infinitam software could not process the whole RGB-D image sequence and failed in producing a complete 3D model with both data LIBRARY and LIBRARY- 2.

To illustrate the usefulness of combining both depth and color images for tracking camera motion, we captured an indoor scene called KITCHEN (of about 5 m × 4 m × 2.5 m) containing parts that do not have rich geometric features (as shown in Fig. 13) and we compared results by our method with and without using color and results obtained with Infinitam (which does not use color). In such a case using depth alone fails, as shown in Fig. 14b, c. However, by combining both depth and color images, as expected, we could successfully track the camera and reconstruct the 3D scene as shown in Fig. 14a.

Figure 15 shows reconstruction results of a scene with non-planar objects. We can see from Fig. 15 that, in this scenario, the choice of the plane as the underlying parametric surface is not well adapted to reconstruct the human body. In particular the head is poorly reconstructed: only half of the head is reconstructed and with poor resolution. The results obtained using Infinitam, which allows for arbitrary shape, were better than those obtained with our proposed method in this scenario. This experiment indicates that depending on the scene to reconstruct, it may be better to use more complicated 3D surfaces than the 3D plane. Actually it has been shown in Anasosalu et al. (2013) that to accurately reconstruct the human head, a sphere as the underlying parametric shape is a better choice.

To show the advantage, in this case, of using a different parametric shape to build our 3D representation we implemented a simple example for the reconstruction of the human head. As shown in Fig. 16, we manually fit a sphere over the head from the first RGB-D frame of the HUMAN sequence. We then ran our proposed method by adding this sphere as an additional surface patch. Note that in order not to have duplicated points we did not detect any planar patches in the region around the sphere. Results shown in Fig. 17 show that the sphere is a more appropriate underlying parametric shape to reconstruct the human head. By using the sphere we could reconstruct a denser and more precise 3D model of the human head. Note that how to automatically detect and fit the best parametric shapes to support the 3D reconstruction using our proposed 3D representation is another issue, and left for future work.

To demonstrate the advantage of building structured 3D models (like our proposed 3D representation) over unstructured 3D models (like TSDF, meshes or cloud of points as those built by KinectFusion or Infinitam) we captured an RGB-D image sequence called CORRIDOR where parts of the sequence have neither geometric nor color features. This situation often happens when dealing with uniform indoor scenes. For example when building a 3D model of a building, one has to walk through corridors as shown in Fig. 18a,

[8] The memory usage at run-time with data LIBRARY never exceeded 447 MB in the GPU and 2180 MB in the CPU. The memory usage at run-time with data LIBRARY- 2 never exceeded 311 MB in the GPU and 881 MB in the CPU.

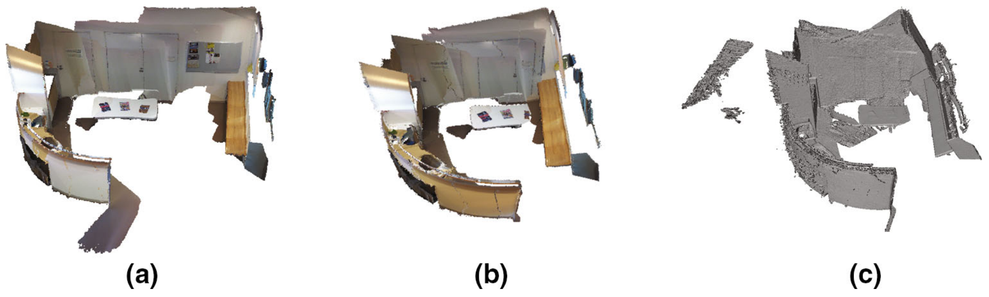


Fig. 14 Results obtained by our method with and without using color and Infinitam with the KITCHEN data-set. The low memory usage of our proposed method allows us to combine geometric and color information for robust camera tracking. **a** Our method *with color*. **b** Our method *without color*. **c** Infinitam

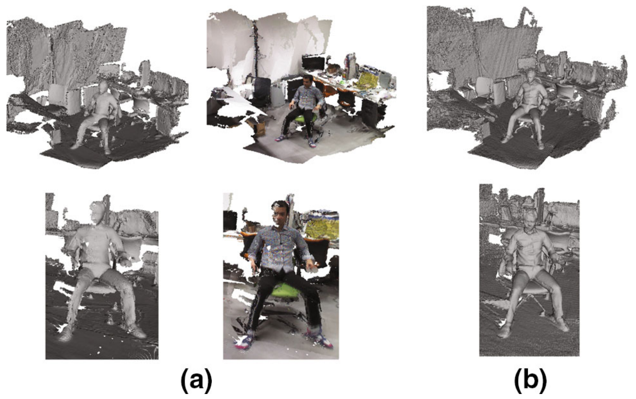


Fig. 15 Results obtained with data HUMAN with our method and with Infinitam. **a** Our method. **b** Infinitam

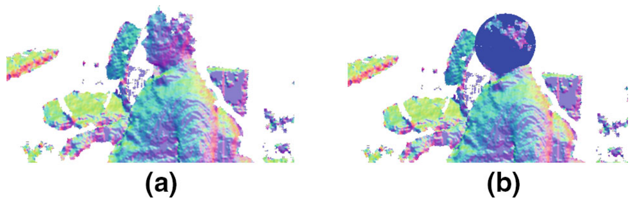


Fig. 16 Manual initialisation of the sphere to build the human head. **a** First input image. **b** A sphere is fit to the head

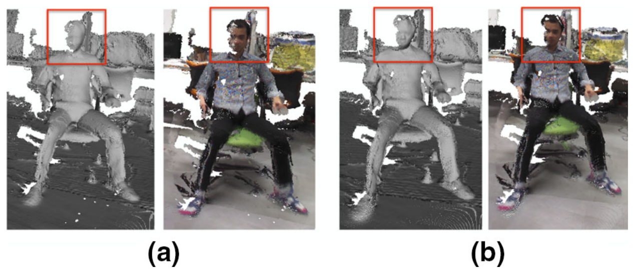


Fig. 17 Results obtained with our method using only planes (**a**) for 3D reconstruction, or with adding a sphere for better reconstruction of the head (**b**)

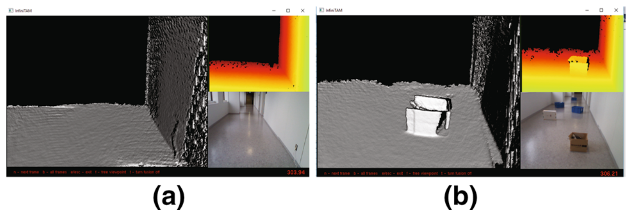


Fig. 18 Two input images from data CORRIDOR with and without dummy objects captured from the Infinitam output. **a** Corridor without dummy objects. **b** Corridor with dummy objects

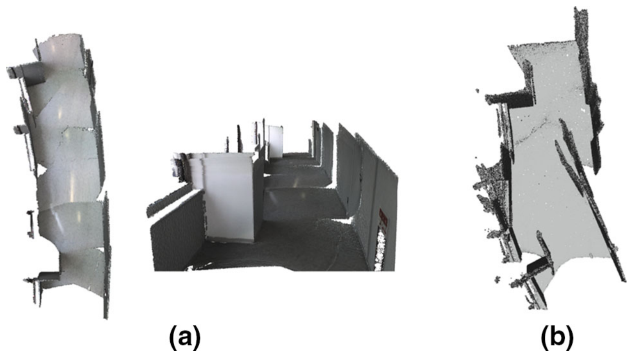


Fig. 19 Results obtained with our method and with Infinitam for the sequence CORRIDOR that contains parts without either rich geometric or color features. The reconstruction failed because the camera motion could not be tracked. **a** Our method. **b** Infinitam

which generally lack in salient features. In such a case, as shown in Fig. 19, camera tracking is not possible and reconstruction fails. To help tracking the camera motion, adding dummy objects in the scene is possible (Fig. 18b). Then, thanks to successful camera tracking, any 3D reconstruction technique can be applied (note that Infinitam did not perform so well in Fig. 20b probably because color information was also needed to correctly track the camera in this case). However the built 3D models then contains the dummy objects as shown in Fig. 20, which is not desirable since the objective is to reconstruct the corridor without dummy objects. If the built 3D model is an unstructured 3D mesh or cloud of points as those produced by Inifnitam, then editing the 3D model to remove the dummy objects and close holes in the ground is difficult. By contrast, thanks to our output that is structured into several planar patches, it is straightforward to remove

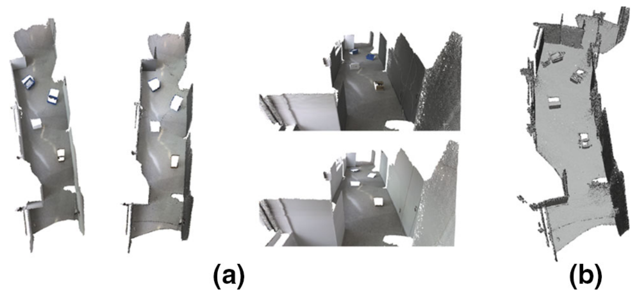


Fig. 20 Results obtained with our method and with Infinitam for the sequence CORRIDORDUMMY that contains several additional objects compared to the sequence CORRIDOR. Accurate camera trajectory could be estimated thanks to the additional dummy objects. **a** Our method. **b** Infinitam

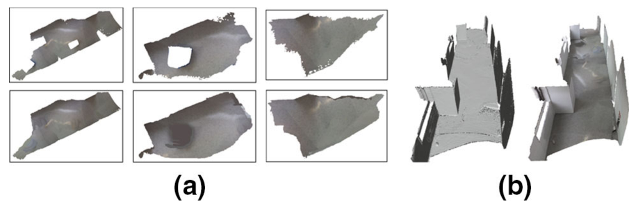


Fig. 21 After running our proposed method, additional objects can be easily removed from the built 3D model to obtain the desired (featureless) 3D scene. **a** Texture images before (*up*) and after (*down*) inpainting. **b** Edited 3D model

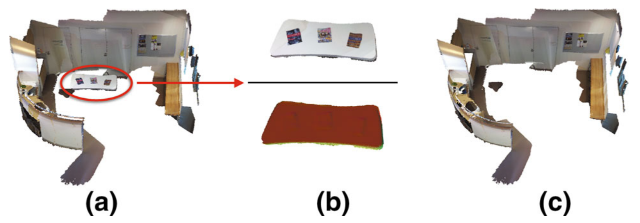


Fig. 22 Results obtained with our method for the data-set KITCHEN. Our proposed method directly generate segmented 3D models. It is then straightforward to isolate or remove specific objects. **a** Our 3D model. **b** Table isolated. **c** Table removed

the dummy objects. Moreover, because editing our built 3D model comes down to manipulating 2D images, it is easy to simply inpaint the Bump, Mask and Color images to close holes as shown in Fig. 21a. Actually, by using our method only we could successfully build the corridor without the dummy objects, as shown in Fig. 21b.

In Fig. 22, we illustrate one interesting output of our scene representation. That is, more than 3D meshes, our method directly outputs roughly segmented 3D meshes. Therefore, we can easily remove some parts of the reconstructed scene directly after reconstruction; this enriches possible usage of our method as seen below. The example shown in Fig. 22 was obtained by simply hiding mesh layers in MeshLab. As we can see it becomes easy to remove undesired objects from the reconstructed scene (see Fig. 22c), or on the contrary to keep only objects of interest (see Fig. 22b).

6 Discussion

3D imaging is still at its blabbering stage compared with 2D imaging. Recently, tools became available that allow to automatically build high fidelity 3D models from inexpensive consumer depth cameras (as shown above). However, simply outputting raw data alone (such as an unstructured dense cloud of 3D points) is largely insufficient to broaden the range of applications for automatic 3D modeling. For example, most of applications for 3D modeling (e.g., robot navigation, scientific simulations or entertainment) require to navigate in the 3D model at interactive frame rate. This requires access to multiple levels of details for each part of the 3D model. Moreover, the possibility to easily edit the generated 3D model is of outmost importance for applications in virtual reality or artistic design. In this section, we discuss about the advantages of using our proposed 3D scene representation (i.e. parametric surface patches augmented by geometric and color texture images) against state-of-the-art 3D representations (e.g., cloud of point, 3D meshes or volumetric TSDF) for various post-processing operations.

6.1 Fast Rendering at Multiple Level of Details

State-of-the-art 3D modeling methods can output 3D models with resolution (i.e. average distance between neighbouring points) about the milimeter. As shown in Fig. 9 a 3D model generated by Infinitam for a small scene of about 2 by 1 m already contains more than two hundred thouthands of points. When the size of the scene is increasing, rendering the 3D surface quickly requires a huge amount of resources. This limits usage of 3D models built using KinectFusion-like softwares to small scale scenes only. Simplifying the output 3D mesh as done in Zhou et al. (2013) works only for untextured rendering. However, when textured rendering is required (which drastically improves visual impression), simplifying the 3D mesh results in dramatically decreasing the amount of details in the rendered image.

In computer graphics, to achieve fast rendering with keeping sufficient level of details in both color and geometry, independently of the size of the scene, rendering at multiple levels of details is commonly used. The concept is that surfaces far from the camera are rendered with low level of details, while surfaces close to the camera are rendered with high level of details. This drastically reduces rendering processing time. In this section, we show that by using our proposed 3D scene representation, it becomes possible to map color and normal images with various resolutions onto coarse meshes to perform rendering at multiple levels of details and at interactive frame-rate independently of the scale of the scene.

Figure 23 shows an example of full resolution Bump, Color, Alpha and Normal images for a planar patch generated

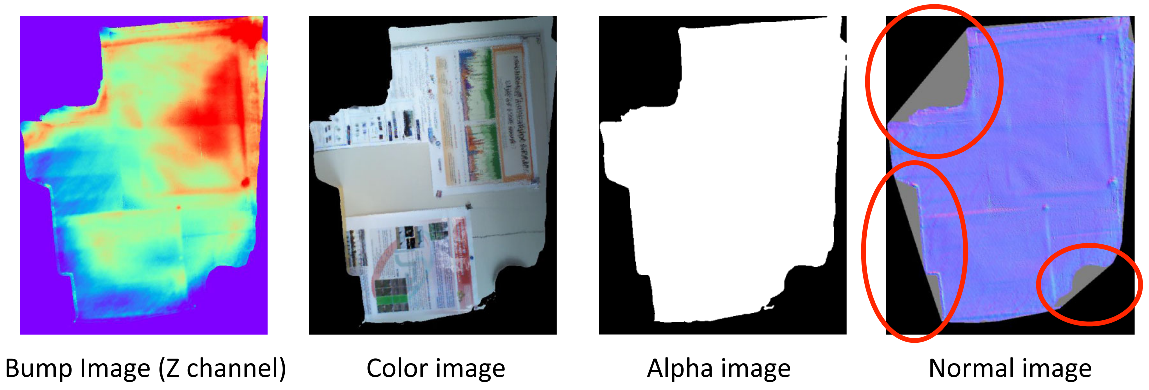


Fig. 23 Illustration of the computed alpha and normal images for a given Color and Bump image. Empty regions in the Bump image that are covered by a *triangle* are visible in the Normal image with *grey color* (inside the *red circles*) (Color figure online)

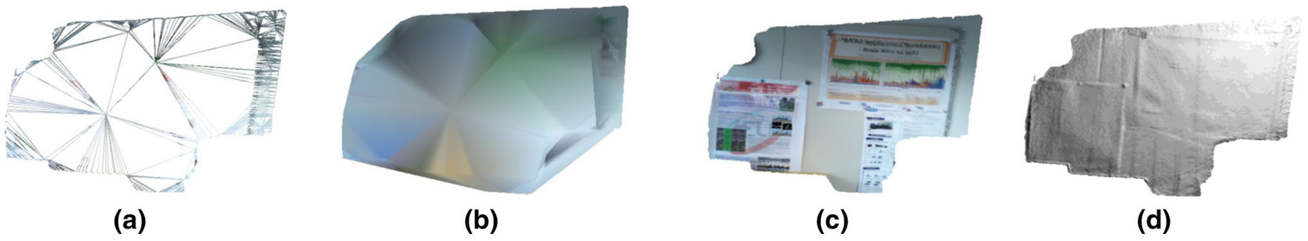


Fig. 24 A light, yet detailed rendering of our built 3D model by taking advantage of texture mapping. **a** The geometrically simplified 3D mesh has a sparse triangulation. **b** Rendering the simplified mesh without texture mapping produces unsatisfactory results because of the loss in *color information*. **c**, **d** Visually detailed rendering with relighting and color could be obtained by mapping the color and normal images produced by our proposed 3D modeling method onto the simplified 3D mesh. **a** Sparse triangulation. **b** No texture mapping. **c** Detailed texture. **d** Detailed geometry

by our proposed method (as explained in Sect. 5). Figure 24 shows an example of a light, yet detailed 3D rendering of this planar patch. This example also illustrates the limitation of standard mesh simplification. A geometric mesh simplification will remove most of points inside the wall, which do not contain geometric details (Fig. 24a). As a consequence the 3D mesh generated in this way will lose most of its color details (Fig. 24b). However, with our method, it was possible to map the color and normal image onto the simplified mesh to generate detailed rendering (Fig. 24c, d). In the remaining of this section, we detail how to produce such a rendered image in this example.

Note that in a post-process, it would be possible to compute texture images that could be mapped over a 3D mesh built from a TSDF 3D representation of the scene. This would then allow rendering the scene at multiple levels of detail by modifying the resolution of the texture image and by simplifying the 3D mesh. However, our proposed method has the advantage that it allows building the texture images on-the-fly independently of the size of the scene to be reconstructed.

To achieve scale independence, loading the whole 3D model at the full level of details is not realistic. If the scene is large, loading all data at once may require a huge amount of resources. Rather, for each surface patch that composes the scene, we load its attributes at the level of details that corresponds to the current viewpoint. Namely, we employ three levels of details, from the coarsest level lvl 1 (for patches farther than 10 meters from the camera centre), to the finest level lvl 3 (for patches closer than 3 meters from the camera centre). At $lvl1$ the attribute images are down sampled to 32×32 pixels, and at $lvl2$ the images are down sampled to half the resolution of the original images in both dimensions. Every time sufficient movement of the camera is observed, attributes of surface patches are updated (if needed). At every update of a surface patch's attributes, we have to (1) compute the mesh simplification and (2) compute the normal image.

3D mesh simplification For each surface patch, we build a coarse triangular mesh onto which its attribute images will be mapped. For the case of planar patches, the triangulation is computed from the Bump image (which is of a different resolution depending on the level of details required). First we detect salient points in the Bump image that will be the mesh vertices. To do so, we compute the gradient image of the third channel of the Bump image (the channel that encodes the vertical displacement) and binarize it using simple thresholding. We then compute the triangulation of the salient vertices using the standard 2D Delaunay triangula-

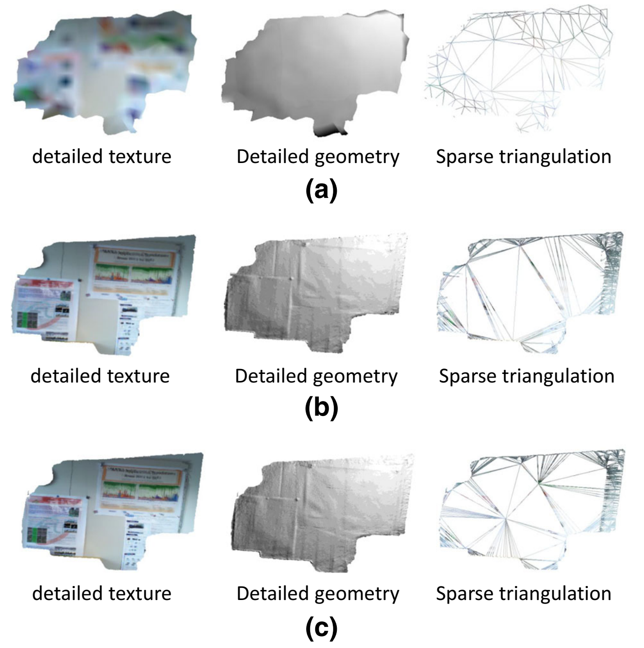


Fig. 25 A planar patch rendered at the three different levels of details. At level 1 the triangulation contains only a few vertices and the texture images have low resolution, which produces low detailed rendered images but with low computational burden. At level 3, there are more vertices in the triangulation than at level 1 and the texture images have the full resolution. This produces highly detailed rendered images but with more expensive computational cost that at level 1. **a** Lvl 1. **b** Lvl 2. **c** Lvl 3

tion (with the 2D coordinates of all vertices being the pixel indices in the Bump image). Figure 25 illustrates the mesh simplification for a planar patch at the thee different levels of details (its attribute color images at the three levels of details are shown in Fig. 26). The 3D mesh had about 1.5 million vertices before simplifications, about 3000 vertices after simplification at level 3, about 2000 vertices after simplification at level 2 and about 200 vertices after simplification at level 1. Note that different sampling strategies can be used depending on the underlying parametric shape in use.

Normal image computation In the triangular mesh computed above, each vertex is identified from a salient pixel (u, v) in the Bump image. Therefore, each vertex has its corresponding 2D coordinates (also called texture coordinates) in the attribute images. The texture coordinates are simply $\left(\frac{u}{height}, \frac{v}{width}\right)$, where $height$ and $width$ are the height and width of the Bump image). Mapping the color image onto the mesh is thus straightforward. However, mapping the color image only is not sufficient because we also have to render the geometric details for realistic lighting or untextured visualisation. To do so, we need to map a normal image that encodes variations of normal vectors at pixels of the attribute images. This normal image is computed from the

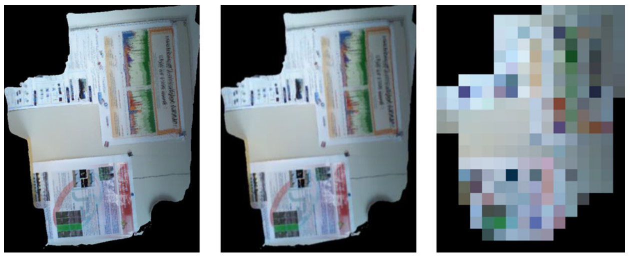


Fig. 26 The attribute color images at the three levels of details used to generate the renderings in Fig. 25. From *left to right*, level 3, 2 and 1

Bump image (Fig. 23 illustrates such a normal image). We detail in the following how to compute the normal image.

For each vertex **P** of the simplified 3D mesh we compute the orthonormal basis consisting of the normal, the tangent **Tgt** and the bitangent **BTgt** vectors, according to Lengyel (2001). The normal vector is identical to the normal of the surface patch. The tangent space has to be aligned such that the X axis corresponds to the U direction in the Bump image and the Y axis corresponds to the V direction in the Bump Image. By doing so, for each point **Q** in the tangent space of **P** we can write

$$\mathbf{Q} - \mathbf{P} = (\mathbf{uv_Q}[0] - \mathbf{uv_P}[0])\mathbf{Tgt} \\ + (\mathbf{uv_Q}[1] - \mathbf{uv_P}[1])\mathbf{BTgt},$$

where $\mathbf{uv_P}$ and $\mathbf{uv_Q}$ are the 2D texture coordinates of **P** and **Q** respectively, and $\mathbf{uv_P}[0]$ and $\mathbf{uv_P}[1]$ denote the U and V texture coordinates of **P**, respectively. Figure 27a illustrates such an orthonormal basis with a simple example.

For each triangle $(\mathbf{s_0}, \mathbf{s_1}, \mathbf{s_2})$ in the mesh we compute its corresponding tangent space and accumulate the base vectors **Tgt** and **BTgt** in its three summits (note that one vertex can be a summit of multiple triangles). The process is illustrated in Fig. 27b. Let $\mathbf{\Delta Pos_1} = \mathbf{s_1} - \mathbf{s_0}$, $\mathbf{\Delta Pos_2} = \mathbf{s_2} - \mathbf{s_0}$, $\mathbf{\Delta UV_1} = \mathbf{uv_1} - \mathbf{uv_0}$ and $\mathbf{\Delta UV_2} = \mathbf{uv_2} - \mathbf{uv_0}$, where $\mathbf{uv_0}$, $\mathbf{uv_1}$ and $\mathbf{uv_2}$ are the 2D texture coordinates of $\mathbf{s_0}$, $\mathbf{s_1}$ and $\mathbf{s_2}$, respectively. **Tgt** and **BTgt** are obtained by solving

$$\mathbf{\Delta Pos_1} = \mathbf{\Delta UV_1}[0]\mathbf{Tgt} + \mathbf{\Delta UV_1}[1]\mathbf{BTgt},$$
$$\mathbf{\Delta Pos_2} = \mathbf{\Delta UV_2}[0]\mathbf{Tgt} + \mathbf{\Delta UV_2}[1]\mathbf{BTgt}.$$

We now have the solution to this system:

$$\mathbf{Tgt} = \frac{\mathbf{\Delta Pos_1} \times \mathbf{\Delta UV_2}[1] - \mathbf{\Delta Pos_2} \times \mathbf{\Delta UV_1}[1]}{r},$$
$$\mathbf{BTgt} = \frac{\mathbf{\Delta Pos_2} \times \mathbf{\Delta UV_1}[0] - \mathbf{\Delta Pos_1} \times \mathbf{\Delta UV_2}[0]}{r},$$

where $r = \mathbf{\Delta UV_1}[0] \times \mathbf{\Delta UV_2}[1] - \mathbf{\Delta UV_1}[1] \times \mathbf{\Delta UV_2}[0]$, and $\cdot$ denotes the dot product and $\Delta\mathbf{UV_i}[0]$ and $\Delta\mathbf{UV_i}[1]$ denotes the U and V texture coordinates of the ith summit.

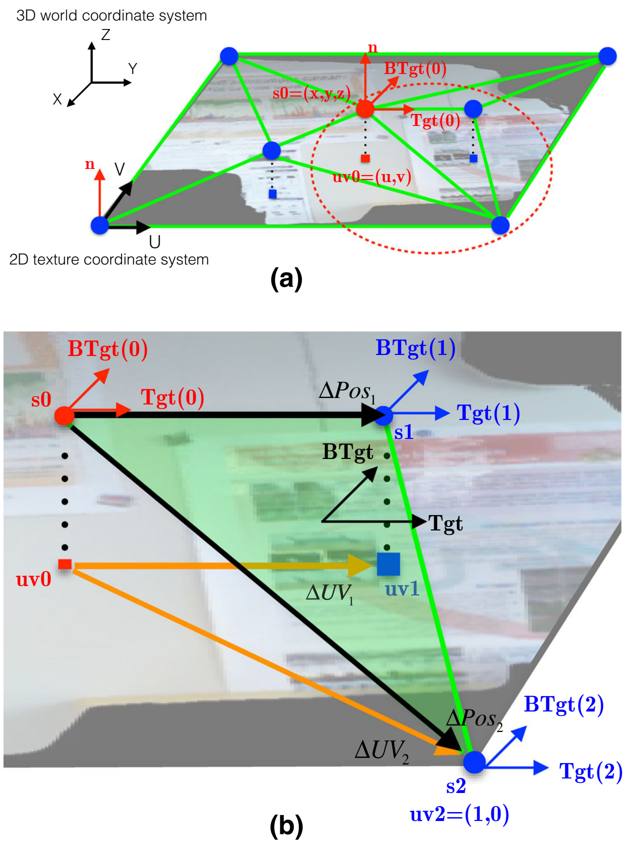


Fig. 27 A toy example that illustrates the computation of the orthonormal basis used to compute the Normal image. **a** For each vertex of the triangulation, we compute the orthonormal basis that consists of the normal, the tangent and the bitangent vectors. **b** Illustration of the computation of the tangent and bitangent vectors for the *triangle circled in red* in (**a**). These vectors are accumulated into the tangent and bitangent vectors of each summit (Color figure online)

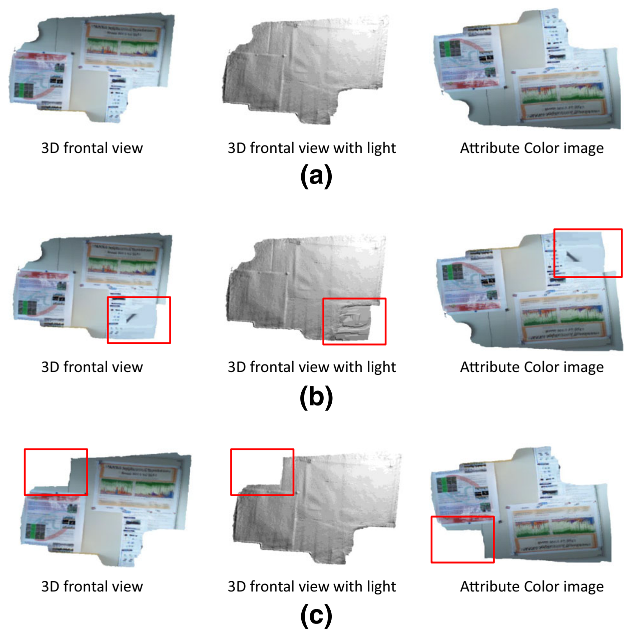


Fig. 28 Example of possible edition of a 3D model constructed using our proposed method. We implemented a very basic 2D editing tool (more advanced 2D edition is out of the scope of this paper) that allows inpainting missing data in both color and geometry shown in (**a**). **a** Initial 3D model and its attribute Color image. **b** Edited 3D model after painting to fill part of missing data. **c** Edited 3D model after removing some undesired parts

Then, we recompute the tangent and bitangent vectors of the summits of the triangle. Namely, for each $i \in \{0, 1, 2\}$,

$$\mathbf{Tgt}(\mathbf{s}_\mathrm{i}) \leftarrow \mathbf{Tgt}(\mathbf{s}_\mathrm{i}) + \mathbf{Tgt},$$
$$\mathbf{BTgt}(\mathbf{s}_\mathrm{i}) \leftarrow \mathbf{BTgt}(\mathbf{s}_\mathrm{i}) + \mathbf{BTgt},$$

where $\mathbf{Tgt}(\mathbf{s}_\mathrm{i})$ and $\mathbf{Btgt}(\mathbf{s}_\mathrm{i})$ are the tangent and bitangent vectors, respectively, of the ith summit. We note that the two vectors are both initialised to be the $\mathbf{0}$ vector. Once each face of the mesh has been processed, for each vertex the basis is made orthonormal.[9]

Thereafter, for each pixel in the Bump image we compute the coordinates of its normal vector $\mathbf{n}$ in its local normal, tangent and bitangent basis. To quickly identify which triangle a pixel belongs to, we draw each triangle (in the 2D space of texture coordinates) with its index as color. Then triangle identification reduces to looking for the color value in an index image. First, for each pixel we compute its barycentric coordinates (in texture coordinate space) in the triangle to which the pixel belongs and identify the corresponding interpolated normal $\mathbf{N}$, tangent $\mathbf{Tgt}$ and bitangent $\mathbf{BTgt}$ vectors at the pixel location (the three vectors are made orthonormal). Second, we compute the coordinates $\hat{\mathbf{n}}$ of the actual normal vector $\mathbf{n}$ at the pixel in its local basis:

$$\hat{\mathbf{n}}[\mathbf{0}] = \mathbf{Tgt} \cdot \mathbf{n},$$
$$\hat{\mathbf{n}}[\mathbf{1}] = \mathbf{BTgt} \cdot \mathbf{n},$$
$$\hat{\mathbf{n}}[\mathbf{2}] = \mathbf{N} \cdot \mathbf{n}.$$

Finally, we map each coordinate value so that it ranges between 0 and 255 (namely, the mapped coordinates are $\mathbf{NMap}[i] = 255 \times \frac{\hat{\mathbf{n}}[\mathbf{i}]+1}{2}$, $i \in \{0, 1, 2\}$) and store it into the pixel values of the normal image.

Transparency effect Because empty regions of the Bump images (i.e., regions where the mask values are 0) may be covered by triangles in the (convex) 2D Delaunay triangulation (as shown in the red circles in Fig. 23), we employ the Mask image to compute the alpha channel for transparency effects. The alpha channel of the color image (mapped onto the 3D mesh) is computed by setting alpha values to 0 for pixels with Mask value of 0 and to 255 otherwise. As shown

[9] (1) The tangent vector is made orthogonal to the normal vector and normalised and (2) the bitangent vector is made orthogonal to both the normal and tangent vectors and then normalised.

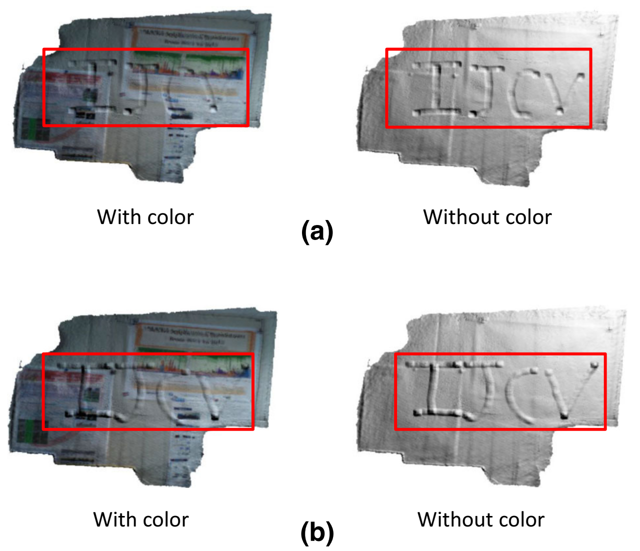


Fig. 29 Example of possible modifications of the 3D geometry of a 3D model constructed using our proposed method. We simply modified the Z components (increase or decrease its value with a Gaussian pattern) of the Bump image to extrude or dig holes in the wall. Note that each edition was obtained in a few seconds, without much practicing. **a** Edited 3D model after modifying its 3D geometry (digging holes in this case). **b** Edited 3D model after modifying its 3D geometry (extruding moles in this case)

in Fig. 24, this allows to obtain correct rendering of the 3D model.

6.2 Easy Editing of the 3D Model

Because of imperfect tracking, imperfect noise correction or incomplete scanning, constructed 3D models often contain remaining outliers and holes. Enabling the function of easily correcting undesired artefacts is of great interest to produce high quality 3D models, even if the corrections have to be made interactively. Thanks to the parametric surface representation we output with our proposed method, this function becomes executable. This is because for each object in the scene, which is modelled using a surface patch, its local geometry and appearance is encoded in two 2D images. Therefore, by simply modifying the 2D attribute images we can modify the 3D geometry and appearance of the object. Editing our produced 3D model then reduces to editing its 2D image attributes, which can be done easily and intuitively. Figures 28 and 29 show examples (with a simple 3D model) of possible editing that can be done using our produced 3D models.

7 Conclusion

We proposed a novel 3D representation using a set of parametric surface patches that achieves accurate, compact and efficient 3D reconstruction from an RGB-D image sequence. By projecting the scene or objects onto different parametric surface patches that segment the scene, we introduced attributes to each surface patch in the scene, with which we can reduce significantly the size required for the scene representation and thus we can generate a global textured model that is light in memory use and, nevertheless, accurate and easy to update with live RGB-D measurements. We remark that it is important to choose the most appropriate parametric surface depending on the input data to ensure lossless representation of the scene. Developing a strategy for finding parametric surfaces that best fit the input data (which is out of the scope of this paper) is an interesting direction for future work. We also remark that our proposed flexible 3D representation has an advantage against standard TSDF representations at very large scale. In these situations, the standard TSDF allows loop closure (crucial to correct drift errors) at the cost of decreasing resolution (thus loosing details). As we have already attempted using our proposed 3D scene representation, it is possible to achieve efficient and accurate loop closure without any loss in details even at very large scales (Thomas and Sugimoto 2014). More detail in this direction will be reported in another venue.

Acknowledgements This work is in part supported by Grant-in-Aid for Scientific Research of the Ministry of Education, Culture, Sports, Science and Technology of Japan.

References

Anasosalu, P.K., Thomas, D., & Sugimoto, A. (2013). Compact and accurate 3-D face modeling using an RGB-D camera: Let s open the door to 3-D video conference. In: *Proc. of CDC4CV*.

Besl, P. J., & McKay, N. D. (1992). A method for registration of 3-D shapes. *IEEE Transactions on PAMI*, *14*(2), 239–256.

Blanz, V., & Vetter, T. (2003). Face recognition based on fitting a 3D morphable model. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *25*(9), 1063–1074.

Canny, J. (1986). A Computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *6*, 679–698.

Chen, J., Bautembach, D., & Izadi, S. (2013). Scalable real-time volumetric surface reconstruction. *ACM Transactions on Graphics*, *32*(4), 113:1–113:8.

Davison, A., Reid, I., Molton, N., & Stasse, O. (2007). Monoslam: Real-time single camera slam. *IEEE Transactions on PAMI*, *29*, 1052–1067.

Henry, P., Fox, D., Bhowmik, A., & Mongia, R. (2013). Patch volumes: Segmentation-based consistent mapping with RGB-D cameras. In: *Proceedings of 3DV'13*.

Henry, P., Krainin, M., Herbst, E., Ren, X., & Fox, D. (2012). RGB-D mapping: Using Kinect-style depth cameras for dense 3D modelling of indoor environments. *International Journal of Robotics Research*, *31*(5), 647–663.

Hernandez, M., Choi, J., & Medioni, G. (2012). Laser scan quality 3-D face modeling using a low-cost depth camera. In: *Proceedings of the 20th European signal processing conference (EUSIPCO)*, pp. 1995–1999.

Jaeggli, T., Konenckx, T., & Gool, L. (2003). Online 3D acquisition and model integration. In: *Proceedings of Procam'03*.

Kahler, O., Prisacariu, V. A., Ren, C. Y., Sun, X., Torr, P. H. S., & Murray, D. W. (2015). Very high frame rate volumetric integration of depth images on mobile device. In: *IEEE Transactions on Visualization and Computer Graphics (proceedings international symposium on mixed and augmented reality)*.

Kazhdan, M., Bolitho, M., & Hoppe, H. (2006). Poisson surface reconstruction. In: *Proceedings of Eurographics symposium on geometry*.

Lengyel, E. (2001). Computing tangent space basis vectors for an arbitrary mesh. In: *Terathon Software 3D Graphics Library*.

Lowe, D. G. (1999). Object recognition from local scale-invariant features. In: *Proceedings of ICCV*, pp. 1150–1157.

Neibner, M., Zollhofer, M., Izadi, S., & Stamminger, M. (2013). Real-time 3D reconstruction at scale using voxel hashing. *ACM Transactions on Graphics*, *32*(6), 169:1–169:11.

Newcombe, R., Izadi, S., Hilliges, O., Molyneaux, D., Kim, D., Davison, A., Kohli, P., Shotton, J., Hodges, S., & Fitzgibbon, A. (2011). Kinectfusion: Real-time dense surface mapping and tracking. In: *Proceedings of ISMAR'11*, pp. 127–136.

Nguyen, C., Izadi, S., & Lovell, D. (2012). Modeling kinect sensor noise for improved 3D reconstruction and tracking. In: *Proceedings of 3DIM/PVT'12*, pp. 524–530.

Pfister, H., Zwicker, M., Baar, J., & Gross, M. (2000). Surfels: Surface elements as rendering primitives. In: *ACM Transactions on Graphics (Proceedings of SIGGRAPH'00)*.

Roth, H., & Vona, M. (2012). Moving volume kinectfusion. In: *Proceedings of BMVC*.

Segal, A., Haehnel, D., & Thrun, S. (2009). Generalized-ICP. In: *Robotics: Science and systems*.

Steinbrucker, F., Kerl, C., Sturm, J., & Cremers, D. (2013). Large-scale multi-resolution surface reconstruction from RGB-D sequences. In: *Proceedings of international conference on computer vision (ICCV 13)*.

Thomas, D., & Sugimoto, A. (2013). A flexible scene representation for 3D reconstruction using an RGB-D camera. In: *Proceedings of ICCV*.

Thomas, D., & Sugimoto, A. (2014). A two-stage strategy for real-time dense 3D reconstruction of large-scale scenes. In: *Proceedings of ECCV workshops'14 (CDC4CV)*.

Weise, T., Wismer, T., Leibe, B., & Gool, L. (2009). In-hand scanning with online loop closure. *Proceedings of ICCV Workshops'09*, pp. 1630–1637.

Whelan, T., McDonald, J., Kaess, M., Fallon, M., Johansson, H., & Leonard, J. (2012). Kintinuous: Spatially extended kinectfusion. *Proceedings of RSS Workshop on RGB-D: Advanced reasoning with depth camera*.

Zeng, M., Zhao, F., Zheng, J., & Liu, X. (2013). Octree-based fusion for realtime 3D reconstruction. *Transaction of Graphical Models*, *75*(3), 126–136.

Zhou, Q.-Y., & Koltun, V. (2013). Dense scene reconstruction with points of interest. *ACM Transaction on Graphics*, *32*(4), 112:1–112:8.

Zhou, Q.-Y., Miller, S., & Koltun, V. (2013). Elastic fragments for dense scene reconstruction. In: *Proceedings of ICCV*.