Very High Frame Rate Volumetric Integration of Depth Images on Mobile Devices

Olaf Kähler*, Victor Adrian Prisacariu*, Carl Yuheng Ren, Xin Sun, Philip Torr, David Murray

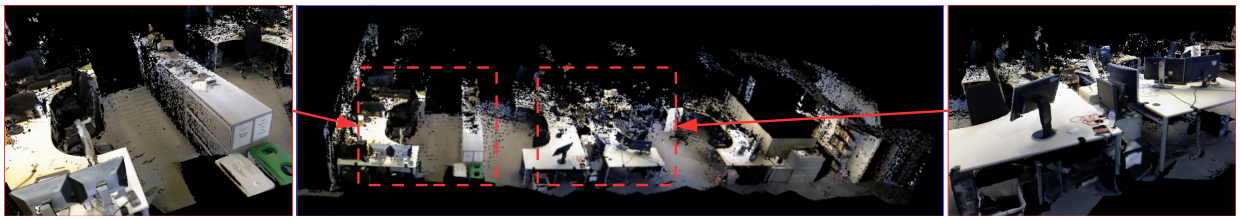

Fig. 1: Sample of the reconstruction result obtained by our system.

Abstract— Volumetric methods provide efficient, flexible and simple ways of integrating multiple depth images into a full 3D model. They provide dense and photorealistic 3D reconstructions, and parallelised implementations on GPUs achieve real-time performance on modern graphics hardware. To run such methods on mobile devices, providing users with freedom of movement and instantaneous reconstruction feedback, remains challenging however. In this paper we present a range of modifications to existing volumetric integration methods based on voxel block hashing, considerably improving their performance and making them applicable to tablet computer applications. We present (i) optimisations for the basic data structure, and its allocation and integration; (ii) a highly optimised raycasting pipeline; and (iii) extensions to the camera tracker to incorporate IMU data. In total, our system thus achieves frame rates up 47 Hz on a Nvidia Shield Tablet and 910 Hz on a Nvidia GTX Titan X GPU, or even beyond 1.1 kHz without visualisation.

Index Terms—3D modelling, volumetric, real-time, mobile devices, camera tracking, Kinect

1 Introduction

Volumetric representations have been established as a powerful method for computing dense, photorealistic 3D models from image observations [14, 8]. The main advantages are their computational efficiency, topological flexibility and algorithmic simplicity. Accordingly, a large number of publications based on volumetric representations has appeared in recent years, as we will discuss in Section 1.1.

For some time, one of the main disadvantages of volumetric representations has been their large memory footprint, as the required memory grows linearly with the overall space that is represented rather than with the surface area. This important issue has been addressed in recent years by sparse volumetric representations like [28, 2, 24] and [16]. As their key idea, they only represent parts of the 3D space near the observed surface, discard the empty space further away, and address the allocated voxels using octrees or hash functions.

While volumetric methods therefore provide high quality, large scale 3D models in real-time for desktop and workstation applications, their computational efficiency is mostly owed to the fact that the operations on individual voxels can be parallelised on modern GPU hardware. Since such GPU hardware is only available to a limited extent on mobile devices, volumetric reconstruction methods are not directly suited for untethered live scanning or augmented reality applications with immediate feedback in mobile environments. Factors like portability, battery life and heat dissipation inherently set a limit on the available computational resources, and without these resources it remains challenging to provide the freedom of movement and instantaneous feedback of results that is crucial for fully immersive applications and augmented reality.

Rather than completely overhauling or even abandoning the previous works on volumetric reconstruction methods, we therefore present ways of optimising them. We show that our extensions enable a speedup by an order of magnitude over previous methods [14, 16] and allow us to run them in real-time on commodity mobile hardware, opening up a whole new range of applications. This requires a number of design choices early on to pick an efficient data structure along with ideas to perform individual processing steps more efficiently and last but not least some low-level refinements to the implementation. Taking all of these together we achieve a system for tracking and integrating IMU augmented 320×240 depth images at up to 47 Hz on a Nvidia Shield Tablet and 20 Hz on an Apple iPad Air 2. Incidentally, the same implementation achieves a framerate up to 910 Hz on a Nvidia GTX Titan X GPU or even beyond 1.1 kHz if visualisation is not required. Our complete and highly optimised implementation is available online at `http://www.infinitam.org`, and we aim to present the design choices that went into it throughout the paper.

1.1 Previous Works

The original ideas of volumetric 3D reconstruction from depth images date back to [3]. Later the advent of inexpensive depth cameras like the Microsoft Kinect and massively parallel processors in GPUs led to the seminal Kinect Fusion system [14, 8], which has been reimplemented a number of times [20, 1, 21] and inspired a wide range of further work [6, 12, 5]. The insights have also been applied to 3D reconstruction from monocular cameras [13, 17]. While we keep such monocular applications in mind, we focus on depth cameras as input in this work.

One of the major limitations of volumetric approaches is the large memory footprint, and the aforementioned early works can therefore only handle small scenes. Various ideas have been investigated to overcome this limitation. One of the first was the use of a moving volume, where only the current view frustum of the camera is

*Olaf Kähler and Victor Adrian Prisacariu contributed equally to this work.
All authors are with the Department of Engineering Science, University of Oxford, Oxford, OX1 3PJ. E-mail: {olaf,victor,carl,dwm}@robots.ox.ac.uk xin.sun@st-hughs.ox.ac.uk philip.torr@eng.ox.ac.uk

Manuscript received 18 Sept. 2014; accepted 10 Jan. 2015. Date of Publication 20 Jan. 2015; date of current version 23 Mar. 2015.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org.

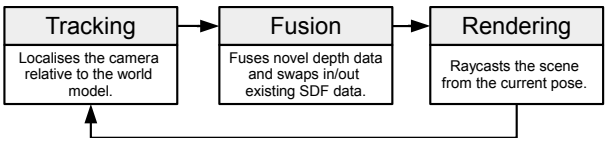

Fig. 2: Overview of the three main processing stages of the proposed system.

kept in the active memory and other parts get swapped out and converted to a more memory efficient triangle mesh [22, 4, 27]. Other approaches try to model the scene with several blocks of volumes, that are aligned with dominant planes [7], or as a bump map representing heights above planes [26]. A third line of approaches uses dense volumes only in blocks around the actual surface and not for the empty space in between. To index and address the allocated blocks, both octrees [28, 2, 24] and hash tables [16, 18] have been used successfully.

While all of the above methods show good or real-time performance on workstation machines with powerful GPUs, mobile devices with limited processing power are not considered in any of them. However, mobile cameras and handheld scanners provide significant advantages over tethered ones, and allow a whole new range of novel applications particularly in the domain of mixed and augmented reality. Dense, photorealistic models of small scale 3D objects have been obtained on mobile devices before in [19], and also at a slightly larger scale and interactive framerates in [25, 9, 23]. Large-scale modelling with the help of depth cameras has so far not been addressed. This is exactly what the present work is about, optimising the wide range of dense volumetric 3D reconstruction methods from depth cameras and bringing them to mobile devices with their flexibility.

1.2 System Outline

In line with many of the previous works, we model the world using a signed distance function, that we represent volumetrically. We adopt the concept of voxel block hashing as introduced by [16] to store the volumetric data efficiently. However, in this work we have adapted and redesigned the data structure to allow for much faster read and write operations, as we will show in Section 2.

Based on this representation we develop our SLAM system for dense mapping, which consists of three stages, as shown in Figure 2: (i) a *tracking* stage to localise incoming new images, (ii) a *fusion* stage to integrate new data into the existing 3D world model and swap data between the compute device (e.g. GPU) and long term storage (e.g. host memory or disk), and (iii) a *rendering* stage to extract the information from the world model that is relevant for the next tracking step. These stages are outlined respectively in Sections 5, 3 and 4. At a coarse level our system therefore follows the well established dense SLAM pipeline of e.g. [14]. Each of the stages however has been heavily optimised to allow for very high frame rates and mobile device operation. We will briefly summarise implementation details in Section 6 and present experimental evaluations in Section 7. Finally we conclude our paper in Section 8.

2 World Representation

As in many previous works such as e.g. [14], we model the world using a truncated signed distance function (T-SDF) D, that maps each 3D point to a distance from the nearest surface. Being truncated, points with distances outside the truncation band $[-\mu...\mu]$ are mapped to a value indicating out-of-range. The surface $\mathcal{S}$ of the observed scene is captured implicitly as the zero level set of this T-SDF, i.e.:

$$\mathcal{S} = \{\mathrm{X} | D(\mathrm{X}) = 0\}. \quad (1)$$

The function values of the T-SDF D are stored volumetrically by sampling with a regular grid.

Since large parts of the world are outside the truncation band μ, most of the grid points will be set to the out-of-range value indicating empty space. A considerable reduction in memory usage is therefore achieved by using a sparse volumetric representation. In our case we use small blocks of usually $8 \times 8 \times 8$ voxels that *only* represent the scene parts inside the truncation band densely. The management and indexing of these blocks can be done using an octree as in [28, 2, 24] or a hash table [16]. As tree based approaches require lots of branching to traverse the full depth of the trees, we decided to use the hash table approach in our work. This promises constant lookup times in case of no hash collisions and only little overhead to resolve such collisions.

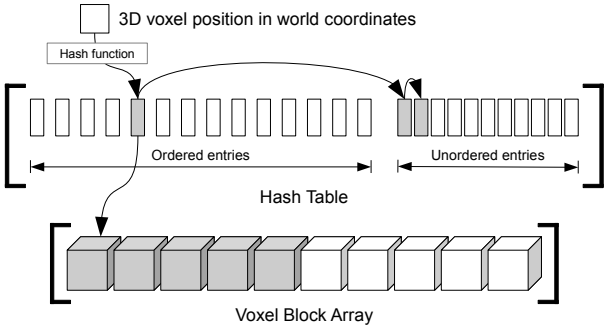

Fig. 3: Logical representation of the underlying data structure for voxel block hashing.

The logical structure of this representation is illustrated in Figure 3. Each 3D location in world coordinates falls into one block of $8 \times 8 \times 8$ voxels. The 3D voxel block location is obtained by dividing the voxel coordinates with the block size along each axis. We then employ a hash function to associate 3D block coordinates with entries in a hash table, which in our current implementation is the same as in [16] i.e.:

$$h = ((b_x \times P_1) \oplus (b_y \times P_2) \oplus (b_z \times P_3)) \bmod K, \quad (2)$$

where mod and $\oplus$ are the modulo and logical XOR operators, (b_x, b_y, b_z) are the block coordinates, (P_1, P_2, P_3) are the prime numbers $(73856093, 19349669, 83492791)$, K is the number of buckets in the hash table, and h is the resulting hash table index. At the corresponding bucket in the hash table, we store an entry with a pointer to the location in a large voxel block array, where the T-SDF data of all the $8 \times 8 \times 8$ blocks is serially stored.

The hash function will inevitably map multiple different 3D voxel block locations to the same bucket in the hash table. Such collisions cannot be avoided in practice, and we use a so called chained table with list head cells to deal with them. This means that the entries in the hash table not only contain the pointer to the voxel block array, but also the 3D block position and an index pointing to the next entry in a linked list of excess entries. In Figure 3 the list head cells and the excess list entries are labelled as the *ordered* and *unordered* parts of the hash table, respectively. This structure is in stark contrast with [16], where two or more entries of each bucket are explicitly stored in the hash table and collisions are resolved with open addressing, storing the excess entries in neighbouring buckets. The reasoning behind our design choice, and we experimentally investigate this in Section 7.3, is that the additional storage space invested into two or more entries per bucket is better spent on creating a hash table with two or more times the number of buckets, hence reducing the number of hash collisions and improving the overall system performance. Also the code for accessing the hash table and excess list is significantly simplified, which pays off at the many random access reads required during raycasting.

The main operations for working with the above hash table are (i) the retrieval of voxel blocks, (ii) the allocation and insertion of new voxel blocks and (iii) the deletion of hash table entries. We provide details of these operations in the following.

2.1 Retrieval

The retrieval function first computes the hash index h using the hash function from equation 2. Starting from the head entry of the hash bucket, all elements within the linked list are checked for a match with

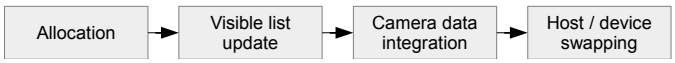

Fig. 4: Outline of the individual processing steps in the fusion stage of the system.

the desired voxel block. Once found, the corresponding voxel block is returned. Depending on the number of hash collisions, this operation generally has constant complexity, unless the load factor of the hash table is disproportionally high, as investigated in Section 7.3.

2.2 Allocation

In the fusion step new information is incorporated into the existing T-SDF, and the above data structure has to be modified to ensure that all blocks observed in the new depth image are allocated. We break this step down into two parts, both of which are trivially parallelisable, require only a minimal amount of atomic operations and do not need any critical sections with nested atomics. This design allows us to achieve optimum performance on parallel processors.

In the first of the two steps we iterate through each pixel in the depth image. For each of the measured depths d we create a segment on the line of sight in the range of the T-SDF truncation band $d-\mu$ to $d+\mu$. We take the block coordinates of voxels on this line segment, compute their hash values, and check whether the block has already been allocated. If it has not, we simply mark the location in the hash table as "*to be allocated with this specific new block*". We do not synchronise the write operations for this marking and if multiple threads require allocations at the same bucket in the hash table, only one of them is selected at random, depending on the execution order. This corresponds to a hash collision within the newly allocated blocks of a single frame, which is exceptionally rare in normal operation, but does happen at the very beginning of the processing, where large parts of the scene suddenly have to be allocated at once. Such intra-frame hash collisions still do not cause lasting artefacts, as the missing blocks are usually cleaned up in the next frame. Should problems ever arise, a trivial workaround would be to to run the allocation twice for the same frame, but for normal operation this is not necessary. The complete elimination of all synchronisation efforts leads to a significantly better runtime performance and also simplifies the second step of the allocation, which can now be parallelised equally trivially.

In the second step we go through the previously created list of marked hash entries that need allocating and update the hash table accordingly. Atomic operations are only required (i) to update a list of available entries in the voxel block array, which we maintain to deal with fragmentation, and (ii) if a new entry has to be allocated in the excess list dealing with hash collisions. Unlike in [16], no critical sections of nested atomic operations are required.

2.3 Deletion

Sometimes entries have to be removed from the hash table and voxel block array. This is the case, for example, when swapping data out of the active list of voxel blocks to a long term storage. Whenever an element has to be removed we simply mark the corresponding hash entry as "*removed*" and add the respective voxel block array pointer to the list of unallocated entries in the voxel block array. Note therefore that (i) we do not reorganise the linked list, thus again avoiding the need for critical sections and (ii) we only require a single atomic operation to update the list of available entries in the voxel block array.

3 Fusion Stage

At the fusion stage we are given a new depth image along with the corresponding camera pose from the tracker, and we update our model of the 3D world to incorporate the novel information. An outline of the required operations is illustrated in Figure 4, and these steps are:

- Allocation: Creates new entries in the hash table and voxel block array.
- Visible list update: Maintains a list of voxel blocks that are currently visible.
- Camera data integration: Incorporates the new camera data into each of the visible voxel blocks.
- Swapping: Optionally swaps in data from a long storage into the active memory and swaps out blocks that are no longer visible.

The allocation step has already been presented in Section 2.2. In the following we discuss the other three steps.

3.1 Visible List Update

At any given time, a large portion of the world map is far from the camera and outside the current field of view. As noted by [16], this can be used to considerably reduce the workload of the integration and rendering stages. Naively, the visible parts of the scene can be identified by directly projecting the eight corners of *all* allocated voxel blocks into the current camera viewpoint and checking their visibility. This guarantees that all visible blocks will be found, but the processing time increases linearly with the number of allocated voxel blocks.

In our work we therefore build and maintain the list of visible blocks incrementally. We assume that voxel blocks can only be visible in a frame, if they were either visible at the previous frame or if they are observed in the current depth frame. The observations at the current frame are already evaluated in the allocation step, and so we only run the visibility check for the voxel blocks that were visible at the previous frame and have not been marked by the allocation stage. This way the list of visible blocks is obtained with very little extra computation and the extra computations are independent of the scene size.

3.2 Camera Data Integration

The camera data integration step now essentially performs the same computations as in the original Kinect Fusion system and its predecessors [14, 3]. Each voxel X maintains running averages of the T-SDF value $D(\mathrm{X})$ and optionally of the RGB values $\mathrm{C}(\mathrm{X})$, which are stored in the voxel block array. The voxel location is transformed into the depth camera coordinate system as $\mathrm{X}_d = \mathrm{R}_d\mathrm{X} + \mathrm{t}_d$, using the known camera pose R_d and t_d. It is then projected into the depth image I_d using the intrinsic parameters K_d, and we compute:

$$\eta = I_d(\pi(K_d\mathrm{X}_d)) - \mathrm{X}_d^{(z)}, \tag{3}$$

where π computes inhomogeneous 2D coordinates from the homogeneous ones and the superscript (z) selects the Z-component of X_d. If $\eta \geq -\mu$, the SDF value is updated as:

$$D(\mathrm{X}) \leftarrow \left(w(\mathrm{X})D(\mathrm{X}) + \min\left(1, \frac{\eta}{\mu}\right)\right) / (w(\mathrm{X})+1), \tag{4}$$

where w is a field counting the number of observations in the running average, which is simultaneously updated and capped to a fixed maximum value. If desired and available, the field of RGB values $\mathrm{C}(\mathrm{X})$ is similarly updated by projection into an RGB image.

As in [16], only voxels in blocks that are currently visible have to be considered. This means a significant reduction in the computational complexity compared to systems like [14].

3.3 Swapping

Sparse volumetric representations enable a very efficient use of memory. While this allows for room-size reconstructions to be kept within the graphics card memory or RAM of a tablet processor, larger scale maps still might not fit. One established approach for dealing with this problem [16, 2] is to swap data between the limited device memory and some much larger long term storage, which could be some host (CPU) memory, or the disk or memory card of a tablet.

We adopt a similar approach and while in general, the active and long term memories can reside on a device (GPU) and host (CPU) with separate memory spaces, the same pattern can be applied to unified memory architectures and long term storage on disk. In our system we

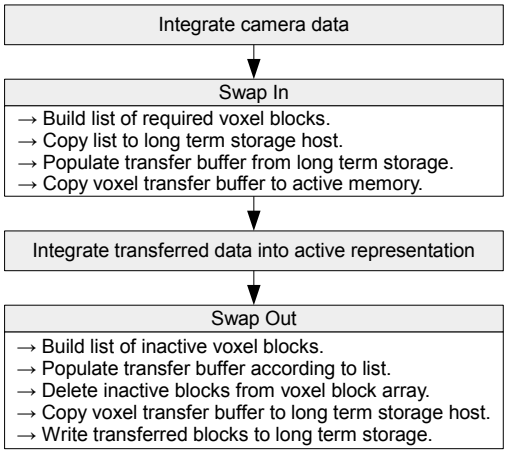

Fig. 5: Outline of the process for swapping data in and out.

use transfer buffers of fixed size. This allows us to set upper bounds on the time spent on data transfers at each frame and implicitly ensures that the data swap does not add a substantial lag, irrespective of the speed of the long term storage device.

We configure the long term storage as a voxel block array with a size equal to the number of buckets in the hash table. Therefore, to check if a hash entry has a corresponding voxel block in the host memory, we only transfer and check the hash table index. The host does not need to perform any further computations, as it would have to do if a separate host hash table was used. Furthermore, whenever a voxel block is deallocated from device memory, its corresponding hash entry is not deleted, but rather marked as unavailable in active memory, and, implicitly, available in the long term storage. This (i) helps maintain consistency between active hash table and long term voxel block storage and (ii) enables a fast visibility check for the parts of the map stored only in the long term storage.

As mentioned, we want to accommodate long term storage systems, that are considerably slower than the online reconstruction. To enable stable tracking and maintain overall reconstruction accuracy, the online system therefore constantly integrates new live depth data even for parts of the scene that are known to have data in the long term storage that is not yet in the active memory. This ensures that we can always track the camera, at least relative to a temporary model that we build in the active memory. By the time the transfer from the long term storage is complete, some voxel blocks might however already hold new data integrated by the online system. Instead of discarding either source of information, we run a secondary integration after the swapping and fuse the transferred voxel block data from the long term storage with the active online data. Effectively we treat the long term storage simply as a secondary source of information.

The overall outline of our swapping operations is shown in Figure 5. This takes place at each frame, but incurs only little speed penalty on the overall integration, as the data transfer is limited to a small size transfer buffer. The list of voxel blocks that needs to be swapped in is built in the same fashion and at the same time as we update the visible list. A voxel block will be *swapped in* if it projects within a pre-defined distance from the boundaries of the currently tracked live frame and *swapped out* once it projects outside the same boundaries.

An example for swapping in a single block is given in Figure 6. The indices of the hash entries that need to be swapped in are stored in the device transfer buffer, filling it up to its capacity. Next, this list is transferred to the host transfer buffer. There the indices are used as addresses inside the long term voxel block array and the target blocks are copied to the host transfer buffer. Finally, the host transfer buffer is copied to the device memory, where a single kernel integrates directly from the transfer buffer into the active voxel block memory.

An example for the step of swapping out a single voxel block is shown in Figure 7. The indices and voxel blocks that are selected for swapping are copied to the device transfer buffer. This is then copied to the host transfer buffer and then to the long term voxel memory.

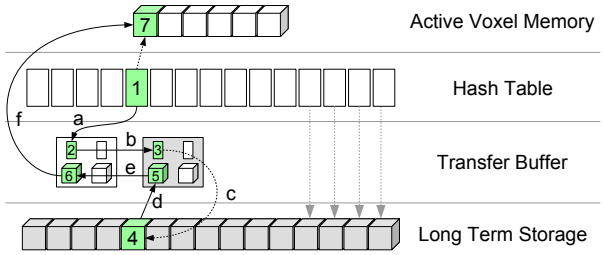

Fig. 6: Swapping in: The hash table entry at address **1** is copied into the device transfer buffer at address **2** (arrow **a**). This is copied to address **3** in the host transfer buffer (arrow **b**) and used as an address inside the host voxel block array, pointing to the block at address **4** (arrow **c**). This block is copied back to location **7** (arrow **f**) inside the device voxel block array, passing through the host transfer buffer (location **5**, arrow **d**) and the device transfer buffer (location **6**, arrow **e**).

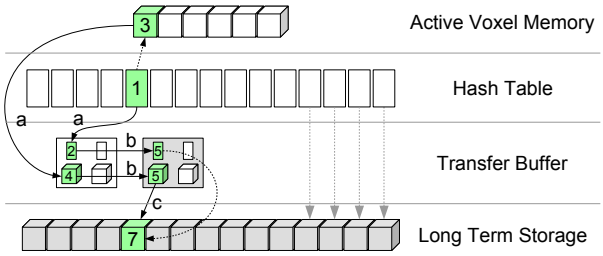

Fig. 7: Swapping Out. The hash table entry **1** and the voxel block **3** are copied into the device transfer buffer locations **2** and **4**, respectively (arrows **a**). The entire transfer buffer is then copied to the host, to location **5** (arrows **b**), and the hash entry index is used to copy the voxel block into location **7** inside the host voxel block array (arrow **c**).

4 Rendering Stage

Extracting information from the implicit representation as a T-SDF is a crucial step that we perform at the rendering stage. We will essentially determine the part of the zero level set $\mathcal{S}$ from Equation 1 that is visible from a given camera viewpoint. In normal operation, this is required primarily to extract information for the tracking stage, but it will also be a fundamental prerequisite for visualisation in a user interface.

The basic operation for extracting the surface information from the T-SDF is raycasting [14]. Given the desired output camera parameters, we cast a ray into the scene for each pixel of the output image and once we found the first intersection of the ray with the surface, we render this point. In our implementation there are four distinct aspects of the rendering stage that we will elaborate in the following:

- Raycasting range: Given the voxel block structure, we determine minimum and maximum ray lengths for each pixel;
- Raycasting: This is the central step, which finds the intersection of each ray with the 3D surface;
- Surface normal: These have to be extracted for some tracking applications and for realistic shading of the rendering;
- Forward projection and other considerations: Raycasting is not always necessary, optionally sacrificing accuracy for speed.

4.1 Raycasting Range

As noted in [16] and in contrast to the original Kinect Fusion works [14], the sparse representation with voxel blocks has a tremendous advantage in the raycasting stage: it allows us to determine the minimum and maximum search range for the rays in each pixel.

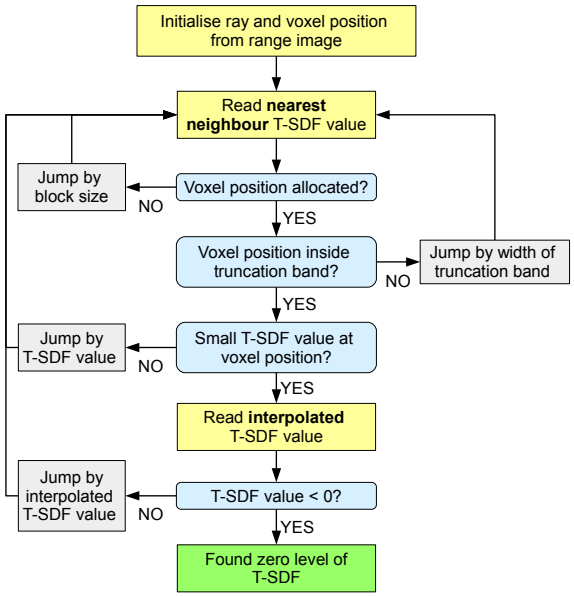

Fig. 8: Flow chart of the raycasting procedure.

In [16], triangle meshes for the boxes around all visible blocks are created and projected into a modified Z-buffer image, which maintains for each pixel the minimum and maximum observed depths. Outside of these bounds the T-SDF contains no information. As we will explain in the following, we modify this approach in two ways to improve the performance. First, we avoid triangle meshes. Second we only compute the raycasting range at a much coarser resolution than the final raycast.

In a first parallelised operation, we project all visible blocks into the desired image. For each block, we compute the bounding box of the projection of all eight corners in image space and store the minimum and maximum Z coordinates of the block in the camera coordinate system. Akin to typical rendering architectures, the bounding box is then split into a grid of fragments of at most 16×16 pixels each, and a list of these fragments is accumulated using a parallel prefix sum. In a second parallelised operation, the fragments are drawn into the modified Z-buffer image, requiring atomic minimum and maximum operations to correctly update the stored values. Due to the subdivision into fragments, the number of collisions in these atomic operations is minimised.

The bounds on the raycasting range determined by projection of the blocks is always at best an approximation. Our approximation with the bounding box of the projection provides slightly more conservative bounds, but is significantly less complex. Furthermore it is not necessary to compute the bounds at the full resolution of the raycast, and we get an even coarser and more conservative approximation by subsampling. There is a tradeoff between spending more time on computing tighter bounds vs. computing more steps in the raycast, and empirically we achieve best results by computing the raycasting bounds at a resolution of 80×60 pixels for a typical raycast of 640×480 pixels.

4.2 Raycasting

In the main step of the rendering stage, we cast rays from the optical centre through each pixel and find their first intersection with the observed 3D surface. In its simplest form, this requires evaluating the SDF along all points of the ray. However, as already stated in [14], the values $D(\mathrm{X})$ in the T-SDF give us a hint about the minimum distance to the nearest surface and allow us to take larger steps.

Due to the further complexity of our T-SDF representation using voxel block hashing, we employ a whole cascade of decisions to compute the step length, as illustrated in Figure 8. First we try to find an allocated voxel block, and the length of the steps we take is determined by the block size. If we found an allocated block, but we are outside the truncation band μ, our step size is determined by the truncation band. Once we are inside the truncation band, the values from the SDF give us conservative step lengths. Finally, once we read a negative value from the SDF, we found the intersection with the surface.

The points along the ray will hardly coincide with the grid points of the discretised T-SDF, hence requiring interpolation. For most of the ray, no sophisticated interpolation is necessary, and it only becomes important once we are close to the surface. For a standard trilinear interpolation step, 8 values have to be read from the T-SDF, as opposed to a single value for nearest neighbour interpolation, and for sake of computational complexity, the selection of interpolation schemes has to be considered carefully. In our implementation we have achieved best results by using trilinear interpolation only if we have read values in the range $-0.5 \leq D(\mathrm{X}) \leq 0.1$ with a nearest neighbour interpolation. The asymmetric boundaries are chosen because errors on the negative side could cause premature termination of rays, whereas errors on the positive side can be corrected at the next iteration. If the T-SDF value is negative after such a trilinear interpolation, the surface has indeed been found and we terminate the ray, performing one last trilinear interpolation step for a smoother appearance.

The read operations in the raycast frequently access voxels located within the same block. To make use of this, we cache the lookup of the blocks, which frequently allows us to completely bypass the hash table in consecutive reads, particularly during trilinear interpolation.

4.3 Surface Normal

Tracking with the ICP method and visualisation of the surface with shading requires to extract a surface normal for each point in the raycasting result, which can be done in two principal ways: Via a 3D gradient computation in the T-SDF, or with a gradient computation in the raycasted image space.

The T-SDF, due to the way it is accumulated in the integration stage, is only ever enforced to be a true SDF along the viewing rays of the integrated depth images. In the orthogonal direction the accumulated values are not necessarily a distance function, and hence their gradients are not necessarily a good approximation of the surface normal. In image space, if the normal is computed with a cross product between the 3D points in neighbouring pixels, the results are independent of this T-SDF artefact, but they are inaccurate at depth discontinuities.

While both ways therefore have their problems, there is a clear winner in terms of complexity. Evaluating the gradient of the T-SDF at a point, that does not fall exactly onto the grid of the discretisation, requires reading 6 interpolated T-SDF values. In image space, since the normals are computed on a regular grid, there are only 4 uninterpolated read operations followed by a cross-product. It is therefore significantly faster to compute surface normals in image space, while empirically the results differ only little.

4.4 Forward Projection and Other Considerations

As it turns out, it is not necessary to perform raycasting at every single frame. The tracker needs 3D surface information to align incoming new images, but it can also align the new images to the surface extracted in a previous frame, as long as the viewpoints are not too different. As an optional approximation we therefore skip the raycast unless the camera has moved by more than a fixed threshold, or unless the most recent raycast is more than a fixed number of e.g. 5 frames in the past. This strategy is illustrated in Figure 9 and evaluated experimentally in Section 7.5, and it turns out that this approximation does not affect tracking accuracy.

For immediate online visualisation, if desired, a rendering of the scene can usually be generated as a byproduct of the raycasting that is required by the tracker. If this raycasting is switched off, as explained above, a forward projection step of the most recent raycasting result can be used for the visualisation, as shown in Figure 9. To this end, we treat the extracted raycasting result as a point cloud and simply project each point into the visualisation image. Since such a simple forward projection of 3D points leads to holes and image areas with missing data, a postprocessing step performs a raycast only for these gaps.

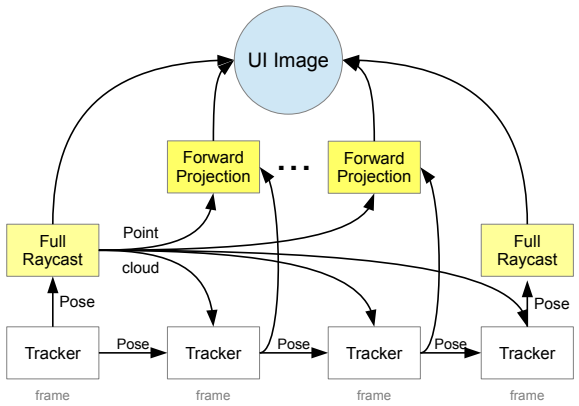

Fig. 9: Speedup by omitting raycasting: The tracker continues localising incoming images relative to the point cloud extracted during the most recent raycast. If a visualisation is desired, the point cloud is also forward projected using the pose from the tracker.

For a more sophisticated, offline visualisation from arbitrary viewpoints, a full raycast similar to the standard approach is required. First we determine a list of visible voxel blocks by projecting all allocated voxel blocks into the desired image. Then the raycasting range image is determined using the modified Z-buffer as described in Section 4.1. Based on this the surface is computed by finding the intersections of rays with the zero level set of the T-SDF, as in Section 4.2. Finally a coloured or shaded rendering of the surface is trivially computed, as desired for the visualisation.

5 Tracking Stage

Various strategies for tracking the camera have been proposed in the context of T-SDF representations. In our system, we implemented an ICP-like tracker as in [14] and a colour based direct image alignment.

In the ICP case we minimise the standard point-to-plane distance:

$$\sum_{\mathrm{p}} \left((\mathrm{Rp} + \mathrm{t} - \mathcal{V}(\bar{\mathrm{p}}))^{\top} \mathcal{N}(\bar{\mathrm{p}}) \right)^2 \tag{5}$$

where p are the 3D points observed in the depth image I_D, (R and t) are the rotation and translation of the camera, $\mathcal{V}$ and $\mathcal{N}$ are maps of surface points and normals, as produced by the raycasting stage, and $\bar{\mathrm{p}}$ are the projections of p into raycasting viewpoint.

In the colour case, we minimise:

$$\sum_{\mathrm{p}} \| I_C(\pi(\mathrm{Rp} + \mathrm{t})) - \mathrm{C}(\mathrm{p}) \|_2 \tag{6}$$

where p and C(p) are the 3D points and their colours, as extracted in the raycasting stage, and I_C is the current colour image.

Either of these energy functions is minimised using the Gauss-Newton approach, and in both cases we use resolution hierarchies. At the coarser levels we optimise only for the rotation matrix R.

Implementing a framework to run on tablet computers has an additional benefit, however: We can make use of the integrated inertial measurement units (IMUs), which will cheaply provide us with the rotational component of the pose. In this case we replace the visual rotation optimisation in both ICP and colour-based trackers with inertial sensor-based rotation obtained using the fusion algorithm of [10]. This is different from e.g. [19] or [15], where the inertial sensor is primarily used as initialisation for the visual tracker. While this does accelerate convergence, it does not reduce rotation drift, as it does not change the local minima of the visual tracker energy function. Our approach, as shown in Section 7.4, has considerably less rotation drift.

6 System Implementation

We have implemented our system and all the optimisations within the InfiniTAM framework from [18]. The framework allows us to deploy our ideas on a wide range of platforms and devices, where platforms include Windows, Linux, MacOS, iOS and Android and devices include CPUs, multi-core CPUs supporting OpenMP, GPUs supporting Nvidia CUDA and APUs supporting Apple Metal.

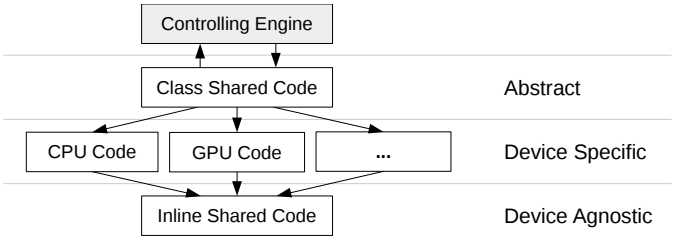

Fig. 10: Schematic of the InfiniTAM cross device architecture.

Table 1: Average computation times per frame for different devices and visualisation strategies. Details on the full raycast, forward projection and no visualisation strategies are given in Sections 4.4 and 7.1.

(a) `teddy` sequence

Device	full	forward	none	[14]	[16]
Nvidia Titan X	1.91ms	1.74ms	1.38ms	26.15ms	25.87ms
Nvidia Tegra K1	36.53ms	31.38ms	26.79ms	-	-
Apple iPad Air 2	82.60ms	65.55ms	56.10ms	-	-
Intel Core i7-5960X	45.28ms	46.75ms	35.40ms	502.69ms	-

(b) `couch` sequence

Device	full	forward	none	[14]	[16]
Nvidia Titan X	1.17ms	1.10ms	0.87ms	19.34ms	15.18ms
Nvidia Tegra K1	25.58ms	21.04ms	19.38ms	-	-
Apple iPad Air 2	56.65ms	48.43ms	41.58ms	-	-
Intel Core i7-5960X	23.43ms	23.38ms	19.94ms	312.86ms	-

Most of the implementation is actually shared across all of these platforms and devices, making use of the InfiniTAM cross device architecture. As illustrated in Figure 10, the implementation is split into three levels, an Abstract level, a Device Specific level and a Device Agnostic level. The Abstract level provides an abstract, common interface and control functions e.g. for the iterative optimisation in the tracker. The Device Specific part executes a for loop or launches a CUDA kernel or similar functionality to perform some computation for example for all pixels in an image. Finally the Device Agnostic part, which is shared as inline code on all devices, performs the actual computations, such as evaluating an error function for a single pixel.

The source code of InfiniTAM along with our extensions and optimisation is available from `http://www.infinitam.org`.

7 Experiments

Since our focus is real time performance on mobile devices, we first present various experiments on the runtime in Sections 7.1 and 7.2. We continue in Section 7.3 with a usage analysis of our hash table, which shows that our argument for simplifying this data structure to a single list head entry indeed holds. We next benchmark the performance of our IMU-based tracker in Section 7.4, and finally investigate the overall accuracy quantitatively and qualitatively in Section 7.5.

7.1 Runtime on Different Devices

As mentioned in Section 6, our system runs on a wide range of different devices, and in the following we compare the performance on these different devices. We recorded two sequences, one of a desk scene with a toy bear, called `teddy` in the following, and one of a living room environment, referred to as `couch`. Samples of both sequences are shown in Figure 11. The `teddy` sequence consists of colour and disparity images from a Microsoft Kinect for XBOX 360, both recorded at 640×480 pixels. The `couch` sequence consists of depth images from an Occipital Structure Sensor with a resolution of 320×240 and orientation information measured with the IMU of an

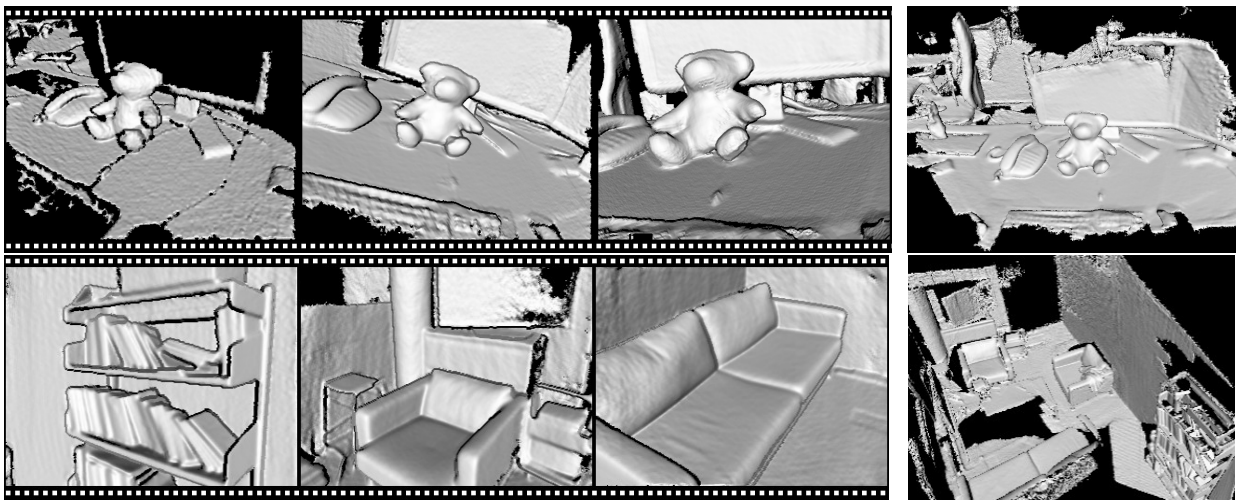

Fig. 11: Samples of the `teddy` (top) and `couch` sequences (bottom) used in some experiments with the final reconstruction results on the right.

Apple iPad Air 2. Accordingly, the `teddy` sequence is representative of reconstruction applications performed at a workstation, while the `couch` sequence is representative of mobile scanning applications with tablet computers and IMU sensors.

In Table 1, we list the measured computation times per frame as averaged over the whole sequences. We evaluate the times on two tablet devices, namely the Nvidia Shield Tablet and the Apple iPad Air 2, as well as on two workstation devices, one CPU only implementation that partly makes use of multi-core parallelisation with OpenMP and a GPU implementation on a Nvidia GTX Titan X. We also compare the runtimes with full raycasting at every frame, forward projection as in 4.4 and with the visualisation turned off completely.

The numbers show that our implementation is running well within real time on tablet computers at 47 Hz. This is based on depth images with a resolution of 320×240 augmented with IMU data and exploiting our proposed forward projection. For 640×480 images and without IMU, it still achieves more than 31 Hz on the same tablet hardware. On workstations with a powerful GPU, even framerates beyond 1.1 kHz are possible, if visualisation is not required. With full raycasting at 640×480 resolution, the implementation still reaches 523 Hz. This is a significant speedup by a whole order of magnitude compared to the previous works presented in [14] and [16]. Furthermore, the fixed volume in the implementation of [14] still fails to capture the entire `couch` scene at a volume size of $512 \times 512 \times 512$. Reducing the volume size of that implementation would result in a significant speedup [12], but only at the cost of an even more reduced scene size or a lower 3D resolution. Mobile, untethered handheld operation on the two tested tablet computers is currently only possible with our implementation.

7.2 Breakdown of Runtime

Figure 12 shows the processing time for a typical frame of the `teddy` sequence, broken down into individual steps. These were measured on a Nvidia GTX Titan X GPU, and the processing is therefore surrounded by transfers of image data between host memory and GPU.

The first major part is tracking and it consists of a short preprocessing step for the resolution hierarchies and a number of ICP iterations, taking less time at coarse resolutions and more at finer levels. At each iteration the error function is evaluated on the GPU and the results are transferred to the CPU, where we compute the actual step and check for convergence. Overall tracking takes usually around 1ms per frame.

The next major part is the fusion stage, which consists of allocation and integration. Taken together, these typically take well under 0.5ms, although at times when large new parts of the scene become visible, allocation can occasionally take slightly longer.

The final part in Figure 12 is the rendering stage. As discussed in Section 4.4, this does not necessarily occur at every frame and there are three options: (i) no raycast at all, (ii) forward projection for visualisation and (iii) full raycast. The figure shows a full raycast. In a first, negligibly fast step, the raycasting bounds are computed. This is followed by the actual raycast, which can optionally be replaced by a faster forward projection. Finally the surface normals are computed in image space. Overall the rendering typically takes under 1ms.

Table 2: Usage of hash table for different numbers of buckets, along with observed and expected number of collisions and load factor.

Buckets	1048576	524288	65536	32768	16384
empty	1037199	512970	55001	23178	8179
1 entry	11265	11150	9624	7928	5657
2 entries	111	166	847	1451	1959
3 entries	1	2	61	193	487
4 entries			3	14	81
≥ 5 entries				4	21
Collisions obs.	225	338	1889	3557	5809
Collisions exp.	125.2	249.0	1854.7	3395.5	5771.0
Load factor	0.011	0.022	0.176	0.350	0.700

Thanks to the efficient representation using voxel block hashing from [16], the fusion is the least time critical stage, and thanks to our optimisation from Section 4, the rendering is also not the major concern anymore. Currently the tracking stage takes up most time, but on typical mobile devices this can be improved by incorporating IMU information as discussed in Section 5.

7.3 Hash Table Usage

In Section 2 we explained our rationale for using a chained hash table with list head cells rather than the open addressing proposed in [16]. In [16], each hash bucket has two or more entries and collisions are resolved by storing the excess entries in neighbouring buckets. Instead we argue that the extra memory of a second entry per bucket is better spent on a hash table with twice the number of buckets, which reduces the overall number of hash collisions and therefore improves overall performance. To underline this argument, we investigate the usage of the hash table and the number of empirically observed hash collisions.

Note that according to the well known Birthday Paradox, the number of expected collisions for a hash table is

$$n \cdot \left(1 - \left(1 - \frac{1}{K}\right)^{n-1}\right), \tag{7}$$

where n is the number of used elements and K is the overall number of buckets in the hash table. We have listed the number of collisions

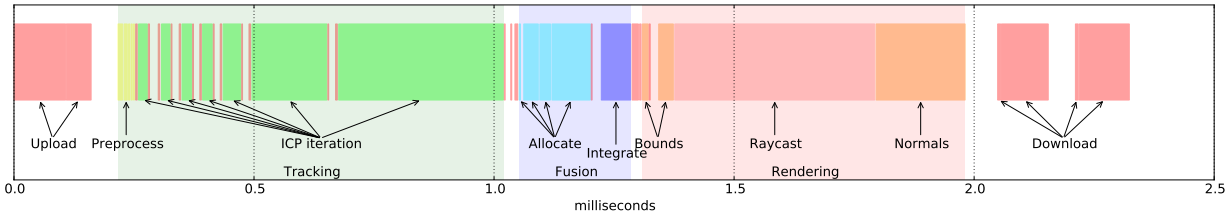

Fig. 12: Timeline of processing for a typical frame on a Nvidia GTX Titan X GPU. The individual sections are explained in the text.

Fig. 13: Photo of the setup used for the IMU experiment.

empirically observed for the `teddy` sequence for hash tables of different sizes along with the expected number of collisions in Table 2. We tried to cover a wide range of load factors in this experiment to simulate different kinds of application scenarios.

As expected, the number of collisions grows significantly with high load factors, but even at very low load factors they can not be completely avoided. The number of observed collisions is consistently above the number of collisions expected for an ideal uniform distribution of hash keys. However, the margin is acceptably small and the hash function seems well suited for the application. As another important observation, the number of buckets with two or more entries is small compared to the overall size of the hash table, even at a load factor of 70%. It therefore seems justified to have no more than one head entry for each bucket, at least if the collisions are stored in an explicit excess list rather than with open addressing.

7.4 Inertial Sensor

Optionally and if available, our tracking stage uses the inertial sensor to provide rotation estimates. To measure the accuracy of this approach we fixed a tablet to a swivel chair (see Figure 13).

We then performed a full rotation of the chair, and reconstructed the scene on the tablet. Figure 14 shows both qualitative and quantitative results obtained for this experiment. The ICP tracker drifts on all three axes by as much as 23°, which leads to an incorrect reconstruction. The IMU based tracker has a maximum drift of 0.09°, thus producing a much more accurate result.

7.5 Quantitative and Qualitative Results

As discussed in the sections above, the runtime of system is highly optimised. In this section we show that this does not lead to any reductions in accuracy. We benchmark our approach on the dataset provided by [6, 12] and list the obtained tracking errors in Tables 3 and 4. Our system consistently outperforms the benchmark reference and the "approximate raycast", which skips the raycast at some frames, does not lead to a significant reduction in accuracy. Samples of the trajectories obtained with our system using the "approximate raycast" strategy are shown in Figure 15. Finally, a comparison between the reconstruction results is shown in Table 5. Again our method outperforms the benchmark reference in virtually all cases.

Further to the quantitative analysis, we show examples of qualitative results in Figures 1 and 16. These large scale results were obtained using colour tracking and integration.

Table 3: Tracking accuracy results on the living room sequence from [6]. Bold shows the best result, in all cases obtained by our method, ITM-Full (full raycast) and ITM-ARC (approximate raycast).

system	kt0-LR	kt1-LR	kt2-LR	kt3-LR
Handa - best	0.0724	0.0054	0.0104	0.3554
ITM-Full	**0.0089**	**0.0029**	0.0088	**0.041**
ITM-ARC	0.0145	0.0047	**0.0072**	0.119

Table 4: Tracking accuracy results on the office sequence of [6]. Bold shows the best result, in all cases obtained by our method, ITM-Full (full raycast) and ITM-ARC (approximate raycast).

system	kt0-OF	kt1-OF	kt2-OF	kt3-OF
Handa - best	0.0216	0.3778	0.0109	0.0838
ITM-Full	**0.0055**	**0.0277**	**0.0079**	0.0047
ITM-ARC	**0.0055**	0.1008	**0.0079**	**0.0042**

8 Conclusion

In this work we have shown a range of ideas and design choices that allow us to run a dense, photorealistic 3D reconstruction algorithm in real time on a tablet computer, based on input from a depth sensor. This requires first of all an efficient representation and data structure and we picked up the basic idea of Voxel Block Hashing [16]. This data structure provides a very efficient way of integrating new data, as we have shown in Section 3. A crucial step is the extraction of surface information for rendering, and we have provided details about our implementation of raycasting in Section 4. Finally we have briefly mentioned the tracker that aligns incoming new images with the stored world representation, and in particular presented details about the integration of IMU data from mobile devices in Section 5. In a series of experiments we then elaborated in Section 7 that (i) the various approximations and shortcuts do not affect the state-of-the-art performance in terms of accuracy, and (ii) that the computational complexity allows processing at up to 47 Hz on a Nvidia shield tablet and up to 910 Hz on a Nvidia GTX Titan X GPU with visualisation, or even beyond 1.1 kHz without.

While we achieve good performance in terms of accuracy and runtime, further work is required to harness the full potential of a system such as ours on mobile devices. Most importantly, truly large scale reconstructions will inevitably accumulate drift and hence result in erroneous reconstructions. In some recent works this is ad-

Table 5: Reconstruction results obtained on the living room sequence [6]. Our method obtains more accurate results in virtually all cases.

Error (m)	kt0	kt1	kt2	kt3
Mean - Handa best	0.0114	0.0080	0.0085	0.1503
Mean - ITM	0.0060	0.0057	0.0048	0.0585
Median - Handa best	0.0084	0.0048	0.0071	0.0124
Median - ITM	0.0060	0.0049	0.0038	0.0535
Std. - Handa best	0.0171	0.0286	0.0136	0.2745
Std. - ITM	0.0046	0.0038	0.0037	0.0415
Min - Handa best	0.0000	0.0000	0.0000	0.0000
Min - ITM	0.0000	0.0000	0.0000	0.0000
Max - Handa best	1.0377	1.0911	1.0798	1.0499
Max - ITM	0.0405	0.0261	0.0429	0.3522

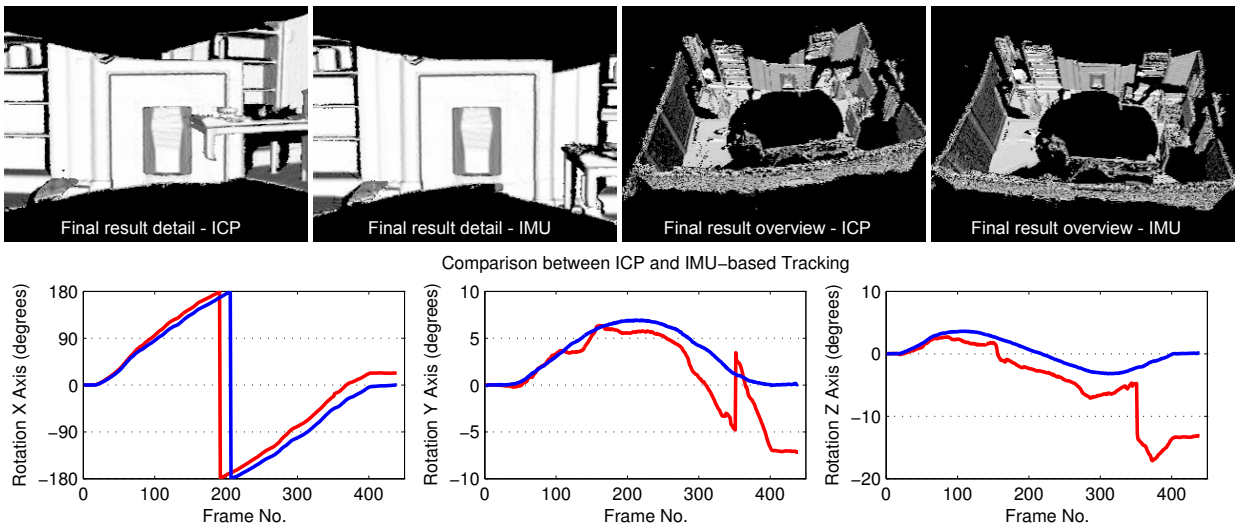

Fig. 14: We compare tracking of a full rotation on a swivel chair with ICP alignment and IMU data. Qualitative results are shown above and quantitative underneath. Poses estimated by the the IMU tracker (blue) start and end very close to the zero level, indicating full rotation (error 0.046°, 0.052° and 0.091° on X, Y, Z axes, respectively), wheres the ICP tracker (red) has a larger drift (23°, 7° and 13° on X, Y, Z axes).

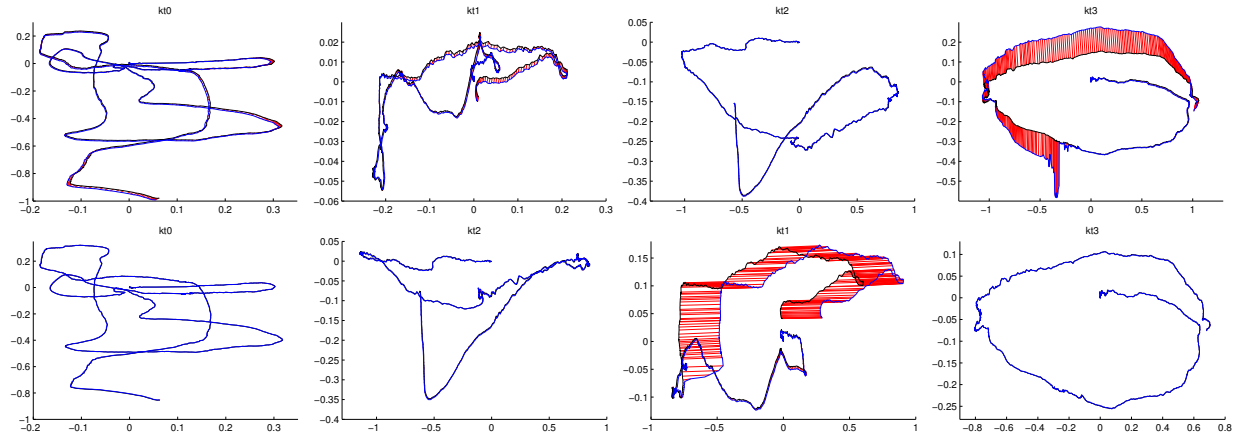

Fig. 15: Trajectories obtained by our system on the dataset from [6], when using approximate raycast, our fastest configuration, first row for the living room sequence and second row for the office sequence. In most of the test sequences our method produces results very close to the ground truth. There are two cases of drift, both caused by non-informative depth information for the ICP tracker.

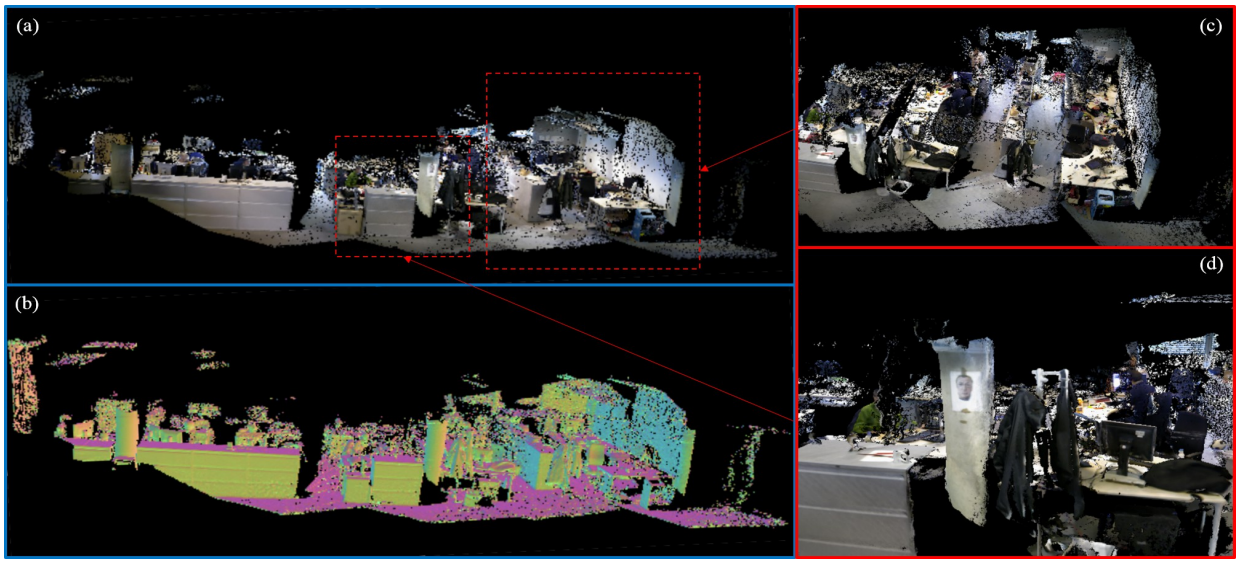

Fig. 16: Reconstruction of a large space using our approach: (a) shows overview, (b) shows surface normals and (c) and (d) are close-up results.

dressed by incorporating loop-closure detection and pose graph optimisation [27, 11] or similar techniques [5]. And while we do not address this problem at all in our current implementation, the excellent runtime performance we achieve means that there is some spare time to accommodate such additional computations. Another major direction of future work is the use of a single RGB camera instead of depth cameras. As shown in [13, 17], the same or very similar volumetric approaches can be used for depth images computed from images of a single, moving camera. And while depth cameras are nowadays widely available, colour cameras are still much more ubiquitous.

ACKNOWLEDGMENTS

This work is supported by UK Engineering and Phyiscal Sciences Research Council (grant EP/J014990).

REFERENCES

[1] A. Baskeheev. An open source implementation of kinectfusion. http://play.pointclouds.org/news/2011/12/08/kinectfusion-open-source/. [accessed 4th September 2014].

[2] J. Chen, D. Bautembach, and S. Izadi. Scalable real-time volumetric surface reconstruction. *Transactions on Graphics*, 32(4):113:1–113:16, 2013.

[3] B. Curless and M. Levoy. A volumetric method for building complex models from range images. In *Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 303–312, 1996.

[4] R. Favier and F. Heredia. Using kinfu large scale to generate a textured mesh. http://pointclouds.org/documentation/tutorials/using_kinfu_large_scale.php. [accessed 4th September 2014].

[5] N. Fioraio, J. Taylor, A. Fitzgibbon, L. D. Stefano, and S. Izadi. Large-scale and drift-free surface reconstruction using online subvolume registration. In *Computer Vision and Pattern Recognition*, pages 4475–4483, 2015.

[6] A. Handa, T. Whelan, J. McDonald, and A. Davison. A benchmark for rgb-d visual odometry, 3d reconstruction and slam. In *International Conference on Robotics and Automation*, pages 1524–1531, 2014.

[7] P. Henry, D. Fox, A. Bhowmik, and R. Mongia. Patch volumes: Segmentation-based consistent mapping with rgb-d cameras. In *International Conference on 3D Vision*, pages 398–405, 2013.

[8] S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison, and A. Fitzgibbon. Kinectfusion: Real-time 3d reconstruction and interaction using a moving depth camera. In *User Interface Software and Technology*, pages 559–568, 2011.

[9] K. Kolev, P. Tanskanen, P. Speciale, and M. Pollefeys. Turning mobile phones into 3d scanners. In *Computer Vision and Pattern Recognition*, pages 3946–3953, 2014.

[10] S. Madgwick, A. Harrison, and R. Vaidyanathan. Estimation of imu and marg orientation using a gradient descent algorithm. In *International Conference on Rehabilitation Robotics*, pages 1–7, 2011.

[11] M. Meilland and A. Comport. On unifying key-frame and voxel-based dense visual slam at large scales. In *International Conference on Intelligent Robots and Systems*, 2013.

[12] L. Nardi, B. Bodin, M. Z. Zia, J. Mawer, A. Nisbet, P. H. J. Kelly, A. J. Davison, M. Luján, M. F. P. O'Boyle, G. Riley, N. Topham, and S. Furber. Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM. In *International Conference on Robotics and Automation*, 2015.

[13] R. Newcombe, S. Lovegrove, and A. Davison. Dtam: Dense tracking and mapping in real-time. In *International Conference on Computer Vision*, pages 2320–2327, 2011.

[14] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, and A. Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *International Symposium on Mixed and Augmented Reality*, pages 127–136, 2011.

[15] M. Nießner, A. Dai, and M. Fisher. Combining inertial navigation and icp for real-time 3d surface reconstruction. In *Eurographics (Short Papers)*, pages 13–16, 2014.

[16] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger. Real-time 3d reconstruction at scale using voxel hashing. *Transactions on Graphics*, 32(6):169:1–169:11, 2013.

[17] V. Pradeep, C. Rhemann, S. Izadi, C. Zach, M. Bleyer, and S. Bathiche. Monofusion: Real-time 3d reconstruction of small scenes with a single web camera. In *International Symposium on Mixed and Augmented Reality*, pages 83–88, 2013.

[18] V. A. Prisacariu, O. Kähler, M. M. Cheng, J. Valentin, P. H. Torr, I. D. Reid, and D. W. Murray. A framework for the volumetric integration of depth images. arXiv:1410.0925 [cs.CV], 2014.

[19] V. A. Prisacariu, O. Kähler, D. Murray, and I. Reid. Real-time 3d tracking and reconstruction on mobile phones. *Transactions on Visualization and Computer Graphics*, 21(5):557–570, 2015.

[20] PROFACTOR. Reconstructme. http://reconstructme.net/. [accessed 4th September 2014].

[21] G. Reitmayr. Kfusion. https://github.com/GerhardR/kfusion. [accessed 15th October 2014].

[22] H. Roth and M. Vona. Moving volume kinectfusion. In *British Machine Vision Conference (BMVC)*, pages 112.1–112.11, 2012.

[23] T. Schöps, J. Engel, and D. Cremers. Semi-dense visual odometry for AR on a smartphone. In *International Symposium on Mixed and Augmented Reality*, pages 145–150, 2014.

[24] F. Steinbruecker, J. Sturm, and D. Cremers. Volumetric 3d mapping in real-time on a cpu. In *International Conference on Robotics and Automation*, 2014.

[25] P. Tanskanen, K. Kolev, L. Meier, F. Camposeco, O. Saurer, and M. Pollefeys. Live metric 3d reconstruction on mobile phones. In *International Conference on Computer Vision*, pages 65–72, 2013.

[26] D. Thomas and A. Sugimoto. A flexible scene representation for 3d reconstruction using an rgb-d camera. In *International Conference on Computer Vision*, pages 2800–2807, 2013.

[27] T. Whelan, M. Kaess, H. Johannsson, M. Fallon, J. J. Leonard, and J. McDonald. Real-time large-scale dense rgb-d slam with volumetric fusion. *International Journal of Robotics Research*, 2014.

[28] M. Zeng, F. Zhao, J. Zheng, and X. Liu. Octree-based fusion for realtime 3d reconstruction. *Graphical Models*, 75(3):126–136, 2013.