Auton Robot
DOI 10.1007/s10514-017-9624-2

Large-scale, real-time 3D scene reconstruction on a mobile device

Ivan Dryanovski[1] · **Matthew Klingensmith**[2] · **Siddhartha S. Srinivasa**[2] · **Jizhong Xiao**[3]

Received: 9 December 2015 / Accepted: 11 January 2017

Abstract Google's *Project Tango* has made integrated depth sensing and onboard visual-intertial odometry available to mobile devices such as phones and tablets. In this work, we explore the problem of large-scale, real-time 3D reconstruction on a mobile devices of this type. Solving this problem is a necessary prerequisite for many indoor applications, including navigation, augmented reality and building scanning. The main challenges include dealing with noisy and low-frequency depth data and managing limited computational and memory resources. State of the art approaches in large-scale dense reconstruction require large amounts of memory and high-performance GPU computing. Other existing 3D reconstruction approaches on mobile devices either only build a sparse reconstruction, offload their computation to other devices, or require long post-processing to extract the geometric mesh. In contrast, we can reconstruct and render a global mesh on the fly, using only the mobile device's CPU, in very large ($300\,m^2$) scenes, at a resolutions of 2–3 cm. To achieve this, we divide the scene into spatial volumes indexed by a hash map. Each volume contains the truncated signed distance function for that area of space, as well as the mesh segment derived from the distance function. This approach allows us to focus computational and memory resources only in areas of the scene which are currently observed, as well as leverage parallelization techniques for multi-core processing. Furthermore, we describe an on-device post-processing method for fusing datasets from multiple, independent trials, in order to improve the quality and coverage of the reconstruction. We discuss how the particularities of the devices impact our algorithm and implementation decisions. Finally, we provide both qualitative and quantitative results on publicly available RGB-D datasets, and on datasets collected in real-time from two devices.

Keywords 3D reconstruction · Mobile technology · SLAM · Computer vision · Mapping · Pose estimation

This is one of several papers published in *Autonomous Robots* comprising the "Special Issue on Robotics Science and Systems".

This work is supported in part by U.S. Army Research Office under Grant No. W911NF0910565, Federal Highway Administration (FHWA) under Grant No. DTFH61-12-H-00002, Google under Grant No. RF-CUNY-65789-00-43, Toyota USA Grant No. 1011344 and U.S. Office of Naval Research Grant No. N000141210613.

Electronic supplementary material The online version of this article (doi:10.1007/s10514-017-9624-2) contains supplementary material, which is available to authorized users.

✉ Jizhong Xiao
jxiao@ccny.cuny.edu

Ivan Dryanovski
idryanovski@gradcenter.cuny.edu

Matthew Klingensmith
mklingen@andrew.cmu.edu

Siddhartha S. Srinivasa
siddh@cs.cmu.edu

[1] Department of Computer Science, The Graduate Center, The City University of New York (CUNY), 365 Fifth Avenue, New York, NY 10016, USA

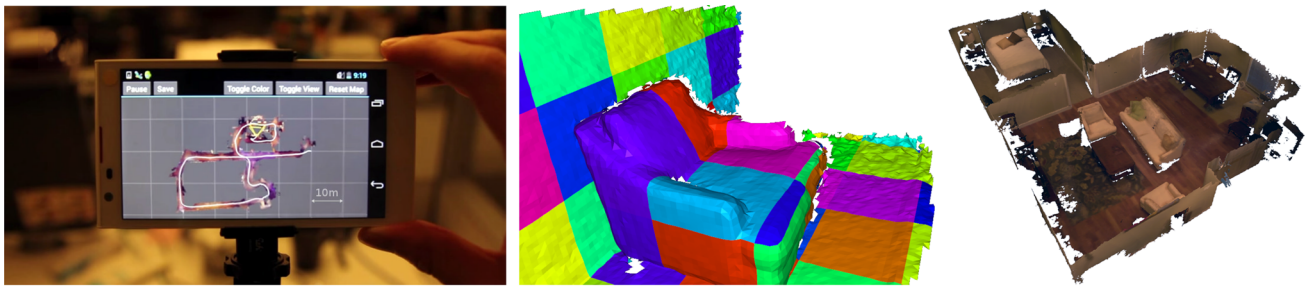
[2] Carnegie Mellon Robotics Institute, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA

[3] Electrical Engineering Department, The City College of New York, 160 Convent Avenue, New York, NY 10031, USA

1 Introduction

Recently, mobile phone manufacturers have started adding embedded depth and inertial sensors to mobile phones and tablets (Fig. 1). In particular, the devices we use in this work,

Published online: 24 February 2017



(a) (b) (c)

Fig. 1 **a** Our system reconstructing a map of an office building floor on a mobile device in real-time. **b** Visualization of the geometric model consististing of mesh segments corresponding to spatial volumes. **c** Textured reconstruction of an apartment, at a resolution of 2 cm

Fig. 2 Google's *Project Tango* developer devices: mobile phone (*left*) and tablet (*right*)

Google's *Project Tango* (2014) phone and tablet (Fig. 2) have very small active infrared projection depth sensors combined with high-performance IMUs and wide field of view cameras. Other devices, such as the Occipital Inc. *Structure Sensor* (2014) have similar capabilities. These devices offer an onboard, fully-integrated sensing platform for 3D mapping and localization, with applications ranging from mobile robots to handheld, wireless augmented reality (AR).

Real-time 3D reconstruction is a well-known problem in computer vision and robotics, and is a subset of Simultaneous Localization and Mapping (SLAM). The task is to extract the true 3D geometry of a real scene from a sequence of noisy sensor readings online, while simultaneously estimating the pose of the camera. Solutions to this problem are useful for robotic and human-assistive navigation, mapping, object scanning, and more. The problem can be broken down into two components: localization (i.e. estimating the sensor's pose and trajectory), and mapping (i.e. reconstructing the scene geometry and texture).

Consider house-scale (300 m^2) real-time 3D mapping and localization on a *Tango* device. A user moves around a building, scanning the scene. At house-scale, we are only concerned with features with a resolution of about 2–3 cm (walls, floors, furniture, appliances, etc.). We impose the following requirements on the system:

– To facilitate scanning, real-time feedback must be given to the user on the device's screen.
– The entire 3D reconstruction must fit inside the device's limited (2–4 GB) memory. Furthermore, the entire mesh model, and not just the current viewport, must be available at any point. This is in part to aid the feedback requirement by visualizing the mesh from different perspectives. More importantly, this enables applications like AR or gaming to take advantage of the scene geometry for tasks such as collision detection, occlusion-aware rendering, and model interactions.
– The system must be implemented without using GPU resources. The motivating factors for this restriction is that GPU resources might not be available on the specific hardware. Even if they are, there are many algorithms already competing for the GPU cycles: in the case of the *Tango* tablet, feature tracking for pose estimation and structured light pattern-matching for depth image extraction are already using the GPU. We want to leave the rest of the GPU cycles free for user applications built on top of the 3D reconstruction system.

3D mapping algorithms involving occupancy grids Elfes (1989), keypoint mapping (Klein and Murray 2007) or point clouds (Rusinkiewicz et al. 2002; Tanskanen et al. 2013; Weise et al. 2008) already exist for mobile phones at small scale—but at the scales we are interested in, their reconstruction quality is limited. Occupancy grids suffer from aliasing and are memory-dense, while point-based methods cannot reproduce surface or volumetric features of the scene without intensive post-processing.

Furthermore, many of the 3D mapping algorithms that use RGB-D data are designed to work with sensors like the Microsoft Kinect (2015), which produce higher-quality VGA depth images at 30 Hz. The density of the depth data allows for high-resolution reconstructions at smaller scales; its high update frequency allows it to serve as the backbone for camera tracking.

In comparison, the depth data from the *Project Tango* devices is much more limited. For both devices, the data is available at a rate between 3 and 5 Hz. This makes real-time

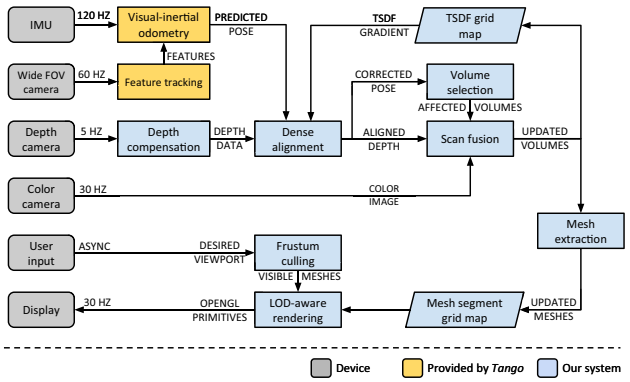


Fig. 3 Overview of te scene reconstruction system. Components contributed by our work are highlighted in *blue* (Color figure online)

pose tracking from depth data a much more challenging problem. Fortunately, *Project Tango* provides an out-of-the-box solution for trajectory estimation based on a visual-inertial odometry system using the device's wide-angle monocular camera. We use depth data only to make small optional refinements to that trajectory. This allows us to focus mainly on the mapping problem.

We present an overview of our entire system in Fig. 3. We discuss each section component in detail, beginning with Sect. 3, which describes the platform's hardware, sensors, and built-in motion tracking capabilities. We also present the depth measurement model we use throughout the paper. Our main contributions here is an analysis of the sensor model and error of our particular hardware. We propose a novel calibration procedure for removing bias in the depth data.

In Sect. 4, we discuss the theory of 3D reconstruction using a truncated signed distance field (TSDF), introduces by Curless and Levoy (1996), and used by manystate-of-the-art real-time 3D reconstruction algorithms (Newcombe et al. 2011; Whelan et al. 2013; Whelan and Kaess 2013; Bylow et al. 2013; Nießner et al. 2013). The TSDF stores a discretized estimate of the distance to the nearest surface in the scene. While allowing for very high-quality reconstructions, the TSDF is very memory-intensive. The size of the TSDF needed to reconstruct a large scene may be on the order of several to tens of gigabytes, which is beyond the capabilities of current mobile devices. We make use of a two-level data structure based on the work of Nießner et al. (2013). The structure consists of coarse 3D volumes that are dynamically allocated and stored in a hash table; each volume contains a finer, fixed-size voxel grid which stores the TSDF values. We minimize the memory footprint of the data by using integers instead of real numbers to do the TSDF filtering. Our main contributions here begin by how we adapt the TSDF fusion algorithms, including dynamic truncation distances and space carving techniques, in order to improve reconstruction quality from the noisy data. We further describe how to use the sensor model in order to switch to the integer representation with the lowest possible data loss from discretization. We discuss how this data discretization impacts the behavior and convergence of the TSDF filter. Finally, we present a dense alignment algorithm which uses the gradient information stored in the TSDF grid to correct for drift in the built-in pose estimation.

In Sect. 5, we present two methods for updating the data structure from incoming depth data: voxel traversal and voxel projection. While each of the algorithms has been described before, we analyze their applicability to the different data formats that we have (mobile phone vs. tablet). Furthermore, we discuss how we can leverage the two-tier data structure to efficiently parallelize the update algorithms in order to take advantage of multi-core CPU processing.

In Sect. 6, we discuss how we extract and visualize the mesh. We use an incremental version of Marching Cubes (Lorensen and Cline 1987), adapted to operate only on relevant areas of the volumetric data structure. Marching Cubes is a well-studied algorithm; similar to before, our key contribution lies in describing how to use the data structure for parallelizing the extraction problem, and techniques for minimizing the amount of recalculations needed. We also discuss efficient mesh rendering. Some of our bigger mesh reconstructions (Fig. 13) reach close to 1 million vertices and 2 million faces, and we found that attempting to directly render meshes of this size quickly overwhelms the mobile device, leading to overheating and staggered performance.

In Sect. 7, we describe offline post-processing which can be carried out on-device to improve the quality of the reconstruction for very large environments where the our system is unable to handle all the accumulated pose drift. Furthermore, we present our work on extending scene reconstruction beyond a single dataset collection trial. A reconstruction created from a single trial is limited by the device's battery life. The operator might not be able to get coverage of the entire scene in a single trial, requiring to revisit parts later. Thus, we present a method for fusing multiple datasets in order to create a single reconstruction as an on-device post-processing step. The datasets can be collected independently from different starting locations.

In Sect. 8, we present qualitative and quantitative results on publicly available RGB-D datasets (Sturm et al. 2012), and on datasets collected in real-time from two devices. We compare different approaches for creating and storing the TSDF in terms of memory efficiency, speed, and reconstruction quality. Finally, we conclude with Sect. 9, which discusses possible areas for further work.

2 Related work

Mapping paradigms generally fall into one of two categories: landmark-based (or *sparse*) mapping, and high-resolution *dense* mapping. While sparse mapping generates a metri-

cally consistent map of landmarks based on key features in the environment, dense mapping globally registers all sensor data into a high-resolution data structure. In this work, we are concerned primarily with dense mapping, which is essential for high quality 3D reconstruction.

Because mobile phones typically do not have depth sensors, previous works (Tanskanen et al. 2013; Newcombe et al. 2011; Engel et al. 2014) on dense reconstruction for mobile phones have gone to great lengths to extract depth from a series of registered monocular camera images. Our work is limited to new mobile devices with integrated depth sensors [such as the Google *Tango* devices (2014) and Occiptal Structure (2014) devices]. This allows us to save our memory and CPU budget for the 3D reconstruction itself, but limits the kinds of hardware we consider. Recent work by Schöps et al. (2015) integrates the system described in this paper with monocular depth estimation from motion, extending our mapping method to domains where a depth sensor is unavailable.

One of the simplest means of dense 3D mapping is storing multiple registered point clouds. These point-based methods (Rusinkiewicz et al. 2002; Tanskanen et al. 2013; Weise et al. 2008; Engel et al. 2014) naturally convert depth data into projected 3D points. While simple, point clouds fail to capture local scene structure, are noisy, and fail to capture *negative* (non-surface) information about the scene. This information is crucial to scene reconstruction under high levels of noise (Klingensmith et al. 2014). To deal with these problems, other recent works have added surface normal and patch size information to the point cloud (called *surfels*), which are filtered to remove noise. Surfels have been used in other large-scale 3D mapping systems, such as Whelan et al. (2015) *Elastic Fusion*. However in this work, we are interested in volumetric, rather than point-based approaches to surface reconstruction.

Elfes (1989) introduced Occupancy Grid Mapping, which divides the world into a voxel grid containing occupancy probabilities. Occupancy grids preserve local structure, and gracefully handle redundant and missing data. While more robust than point clouds, occupancy grids suffer from aliasing, and lack information about surface normals and the interior/exterior of obstacles.

Attempts to extend occupancy grid maps to 3D have sometimes relied on octrees. Rather than storing a fixed-resolution grid, octrees store occupancy data in a spatially organized tree. In typical scenes, octrees reduce the required memory over occupancy grids by orders of magnitude. Octomap (Wurm et al. 2010) is a popular example of the octree paradigm. However, octrees containing only occupancy probability suffer from many of the same problems as occupancy grids: they lack information about the interior and exterior of objects, and suffer from aliasing. Further, octrees suffer from logarithmic reading, writing, and iteration times, and have very poor memory locality characteristics.

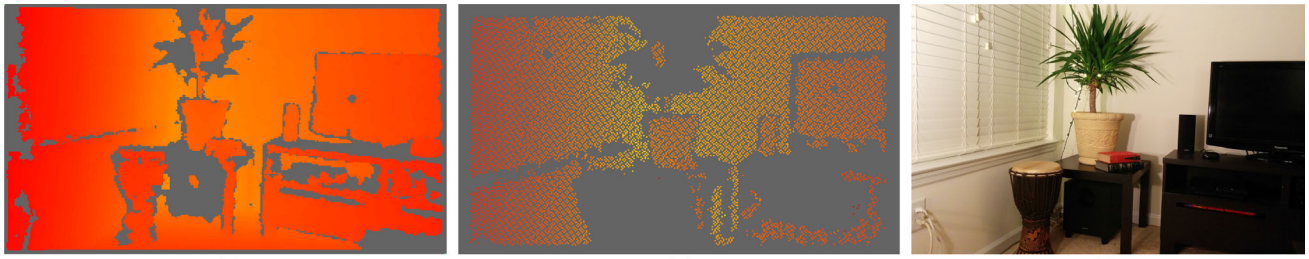
Curless and Levoy (1996) created an alternative to occupancy grids called the Truncated Signed Distance Field (TSDF), which stores a voxelization of the signed distance field of the scene. The TSDF is negative inside obstacles, and positive outside obstacles. The surface is given implicitly as the zero isocontour of the TSDF. While using more memory than occupancy grids, the TSDF creates much higher-quality surface reconstructions by preserving local structure.

Newcombe et al. (2011) uses a TSDF to simultaneously extract the pose of a moving depth camera and scene geometry in real-time. Making heavy use of the GPU for scan fusion and rendering, *Fusion* is capable of creating extremely high-quality, high-resolution surface reconstructions within a small area. However, like occupancy grid mapping, the algorithm relies on a single fixed-size 3D voxel grid, and thus is not suitable for reconstructing very large scenes due to memory constraints. This limitation has generated interest in extending TSDF fusion to larger scenes. Moving window approaches, such as *Kintinuous* Whelan et al. (2013) extend Kinect Fusion to larger scenes by storing a moving voxel grid in the GPU. As the camera moves outside of the grid, areas which are no longer visible are turned into a surface representation. Hence, distance field data is prematurely thrown away to save memory. As we want to save distance field data so it can be used later for post-processing, motion planning, and other applications, a moving window approach is not suitable.

Recent works have focused on extending TSDF fusion to larger scenes by compressing the distance field to avoid storing and iterating over empty space. Many have used *hierarchal* data structures such as octrees or KD-trees to store the TSDF (Zeng et al. 2013; Chen et al. 2013). However, these structures suffer from high complexity and complications with parallelism.

An approach by Nießner et al. (2013) uses a two-layer hierarchal data structure that uses spatial hashing (Teschner et al. 2003) to store the TSDF data. This approach avoids the needless complexity of other hierarchical data structures, boasting $\mathcal{O}(1)$ queries, and avoids storing or updating empty space far away from surfaces.

Our system adapts the spatially-hashed data structure of Nießner et al. (2013) to *Tango* devices. By carefully considering what parts of the space should be turned into distance fields at each timestep, we avoid needless computation and memory allocation in areas far away from the sensor. Unlike Nießner et al. (2013), we do not make use of any general purpose GPU computing. All TSDF fusion is performed on the mobile processor, and the volumetric data structure is stored on the CPU. Instead of rendering the scene via raycasting, we generate and maintain a polygonal mesh representation, and render the relevant segments of it. Since the depth sen-



(a) **(b)** **(c)**

Fig. 4 Comparison of depth output from the *Project Tango* mobile phone (**a**) and tablet (**b**), as well as a reference color image (**c**). The phone produces a depth image similar in density and quality to a Kinect camera. The tablet produces a point cloud with much fewer readings than on the mobile phone. Creating a depth image using the tablet's infra-red camera intrinsic parameters results in significant image gaps

sor found on the *Tango* device is significantly noisier than other commercial depth sensors, we reintroduce space carving (Elfes 1989) from occupancy grid mapping (Sect. 4.3) and dynamic truncation (Nguyen et al. 2012) (Sect. 4.2) into the TSDF fusion algorithm to improve reconstruction quality under conditions of high noise. The space carving and truncation algorithms are informed by a parametric noise model trained for the sensor using the method of Nguyen et al. (2012).

A preliminary version of this work was recently published (Klingensmith et al. 2015). Since then, we have released an open-source implementation of the mapping system[1] that has been used by other works to produce large-scale 3D reconstructions, most notably by Schöps et al. (2015). Other mapping systems targeting mobile devices have converged on similar solutions. Notably, Kähler et al. (2015) use a voxel hashing TSDF approach on a similar tablet device.

3 Platform

3.1 Hardware

We designed our system to work with two different platforms—the *Project Tango* cell phone and tablet (Fig. 2). The cell phone has a Qualcomm Snapdragon CPU and 2 GB of RAM. The tablet has an NVIDIA Tegra K1 CPU with 4 GB of RAM.

The two devices have a very similar set of integrated sensors. The sensors are listed in Fig. 3 (left). The devices have a low-cost inertial measurement unit (IMU), consisting of a tri-axis accelerometer and tri-axis gyroscope, producing inertial data at 120 Hz. Next, the devices have a a wide-angle (close to 180°), monochrome, global-shutter camera. The camera's wide field of view make it suitable for robust feature tracking. The IMU and wide-angle camera form the basis of the built-in motion tracking and localization capabilities of the *Tango* devices. We do not use the IMU data or this camera's images directly in our system.

Next, the devices have a high-resolution color camera, capable of producing images at 30 Hz. This camera is not used for motion tracking or localization, and we employ it in our system for adding color to the reconstructions.

Finally, both devices have a built-in depth sensor, based on structured infrared light. Both depth sensors on the two devices operate at a frequency of 5 Hz, but produce data with different characteristics. The cell phone produces depth images similar in density and coverage to those from a Kinect (albeit at a lower, QVGA resolution). On the other hand, the tablet produces data which is much sparser, as well as with poorer coverage on IR-absorbing surfaces (see Fig. 4). The data on the tablet arrives directly in the form of a point cloud. Point observations are calculated by using an IR camera and detecting features from an IR illumination pattern at sub-pixel accuracy. Converting to a depth image can be done by using the focal length of the tablet's IR camera as a reference. However, this produces a depth image with significant gaps (Fig. 4b). Creating a depth image with fewer and larger pixels eliminates the gaps, at the cost of the loss of angular resolution for each returned 3D point. Any algorithm we use must be aware of the data peculiarity.

3.2 Integrated motion estimation

Project Tango devices come with a built-in 30 Hz visual-inertial odometry (VIO) system, which we use as the main source of our pose estimation. The system is based on the work of Hesch et al. (2014). It fuses information from the IMU with observations of point features in the wide-angle camera images using a multi-state constraint Kalman filter (Mourikis and Roumeliotis 2007). The VIO system provides the pose of the device base frame B with respect to a global frame G at time t: ${}^{G}_{B}T_{t}$. We are also provided the (constant) poses of the depth camera D and the color camera C with respect to the device base, ${}^{B}_{D}T$ and ${}^{B}_{C}T$ respectively.

The first issue worth noting is that the VIO pose information, depth data, and color images all arrive asynchronously.

[1] http://www.github.com/personalrobotics/OpenChisel.

To resolve this, when a depth or color data arrives with a timestamp t', we buffer it and wait until we receive at least one VIO pose ${}^{\mathrm{G}}_{\mathrm{B}}T_t$ such that $t \geq t'$, and at least one pose such that $t \leq t'$. Then, we linearly interpolate for the pose between the two VIO readings.

The second issue is that like any open-loop odometry system, *Project Tango*'s VIO pose is subject to drift. This can result in model inaccuracies as the user revisits previously-scanned parts of the scene.

4 Theory

4.1 The signed distance function

We model the world based on a volumetric Signed Distance Function (SDF), denoted by $\mathrm{D} : \mathbb{R}^3 \to \mathbb{R}$ (Curless and Levoy 1996). For any point in the world $\mathbf{x}$, $\mathrm{D}(\mathbf{x})$ is the distance to the nearest surface, *signed* positive if the point is outside of objects and negative otherwise. Thus, the zero isocontour ($\mathrm{D} = 0$) encodes the surfaces of the scene. The Truncated Signed Distance Function (TSDF), denoted by $\Phi : \mathbb{R}^3 \to \mathbb{R}$, is an approximation of the SDF which stores distances to the surface within a small truncation distance τ around the surface (Curless and Levoy 1996):

$$\Phi(\mathbf{x}) = \begin{cases} \mathrm{D}(\mathbf{x}) & \text{if } |\mathrm{D}(\mathbf{x})| \leq \tau \\ \tau & \text{if } \mathrm{D}(\mathbf{x}) > \tau \\ -\tau & \text{if } \mathrm{D}(\mathbf{x}) < \tau \end{cases} \tag{1}$$

Consider an ideal depth sensor observing a point $\mathbf{p}$ on the surface of an object, which is a distance r away from the sensor origin.

$$r \equiv \|\mathbf{p}\| \tag{2}$$

Let $\mathbf{x}$ be a point which lies along the observation ray. The points $\mathbf{p}$ and $\mathbf{x}$ are expressed with respect to the sensor's frame of reference. We define the auxiliary parameters d (distance from sensor to $\mathbf{x}$) and u (distance from $\mathbf{x}$ to $\mathbf{p}$):

$$d \equiv \|\mathbf{x}\| \tag{3a}$$

$$u \equiv r - \|\mathbf{x}\| \tag{3b}$$

Thus, u is positive when $\mathbf{x}$ lies between the camera and the observation, and negative otherwise (see Fig. 5).

Let $\phi(\mathbf{x}, \mathbf{p})$ be the *observed* TSDF for a point $\mathbf{x}$, given an observation $\mathbf{p}$. Near the surface of the object, where $|u|$ is small (within $\pm\tau$), the signed distance function can be approximated by the distance along the ray to the endpoint of the ray. This is because we know that there is at least one point of the surface, namely the endpoint of the ray.

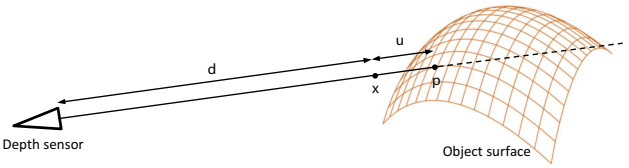


Fig. 5 Observation ray geometry. The depth sensor is observing a point $\mathbf{p}$ on the surface of an object. A point $\mathbf{x}$, which lies on the observation ray, is a distance d from the sensor origin, and a distance u from the observation $\mathbf{p}$

$$\phi(\mathbf{x}, \mathbf{p}) \approx u \tag{4}$$

Note that this approximation is better whenever the ray is approximately perpendicular to the surface, and worst whenever the ray is parallel to the surface. Because of this, some works (Bylow et al. 2013) instead approximate ϕ by fitting a plane around $\mathbf{p}$, and using the distance to that plane as a local approximation:

$$\phi(\mathbf{x}, \mathbf{p}) \approx -u(\mathbf{p} \cdot \mathbf{n_p}) \tag{5}$$

where $\mathbf{n_p}$ is the local surface normal at $\mathbf{p}$. In general, the planar approximation (5) is much better than the point-wise approximation (4), especially when surfaces are nearly parallel to the sensor but computing surface normals is not always computationally feasible. Both approximations of ϕ are defined only in the truncation region of $d \in [r - \tau, r + \tau]$.

We are interested in estimating the value of the TSDF from subsequent depth observations at discrete time instances. Curless and Levoy (1996) show that by taking a weighted running average of the distance measurements over time, the resulting zero-isosurface of the TSDF minimizes the sum-squared distances to all the ray endpoints. Following their work, we introduce a weighting function $\mathrm{W} : \mathbb{R}^3 \to \mathbb{R}^+$. Similarly to the TSDF, the weighting function is defined only in the truncation range. We draw the weighting function from the uncertainty model of the depth sensor. The filter is defined by the following transition equations:

$$\Phi_{k+1}(\mathbf{x}) = \frac{\Phi_k(\mathbf{x})\mathrm{W}_k(\mathbf{x}) + \phi_k(\mathbf{x}, \mathbf{p})w_k(\mathbf{x}, \mathbf{p})}{\mathrm{W}_k(\mathbf{x}) + w_k(\mathbf{x}, \mathbf{p})} \tag{6a}$$

$$\mathrm{W}_{k+1}(\mathbf{x}) = \mathrm{W}_k(\mathbf{x}) + w_k(\mathbf{x}, \mathbf{p}) \tag{6b}$$

where for any point $\mathbf{x}$ within the truncation region, $\Phi(\mathbf{x})$ and $\mathrm{W}(\mathbf{x})$ are the state TSDF and weight, and ϕ and w are the corresponding observed TSDF and observation weight according to the depth observation $\mathbf{p}$. The filter is initialized with

$$\Phi_0(\mathbf{x}) = \text{undefined} \tag{7a}$$

$$\mathrm{W}_0(\mathbf{x}) = 0 \tag{7b}$$

for all points $\mathbf{x}$.

Ideally, the distance-weighting function $w(\mathbf{x}, \mathbf{p})$ should be determined by the probability distribution of the sensor, and should represent the probability that $\phi(\mathbf{x}, \mathbf{p}) = 0$. It is possible (Nguyen et al. 2012) to directly compute the weight from the probability distribution of the sensor, and hence compute the actual expected signed distance function. In favor of better performance, linear, exponential, and constant approximations of w can be used (Bylow et al. 2013; Curless and Levoy 1996; Newcombe et al. 2011; Whelan et al. 2013).

In our work, we use a constant approximation. The function is only defined in the truncation region $d \in [r - \tau, r + \tau]$, and normalizes to 1:

$$\int_{r-\tau}^{r+\tau} w(\mathbf{x}, \mathbf{p})\, \mathrm{d}d = 1 \tag{8}$$

4.2 Dynamic truncation distance

Following the approach of Nguyen et al. (2012), we use a dynamic truncation distance based on the noise model of the sensor rather than a fixed truncation distance. In this way, we have a truncation function:

$$\tau(z_{uv}) = \beta\sigma(r) \tag{9}$$

where $\sigma(r)$ is the standard deviation for an observation with a range r, and β is a scaling parameter which represents the number of standard deviations of the noise we want to integrate over. For the sake of simplicity, we approximate the range-based deviation using the sensor model for the depth-depth based deviation $\sigma(z_{uv})$, which we derived previously (34b).

$$\sigma(r) \approx \sigma(z) \tag{10}$$

Using the dynamic truncation distance has the effect that further away from the sensor, where depth measurements are less certain and sparser, measurements are smoothed over a larger area of the distance field. Nearer to the sensor, where measurements are more certain and closer together, the truncation distance is smaller.

The constant weighting function thus becomes

$$w(\mathbf{x}, \mathbf{p}) = \frac{1}{2\,\tau(r)} = \frac{1}{2\,\beta\,\sigma(z)} \tag{11}$$

where z is the z-component of $\mathbf{p}$.

4.3 Space carving

When depth data is highly noisy and sparse, the relative importance of negative data (that is, information about what parts of the space do not contain an obstacle) increases over positive data (Elfes 1989; Klingensmith et al. 2014). This is because rays passing through empty space constrain possible values of the signed distance field to be positive at all points along the ray, whereas rays hitting objects only inform us about the presence of a surface very near the endpoint of the ray. In this way, rays carry additional information about the value of SDF.

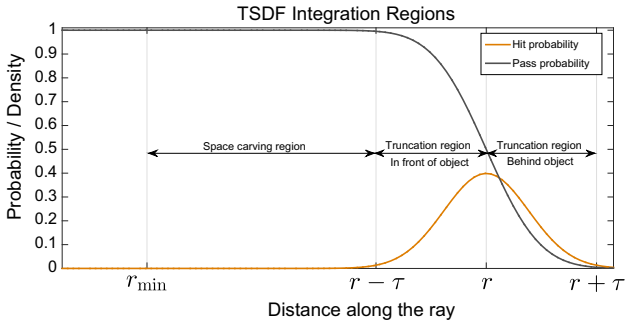


Fig. 6 The probability model for a given depth observation $\mathbf{p}$. The model consist of a hit probability $P(d = r)$ and pass probability $P(d < r)$, plotted against the distance d along the ray. We define two integration regions: a space carving region $[r_{\min}, r - \tau)$ and the truncation region $[r - \tau, r + \tau]$, where $r_{\min}$ is the minimum depth camera range, and τ is the truncation distance

So far, we have defined the TSDF observation function ϕ only within the *truncation region*, defined by $d \in [r - \tau, r + \tau]$. We found that it's highly beneficial to augment the region in which the filter operates by $[r_{\min}, r - \tau)$, where $r_{\min}$ is the minimum observation range of the camera (typically around 40 cm). We refer to this new region as the *space carving* region. The entire operating region for filter updates thus becomes the union of the space carving and truncation regions: $[r_{\min}, r + \tau]$ (Fig. 6).

One way to incorporate space-carving observations into the filter is to treat them as an absolute constraint on the values of the TSDF, and "reset" the filter.

$$\Phi_{k+1}(\mathbf{x}) = \tau(r) \tag{12a}$$
$$\mathrm{W}_{k+1}(\mathbf{x}) = 0 \tag{12b}$$

However, since it only takes one noisy depth measurement to incorrectly clear a voxel in this way (even when many previous observations indicate that a voxel is occupied), we instead choose to incorporate space-carving observations into the TSDF filter by treating them as regular observations. To do so, we must assign a TSDF value ϕ and weight w to points within the space-carving region (ϕ and w have so far been defined only within the truncation region). We choose the following values:

$$w(\mathbf{x}, \mathbf{p}) = \tau \qquad \text{if } d \in [r_{\min}, r - \tau) \tag{13a}$$
$$w(\mathbf{x}, \mathbf{p}) = w_{\mathrm{sc}} \qquad \text{if } d \in [r_{\min}, r - \tau) \tag{13b}$$

where w_{sc} is a fixed space-carving weight.

Space carving gives us two advantages: first, it dramatically improves the surface reconstruction in areas of very high noise, especially around the edges of objects (see Fig. 18). Further, it removes some inconsistencies caused by moving objects and localization errors. If space carving is not used, moving objects appear in the TSDF as permanent blurs, and localization errors result in multiple overlapping isosurfaces appearing. With space carving, old inconsistent data is removed over time.

4.4 Colorization

As in Bylow et al. (2013) and Whelan et al. (2013), we create textured surface reconstructions by directly storing color as volumetric data. We augment our model of the scene to include a color function $\Psi : \mathbb{R}^3 \rightarrow \mathbf{RGB}$, with a corresponding weight $\mathrm{G} : \mathbb{R}^3 \rightarrow \mathbb{R}^*$. We assume that each depth observation ray also carries a color observation, taken from a corresponding RGB image. As in Bylow et al. (2013), we have chosen RGB color space for the sake of simplicity, at the expense of color consistency with changes in illumination.

Color is updated in exactly the same manner as the TSDF. The update equation for the color function is

$$\Psi_{k+1}(\mathbf{x}) = \frac{\Psi_k(\mathbf{x})\mathrm{G}_k(\mathbf{x}) + \psi_k(\mathbf{p})\mathrm{g}_k(\mathbf{x},\mathbf{p})}{\mathrm{G}_k(\mathbf{x}) + \mathrm{g}_k(\mathbf{x},\mathbf{p})} \tag{14a}$$

$$\mathrm{G}_{k+1}(\mathbf{x}) = \mathrm{G}_k(\mathbf{x}) + \mathrm{g}_k(\mathbf{x},\mathbf{p}) \tag{14b}$$

where for any point $\mathbf{x}$ within the truncation region, $\Psi(\mathbf{x})$ and $\mathrm{G}(\mathbf{x})$ are the state color and color weight, and ψ and g are the corresponding observed color and color observation weight.

In both Bylow et al. (2013) and Whelan et al. (2013), the color weighting function is proportional to the dot product of the ray's direction and an estimate of the surface normal. But since surface normal estimation is computationally expensive, we instead use the same observation weight for both the TSDF and color filter updates:

$$\mathrm{g}(\mathbf{x},\mathbf{p}) = w(\mathbf{x},\mathbf{p}) \tag{15}$$

We must also deal with the fact that color images and depth data are asynchronous. In our case, depth data is often delayed behind color data by as much as 30 ms. So, for each depth scan, we project the endpoints of the rays onto the nearest color image in time to the depth image, accounting for the different camera poses due to movement of the device. Then, we use bilinear interpolation on the color image to determine the color of each ray.

4.5 Discretized data layout

We use a discrete approximation of the TSDF which divides the world into a uniform 3D grid of voxels. Each voxel contains an estimate of the TSDF and an associated weight. More precisely, we define the Normalized Truncated Signed Distance Function (NTSDF), denoted by $\overline{\Phi} : \mathbb{Z}^3 \rightarrow \mathbb{Z}$. For discrete voxel coordinate $\mathbf{v} = \left[v_i, v_j, v_k\right]^\mathsf{T}$, the NTSDF $\overline{\Phi}(\mathbf{v})$ is the discretized approximation of the TSFD, scaled by a normalization constant λ. We similarly define a discretized weight, denoted by $\overline{\mathrm{W}} : \mathbb{Z}^3 \rightarrow \mathbb{Z}^+$, which is the weight W scaled by a constant γ.

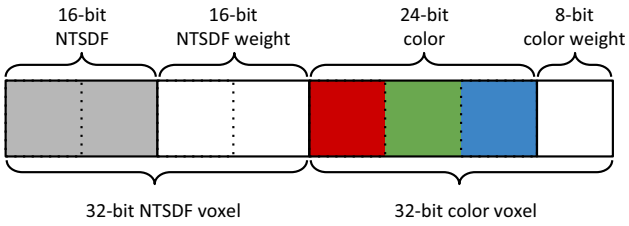


Fig. 7 Memory layout for a single voxel

$$\overline{\Phi}(\mathbf{v}) = \lfloor \lambda \Phi(\mathbf{x}) \rceil \tag{16a}$$

$$\overline{\mathrm{W}}(\mathbf{v}) = \lfloor \gamma \mathrm{W}(\mathbf{x}) \rceil \tag{16b}$$

$$\mathbf{v} = \left\lfloor \frac{1}{s_v}\mathbf{x} \right\rceil \tag{16c}$$

where s_v is the voxel size in meters, and $\lfloor \rceil$ is the round-to-nearest-integer operator.

In our implementation, the NTSDF and weight are packed into a single 32-bit structure. The first 16 bits are a signed integer distance value, and the last 16 bits are an unsigned integer weighting value. Color is similarly stored as a 32 bit integer, with 8 bits per color channel, and an 8 bit weight (Fig. 7). A similar method is used in Bylow et al. (2013), Handa et al. (2015) and Lorensen and Cline (1987) to store the TSDF.

We choose λ in a way to maximize the resolution in the 16-bit range $\left[-2^{15}, 2^{15}\right)$. By definition, the maximum TSDF value occurs at the maximum truncation distance. The truncation distance is a function of the reading range (9). The maximum reading range $r_{\max}$ is a fixed parameter of the depth camera usually chosen at around 3.5–4 m. Thus, we can define λ as:

$$\lambda = 2^{15}\tau(r_{\max}) \tag{17}$$

This formulation guarantees $\overline{\Phi}$ utilizes the entire $\left[-2^{15}, 2^{15}\right)$ range.

We similarly choose γ to maximize the weight resolution. By the definition of our sensor model, the lowest possible weight $\mathrm{W}_{\min}$ occurs at the maximum truncation distance $\tau(r_{\max})$. We define that weight to be 1, and scale the rest of the range accordingly:

$$\gamma = \frac{1}{\mathrm{W}_{\min}} \tag{18}$$

This formulation guarantees $\overline{\mathrm{W}} \geq 1$.

4.6 Discretized data updates

We define the weighted running average filter (WRAF) as a discrete-time filter for an arbitrary state quantity X and weight W with observation x and observation weight w with the following transition equations:

$$X_{k+1} = \frac{X_k W_k + x_k w_k}{W_k + w_k} \qquad X, x \in \mathbb{R} \tag{19a}$$

$$W_{k+1} = W_k + w_k \qquad W, w \in \mathbb{R}^+ \tag{19b}$$

In Sect. 4, we applied this filter to both the TSDF and color updates. However, since we store the data using integers, we need to define a discrete weighted running average filter (DWRAF) as a variation of the ideal WRAF where the state, observation, and weights are integers:

$$\widetilde{X}_k = \frac{X_k W_k + x_k w_k}{W_k + w_k} \qquad X, x \in \mathbb{Z}, \widetilde{X} \in \mathbb{R} \tag{20a}$$

$$X_{k+1} = \lfloor \widetilde{X}_k \rceil \tag{20b}$$

$$W_{k+1} = W_k + w_k \qquad w \in \mathbb{Z}^*, w \in \mathbb{Z}^+ \tag{20c}$$

The observation weight is a strictly positive integer, while the the state weight is a non-negative-integer (allowing the filter to be initialized with zero weight, corresponding to an unknown state).

We further define the the *observation delta* between the observation and the current state as:

$$\Delta x_k \equiv x_k - X_k \tag{21}$$

where $\Delta x \in \mathbb{R}$.

It can be shown that due to the rounding step in (20b), when the state weight is greater than or equal to twice the observation delta, then the state transition delta will be zero (the state remains the same).

$$w_k \geq 2|\Delta x_k| \implies X_{k+1} = X_k \tag{22}$$

This means that for any given state weight, there exists a minimum observation delta, and observations which are too close to the state will effectively be ignored. Since the state weight is monotonically increasing with each filter update, the minimum observation delta grows over time, causing the filter to ignore a wider range of observations.

Consider the example when the filter state $X \in \left[-2^{15}, 2^{15}\right]$ represents TSDF in the range $[-0.10, 0.10]$ m. When the filter has received 100 observations, each with an observation weight of 1, the minimum observation delta will be approximately 0.15 mm. At 10,000 observations, the minimum observation delta is 1.5 cm, etc. If observations with higher weights go in, the minimum observation delta will grow even quicker.

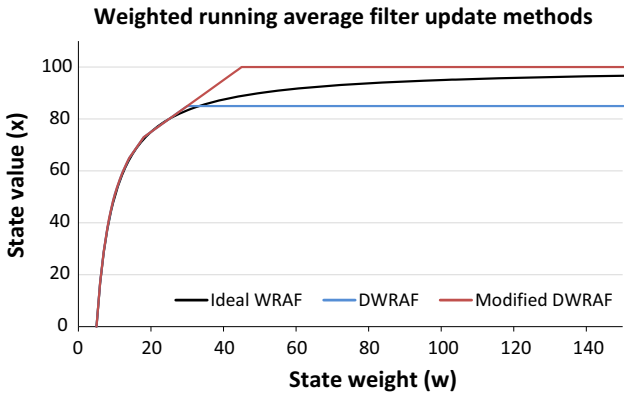


Fig. 8 Convergence behavior of the ideal weighted running average filter (WRAF), discrete WRAF, and modified discrete WRAF. The filter starts with a state weight of 5, and receives a series of updates with a constant observation weight w of 1 and a constant observation x of 100

The problem is much more significant with smaller discrete spaces, such as colors discretized in the $X \in [0, 255]$ range. A filter with a state weight of 100 will require an observation delta of 50, or approximately 20% color difference, in order to change its state. Once the state weight reaches 510, the filter will enter a steady state, and no subsequent observations will ever perturb it.

One way to deal with this issue is to impose an artificial upper maximum on the state weight, and prevent it from growing beyond that with new updates. This guarantees that the minimum observation delta is bound. This is a straightforward solution which might work for the distance filter, but not for the color filter, where the maximum weight would have to be set very low. We propose a different solution, which modifies the state transition equation (20b) to guarantee a response even at high state weights:

$$X_{k+1} = \begin{cases} X_k + 1 & \widetilde{X}_k - X_k \in (0, 0.5) \\ X_k - 1 & \widetilde{X}_k - X_k \in (-0.5, 0) \\ \lfloor \widetilde{X}_k \rceil, & \text{otherwise} \end{cases} \tag{23}$$

This preserves the behavior of the DWRAF, except in the cases when the the rounding would force the new state to be the same as the old one. In those cases, we enforce a state transition delta of 1 (or -1).

The three different filter behaviors (ideal WRAF, DWRAF, and modified DWRAF) are illustrated in Fig. 8. We simulate a filter update scenario where the filter starts with a state weight of 5, and receives a series of updates with a constant

observation weight w of 1 and a constant observation x of 100. The ideal WRAF exhibits an exponential convergence towards 100. The DWRAF approximates the convergence curve using piecewise-linear segments; once it reaches the saturation weight, it becomes clamped at 85. The modified DWRAF behaves like the DWRAF, except in the last segment, where it forces a linear convergence with a rate of 1.

In our implementation, we use the modified DWRAF filter implementation for the NTSDF and color updates.

4.7 Dynamic spatially-hashed volume grid

Unfortunately, a flat volumetric representation of the world using voxels is incredibly memory intensive. The amount of memory storage required grows as $\mathcal{O}(N^3)$, where N is the number of voxels per side of the 3D voxel array. For example, at a resolution of 3 cm, a 30 m TSDF cube with color would occupy 8 GB of memory. Worse, most of that memory would be uselessly storing unseen free space. Further, if we plan on exploring larger and larger distances using the mobile sensor, the size of the TSDF array must grow if we do not plan on allocating enough memory for the entire space to be explored.

For a large-scale real-time surface reconstruction application, a less memory-intensive and more dynamic approach is needed. Confronted with this problem, some works have either used octrees (Wurm et al. 2010; Zeng et al. 2013; Chen et al. 2013), or use a moving volume (Whelan et al. 2013). Neither of these approaches is desirable for our application. Octrees, while maximally memory efficient, have significant drawbacks when it comes to accessing and iterating over the volumetric data (Nießner et al. 2013). Every time an octree is queried, a logarithmic $\mathcal{O}(M)$ cost is incurred, where M is the depth of the Octree. In contrast, queries in a flat array are $\mathcal{O}(1)$. An octree stores pointers to child octants in each parent octant. The octants themselves may be dynamically allocated on the heap. Each traversal through the tree requires $\mathcal{O}(M)$ heap pointer dereferences in the worst case. Even worse, adjacent voxels may not be adjacent in memory, resulting in very poor caching performance (Chilimbi et al. 2000). Like Nießner et al. (2013), we found that using an octree to store the TSDF data to reduce iteration performance by an order of magnitude when compared to a fixed grid.

Instead of using an Octree, moving volume, or a fixed grid, we use a hybrid data structure introduced by Nießner et al. (2013). The data structure makes use of two levels of resolution: volumes and voxels. Volumes are spatially-hashed (Chilimbi et al. 2000) into a dynamic hash map. Each volume consists of a fixed grid of ${N_v}^3$ voxels, which are stored in a monolithic memory block. Volumes are allocated dynamically from a growing pool of heap memory as data is added, and are indexed in a spatial 3D hash map (Chilimbi et al. 2000) by their spatial coordinates. As in Chilimbi et al. (2000) and Nießner et al. (2013) we use the hash function: $\textbf{hash}(x, y, z) = p_1 x \oplus p_2 y \oplus p_3 z \bmod n$, where x, y, z are the 3D integer coordinates of the chunk, p_1, p_2, p_3 are arbitrary large primes, $\oplus$ is the xor operator, and n is the maximum size of the hash map.

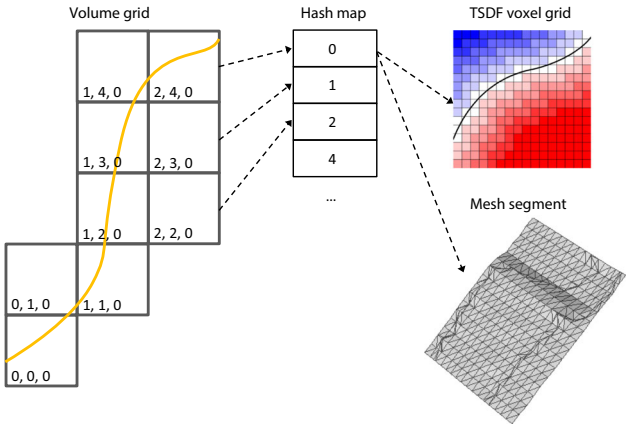


Fig. 9 The dynamic spatially-hashed volume grid data structure. Space is subdivided into coarse volumes (*left*). Volumes are indexed in a hash table (*center*). Each volume contains a bounded voxel grid and a mesh segment, corresponding to the isosurface in that volume (*right*)

Since volumes are a fixed size, querying data from the hybrid data structure involves rounding (or bit-shifting, if N_v is a power of two) a world coordinate to a chunk and voxel coordinate, and then doing a hash-map lookup followed by an array lookup. Hence, querying is $\mathcal{O}(1)$ (Nießner et al. 2013). Further, since voxels within chunks are stored adjacent to one another in memory, cache performance is improved while iterating through them. By carefully pre-selecting the size of volumes so that they corresponding to $\tau_{\max}$, we only allocate volumetric data near the zero isosurface, and do not waste as much memory on empty or unknown voxels.

Additionally, each volume also contains a mesh segment, represented by a list of vertices and face index triplets. The mesh may also optionally contain per-vertex color. Each mesh segment corresponds to the isosurface of the NTSDF data stored in the volume. The layout of the data structure is shown in Fig. 9.

4.8 Gradient-based dense alignment

To account compensate for VIO drift, we calculate a correction to the VIO pose by aligning the depth data to the isosurface ($\Phi = 0$) of the TSDF field. We denote this correction by ${}^{\overline{G}}_{G}T_t$, corresponding to the pose of the the global frame G with respect to the *corrected* global frame $\overline{G}$, at time t. This correction is initialized to identity, and is recalculated with each new depth scan which arrives, as described in the following subsection.

When a new point cloud $P = \{\mathbf{p}_i\}$ is available, we calculate the *predicted* position of all the points, using the previous correction ${}^{\bar{\mathrm{G}}}_{\mathrm{G}}T_{t-1}$ and the current VIO pose ${}^{\mathrm{G}}_{\mathrm{B}}T_t$:

$$ {}^{\bar{\mathrm{G}}}\mathbf{p}_i = {}^{\bar{\mathrm{G}}}_{\mathrm{G}}T_{t-1}\ {}^{\mathrm{G}}_{\mathrm{B}}T_t\ {}^{\mathrm{B}}_{\mathrm{D}}T_t\ {}^{\mathrm{D}}\mathbf{p}_i \tag{24} $$

Let there be some error function $e(P)$ which describes the deviation of the point cloud from the model isosurface, and a corresponding transformation ΔT, which, when applied to the point cloud, minimizes the error.

$$ \arg\min_{\Delta T} e(P) \tag{25} $$

Once we obtain ΔT, we can apply it to the previous corrective transform in order to estimate the new correction:

$$ {}^{\bar{\mathrm{G}}}_{\mathrm{G}}T_t = \Delta T\ {}^{\bar{\mathrm{G}}}_{\mathrm{G}}T_{t-1} \tag{26} $$

It remains to be shown how to formulate the error function and the minimization calculation. We do this by modifying the classic iterative closest point (ICP) problem (Chen and Medioni 1991), using the gradient informaiton from the voxel grid. For each point $\mathbf{p}_i$ in the point cloud P, there exists some point $\mathbf{m}_i$, which is the closest point to $\mathbf{p}_i$ on the isosurface. We can therefore define the per-point error $e(\mathbf{p}_i)$ and total weighted error $e(P)$ as:

$$ e(\mathbf{p}_i) = \|\mathbf{m}_i - \Delta T\ {}^{\bar{\mathrm{G}}}\mathbf{p}_i\|^2 \tag{27a} $$

$$ e(P) = \sum_i \left(w_i\ e(\mathbf{p}_i)\right) \tag{27b} $$

Next, we must find an appropriate value for $\mathbf{m}_i$. The ICP algorithm accomplishes this by doing a nearest-neighbor search into a set of model points. Instead, we will use the TSDF gradient. By the TSDF definition, the distance from any point $\mathbf{p}_i$ to the closest surface is given by the TSDF value. The direction towards the closest point is directly opposite the TSDF gradient. Thus,

$$ \mathbf{m} \approx -\frac{\nabla\Phi(\mathbf{p})}{\|\nabla\Phi(\mathbf{p})\|}\Phi(\mathbf{p}) \tag{28} $$

We can look up the gradient information in $\mathcal{O}(1)$ time per point. Therefore, this algorithm performs faster than the classical ICP formulation, which requires nearest-neighbor computations.

We approximate the gradient $\nabla\Phi$ by taking the difference between the corresponding TSDF for that voxel and and its three positive neighbors along the x, y, and z dimensions. Note that the approximations only holds when the TSDF has a valid value, which only occurs at distances up to the truncation distance τ away from the surface. This limits the convergence basin for the minimization to areas in space where $\Phi(\mathbf{p_i}) < \tau$. However, as long as each new VIO pose has a linear drift magnitude less then τ from the last calculated aligned pose, we are generally able to converge to the right corrected position. In practice, the approximation in (28) becomes better the closer we are to the surface. Thus, we perform the minimization iteratively, recalculating the corresponding points at each iteration.

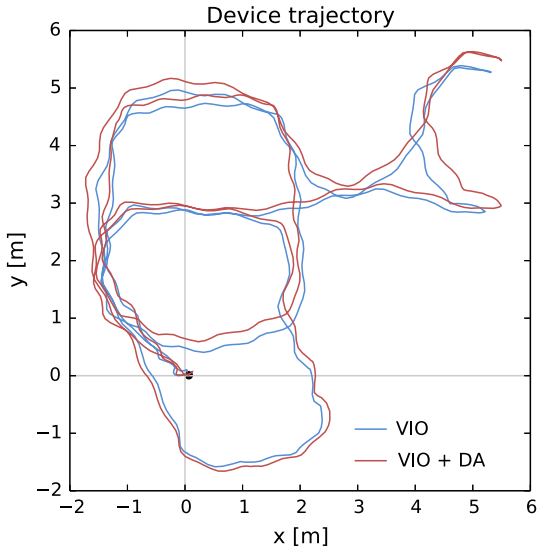


Fig. 10 Trajectories calculated online using *Project Tango*'s visual-inertial odometry (VIO), as well as the proposed dense alignment correction (VIO + DA)

Since we are aligning against a persistent global model, this effectively turns the open-loop VIO pose estimation into a closed-loop pose estimation. We found that in practice, the VIO pose is sufficiently accurate for reconstructing small (room-sized) scenes, if areas are not revisited. Using the proposed dense alignment method, we are able to correct for drift and "close" loops in mid-sized environments such as an apartment with several rooms. Figure 10 shows a comparison between the visual-inertial odometry (VIO) trajectory, as well as the corrected trajectory calculated using the dense alignment (DA). Figure 11 shows different views of the scene from the same experiment. Enabling dense alignment results in superior reconstruction quality.

5 Scan fusion

Updating the data structure requires associating voxels with the observations that affect them. We discuss two algorithms which can be used to accomplish this: *voxel traversal* and *voxel projection*, and compare their advantages and drawbacks.

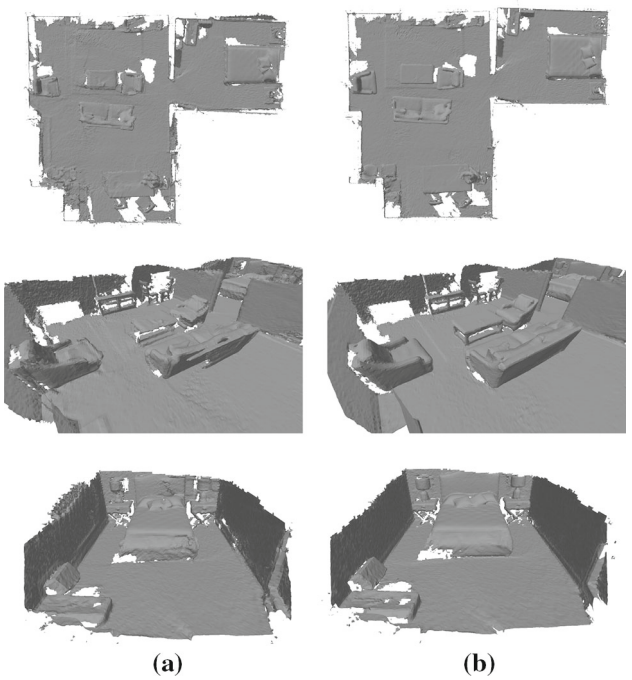


Fig. 11 Comparison of reconstructions using the trajectories in Fig. 10. Using dense alignment results in superior reconstruction quality. **a** Without dense alignment. **b** With dense alignment

Algorithm 1 Voxel traversal fusion

▷ *For each point observation:*
for $\mathbf{p} \in P$ **do**
  ▷ *Find all volumes intersected by observation.*
  $\mathcal{V} \leftarrow$ **RaytraceVolumes**()
  ▷ *Ensure all volumes are allocated*
  AllocateVolumes($\mathcal{V}$)
  ▷ *Find all voxels intersected by observation.*
  $\mathcal{X} \leftarrow$ **RaytraceVoxels**()
  ▷ *For each intersected voxel:*
  for $\mathbf{v} \in \mathcal{X}$ **do**
    $d_{\text{in}} \leftarrow \|\mathbf{v}_{\text{in}} - \mathbf{o}\|$ ▷ *Voxel entry range*
    $d_{\text{out}} \leftarrow \|\mathbf{v}_{\text{out}} - \mathbf{o}\|$ ▷ *Voxel exit range*
    $d \leftarrow 0.5(d_{\text{out}} - d_{\text{in}})$ ▷ *Effective voxel range*
    $r \leftarrow \|\mathbf{p}\|$ ▷ *Measured range*
    $\tau \leftarrow \tau(r)$ ▷ *Truncation distance*
    if $d \in [\tau_{\min}, r - \tau)$ **then**
      ▷ *Inside space carving region:*
      UpdateNTSDF(τ, w_{SC})
    else if $d \in [r - \tau, r + \tau]$ **then**
      ▷ *Inside truncation region:*
      $w \leftarrow \frac{1}{2\tau}$ ▷ *Observation weight*
      $c \leftarrow$ **Color**($\{u, v\}$) ▷ *Observation color*
      UpdateNTSDF(u, w)
      UpdateColor(c, w)

5.1 Voxel traversal

The voxel traversal algorithm is based on raytracing (Amanatides and Woo 1987), and is described by Algorithm 1. For each each depth measurement, we begin by performing a raytracing step on the coarse volume grid to determine the affected volumes and make sure all of them are allocated. Next, we raytrace along the voxel grid of each affected volume, and update each visited voxel's NTSDF (if they lie in the voxel carving region), or NTSDF and color (if they lie in the truncation region). We can choose the raytracing range for each observation to either $[r_{\min}, r + \tau]$ or $[r - \tau, r + \tau]$, depending on whether we want to perform voxel carving or not. The latter range will only allocate volumes and traverse voxels inside the truncation region, gaining speed and memory at the cost of reconstruction quality. For each voxel visited by the traversal algorithm, we calculate the *effective* voxel range d, which is average of the entry range d_{in} and exit range d_{out}, corresponding to the distances at which the ray enters and exits the voxel.

Let ρ_v be the linear voxel density, equal to the inverse of the voxel size s_v. the computational complexity of the voxel traversal algorithm becomes $\mathcal{O}(|P|\,\rho_v)$, where $|P|$ is the number of observations in the point cloud P (or, if using a depth image, the number of pixels with a valid depth reading).

This formulation of voxel traversal makes the approximation that each depth observation can be treated as a ray. As a result, as the depth rays diverge further away from the sensor, some voxels might not be visited. A more correct approximation would be to treat each depth observation as a narrow pyramid frustum, and calculate the voxels which the frustum intersects. However, this comes with a significantly greater computational cost.

5.2 Voxel projection

The voxel projection algorithm, described by Algorithm 2, works by projecting points from the voxel grid onto a 2D image. As such, it requires that the depth data is in the form of a depth image, with a corresponding projection matrix. Voxel projection has been used by Bylow et al. (2013), Klingensmith et al. (2014) and Nguyen et al. (2012) for 3D reconstruction. The algorithm is analogous to shadow mapping (Scherzer et al. 2011) from computer graphics, where raycast shadows are approximated by projecting onto a depth map from the perspective of the light source, except in our case, shadows are regions occluded from the depth sensor.

In order to perform voxel projection, we must iterate over all the voxels in the scene. This is, of course, both computationally expensive and inefficient. The majority of the iterated voxels will not project on the current depth image. Moreover, due to the nature of the two-tier data structure, some voxels of interest might not have been allocated at all. Thus, as a preliminary step, we determine the set of all potentially affected volumes $\mathcal{V}$ in the current view. We do this by calculating the intersection between the current depth camera frustum and

Algorithm 2 Voxel projection fusion

▷ *Find all volumes in current frustum.*
$\mathcal{V} \leftarrow$ **FrustumIntersection**()
▷ *Ensure all volumes are allocated.*
AllocateVolumes($\mathcal{V}$)
▷ *For each intersected volume:*
for $V \in \mathcal{V}$ **do**
  ▷ *For each voxel centroid in volume:*
  for $\mathbf{v}_c \in V$ **do**
    $d \leftarrow \|\mathbf{v}_c - \mathbf{o}\|$ ▷ *Effective voxel range*
    $\{u, v\} \leftarrow$ **Project**($\mathbf{v}_c$) ▷ *Measurement pixel*
    $\mathbf{p} \leftarrow$ **InvProject**(u, v, z_{uv}) ▷ *Measurement 3D point*
    $r \leftarrow \|\mathbf{p}\|$ ▷ *Measured range*
    $\tau \leftarrow \tau(r)$ ▷ *Truncation distance*
    if $d \in [\tau_{\min}, r - \tau)$ **then**
      ▷ *Inside space carving region:*
      UpdateNTSDF(τ, w_{SC})
    else if $d \in [r - \tau, r + \tau]$ **then**
      ▷ *Inside truncation region:*
      $w \leftarrow \frac{1}{2\tau}$ ▷ *Observation weight*
      $c \leftarrow$ **Color**($\{u, v\}$) ▷ *Observation color*
      UpdateNTSDF(u, w)
      UpdateColor(c, w)
▷ *Clean up volumes that did not receive updates.*
GarbageCollection($\mathcal{V}$)

the volume grid. The far plane of the frustum is at a distance of $r_{\max} + \tau(r_{\max})$, where $r_{\max}$ is the maximum camera range.

Next, we iterate over the voxels of all potentially affected volumes. We approximate the effective voxel range d as the distance from the voxel's centroid to the camera origin. By projecting the voxel onto the depth image, we can obtain a pixel coordinate and corresponding depth z_{uv}, from which we can calculate the observed range r. The rest of the algorithm proceeds analogous to voxel traversal: we check the region that this voxel belongs to, and conditionally update its NTSDF and color.

Finally, we perform a garbage collection step, which iterates over all the potentially affected volumes $\mathcal{V}$ and discards volumes that did not receive any updates to any of their voxels.

The computational complexity of the algorithm is $\mathcal{O}(|\mathcal{V}|\,\rho_v^3)$, where $|\mathcal{V}|$ is the number of volume candidates, and ρ_v^3 is the number of voxels in a unit of 3D space.

This formulation of voxel projection approximates each voxel as its centroid point. Thus, the estimation of the effective voxel range can have an error of up to $\sqrt{3}/2\ s_v$ (half the voxel's diagonal). Furthermore, each voxel will only project onto a single depth pixel, receiving a single update. Compare that to voxel traversal, where a single voxel might be updated by multiple depth rays passing through it, thus averaging the observations from all of them. At the resolution that were interested in (around 3 cm) these approximations lead to worse reconstruction quality than the approximations used in voxel traversal fusion.

Table 1 Comparison of fusion algorithms

Algorithm	Voxel traversal	Voxel projection
Complexity	$\mathcal{O}(\lvert P\rvert\,\rho_v)$	$\mathcal{O}(\lvert\mathcal{V}\rvert\,\rho_v^3)$
Approximations	Depth observations approximated as rays	Voxel range approximated by the voxel centroid; Each voxel projects to a single depth observation
Approximation effects	Integration gaps far from the camera	Aliasing effects with larger voxels

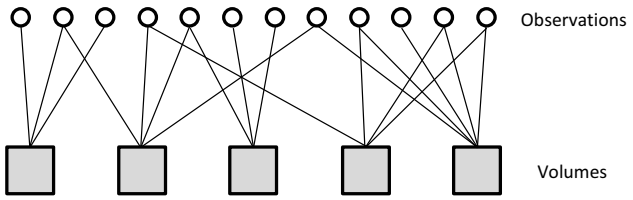


Fig. 12 Bipartite graph showing an example relationship between a set of point observations and the volumes which they affect

We summarize the properties of the two algorithms in Table 1.

5.3 Parallelization

We present a method to parallelize the scan fusion algorithms, so that scan fusion can take advantage of multi-threading and the multi-core architecture of the mobile devices.

Voxel projection fusion is straight-forward to implement in a parallel way. The frustum intersection and volume allocation are carried out serially. The outer loop in Algorithm 2 is parallelized so that each volume receives its own job. Volumes are independent of one another—thus, each job owns its own data, and no synchronization or locking is required. The only data shared between the jobs is the depth image that the volumes project onto and the camera extrinsic and intrinsic parameters. All of these are constant during any given voxel projection execution.

Voxel traversal fusion is more challenging to parallelize. We describe the parallelization problem in terms of a graph problem. Consider the bipartite graph in Fig. 12, split between the observation set P and the volume set $\mathcal{V}$. The serial version of the voxel traversal algorithm we have presented in Algorithm 1 iterates over all the points in P and finds their corresponding affected volumes from $\mathcal{V}$. Thus, we can think of the algorithm as building a *directed* bipartite graph, where the direction is from observation to volume.

A naive way to parallelize the traversal problem is to follow with this direction of the bipartite graph and instantiate a single job for each observation. This has the drawback that each job needs write access to multiple volumes, and thus, a locking mechanism is required in order to implement con-

currency. Instead, we reorganize the problem by inverting the direction of the bipartite graph so that each volume points to all the observations which affect it. We perform this re-indexing step serially. Next, we instantiate a job for each affected volume, and fuse the data from all the observations which it points to. Thus, we have restructured the problem so that, similarly to parallelized voxel projection, we have the volume as the unit of job separation. Choosing volumes over observations as the job unit has another advantage: in the datasets we used, there were anywhere between one and two orders of magnitude more observations than corresponding affected volumes. Having fewer jobs that run for longer is preferable over having a greater number of shorter jobs, as there is an overhead incurred by creating a job thread, as well as frequent context switching.

It remains to be shown how the graph re-indexing step affects the computational complexity of the voxel traversal algorithm, which we have so far established to be $\mathcal{O}(|P|\ \rho_v)$. The added complexity of this step is equal to the number of edges in the graph. Since calculating an edge requires a raytracing operation over the coarse volume grid for a given observation, this gives us a total complexity of $\mathcal{O}(|P|\ \rho_{\text{vol}})$, where ρ_{vol} is the linear volume density. By definition, each volume is bigger than the voxels it contains, and therefore $\rho_{\text{vol}} < \rho_v$. Thus, the total computational complexity remains the same. In practice, raytracing over the coarse grid is much faster than raytracing over the voxel grids; therefore, the computational overhead added by the re-indexing step is minimal, and outweighed by the gain in parallelizing the fusion.

6 Mesh extraction and rendering

6.1 Online mesh extraction

We generate a mesh of the isosurface using the Marching Cubes algorithm (Lorensen and Cline 1987). The mesh vertex location inside the voxel is determined by linearly interpolating the TSDF of its neighbors. Similarly, as in Bylow et al. (2013) and Whelan et al. (2013), colors are computed for each vertex by trilinear interpolation of the color field. We can optionally also extract the per-face normals, which are utilized in rendering.

We maintain a separate mesh for each segment for each instantiated volume (Fig. 9). Meshing a given volume requires iterating over all its voxels, as well as the border voxels of its 7 *positive* neighbors (volumes with at least one greater index along the x, y, or z dimension, and no smaller indices). The faces which span the border across to positive neighbors are stored with the mesh of the center volume. Therefore, the top-most, right-most, and forward-most bounding vertices of each mesh segment are duplicates of the starting vertices of the meshes stored in its positive neighbors. This is done to ensure that there are no gaps between the mesh segments, and that when rendered, the mesh appears seamless. Vertex duplication isn't strictly necessary, as we could accomplish this by indexing directly into the neighboring mesh. However, direct indexing would introduce co-dependencies between parallel jobs.

We perform mesh extraction every time a new depth scan is fused. As we have shown in Algorithms 1 and 2, the voxel fusion algorithms estimate a set of volumes $\mathcal{V}$, potentially affected by the depth data. Mesh extraction is performed only for those volumes. Thus, the computational complexity of the meshing algorithm becomes $\mathcal{O}(|\mathcal{V}|\ \rho_v^3)$. We parallelize meshing similarly to how we parallelize fusion, by instantiating a single marching cubes job per volume, and letting jobs run concurrently. Since each job owns the data that it is modifying (the triangle mesh), there are no concurrency issues.

Parallelized meshing of only the affected volumes allows us to maintain an up-to-date representation of the entire isosurface in real time. Still, we observed that a lot of time is spent re-meshing volumes which are completely empty. These volumes occur mostly between the sensor and the surface, especially when we have voxel carving available. To further decrease the time spent for meshing, we propose a lazy extraction scheme which prunes a large portion of the volumes. To do so, we augment the fusion algorithms to keep track of a per-volume flag signifying whether the isosurface has jumped across voxels since the last scan fusion. The flag is set to true when at least one voxel in the current volume receives an NTSDF update which changes its NTSDF sign. The flag is cleared after each re-meshing.

The effect of the lazy extraction flag is that volumes with no iso-surface, or where the isosurface did not move during the last scan fusion, will not be considered for meshing. This decreases the time spent extracting meshes, at the cost that we might not have the most up-to-date mesh available. As we mentioned earlier, the position of the vertex inside a voxel is a calculated using a linear interpolation with the NTSDF of its neighbors. Thus, even if the isosurface doesn't jump between voxels, a depth scan fusion might move the vertices within each voxel. Enabling lazy extraction means that at worst, each vertex can be up to $\sqrt{3}/2\, s_v$ (half a voxel diagonal) away from its optimal position. Further, lazy extraction does not consider updates to color, so color changes that do not come with occupancy changes will not trigger re-meshing.

In practice, we found that the sub-optimality conditions described above are triggered very rarely, and that lazily-extracted meshes were indistinguishable from the optimal meshes during real-time experiments.

At the end of each dataset collection trial, the user may want to save a copy of the entire mesh. In those cases, we run a single marching cubes pass across all the volumes in the hash map to produce a single, non-segmented mesh. Thus the

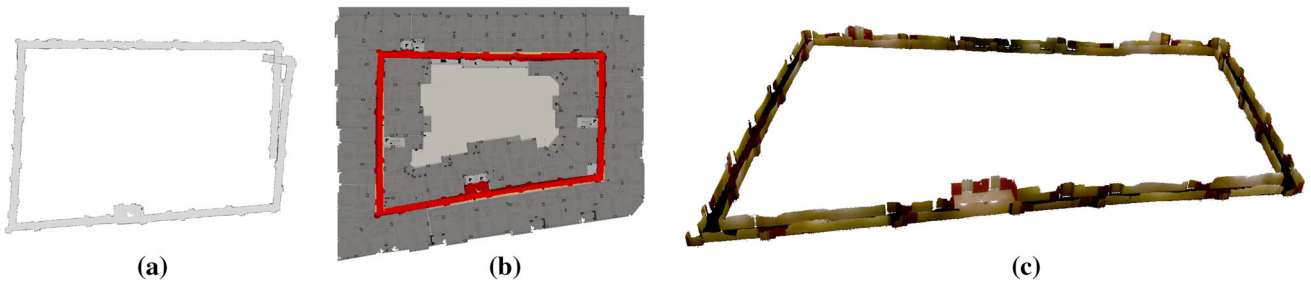


Fig. 13 Comparison of reconstruction performed online versus with offline bundle adjustment. The trajectory length is approximately 200 m. The corridor loop length is approximately 175 m. **a** Online reconstruction, **b** after offline bundle-adjustment, **c** after offline bundle-adjustment (side view)

final output mesh does not incur any of the approximations we made for real-time meshing (duplicate vertices and lazy extraction). After the final mesh is extracted, we perform (optional) mesh simplification via the quadric error vertex collapse algorithm (Garland and Heckbert 1997), which reduces the number of mesh faces by a user-controllable amount. Finally, we remove any isolated vertices which have few neighbors within some small threshold radius. These vertices typically occur when the sensor has a false depth readings, occasionally resulting in "speckle" artifacts floating around the scene.

6.2 Rendering

We render the mesh reconstruction using only simple fixed-function pipeline calls on the devices graphics hardware. To speed up rendering, we only consider mesh segments which are inside the current viewport's frustum. This is done using frustum culling, analogous to how we use it in the voxel projection algorithm (Algorithm 2). Further, we utilize a simple level-of-detail (LOD) rendering scheme. Mesh segments which are sufficiently far away from the camera are rendered as a single box, whose size and color are specified by the mesh segment's bounding box and average vertex color, respectively. "Sufficiently far away" is determined by checking whether the projection of the corresponding mesh volume exceeds a certain number of pixels on the screen.

7 Offline processing

7.1 Offline pose estimation

In larger scenes where the odometry can drift more, the approach described in Sect. 4.8 will not be able to account for drift errors. In situations like this, we perform an offline post-processing step on device. The post processing requires that we record the inertial, visual, and depth data to disk. Given that the *Project Tango* devices have sufficient persistent memory available (60 and 120 GB for the cell phone and tablet, respectively), this is not an issue. Once the data collection is finalized, we replay the data and perform visual-inertial bundle-adjustment on the trajectory (Nerurkar et al. 2014). The bundle adjustment solves a nonlinear weighted least-squares minimization over the visual-inertial data to jointly estimate the device pose and the 3D position of the visual features. Correspondences between non-consecutive keyframes (loop-closure) are established through visual feature matching (Lynen et al. 2014). Once the bundle-adjusted trajectory is calculated, we rebuild the entire TSDF map using the recorded depth data, and extract the final surface once.

We present the results of this process in Fig. 13. We recorded a single, large dataset (approximately 200 m trajectory length). Figure 13a shows a top-down view of the reconstruction performed online. Figure 13b shows the reconstruction after performing the offline bundle adjustment. We overlaid it on the building schematic for reference. Finally, Fig. 13c shows a side view of the offline reconstruction. The surface area of the entire reconstructed mesh is approximately 740 m^2.

7.2 Mutli-dataset reconstruction

Given an accurate trajectory and bias-free measurement model, we found that the biggest limiting factor for the quality of the reconstruction is the number of depth images taken. Each new depth image either adds information about an unobserved space, fills in small gaps in the existing reconstruction, or refines the estimation of the reconstruction's isosurface. As we noted before, depth cameras such as the Kinect provide data with a rate and density much higher than the *Project Tango* mobile devices. We estimated that the Yellowstone tablet, for example, might receive two orders of magnitudes less depth observations for a comparable dataset collection trial. On the other hand, recording very long datasets is limited by battery life and is cumbersome for the operator, who has to walk slowly and spend a long time scanning.

The reconstructions shown so far, both in our paper, and in the works we have referenced, are created using a single dataset collection trial. We extend our workflow by allowing

Fig. 14 Results from a multi-dataset reconstruction from 4 independently-collected datasets. (**a**) through (**d**) the resulting mesh, as reconstructed from a single dataset, as well as the device trajectory and starting point. Each individual reconstruction has gaps in different areas. (**e**) the final, combined reconstruction. **a** Dataset A, **b** datasets B, **c** datasets C, **d** datasets D, **e** datsets A + B + C + D

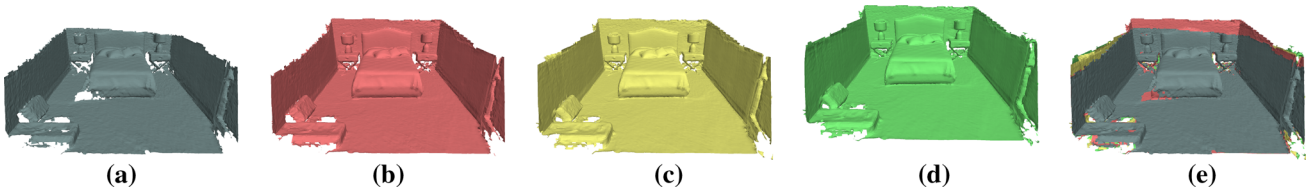


Fig. 15 Closeup of room during the multi-dataset reconstruction from Fig. 14. (**a**) through (**d**) the incremental resulting mesh at the different stages of the reconstruction, as each new dataset is fused into the previous stage. The final reconstruction is shown in (**d**). **e** an overlay of the stages, highlighting the areas which each new stage filled in. **a** Dataset A, **b** datasets A + B, **c** datasets A + B + C, **d** datasets A + B + C + D, **e** overlay

for automated fusing multiple datasets into a single reconstruction. Each dataset can be recorded from a different starting location and cover different areas of the scene, as long as their is sufficient visual overlap so that they can be co-registered together. We repeat the offline procedure described in Sect. 7.1 for each dataset. Then, we query keyframes from each dataset against keyframes from the other ones, until we have sufficient information to calculate the offset between each individual trajectory, bringing them into a single global frame. Finally, we insert the depth data from each dataset into a unified TSDF reconstruction.

The resutls of this process are shown in Figs. 14 and 15. We asked two different operators to create two full reconstructions of an apartment scene. This resulted in four individual reconstructions, each with varying degrees of missing data. Figure 14 shows each individual reconstruction (labeled "A" through "D"), as well as the final combined reconstruction. The final mesh has significantly fewer gaps. Figure 15 shows the incremental resulting mesh at the different stages of the reconstruction, as each new dataset is fused into the previous stage. Each new stage fills in more mesh holes and refines the estimate of the isosurface.

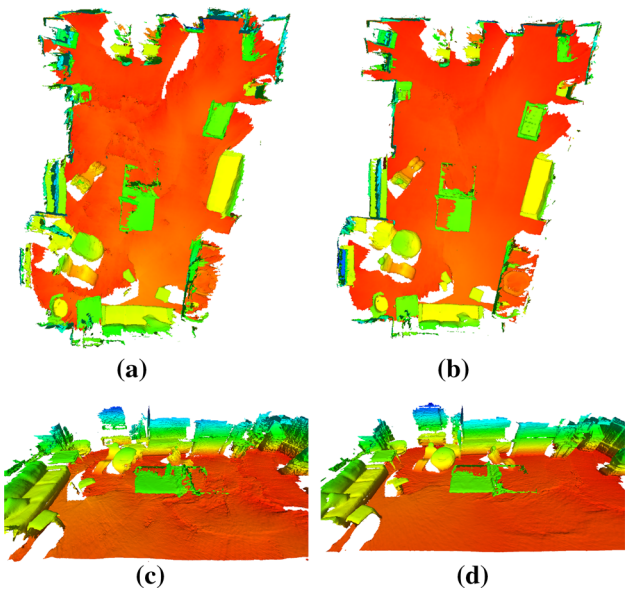


Fig. 16 Comparison of model built with raw cell phone depth data (*left*) versus compensated depth data (*right*). Using compensated data results in cleaner walls and object. **a** Raw depth, **b** compensated, **c** raw depth, **d** compensated detail

8 Experiments

8.1 Reconstruction quality experiments

We tested the qualitative performance of the system on a variety of data sets. So far, we've shown reconstructions from the mobile phone data in Fig. 1a. In this section, we further present a reconstruction in Fig. 16, which demonstrates the effectiveness of our depth compensation model (discussed in Sect. 9).

We have also shown several reconstructions from tablet data: the "Apartment" scene (Figs. 1b, c, 11, 14, 15) and "Corridor scene" (Fig. 13). In this section, we further present

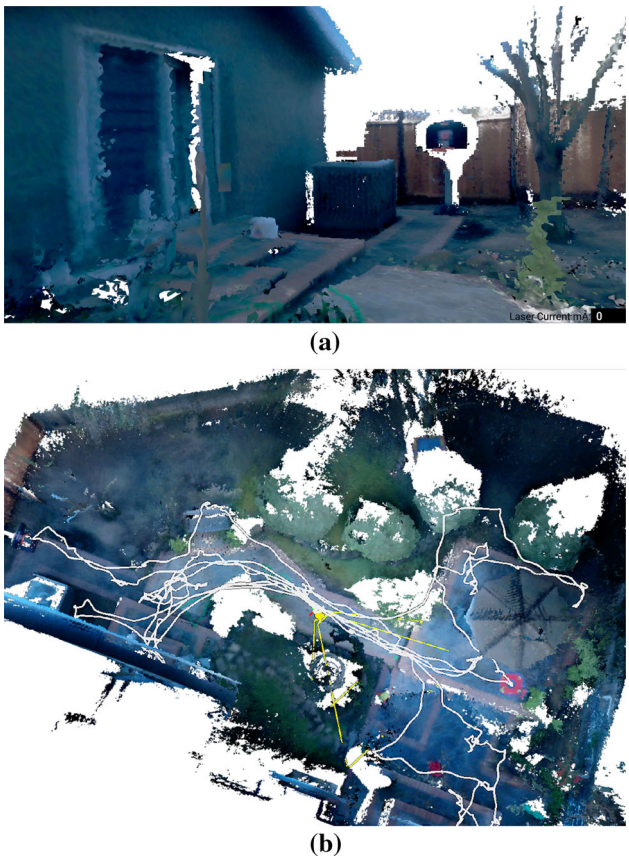


Fig. 17 Outdoors reconstruction using tablet data, performed during an overcast day. Top-down view (**b**) also shows the device trajectory, in *white*. **a** Side view, **b** top-down view

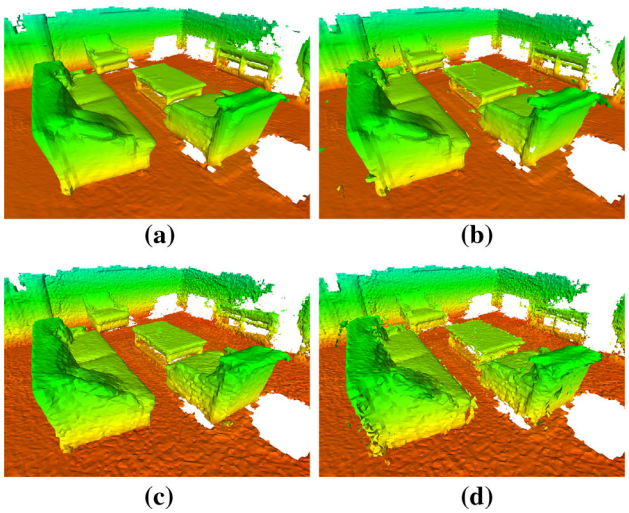


Fig. 18 Comparison of reconstruction quality using the different insertion algorithms (voxel traversal vs. voxel projection) with and without voxel carving applied. The reconstructions was performed with tablet data. **a** Traversal, carving, **b** traversal, no carving, **c** projection, carving, **d** projection, no carving

results from an outdoor reconstruction with tablet data (Fig. 17). Since the depth sensor is based on infrared light, outdoor reconstructions work better in scenes without direct sunlight.

We investigated the effects of the different scan fusion algorithms on the reconstruction quality using tablet data. We examined four combinations in total: voxel traversal versus voxel projection, with voxel carving enabled or disabled (Fig. 18). We found that voxel traversal gives overall better quality at the resolution we are working with. Enabling voxel carving (for both traversal and projection algorithms) removes some random artifacts, and significantly improves reconstruction quality around the edges of objects).

To assess the reconstruction quality of the different scan fusion algorithms, we reconstructed a scene using a simulated dataset from SceneNet (Handa et al. 2015). Quadratically attenuated Gaussian noise (see Sect. 9) was added to the depth images produced by the simulator, using the noise model learned from the *Tango* tablet. Additionally, 25% of the pixels were thrown out at random to simulate further data loss/sparsity in the depth image. Perfect pose estimation is assumed. Figure 19 shows 3 cm resolution reconstructed meshes using each of the scan fusion algorithms colored by their nearest distance to the ground truth mesh. Table 2 shows the mean and 95% confidence interval of the reconstruction error. The raw noisy data are shown as a baseline. Overall, voxel projection with carving had the least reconstruction error, and voxel traversal without carving had the most error. Figure 19 demonstrates the charachteristic errors of each reconstruction method. Where the data are sparser and noisier, voxel projection seems to do a better job at filling in smooth surfaces. This is because each voxel is updated in voxel projection regardless of whether a depth ray passes through it. Voxel projection also characteristically suffers from reconstruction errors along the edges of objects where voxels are partially oc- cluded by surfaces, and voxel aliasing is clearly present. Voxel traversal does not suffer from these issues, but in general the reconstruction is noisier and sparser. Voxel carving improves both scan fusion algorithms by eras- ing random oating structures (visible as faint yellow dots in Fig. 19).

Finally, we wanted to verify the applicability of the presented method with a third, non-*Project Tango* data source. We chose the "Freiburg" datasets, recorded with a high-quality RGB-D camera and ground-truth trajectories from a motion-capture system (Sturm et al. 2012). The reconstruction results are presented in Fig. 20.

8.2 Performance experiments

We performed several quantitative experiments to analyze the performance of the different algorithms we use. In the first experiment, we compare the number of grid volumes

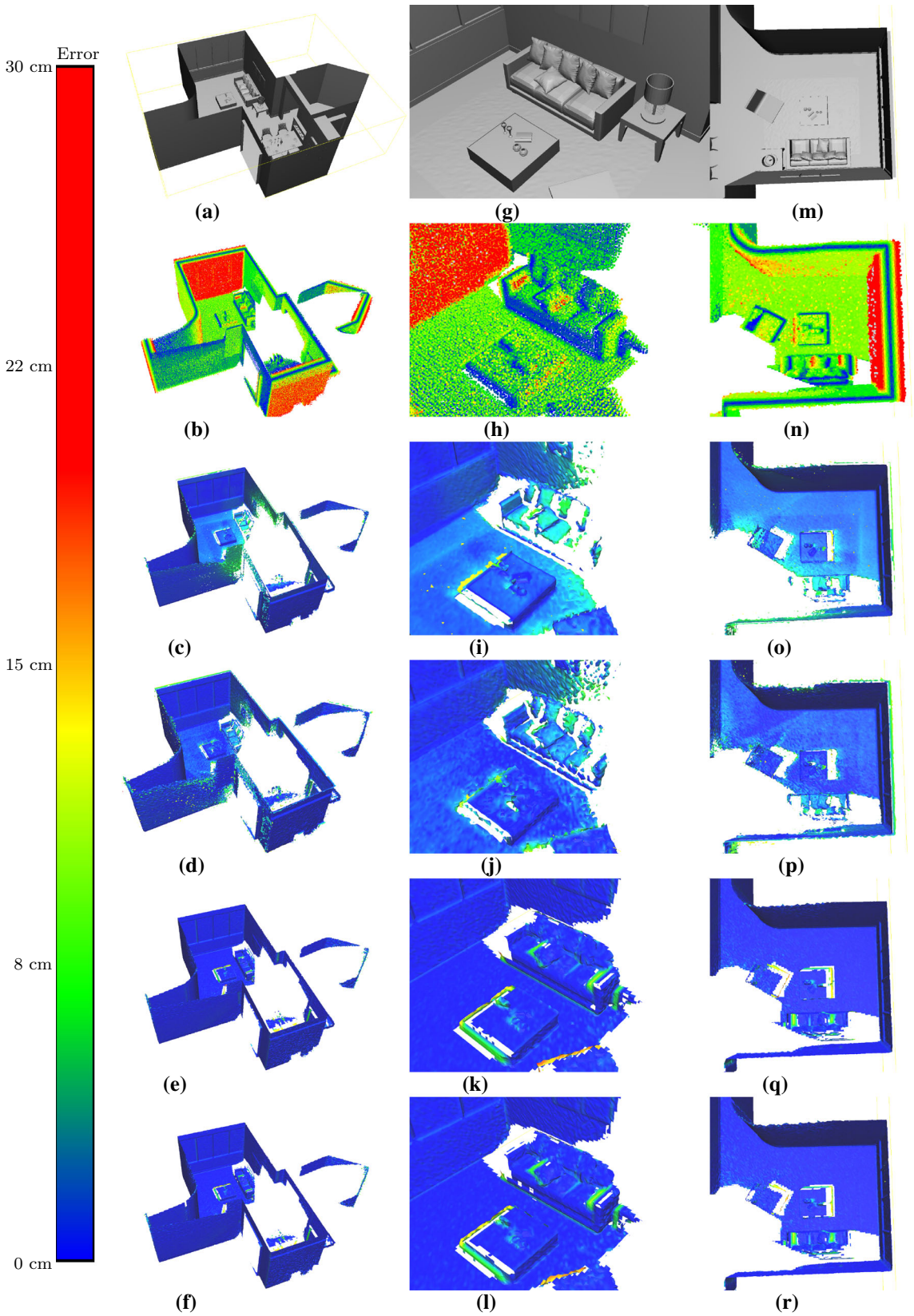


Fig. 19 Reconstruction error in a simulated dataset from SceneNet (Handa et al. 2015), with simulated depth noise. An overview, closeup, and top-down view are shown (*columns*), comparing the ground truth, raw point cloud, voxel projection and voxel traversal reconstruction error (*rows*). Lower error (*blue*) is better than higher error (*red*/*green*). The voxel resolution is 3 cm (Color figure online)

Table 2 Reconstruction quality comparison (simulated SceneNet dataset Fig. 19)

Algorithm	Error (cm)
Raw point cloud	
–	7.46 ± 6.0
Voxel traversal	
Carving	1.46 ± 2.3
No carving	1.67 ± 2.0
Voxel projection	
Carving	$\mathbf{0.45 \pm 1.0}$
No carving	0.47 ± 1.1

95% confidence interval shown

Fig. 20 Reconstruction using the *Freiburg desk* public dataset Sturm et al. (2012), based on the motion-capture ground-truth trajectory. The maxiumum depth used is 2 m. Performed on a desktop machine

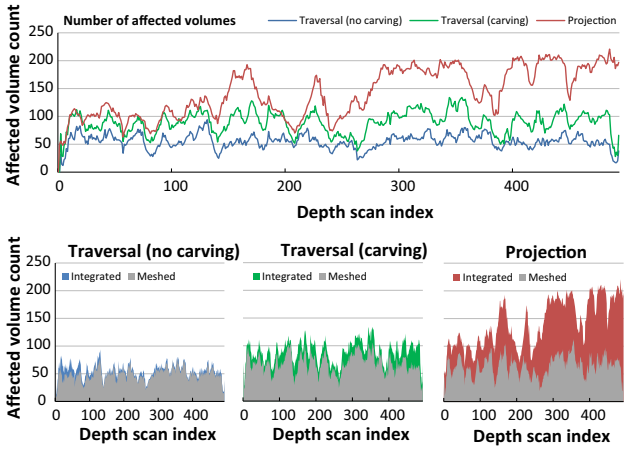


Fig. 21 *Top* analysis of the number of affected volumes per depth scan. We compare the voxel traversal versus voxel projection algorithms. The results for voxel traversal depend on whether voxel carving has been enabled. *Bottom* comparison of the number of meshed volumes versus all the affected volumes. The difference between the two is due to the lazy meshing scheme

Table 3 Scan fusion time [ms] (tablet dataset)

Algorithm	Mobile	Desktop
Voxel projection		
Carving	83.7 ± 13.0	18.0 ± 2.4
No carving	72.1 ± 10.8	16.0 ± 2.2
Voxel traversal		
Carving	121.9 ± 21.1	23.4 ± 5.8
No carving	85.6 ± 12.6	13.3 ± 3.2

Table 4 Scan fusion time (ms) (cell phone dataset)

Algorithm	Mobile	Desktop
Voxel projection		
Carving	80.2 ± 8.0	18.5 ± 2.8
No carving	67.5 ± 6.9	16.9 ± 2.0
Voxel traversal		
Carving	211.1 ± 67.9	49.5 ± 14.1
No carving	113.4 ± 31.6	24.3 ± 7.6

which are affected by each depth scan. As we discussed in Sect. 5, this number is important because we would like to keep our computations focused on a small number of local volumes. The results are shown in Fig. 21. Voxel projection uses frustum culling to select volume candidates, which is a conservative approximation - thus, it ends up considering more candidates overall than voxel traversal. When using voxel traversal, the number of volume candidates depends on whether voxel carving is enabled. Regardless of the choice of algorithm, fusion is performed on each candidate volume. However, as we discussed before, meshing is performed lazily, and might prune some candidate volumes. We demonstrate this in Fig. 21 (bottom). We overlaid the number of meshed volumes over the number of total fused volumes over time. As can be seen, voxel projection benefits the most from the lazy meshing scheme.

We also analyzed the exact run-time of voxel projection and voxel traversal, with and without carving enabled. We carried out the experiments with two sets of data (cell phone dataset and tablet dataset), since depth data affects algorithm performance. We further carried out the experiment on both a mobile device and on a desktop machine, to determine the effect of the mobile constraint. The results are shown in Table 3 (tablet dataset) and Table 4 (cell phone dataset). The tables show the average and standard deviation of the total processing time for each depth scan, in milliseconds, including: depth scan compensation, dense alignment, scan fusion, and mesh extraction. The fusion and extraction steps are performed using the parallelized algorithms described in Sect. 5.3. We used a voxel size of 3 cm, and $16 \times 16 \times 16$ voxels per volume.

The timing experiments reveal that when tablet data is used, all of our insertion algorithms perform in real time

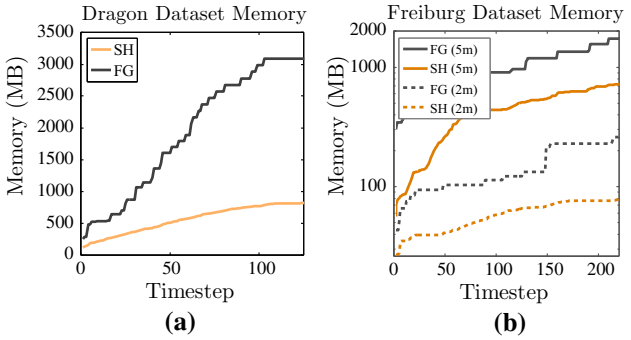


Fig. 22 Comparison of memory consumption between our spatially-hashed volume data structure (SH) versus a fixed-resolution grid approach (FG). We compare two datasets: from a *Tango* cell phone device (**a**) and from a Kinect camera (**b**). In the case of the Kinect camera, we tested both a short-range insertion (observations up to 2 m) and a long-range one (observations up to 5 m). Note the logarithmic scale on the right graph

at 5 Hz or higher. Thus, we can choose the algorithm which gives us the best quality (voxel traversal with carving). When using a the cell phone dataset, voxel projection takes around the same time, but voxel traversal is significantly more expensive. This is because the number of observations in a the cell phone depth scan is much higher (Fig. 4). At this resolution, we cannot run the best-quality algorithm, so we choose either projection with carving or traversal without carving, depending the type of quality approximation we want to achieve.

Finally, we profiled the memory performance of our system on two datasets, from a *Project Tango* device (Fig. 22a) and a "Freiburg" RGB-D dataset (Sturm et al. 2012) (Fig. 22b). For the Freiburg dataset, we tested two different maximum camera ranges (2 and 5 m). The figures show the total memory consumption using the spatially hashed volume grid data structure described in Sect. 4.7. We used a voxel size of 3 cm, and $16 \times 16 \times 16$ voxels per volume. For reference, we also include the memory for a static, 3 cm fixed-grid approach.

9 Conclusions and future work

We have demonstrated a fully-integrated system capable of creating and rendering large-scale 3D reconstructions of scenes in real-time using only the limited computational resources of a Tango device. We accomplish this without any GPU computing, and use only the fixed function pipeline to render the scene. We do this out of necessity to meet the computing requirements of a mobile device. We discuss our contributions in terms of sensor calibration, discrete filtering, trajectory correction, data structure layout, and parallelized TSDF fusion and meshing algorithms algorithms. Our work is heavily influenced by the characteristics of the depth data we receive (high noise and bias, lower frequency, variable observation density) We have successfully applied it to two different mobile devices, as well as datasets from other RGB-D sensors, to efficiently produce accurate geometric models.

Where appropriate, we augment our system with post-processing tools to overcome limitations in the sensors and the device. Notably, we employ place recognition and sparse feature-based bundle adjustment to correct very large trajectories, as well as merge multiple datasets, in order to get the best possible model and coverage. This can be seen as one of the major potential ares of improvement. Further research into this area should allow for the depth measurements, and dense alignment information they provide, to be closely integrated with the sparse features for a more robust SLAM solution. This can include methods for real-time mesh deformation for long trajectories, eliminating the need for post-processing.

As more and more consumer and research devices come with embedded depth sensors, we expect our work to be of interest to developers and researchers working on human or robotic navigation, augmented reality, gaming.

Appendix: Calibration

Depth measurement model

We begin by considering the data from the device's depth sensor. It may be available in the form of a depth image I_D or a point cloud P. In the first case, the depth image consists of a set of pixels $\mathbf{q}$ where $\mathbf{q}(u, v) = z_{uv}$ is the *depth* measured at pixel coordinates u, v. In this context, the depth z_{uv} is defined as the distance from the camera to the observation, along the camera's optical axis. When dealing with point clouds, we define the cloud P as a set of points $\mathbf{p}_i$, where $\mathbf{p}_i = [x, y, z]^\mathsf{T}$, and x, y, z are the coordinates of the point in the depth camera's frame of reference. We make a distinction between the *depth* and the *range*, the latter being the Euclidean distance $\|\mathbf{p}\|$ to an observation $\mathbf{p}$.

Given the depth camera's intrinsic parameters, we can switch between the two interpretations freely using the ideal pinhole camera model:

$$x = \frac{z_{uv}}{f_x}(u - c_x) \tag{29a}$$

$$y = \frac{z_{uv}}{f_y}(v - c_y) \tag{29b}$$

As we mentioned earlier (Fig. 4b), converting from point cloud to depth image representation offers a tradeoff: the resulting depth image may have significant gaps between pixels, or will have dense pixel coverage, but lose some angular accuracy of the observations.

Each depth reading z_{uv} may be corrupted by random noise, systematic biases, or missing data. We treat the depth as a random variable Z. We define Z to have the following form:

$$Z = f(z_{uv}) + \epsilon(z_{uv}) \tag{30}$$

The functions f and ϵ model the systematic bias and random noise, respectively, and we discuss them in the following passages.

Systematic bias model and calibration

We found that the depth data returned by *Tango* devices exhibits a strong systematic bias. The bias varies in different areas of the image, and is generally worse closer to the edge. For modeling the systematic bias, we choose a second-degree polynomial:

$$\widetilde{z}_{uv} = f(z_{uv}) = a_{uv} + b_{uv} z_{uv} + c_{uv} z_{uv}^2 \tag{31}$$

The model depends on the per-pixel coefficients a_{uv}, b_{uv}, c_{uv}. This allows us to apply different corrections to each pixel. Given the model coefficients, we can calculate the *compensated*, or bias-free measurement $\widetilde{z}_{uv}$.

For performing the depth calibration, we use a large checkerboard printed on a flat surface. We record RGB and depth image pairs of the checkerboard at various distances to the camera and locations in the image. Since there is a limit on how large we can print the checkerboard on a flat surface, when we move the checkerboard further away from the camera it does not cover the entire field of view of the depth image. Therefore, we must take multiple images with the checkerboard covering different areas in the image view. Alternatively, if a significantly large, flat wall is available, one could attach the checkerboard to the wall and observe it from various distances.

To mitigate the effects of the random noise in the depth readings, the depth image used is the result of averaging 1000 consecutive depth images of the same static scene. From each RGB image, we detect the checkerboard corners and calculate the reference depth of the checkerboard using the projective-n-points (PNP) algorithm (Lepetit et al. 2009). Figure 23 shows two pairs of training images, with the reference checkerboard in different positions.

We performed the calibration by taking multiple training pairs at different ranges and different locations in the camera view. Once all K training images are collected, we create a set of training points for each checkerboard pixel in each image. We denote the measured distance at a pixel $\mathbf{q}(u, v)$ as $z_{uv}^{(k)}$, and the corresponding reference distance $\mathcal{Z}_{uv}^{(k)}$. Since the same pixel $\mathbf{q}$ will be observed at multiple images, we use k to express the index of the training image. The total error function for a given pixel location u, v can be defined as

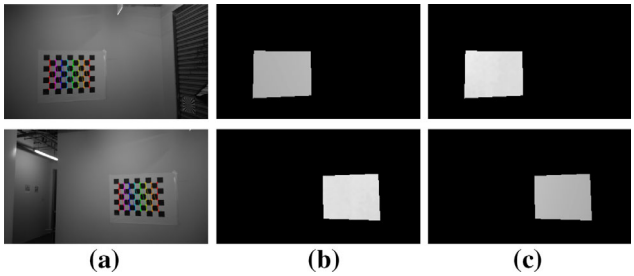


Fig. 23 Two sets of training images used in the calibration for systematic bias in depth readings (*top* and *bottom*). Each training set consists of 3 images: the greyscale image **a** is used to compute a linear plane model **b** for the reference depth. The reference depth is compared to the measured depth data (**c**). We repeat for multiple checkerboard poses at different distances and view positions

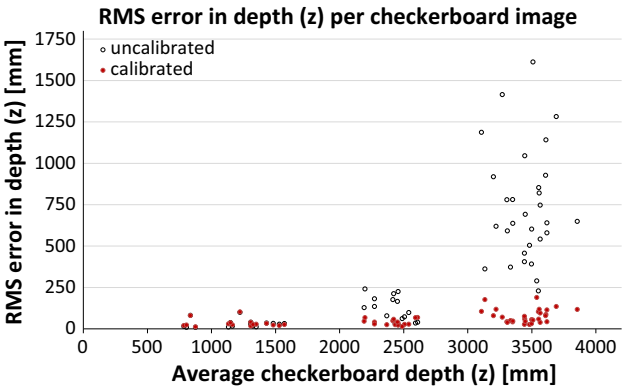


Fig. 24 RMS error in z before and after depth bias calibration on the *Tango* cell phone. Each data point represents the average error over all checkerboard pixels in the kth checkerboard image. Multiple checkerboard images at varying distances z from the camera are used

$$e(u, v) = \sum_k \left(\mathcal{Z}_{uv}^{(k)} - \widetilde{z}_{uv}^{(k)} \right)^2 \tag{32}$$

where k iterates across all the images where the pixel u, v was inside the checkerboard area. We wish to find the per-pixel coefficients a_{uv}, b_{uv} and c_{uv} which minimize the error for each pixel.

$$\arg\min_{a,b,c} e(u, v) \tag{33}$$

We accomplish this by fitting a second-degree polynomial to the data.

Using this procedure, we can remove a significant amount of the bias in the depth reading. Figure 24 presents the results for the *Project Tango* mobile phone, which exhibited the worse systematic bias of the two devices. Each data point in the figure represents the RMS error of z over all the pixels in a given checkerboard test image, versus the average depth of the checkerboard. The results with and without calibration are displayed. The figure shows that the RMS error is significantly lower when the polynomial compensation is

performed on the depth images, and the error improvement becomes more pronounced as the distance between the object and the camera grows.

Random noise model

We model the random noise as a Gaussian random variable, centered at the measurement z_{uv}, with a standard deviation of $\sigma(z_{uv})$. Similar to the approach of Nguyen et al. (2012), we model $\sigma(z_{uv})$ as a quadratic polynomial.

$$\epsilon(z_{uv}) = \mathcal{N}(z_{uv}, \sigma(z_{uv})) \tag{34a}$$

$$\sigma(z_{uv}) = \sigma_0 + \sigma_1 z_{uv} + \sigma_2 z_{uv}^2 \tag{34b}$$

The coefficients $\sigma_{0:2}$ are computed by taking the measured standard deviation from the same training data that we used for the systematic bias, and then performing quadratic regression on the result. A single set of coefficients is used for the entire image, unlike the distortion model which is computed on a per-pixel basis.

Acknowledgements This work was done with the support of Googles Advanced Technologies and Projects division (ATAP) for *Project Tango*. The authors thank to Johnny Lee, Joel Hesch, Esha Nerurkar, Simon Lynen, Ryan Hickman and other ATAP members for their close collaboration and support on this project.

References

Amanatides, J., & Woo, A. (1987). A fast voxel traversal algorithm for ray tracing. *Eurographics*, *87*, 3–10.

Bylow, E., Sturm, J., Kerl, C., Kahl, F., & Cremers D. (2013). Real-time camera tracking and 3D reconstruction using signed distance functions. In *Robotics: Science and systems (RSS) conference 2013*.

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

Chen, Y., & Medioni, G. (1991, April). Object modeling by registration of multiple range images. In *Proceedings., 1991 IEEE international conference on robotics and automation* (Vol. 3, pp. 2724–2729).

Chilimbi, T. M., Hill, M. D., & Larus, J. R. (2000). Making pointer-based data structures cache conscious. *Computer*, *33*(12), 67–74.

Curless, B., & Levoy, M. (1996). A volumetric method for building complex models from range images. In *SIGGRAPH 96 conference proceedings* (pp. 303–312). ACM.

Elfes, A. (1989). Using occupancy grids for mobile robot perception and navigation. *Computer*, *22*, 46–57.

Engel, J., Schöps, T., & Cremers, D. (2014, September). LSD-SLAM: Large-scale direct monocular SLAM. In *European conference on computer vision (ECCV)*.

Garland, M., & Heckbert, P. S. (1997). Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on computer graphics and interactive techniques* (pp. 209–216). ACM Press/Addison-Wesley Publishing Co.

Google. Project Tango (2014). https://www.google.com/atap/projecttango.

Handa, A., Patraucean, V., Badrinarayanan, V., Stent, S., & Cipolla, R. (2015). Scenenet: Understanding real world indoor scenes with synthetic data. In *CoRR*. arXiv:1511.07041.

Hesch, J. A., Kottas, D. G., Bowman, Sean L., & Roumeliotis, S. I. (2014). Camera-IMU-based localization: Observability analysis and consistency improvement. *The International Journal of Robotics Research*, *33*(1), 182–201.

Kähler, 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 devices. *IEEE Transactions on Visualization and Computer Graphics*, *21*(11), 1241–1250.

Klein, G., & Murray, D. (2007). Parallel tracking and mapping for small AR workspaces. In *2007 6th IEEE and ACM international symposium on mixed and augmented reality, ISMAR*.

Klingensmith, M., Dryanovski, I., Srinivasa, S., & Xiao, J. (2015, July). Chisel: Real time large scale 3d reconstruction onboard a mobile device using spatially hashed signed distance fields. In *Proceedings of robotics: Science and systems, Rome*.

Klingensmith, M., Herrmann, M., & Srinivasa, S. S. (2014). Object modeling and recognition from sparse: Noisy data via voxel depth carving. In *ISER, number d*.

Lepetit, V., Moreno-Noguer, F., & Fua, P. (2009). Epnp: An accurate o (n) solution to the PnP problem. *International Journal of Computer Vision*, *81*(2), 155–166.

Lorensen, W. E., & Cline, H. E. (1987). Marching cubes: A high resolution 3D surface construction algorithm. In *SIGGRAPH 1987*, (Vol. 21 pp. 163–169). ACM.

Lynen, S., Bosse, M., Furgale, P., & Siegwart, R. (2014). Placeless place-recognition. In *2nd international conference on 3D vision (3DV)*

Microsoft. Kinect for Windows. http://www.microsoft.com/en-us/kinectforwindows/.

Mourikis, A. I., & Roumeliotis, S. I. (2007). A multi-state constraint Kalman filter for vision-aided inertial navigation. In *2007 IEEE international conference on robotics and automation*.

Nerurkar, E. D., Wu, K. J., & Roumeliotis, S. I. (2014). C-KLAM: Constrained keyframe-based localization and mapping. In *2014 IEEE international conference on robotics and automation (ICRA)* (pp. 3638–3643).

Newcombe, R. A., Izadi, S., Hilliges, O., Molyneaux, D., Kim, D., & Davison, A. J. Pushmeet K., Jamie S., Steve H., & Andrew F. (2011) KinectFusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE international symposium on mixed and augmented reality, ISMAR 2011* (pp. 127–136).

Newcombe, R. A., Lovegrove, S. J., & Davison, A. J. (2011). DTAM: Dense tracking and mapping in real-time. *2011 IEEE international conference on computer vision (ICCV)*.

Nguyen, C. V., Izadi, S., & Lovell, D. (2012). Modeling kinect sensor noise for improved 3D reconstruction and tracking. In *Proceedings—2nd joint 3DIM/3DPVT conference: 3D imaging, modeling, processing, visualization and transmission, 3DIMPVT 2012* (pp. 524–530).

Nieß ner, M., Zollhöfer, M., Izadi, S., & Stamminger, M. (2013). Real-time 3D reconstruction at scale using voxel hashing. In *ACM transactions on graphics (TOG)*.

Rusinkiewicz, S., Hall-Holt, O., & Levoy, M. (2002). Real-time 3D model acquisition. In *ACM transactions on graphics* (Vol. 21, pp. 438–446). ACM

Scherzer, D., Wimmer, M., & Purgathofer, W. (2011). A survey of real-time hard shadow mapping methods. In *Computer graphics forum* (Vol. 30, pp. 169–186). Wiley Online Library.

Schöps, T., Sattler, T., Häne, C., & Pollefeys, M. (2015). 3D modeling on the go: Interactive 3D reconstruction of large-scale scenes on mobile devices. In *International conference on 3D vision (3DV)*.

Structure Sensor. http://structure.io/

Sturm, J., Engelhard, N., Endres, F., Burgard, W., & Cremers, D. (2012). A benchmark for the evaluation of RGB-D SLAM systems. In *IEEE international conference on intelligent robots and systems* (pp. 573–580).

Tanskanen, P., Kolev, K., Meier, L., Camposeco, F., Saurer, O., & Pollefeys, M. (2013). Live metric 3D reconstruction on mobile phones. In *2013 IEEE international conference on computer vision* (pp. 65–72).

Teschner, M., Hiedelberger, B., Müller, M., Pomeranets, D., & Gross, M. (2003). *2003*. In: *Vmv: Optimized spatial hashing for collision detection of deformable objects*.

Weise, T., Leibe, B., & Van Gool, L. (2008). Accurate and robust registration for in-hand modeling. In *26th IEEE conference on computer vision and pattern recognition, CVPR* (pp. 1–8).

Whelan, T., Leutenegger, S., Salas-Moreno, R. F., Glocker, B., & Davison, A. J. (2015, July). ElasticFusion: Dense SLAM without a pose graph. In *Robotics: Science and systems (RSS), Rome*.

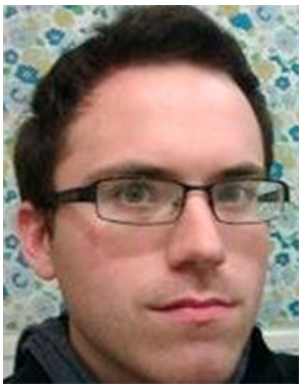
Whelan, T., Johannsson, H., Kaess, M., Leonard, J. J., & McDonald, J. (2013). Robust real-time visual odometry for dense RGB-D mapping. In *2013 IEEE international conference on robotics and automation (ICRA)*.

Whelan, T., & Kaess, M. (2013, November). Deformation-based loop closure for large scale dense RGB-D SLAM. In *2013 IEEE/RSJ international conference on intelligent robots and systems (IROS), Tokyo*.

Wurm, K. M., Hornung, A., Bennewitz, M., Stachniss, C., & Burgard, W. (2010). OctoMap: A probabilistic, flexible, and compact 3D map representation for robotic systems. In *Proceedings of the ICRA 2010 workshop on best practice in 3D perception and modeling for mobile manipulation*.

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

Ivan Dryanovski received a Ph.D. in computer science from the Graduate Center, City University of New York (CUNY). He worked for the CCNY Robotics lab and was advised by Dr. Jizhong Xiao. Previously, Ivan received an M.Sc. in computing science from Imperial College London in 2009, and a BA in Physics from Franklin and Marshall College in 2007. His research interests involve 3D reconstruction, depth cameras, unmanned aerial vehicles and SLAM.



Matthew Klingensmith is a Ph.D. candidate at the Robotics Institute of Carnegie Mellon University, where he works in the Personal Robotics Lab. He is co-advised by Sidd Srinivasa and Michael Kaess. He received a MS in Robotics in 2012 and a BS in Computer Science in 2011, both from Carnegie Mellon University. His research interests involve 3D reconstruction, state estimation and visual robotic manipulation.

Siddhartha S. Srinivasa is the Finmeccanica Associate Professor at Carnegie Mellon's Robotics Institute. His research goal is to to enable robots to robustly and gracefully interact with the world to perform complex manipulation tasks under uncertainty and clutter, with and around people. He founded and directs the Personal Robotics Lab. His research has received numerous awards, including 10 best paper award nominations. He is an Editor of the International Journal of Robotics Research, and IEEE IROS.

Jizhong Xiao is a full professor at the Department of Electrical Engineering at the City College, City University of New York, where he has worked since 2002. He directs the CCNY Robotics Lab. He received a Ph.D. from Michigan State University in 2002, a M.Eng. from Nanyang Technological University in 1999. His research interests include the 3D SLAM, wall-climbing robots, and Micro-UAV.