FusionMLS: Highly dynamic 3D reconstruction with consumer-grade RGB-D cameras

Siim Meerits, Diego Thomas, Vincent Nozick, Hideo Saito

Computational Visual Media
https://doi.org/10.1007/s41095-018-0121-0

Research Article

FusionMLS: Highly dynamic 3D reconstruction with consumer-grade RGB-D cameras

Siim Meerits[1] (✉), **Diego Thomas**[2], **Vincent Nozick**[3,4], **and Hideo Saito**[1]

Abstract Multi-view dynamic three-dimensional reconstruction has typically required the use of custom shutter-synchronized camera rigs in order to capture scenes containing rapid movements or complex topology changes. In this paper, we demonstrate that multiple unsynchronized low-cost RGB-D cameras can be used for the same purpose. To alleviate issues caused by unsynchronized shutters, we propose a novel depth frame interpolation technique that allows synchronized data capture from highly dynamic 3D scenes. To manage the resulting huge number of input depth images, we also introduce an efficient moving least squares-based volumetric reconstruction method that generates triangle meshes of the scene. Our approach does not store the reconstruction volume in memory, making it memory-efficient and scalable to large scenes. Our implementation is completely GPU based and works in real time. The results shown herein, obtained with real data, demonstrate the effectiveness of our proposed method and its advantages compared to state-of-the-art approaches.

Keywords 3D reconstruction; RGB-D cameras; motion capture; GPU

1 Department of Information and Computer Science, Keio University, Yokohama, Japan. E-mail: S. Meerits, meerits@hvrl.ics.keio.ac.jp (✉); H. Saito, saito@hvrl.ics.keio.ac.jp.

2 Department of Advanced Information Technology, Kyushu University, Fukuoka, Japan. E-mail: thomas@ait.kyushu-u.ac.jp.

3 LIGM, UMR 8049, Université Paris-Est Marne-la-Vallée, Champs-sur-Marne, France. E-mail: vincent.nozick@u-pem.fr.

4 Japanese–French Laboratory for Informatics, CNRS, UMI 3527, Tokyo, Japan.

Manuscript received: 2018-03-09; accepted: 2018-05-29

1 Introduction

Three-dimensional reconstruction has found many applications in various fields such as archaeology [1], design [2], and architecture [3]. In the digitization and preservation of cultural heritage, for example, 3D models of ancient artifacts can be built on-site using existing automated tools such as RGB-D SLAM that do not require experts. In contrast, for applications in mixed reality in which real data can be mixed with virtual content [4], it is still difficult to reconstruct accurate 3D models in real time when objects are moving rapidly or when the scene is large.

As the multi-view reconstruction of *static* scenes has matured, the research focus has shifted to multi-view reconstruction of *dynamic* scenes containing moving objects. Such 3D reconstruction remains a challenging problem and well-studied topic in the fields of computer vision, virtual reality, and robotics. The advent of consumer-grade RGB-D cameras that can capture both depth and color information has motivated a wave of research on dynamic 3D scene reconstruction in the last few years. We may divide most existing techniques for dynamic scene reconstruction using RGB-D devices into two main categories: (a) *fusion-based* methods that track the motions of objects in the scene, and accumulate captured data into a canonical representation of the scene, and (b) *frame-based* methods that reconstruct a 3D model independently for each set of images taken at the same time. Both strategies have several advantages and limitations.

While *fusion-based* methods allow reconstruction of visually appealing 3D models with smooth surfaces, smooth motion, and high levels of detail, they

generally fail to reconstruct fast-moving [①] objects. Indeed, tracking fast-moving objects is extremely difficult, and the slightest error may corrupt the canonical 3D representation of the scene. This can initiate a vicious feedback circle of tracking errors, eventually leading to completely inaccurate model reconstruction. Furthermore, using deformable models or object templates causes issues when handling large and complex scene topology changes [5–7]. Moreover, focusing only on either human reconstruction [8] or foreground objects [6, 7, 9–11] restricts applicability of reconstruction approaches. Most fusion-based methods use truncated signed distance field volumes for fusing depth data. This has high memory consumption, limiting the size of the scenes that can be reconstructed. Memory reduction strategies [12] exist, but they tend to considerably increase the algorithmic complexity of reconstruction.

Frame-based methods reconstruct a 3D model of the scene from a set of RGB-D camera images taken at a single moment in time; reconstruction is done independently for each frame. In contrast to fusion-based methods, as long as the input RGB-D images are well synchronized in time, there is no need to track the deformation of the constructed 3D model. This also guarantees that rapid movements can be correctly reconstructed. However, using only a single frame of data is a drawback that generally results in poorer 3D model details than from fusion-based methods.

In this paper, we follow the frame-based strategy and present a 3D reconstruction method designed to be utilized with commodity RGB-D camera hardware. Thus, in our approach, the whole 3D scene is reconstructed from scratch on each frame. This trades some loss of 3D model precision for increased resistance to topological changes. Our method does not require a template, can reconstruct scene backgrounds, and has a small memory footprint. In addition to providing real-time reconstruction, our method allows very flexible playback of recorded data. This includes synthetic 3D slow motion based on interpolation between camera frames. The technical contributions of this work are (1) a new robust and accurate time calibration method for consumer RGB-D cameras, (2) a fast depth map interpolation method to synthesize scene point clouds at arbitrary time, and (3) a real-time moving least squares-based volumetric reconstruction method with a small memory footprint.

[①] It is difficult to quantify fast movement, but empirically we consider position change of 40 pixels or more per frame to be fast.

2 Related work

3D reconstruction research initially focused on off-line reconstruction of static scenes. Gradually the focus has shifted to dynamic scenes and achieving real-time processing speeds. In this paper, we divide related works into two main categories, *fusion-based* methods and *frame-based* methods. A comprehensive review of reconstruction algorithms can be found in Ref. [13].

2.1 Fusion-based methods

Fusion-based methods track the motions of objects in the scene over time and accumulate the captured data into a canonical representation of the scene. Following this strategy, parametric reconstruction methods display impressive results by deforming models of objects known a priori. *Performance capture* systems track motion by deforming a model of a human [8, 14]. Zollhöfer et al. [5] presented a way to scan any 3D object template, which can then be deformed in real time. The obvious drawback of fixed templates and parameterized models is their inability to deal with unexpected scene topology changes or the appearance of unknown objects in the scene. Wang et al. [15] proposed a templateless reconstruction method that can efficiently track non-rigid motions. However, the method does not work in real time and complex topology changes may not correctly be tracked.

Truncated signed distance function (TSDF) volumes [16] have been at the forefront of non-rigid 3D reconstruction research. TSDF volumes allow accumulation of scene details over multiple frames to achieve high-quality reconstruction, but they also require accurate tracking of object movements to avoid corrupting geometry. Newcombe et al. [6] introduced DynamicFusion, wherein depth data is accumulated into a canonical model of a scene, which is subsequently deformed using a warp field to match scene changes in real time. Innmann et al. [7] improved DynamicFusion by estimating a more dense warping field, and Guo et al. [17] made use of albedo information in motion tracking. Zhang and Xu [18] added an option to reconstruct scene backgrounds

by segmenting dynamic and static content. While the reconstruction quality is high, these methods can fail when scene topology changes considerably from that of the initially estimated model. Moreover, these methods were designed to be used with a single RGB-D camera. Dou et al. [10] proposed resetting the deformed model periodically to allow more extensive scene topology changes. The method relies on multiple custom-built capture devices. Follow-up work [11] has improved the reconstruction and motion estimation accuracy via machine learning techniques but retains a similar hardware setup.

2.2 Frame-based methods

Frame-based methods reconstruct a 3D model independently for each set of simultaneously-taken images. Therefore they do not require tracking of the motions of objects in the scene. Poisson surface reconstruction, proposed by Kazhdan and Hoppe [19], is a popular choice for generating high-quality 3D models from point clouds. The method considers the full structure of the scene to produce watertight meshes. An adaptation of the method for dynamic scenes together with a multi-camera capture setup was put forward by Collet et al. [9]. While the reconstruction has very impressive quality, the high computational cost and custom camera setup requirement have made the method far from real-time and hard to use. Wang et al. [20] proposed another Poisson reconstruction-based system that utilizes a much simpler camera setup, but it fails under rapid motion. Alexiadis et al. [21, 22] used Fourier transform-based reconstruction together with an a priori human model. The methods use consumer RGB-D devices and generate high-quality models, but they are restricted to capturing human movement.

Reconstruction methods based on moving least squares (MLS) use local fitting of point clouds to obtain a refined set of points on surfaces. Kuster et al. [23] showed near-real-time multi-camera reconstruction by rendering MLS-processed points using point splatting. In applications, however, triangle mesh models are preferred over points. Meerits et al. [24] used a method inspired by mesh zippering [25] to generate triangle meshes for MLS surfaces. The drawback of this work is that when using many cameras, mesh density becomes unnecessarily high and the mesh joints can present some ambiguities.

3 Proposed method

The major components of our proposed system are outlined in Fig. 1. We use multiple consumer-grade RGB-D devices (such as the Microsoft Kinect 2) to capture the scene to be reconstructed. Cameras are connected to small computers, termed clients. In turn, the clients are connected over a network to a single server machine. The server decompresses and buffers the received data, which is later forwarded to a GPU for reconstruction. The reconstruction process is generally divided into two parts, (1) motion estimation and (2) geometric surface estimation, ending with a visualization of the results.

We use volumetric 3D reconstruction with a hierarchical spatial structure: a volume of space to be reconstructed consists of *blocks* that in turn consist of voxels. This structure speeds up reconstruction as we can quickly determine whether a block contains any surfaces or not. The geometric surface is estimated using an MLS approach that assigns a signed distance function value to each voxel. A triangle mesh is extracted from the volume using the marching cubes

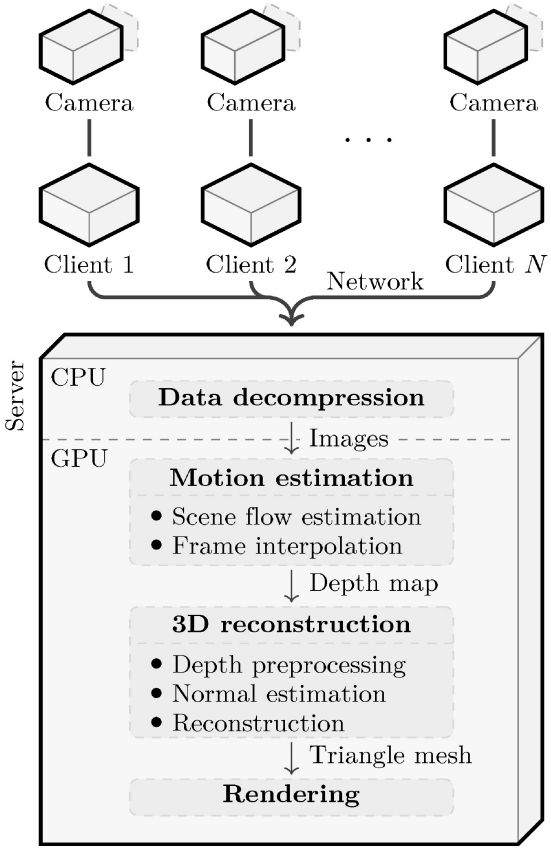


Fig. 1 FusionMLS pipeline.

algorithm. Finally, the reconstructed 3D model is rendered.

For accurate 3D reconstruction, the input images must be temporally consistent. This means that the depth maps should appear as if they were captured at exactly the same time from all cameras. Consumer RGB-D devices, however, typically lack shutter synchronization, so the captured frames are temporally inconsistent. To tackle this issue, we developed a depth interpolation scheme that generates a new temporally consistent set of images from raw RGB-D camera data. This works by (1) calibrating the clocks of all cameras so that accurate timestamps can be assigned to all captured frames, (2) estimating scene flow between each pair of consecutive depth maps for each camera separately, and (3) warping depth maps to the desired point in time using scene flow vectors. Scene flow is computed by estimating the non-rigid transformation between depth maps.

4 System setup

4.1 Hardware topology

The RGB-D devices can be placed in various configurations. When capturing small scale activity, the cameras are usually placed around objects as shown in Fig. 2 to maximize view coverage.

All recent consumer-level RGB-D devices are USB connected, which imposes strict limits on cable length. By attaching a small client computer to each RGB-D camera, every client can be connected over a much more flexible Ethernet network to a server machine, thus avoiding the constraint of RGB-D camera cable length.

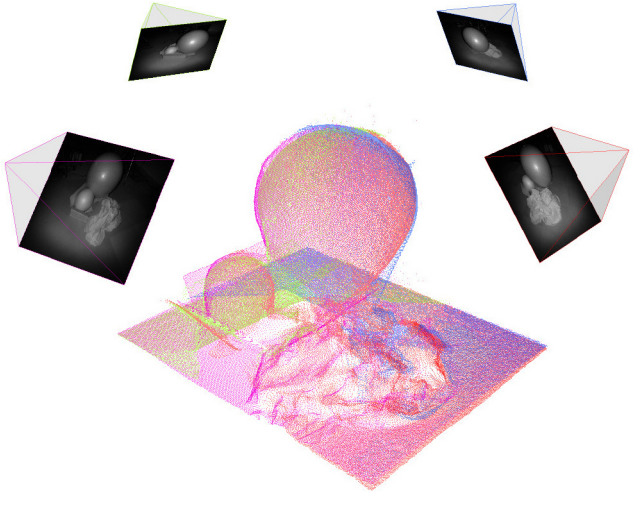


Fig. 2 Example camera setup. Four RGB-D cameras, shown with infrared images, capture a point cloud of a sample scene. The colors of the points indicate capture by different cameras.

The clients compress all depth maps, infrared images, and color images received from the RGB-D cameras and send the data over a custom network protocol to the server. The server decompresses and buffers the images for reconstruction. Network transfers and image compression and decompression can incur variable latency. We therefore perform reconstruction at a defined point in time 500 ms after the actual time. In other words, 3D reconstruction is delayed half a second to accommodate pipeline latency.

The data received from the clients can also be recorded for later playback. When using offline reconstruction, the user is free to specify any playback speed. Because our proposed method can interpolate between frames, we can simulate slow motion playback of the scene.

All scene reconstruction takes place on the GPU to achieve highly parallel processing. At the start of reconstruction, the necessary image frames are buffered and uploaded to the GPU memory. Because the reconstruction process requires at least two consecutive camera frames from each camera, we can achieve further speedup by buffering pairs of frames on the GPU for each camera.

4.2 Time calibration

Consumer-level RGB-D cameras have timing-related drawbacks that must be addressed, including the fact that the shutters of multiple cameras cannot be synchronized and that devices lack time calibration functionality. Because we are unable to force synchronization of image capture by the RGB-D cameras, we must compensate for object movement depending on image capture time. Such motion compensation requires precise timestamps for all captured frames. This brings us to the second RGB-D camera drawback: the camera clocks must be calibrated despite difficulties.

Alexiadis et al. [22] proposed solving the clock synchronization issue by recording audio using the camera's built-in microphone. The audio can then be used to align the video streams of different cameras in post-processing. We follow another route and propose an online method with high calibration accuracy that

works with any RGB-D camera by assigning accurate timestamps to image frames. In the following, the reported results and constants are for the Microsoft Kinect 2 device that was used in our experiments.

The time calibration starts by setting the clocks of all computers using Network Time Protocol (NTP). As is typical for hierarchical networks, the server first obtains the time from a nearby stratum 2 clock and the clients in turn calibrate their clocks by querying the time from the server. Because the network adapters of all machines support hardware timestamping, we can achieve sub-microsecond clock accuracy within the local network. The time synchronization software consistently reported measured offsets below 1 µs with respect to the local timeserver.

Next, we calibrate the client computer clocks with the connected RGB-D cameras. Each camera is assumed to have a precise internal clock that starts during device initialization. Unfortunately, there is no straightforward way to synchronize this clock to computer time. The time is exposed in the data packets sent from the camera to client. These packets include a timestamp with a precision of 125 µs. On the client, we record the arrival time of each USB packet from the camera. Although the USB protocol has variable speed and latency, we can correlate the device and computer clocks over a large enough sample of timestamps.

Let t_c and t_d be the computer and the camera device time, respectively. We model the relation between these timers as

$$t_c = (1 + s_d)t_d + o_d + c_{dc} \quad (1)$$

where s_d is the clock skew, o_d is the offset between the timers, and c_{dc} is the average time between image capture and delivery of the image to the computer. We can recover s_d and o_d using a linear least squares regression analysis. The constant c_{dc}, however, is dependent on the hardware and software being used. Because our capture setup consists of homogeneous hardware, we assume that this time delay is constant across devices. Hence, the relative timestamps of any captured frames remain valid.

From a sample of 3000 timestamps captured over 100 seconds, we found a significant time skew s_d of -179.2 ± 0.4 PPM (confidence level of 95%). If the time calibration is repeated after every 100 seconds, we get time uncertainty of ±40 µs from skew. The timer offset o_d has a confidence interval of ±23 µs. We can conclude that the Kinect 2 RGB-D camera has a reliable internal timer that can be calibrated to sub-millisecond precision. However, the calibration should be repeated periodically, for instance, every 100 seconds, to reduce the time uncertainty from clock skew estimation error.

4.3 Camera setup and calibration

Our 3D reconstruction method requires precise intrinsic and extrinsic camera calibration.

Time-of-flight-based RGB-D cameras such as the Kinect 2 typically have one depth sensor and one color imaging sensor at different viewpoints. Intrinsic parameters for both are calibrated in-factory and are readable from the device. Extrinsic parameters for converting coordinates between sensors are also typically available. In the case of the Kinect 2, the extrinsic transformation is given in a custom high-degree polynomial format. We convert it to a more convenient Euclidean rotation and translation transformation.

Extrinsic calibration between RGB-D cameras is done using color images. Initially, one camera is fixed to the global coordinate system origin. Next, a classical checkerboard-pattern with a known size is used to determine the transformation between initially fixed and other cameras. It is possible that all cameras cannot simultaneously see the checkerboard. In that scenario checkerboard is moved to get a chain of transformations linking all cameras. As the cameras in our experiments are in fixed positions, we can repeat the extrinsic calibration a number of times. With the use of quaternions we can average multiple transformation estimates to achieve more precise results. It also allows measuring standard error of rotation and translation parameters to determine the accuracy of extrinsic calibration.

5 Motion estimation

As detailed in Section 4.2, all image frames received from the cameras are assigned precise timestamps. Due to the consumer-oriented nature of the hardware, we are unable to control shutter trigger time. The result is that the cameras capture fast moving objects at different time. A naive 3D reconstruction of such data would result in one object appearing at slightly different locations simultaneously.

Our solution is to generate new synthetic camera frames such that it appears as if the scene were captured at the same time by all RGB-D devices. Essentially, we interpolate images between consecutive camera frames. Since the interpolation method is continuous, we are free to generate data for any point in time. This also leads to interesting applications such as generating synthetic slow motion videos.

The basic strategy for interpolating depth data from cameras is to estimate scene flow for every camera separately. The depth can then be warped to an interpolated time by applying scaled scene flow vectors to depth points. For an overview of scene flow estimation methods please refer to Yan and Xiang [26].

Unfortunately, the available methods tend to be computationally too expensive to be practical for estimating multiple flow maps in real time. Therefore, we designed our own scene flow estimation method, which trades some estimation quality for speed. In a nutshell, scene flow is estimated by finding correspondences in consecutive depth maps. We generate a mesh of depth points in one frame and warp it iteratively to the closest points on the second depth map frame. Regularization is achieved by imposing some local rigidity constraints on the mesh.

5.1 Scene flow

Scene flow estimation takes as input two consecutive depth maps $\mathcal{D}$ and $\mathcal{D}'$ from a single RGB-D camera. For the first depth map, $\mathcal{D}$, we generate a dense mesh $\mathcal{M}$ from the depth pixels. Essentially, all neighboring depth points with distance less than a user-set threshold m_{t} are connected by an edge. Our aim is to warp this mesh so that it matches depth map $\mathcal{D}'$. The warped mesh is denoted $\mathcal{M}'$. The scene flow vectors for depth points then equal the displacements of vertices between meshes $\mathcal{M}$ and $\mathcal{M}'$.

To reduce the computational cost of finding correspondences between $\mathcal{M}$ and $\mathcal{D}'$, we generate multiple mesh and depth scales, $\{\mathcal{M}_0, \ldots, \mathcal{M}_n\}$ and $\{\mathcal{D}'_0, \ldots, \mathcal{D}'_n\}$, respectively. A mesh scale $\mathcal{M}_i$ is found by downscaling $\mathcal{M}_{i-1}$ to half size. Downscaling works by removing every second vertex in horizontal and vertical directions. Vertices in $\mathcal{M}_i$ are joined by edges only if a path of edges exists with length two or less connecting corresponding vertices in mesh $\mathcal{M}_{i-1}$. The mesh warping strategy begins at the highest scale mesh $\mathcal{M}_n$, which is iteratively matched to target depth map $\mathcal{D}'_n$. The warping parameters are then propagated down to the next mesh scale $\mathcal{M}_{n-1}$. We store correspondences between vertices at levels n and $n-1$ when downsampling. Hence we can simply copy flow data for vertices which exist on both levels. The data for vertices that only exist on level $n-1$ can be generated by averaging flow vectors of neighboring edge connected vertices with already copied data. The warp estimation and propagation procedure is repeated at each scale until we reach mesh $\mathcal{M}_0$.

The mesh warping procedure works in two steps. First, for all vertices in $\mathcal{M}_i$ we find the closest points in $\mathcal{D}'_i$ using a grid search and store the new vertex positions in $\mathcal{M}'_i$. Since warping is carried out at multiple scales, a fairly small search window of 5×5 suffices for finding good correspondences. Secondly we need to regularize the found correspondences to get more accurate flow vectors. This is done by imposing local rigidity constraints on the meshes. Given corresponding vertices $v \in \mathcal{M}_i$ and $v' \in \mathcal{M}'_i$, we minimize the energy function:

$$E_i = \sum_{a,b \in \mathcal{M}_i} \|(v_a - v_b) - (v'_a - v'_b)\|^2 \qquad (2)$$

where the sum is over all pairs of vertices connected by edges in the mesh. We carry out energy optimization by gradient descent due to its simplicity. In practice, each gradient descent iteration makes the mesh more rigid. Hence, we can easily tune the mesh rigidity via the number of iterations.

A single warping pass on each mesh level is typically sufficient for slowly moving objects. In the case of rapidly moving objects, the depth has to be warped a long distance from its original location. Since the point correspondence search just selects closest points, many initial matches can be inaccurate. Regularization cannot completely fix bad initial correspondences. Hence it is best to repeat warping a few times, using the last warping result to increase quality. The major steps of flow estimation are shown in Fig. 3.

5.2 Depth warping

The final step of motion estimation is to interpolate the depth frames based on the depth map meshes calculated in the previous subsection.

Our system runs at a constant frame rate determined by the user: the reconstruction time is

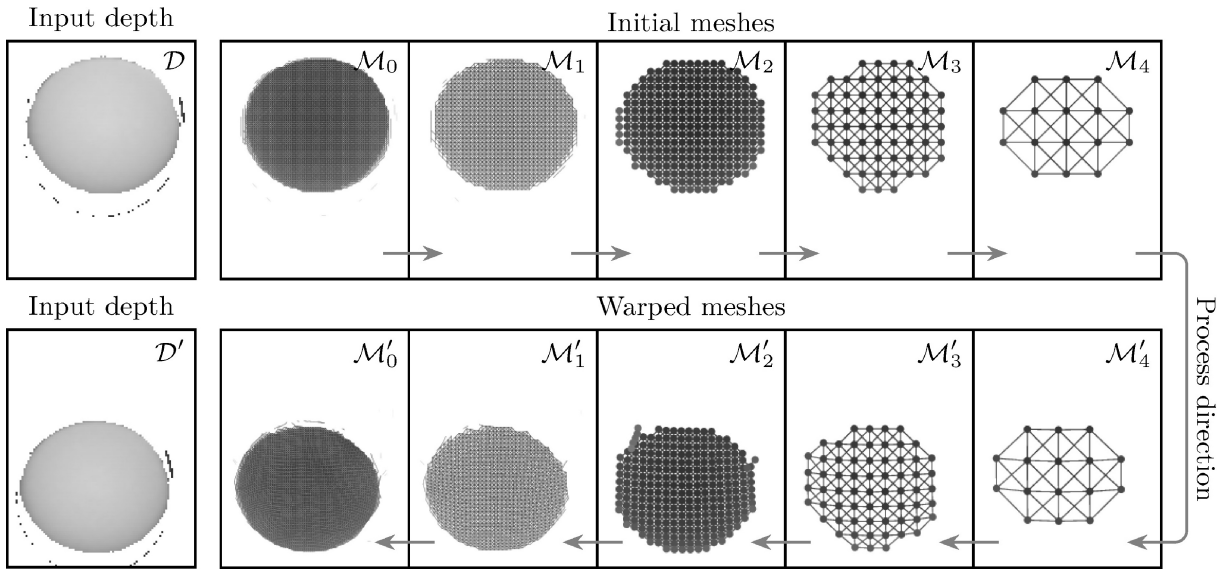


Fig. 3 Scene flow estimation using a rapidly moving spherical object. Mesh $\mathcal{M}_0$ is generated from input $\mathcal{D}$ and is transformed through a series of steps, resulting in mesh $\mathcal{M}'_0$, which closely matches the second input depth map $\mathcal{D}'$.

not influenced by RGB-D camera frame timestamps. Let the reconstruction time be t. In that case, for each camera separately, we find the consecutive depth frames $\mathcal{D}$ and $\mathcal{D}'$ from the buffers at time t_1 and t_2, respectively, such that $t_1 \leqslant t < t_2$. The interpolation ratio between those depth frames is then $r = (t_2 - t) / (t_2 - t_1)$.

Next we generate a new mesh $\mathcal{M}''$ by interpolating the vertex positions of meshes $\mathcal{M}_0$ and $\mathcal{M}'_0$. Given the corresponding vertices $v \in \mathcal{M}_0$ and $v' \in \mathcal{M}'_0$, the new vertex position for mesh $\mathcal{M}''$ is $(1 - r)v + rv'$. The resulting mesh $\mathcal{M}''$ can effectively be rendered using standard computer graphics tools to produce a new interpolated depth map. Figure 4 shows an example of interpolation at three different time points using real data.

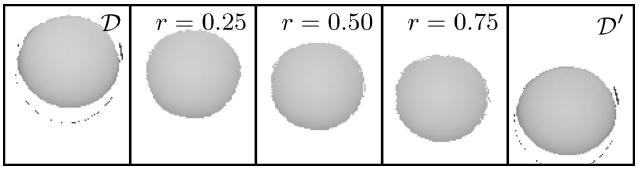


Fig. 4 Depth interpolation. Three interpolated depth maps with interpolation ratios of 0.25, 0.50, and 0.75 are generated from input depth maps $\mathcal{D}$ and $\mathcal{D}'$.

6 Volumetric reconstruction

3D reconstruction of the collected images requires as input multiple RGB-D camera depth maps, with a common timestamp, taken from different cameras. These images are fused into a triangle mesh model of the scene. Our proposed method is designed to reconstruct the whole scene from scratch on each frame (one frame consists of a set of multiple RGB-D images with the same timestamp). That is, no reconstruction data is stored for use in reconstruction of other frames. This approach prevents corruption of the 3D reconstruction by incorrect model tracking.

The reconstruction starts by filtering the depth maps for noisy object edges and estimating the initial surface normals. Next, a block occupancy process finds regions of space that are likely to contain surfaces, in order to reduce the reconstruction cost. Finally, the surface geometry is estimated and a surface mesh is generated in the main reconstruction process.

6.1 Preprocessing

The input to our reconstruction method consists of depth maps generated from the process presented in Section 5.1. We start with simple filtering of the depth maps and estimation of the initial surface normals.

Depending on the type of RGB-D camera used, the depth maps can contain various types of noise. While our reconstruction method can handle per-pixel measurement noise, completely incorrect surfaces should be avoided. A typical issue with RGB-D cameras is that object edges in depth maps can be

noisy or incorrect. Such areas are best avoided, and we alleviate this problem by eroding by one pixel the depth map on object edges. These object edges are found by looking for neighboring depth points separated by a distance greater than a threshold value m_t, also used in Section 5.1. Note that simultaneous use of multiple time-of-flight RGB-D cameras, operating at the same frequency, can cause interference. Most commonly, this interference can make the pixel values periodically vibrate around their true value. In our experiments, the effect was not pronounced, and no specific filter was used to remove such potential artifacts. Details of time-of-flight camera interference are discussed by Li et al. [27], who also proposed a filtering strategy.

The initial surface normals should be estimated at each depth map point location. It is best to calculate the normals for each depth map separately as multiple depth maps may not be perfectly aligned, which may distort the normal estimates. Normal estimation starts by calculating gradients:

$$g_x(x,y) = p(x+1,y) - p(x-1,y) \tag{3}$$

and

$$g_y(x,y) = p(x,y+1) - p(x,y-1) \tag{4}$$

at all depth pixel coordinates. Here $p(x,y)$ is a 3D point corresponding to a depth map pixel at coordinates (x,y). Next, we calculate temporary normals as

$$u(x,y) = g_x(x,y) \times g_y(x,y) \tag{5}$$

Finally, the initial normals are calculated as a spatially weighted sum over a small window around (x,y) as

$$n(x,y) = \sum_{i,j} u(x,y) w(\, \|p(x,y) - p(i,j)\| \,) \tag{6}$$

where $w(\cdot)$ represents spatial weighting. We follow Guennebaud and Gross [28] and define the weight function as

$$w(r) = \begin{cases} \left[1 - \left(\frac{r}{h}\right)^2\right]^4, & r < h \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

where h is a constant spatial smoothing factor. The window size for normal calculation can be derived from the spatial smoothing factor h and RGB-D camera parameters dynamically for each point. However, in terms of GPU code optimization, we found it best to use a hardcoded 7×7 pixel window for our test scenes.

6.2 Volume hierarchy

Our method is a volumetric reconstruction approach; the reconstructed scene area is defined as the reconstruction volume. This volume can be specified by the user or calculated from the camera positions and their parameters. The smallest volume elements are voxels, arranged in a grid-like fashion. We also define a *block* as a sub-volume of voxels of fixed size. A block has a uniform number of voxels in all dimensions, for instance, $8 \times 8 \times 8$.

There are two major reasons for dividing the total volume into blocks. The first reason is that a spatial hierarchy allows us to determine the occupied volume regions, which need to be reconstructed. In other words, we eliminate the need for expensive voxel calculations in areas where there are no depth map points and hence no surfaces. This method is simpler than using octrees or k-d trees and can be computed quickly. The second reason concerns storing voxel values. We prefer not to store voxel data in the GPU main memory as it is expensive and we have no use for this data when reconstructing the next frame. Our method is more light-weight than other proposed volume memory reduction schemes [29, 30]. However, it is important to note that we do need voxel values to generate the mesh. The size of a block of voxels is low enough to permit its temporary storage. Furthermore, we can utilize modern GPU features in this scenario to speed up the processing.

Modern GPUs have multiple types of memory with different characteristics. Global memory is plentiful and is persistent from allocation to deallocation but has slow access time. Shared memory, on the other hand, has limited availability of just tens of kilobytes, and the data is not persistent during program execution, but it has much faster access time. According to GPU manufacturer documentation, recent GPUs such as those in the Nvidia GeForce range have roughly 100 times lower shared memory latency than for uncached global memory. We leverage this to greatly accelerate 3D reconstruction. Typical volumetric reconstruction methods store per-voxel data in GPU global memory. This data needs to be read and written when estimating surfaces. Furthermore, mesh generation also requires multiple lookups of the voxel values. In contrast, our method

is designed to use only shared memory to store voxel data.

Using the shared memory comes with certain restrictions. Most importantly, the data is stored only for the duration of GPU thread group execution. This means that after any voxel value calculation, we must immediately run mesh generation on the same voxels. As the amount of shared memory is very limited, this limits maximum block size.

Our mesh generation method, which uses the marching cubes algorithm, can only generate triangles between neighboring voxels. Essentially, every possible $2 \times 2 \times 2$ voxel sub-volume is processed to yield zero or more triangles. We can run marching cubes inside a block of voxels but not at block edges. However, using the shared memory for block voxel storage means that we cannot look up the voxel values of neighboring blocks. We solve the problem similarly to Ref. [30] and make all blocks overlap each other by one voxel in each direction. As an example, if a block consists of $8 \times 8 \times 8$ voxels, then all blocks are laid out with spacing of seven voxels on all axes.

Because the voxel blocks overlap, some voxels are processed several times as parts of different blocks. We can calculate the theoretical worst-case overhead as $s^3/(s-1)^3$, where s is the size of the block in voxels. In the case of an $8 \times 8 \times 8$ voxel block, we get 49% processing overhead. While the extra processing might seem considerable, the savings from not having to store and load voxels in global memory is greater. However, the block size s should be chosen with the balance of shared memory usage and processing overhead in mind.

6.3 Block occupancy

To determine which blocks of voxels are likely to contain surfaces, we count the number of points found in each block. A three-dimensional array of integers is allocated with one entry per block in the reconstruction volume. Assuming we are using a GPU that supports atomic operations, we project all depth map points to the reconstruction volume. Atomic addition, AtomicAdd(m, n), which adds the value n to some array location m, is used to sum the number of points in each block in parallel. Note that because the blocks overlap by one voxel on each side, we must detect when points are on edges and add them to other block counts as well. The procedure is summarized in Algorithm 1. An example of finding the number of points inside the blocks is depicted in Fig. 5.

Algorithm 1 Block occupancy calculation

Input: camera points $\boldsymbol{p}$
Output: occupancy volume V

reset occupancy volume V to zeros
for every point $\boldsymbol{p}$ from cameras {parallelized} **do**
  $\boldsymbol{p} \leftarrow R\boldsymbol{p} + t$ {transform point to volume}
  $\boldsymbol{b} \leftarrow \boldsymbol{p}/(s-1)$ {block coordinates}
  AtomicAdd$(V(\boldsymbol{b}), 1)$
  if $\boldsymbol{p_x} \bmod (s-1) = 0$ and $\boldsymbol{b_x} > 0$ **then**
    AtomicAdd$(V(\boldsymbol{b} - [1,0,0]^{\mathrm{T}}), 1)$
  end if
  if $\boldsymbol{p_y} \bmod (s-1) = 0$ and $\boldsymbol{b_y} > 0$ **then**
    AtomicAdd$(V(\boldsymbol{b} - [0,1,0]^{\mathrm{T}}), 1)$
  end if
  if $\boldsymbol{p_z} \bmod (s-1) = 0$ and $\boldsymbol{b_z} > 0$ **then**
    AtomicAdd$(V(\boldsymbol{b} - [0,0,1]^{\mathrm{T}}), 1)$
  end if
end for

Next we need to find a list of non-empty blocks so that they can be reconstructed. We iterate over all blocks to determine whether the point count exceeds a constant threshold b_{t}. This threshold acts as a coarse point cloud filter. In practice, however, blocks tend to have relevant surfaces even at quite low point counts and hence setting $b_{\mathrm{t}} = 1$ is recommended. Again we utilize the atomic add operation to create a list of occupied block coordinates. Details are given in Algorithm 2.

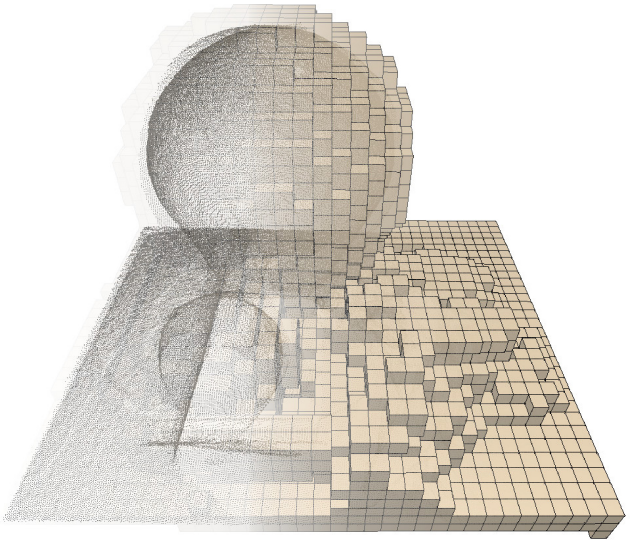


Fig. 5 Visualization of blocks to be reconstructed. Left side shows the input point cloud and the right side shows the corresponding non-empty blocks.

Algorithm 2 Block list generation

Input: occupancy volume V
Output: block index B

```
i ← 0
for every block coordinate b {parallelized} do
  if V(b) ⩾ b_t then
    j ← AtomicAdd(i, 1)
    B(j) ← b
  end if
end for
```

6.4 Reconstruction

The main part of scene reconstruction consists of estimating the surface geometry and generating the respective triangle meshes. The surfaces are defined implicitly using a signed distance function (SDF). The meshing algorithm can then find a zero-level set of SDF and output triangles.

We estimate SDF for each voxel by sampling nearby points from all RGB-D cameras. State-of-the-art dynamic scene reconstructions using signed distance functions (e.g., Refs. [6, 7]) typically use only one depth map point per camera to update single voxel values. This method works well only if the SDF is updated over multiple frames. Because our volume is not stored between reconstructions, this approach does not suit our needs. We choose to estimate the local surfaces using MLS, which samples many points in the neighborhood of the voxel.

To estimate the local surface, we first need to retrieve depth points $\boldsymbol{p}_i$ and the corresponding initial surface normals $\boldsymbol{n}_i$ from the neighborhood of a given voxel position $\boldsymbol{p}$. Following Kuster et al. [23], we project $\boldsymbol{p}$ to each RGB-D depth map and retrieve a $u \times u$ square block of depth points $\boldsymbol{p}_i$ and normals $\boldsymbol{n}_i$ around the projected point.

For the actual surface estimation, we follow the moving least squares formulation put forward in Ref. [31]. Given a voxel position $\boldsymbol{p}$, some corresponding points $\boldsymbol{p}_i$, and some normals $\boldsymbol{n}_i$, we can calculate a new weighted position:

$$a(\boldsymbol{p}) = \frac{\sum_i w(\,\|\boldsymbol{p}-\boldsymbol{p}_i\|\,)\boldsymbol{p}_i}{\sum_i w(\,\|\boldsymbol{p}-\boldsymbol{p}_i\|\,)} \tag{8}$$

and weighted normal:

$$n(\boldsymbol{p}) = \frac{\sum_i w(\,\|\boldsymbol{p}-\boldsymbol{p}_i\|\,)\boldsymbol{n}_i}{\sum_i w(\,\|\boldsymbol{p}-\boldsymbol{p}_i\|\,)} \tag{9}$$

where $w(r)$ is the spatial weighting function previously defined in Eq. (7) for use in normal estimation.

We also define a voxel *confidence* value simply as

$$c(\boldsymbol{p}) = \sum_i w(\,\|\boldsymbol{p}-\boldsymbol{p}_i\|\,) \tag{10}$$

Finally, the implicit surface distance function is given as

$$f(\boldsymbol{p}) = n(\boldsymbol{p})^{\mathrm{T}}(\boldsymbol{p}-a(\boldsymbol{p})) \tag{11}$$

In cases where the confidence value $c(\boldsymbol{p})$ is below a constant user-specified threshold c_t, we mark $f(\boldsymbol{p})$ as invalid. This effectively removes surfaces that do not have enough points for an accurate estimation. We store the SDF value $f(\boldsymbol{p})$, the normal $n(\boldsymbol{p})$, and the confidence value $c(\boldsymbol{p})$ for each voxel. The normal is stored so that there is no need to estimate the surface normal again during meshing. Also the confidence value can be used to generate smooth object edges during rendering.

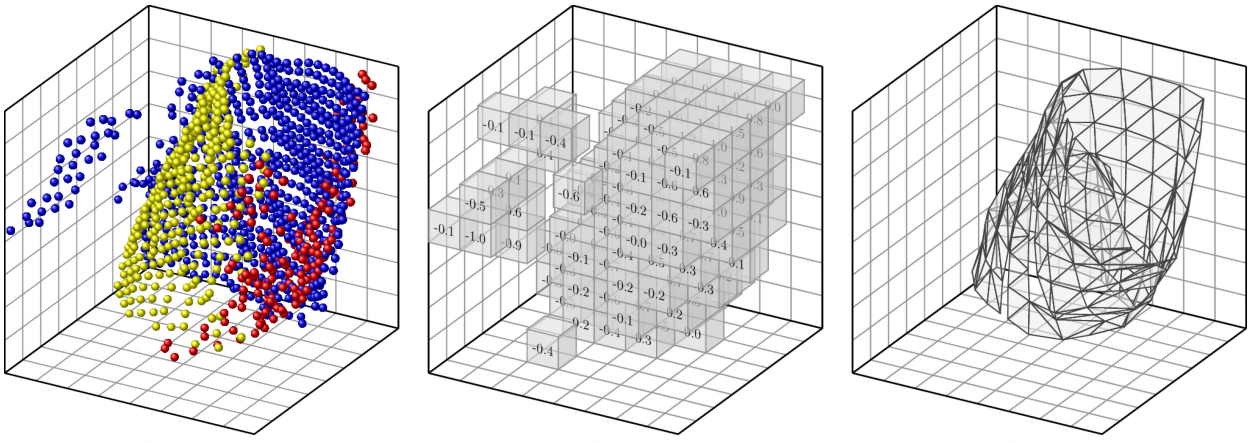
To generate the mesh, we use marching cubes triangulation [32]. Every possible $2 \times 2 \times 2$ voxel sub-volume of the block is passed to triangulation. The marching cubes method decides which triangles to create between each voxel based on SDF values. If any values $f(\boldsymbol{p})$ are found marked as invalid, then no triangles are output. All valid triangles are written to a global buffer and include a point location $\boldsymbol{p}$, a normal $n(\boldsymbol{p})$, and a confidence $c(\boldsymbol{p})$ attribute for each vertex.

Both surface estimation and mesh generation are summarized in Algorithm 3. The major steps of reconstruction are also visualized in Fig. 6.

Algorithm 3 3D reconstruction and mesh generation

Input: block index B
Output: triangle mesh M

```
i ← 0
for every block b in index B {parallelized} do
  ▷ MLS reconstruction
  for voxels c ∈ [0, 1, ..., s]^3 {parallelized} do
    p ← R(c + sb) + T {global coordinates}
    F(c), N(c) ← MovingLeastSquares(p)
  end for
  ▷ Marching cubes triangulation
  for voxels c ∈ [0, 1, ..., s − 1]^3 {parallelized} do
    m ← MarchingCubes(F(c), N(c))
    for all triangles t ∈ m do
      j ← AtomicAdd(i, 1)
      M(j) ← t
    end for
  end for
end for
```



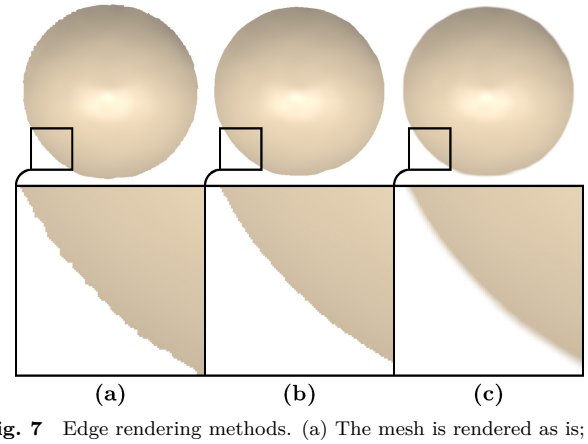
(a) Point cloud **(b)** Voxels **(c)** Mesh

Fig. 6 Reconstruction of an $8 \times 8 \times 8$ voxel block using real-world data. (a) Input point cloud data; colors mark different RGB-D cameras; (b) MLS voxel values (only negative voxel values are visualized); (c) meshing result after marching cubes triangulation. Note that the outlier points on the left side of the block are successfully excluded from the final result.

6.5 Rendering

The reconstructed scene is rendered solely using triangles generated by the process described in the previous subsection. The included normals can be used to shade the 3D model under lighting.

The 3D mesh can contain discontinuities at edges of thin objects or due to limited depth map coverage of the scene. A naive rendering of such areas will result in jagged edges. This is because marching cubes has no native way of handling discontinuities. However, we can use the confidence value $c(\boldsymbol{p})$ of the vertices to smoothly cut off triangles at a user-defined threshold c_r. Optionally, the edges can be smoothly made transparent using alpha blending together with an order-independent transparency technique such as depth peeling [33]. Different ways of handling edge rendering are visualized in Fig. 7.



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

Fig. 7 Edge rendering methods. (a) The mesh is rendered as is; (b) mesh triangles are cut off at a user-defined confidence value; and (c) mesh triangles smoothly transition from opaque to transparent based on confidence value.

7 Results

All of our experiments were conducted on a server with an Intel i7-5930K CPU, 32 GB of RAM, and an Nvidia GeForce GTX 1080 Ti graphics card. The client computers were Intel NUC7i3 machines with an Intel i3-7100U CPU and 16 GB of RAM. For RGB-D cameras, we exclusively used Microsoft Kinect 2 devices.

The system performance characteristics for a typical scene recorded with four RGB-D cameras can be seen in Table 1. Because our pipeline is completely executed on the GPU, precise statistics for each process step can be obtained by OpenGL timer queries. We used the system parameters given in Table 2, which were tuned to obtain maximum reconstruction quality while retaining a real-time frame rate.

An important aspect of our reconstruction system is the ability to handle scenes with significant dynamic content. This includes fast-moving objects as well as changing scene topology. Figure 8 shows

Table 1 Performance

Process	Avg. time	Max. time
Motion estimation		
Scene flow	11.7 ms	12.7 ms
Depth warping	0.7 ms	1.0 ms
Reconstruction		
Preprocessing	2.5 ms	2.9 ms
Block occupancy	0.07 ms	0.05 ms
Reconstruction	17.6 ms	19.3 ms
Total	32.6 ms	36.0 ms

Table 2 Recommended parameters

Parameter	Value
Motion estimation	
Mesh segmentation threshold	$m_t = 1.5$ cm
Mesh layers	$n = 4$
Reconstruction	
Number of voxels in volume	2×10^7
Block size in voxels	$s^3 = 8^3$
Minimum points in block	$b_t = 1$
Neighbor search window	$u = 11$
Spatial weight radius	$h = 4$ cm
Confidence value threshold	$c_t = 30$

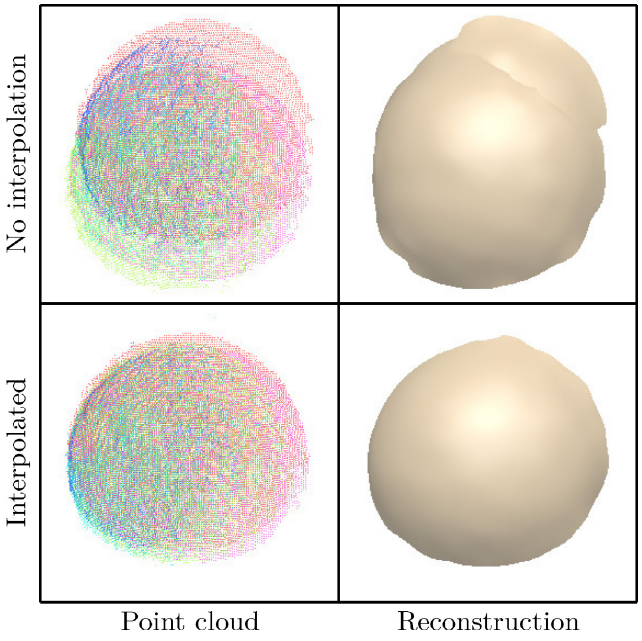


Fig. 8 Frame interpolation using a fast-moving spherical object. Without interpolation (above), the point clouds from the different cameras are not aligned. Using interpolation (below), the point clouds are aligned and the object is correctly reconstructed.

the effectiveness of the approach. A selection of challenging situations is shown in Fig. 9 and in the Electronic Supplementary Material (ESM). Rapid movements of objects are correctly reconstructed thanks to our depth frame interpolation method.

Human actor reconstruction can be seen in Fig. 10 and in the accompanying video in the ESM. Source data was taken from a publicly available dataset, courtesy of Alexiadis et al. [21]. The resulting model has good quality where visibility from cameras is good. However, as our method is designed to be very general, we do not use human templates to fill missing surface areas.

Large scenes can also be reconstructed with our method in real time. Figure 11 shows a room of size 10m × 3m × 3m. Movements can more clearly be seen in the supplementary video in the ESM. The scene can be reconstructed in real time mostly because of the block occupancy test, which avoids the need to reconstruct empty spaces. Moreover, because we do not store the reconstruction volume, the memory cost is low. For a volume with 2×10^7 voxels, we only need to store block occupancy, taking 153 kB, and block index data, taking up to 457 kB of memory. The rest of the memory usage is related to input depth maps and normals, and the output triangle mesh.

The number of systems to which we can compare our method is limited. Recent dynamic scene reconstruction methods, especially ones that utilize truncated signed distance volumes, require fusing depth data over multiple frames and are not designed to handle cameras with unsynchronized shutters. In addition, we are restricted in choosing comparative methods as our scenes contain backgrounds and highly non-rigid objects, such as cloth, for which template generation is very difficult.

We compared our 3D reconstruction method with another MLS method that uses mesh zippering [24] and with Poisson surface reconstruction [19]. Some results for highly dynamic and large scenes are shown in Fig. 11 and Fig. 12, respectively. These methods were chosen for their ability to reconstruct entire scenes from only one depth frame per RGB-D camera. In terms of quality, mesh zippering tends to have rough edges at surface discontinuities. Additionally, joining meshes can fail in some areas with complex geometry, resulting in small holes in the meshes or incorrectly generated triangles protruding from surfaces. The Poisson method works off-line, taking between 2 and 4 s to reconstruct the scenes in Fig. 11 and Fig. 12 when the reconstruction depth parameter is set to 8. This method has the ability to complete a surface even when point cloud data is missing from some scene areas. While this feature can be beneficial in repairing some areas of the generated mesh, it also has drawbacks. Firstly, real-world scenes tend to be topologically open and have many boundaries. These areas are incorrectly reconstructed by the Poisson method. Secondly, repairing surfaces is an under-determined problem and holes in geometry can be filled in various ways. This results in temporally inconsistent surfaces.

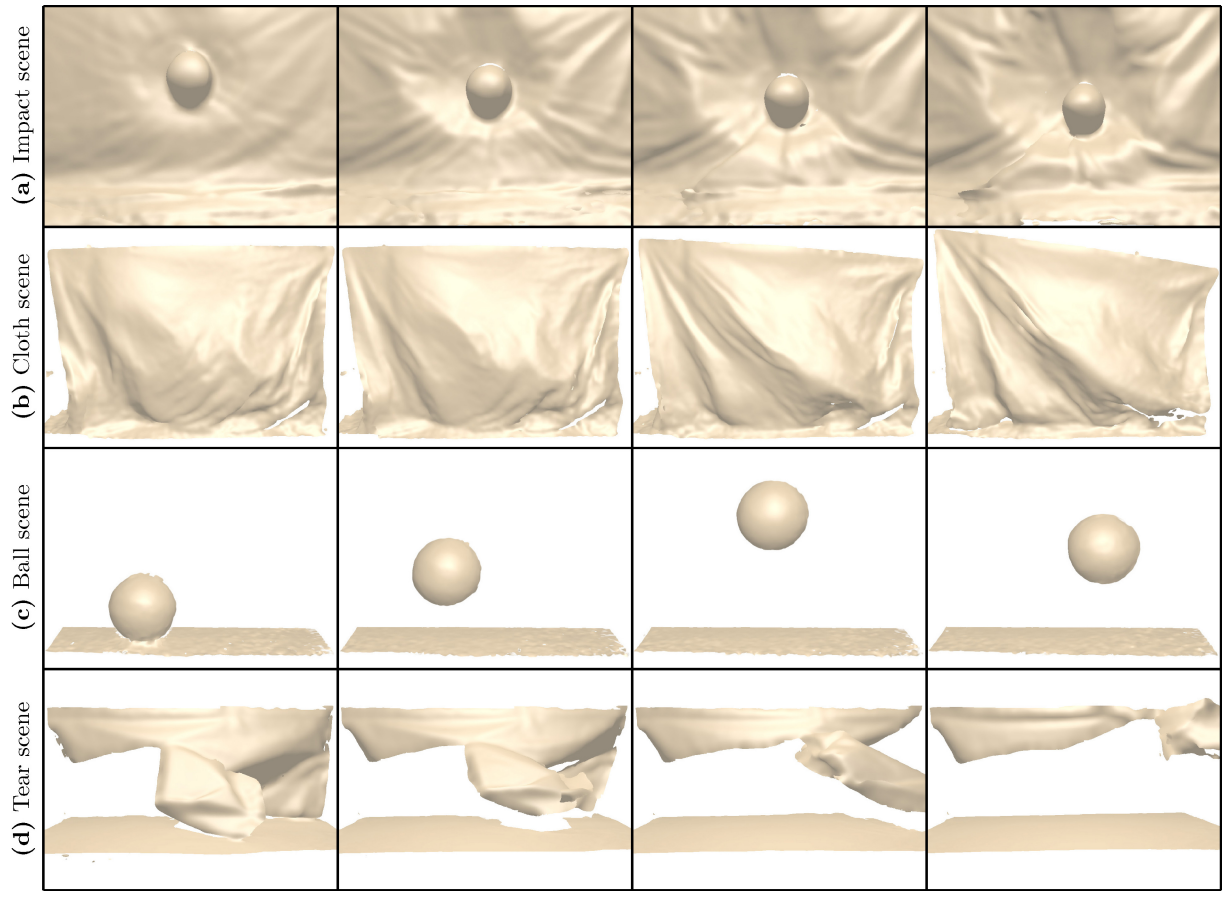


Fig. 9 Reconstructing highly dynamic scenes: (a) a ball is thrown at a cloth curtain; (b) a cloth is shaken; (c) a ball bounces off the ground; and (d) a large piece of paper is torn apart.

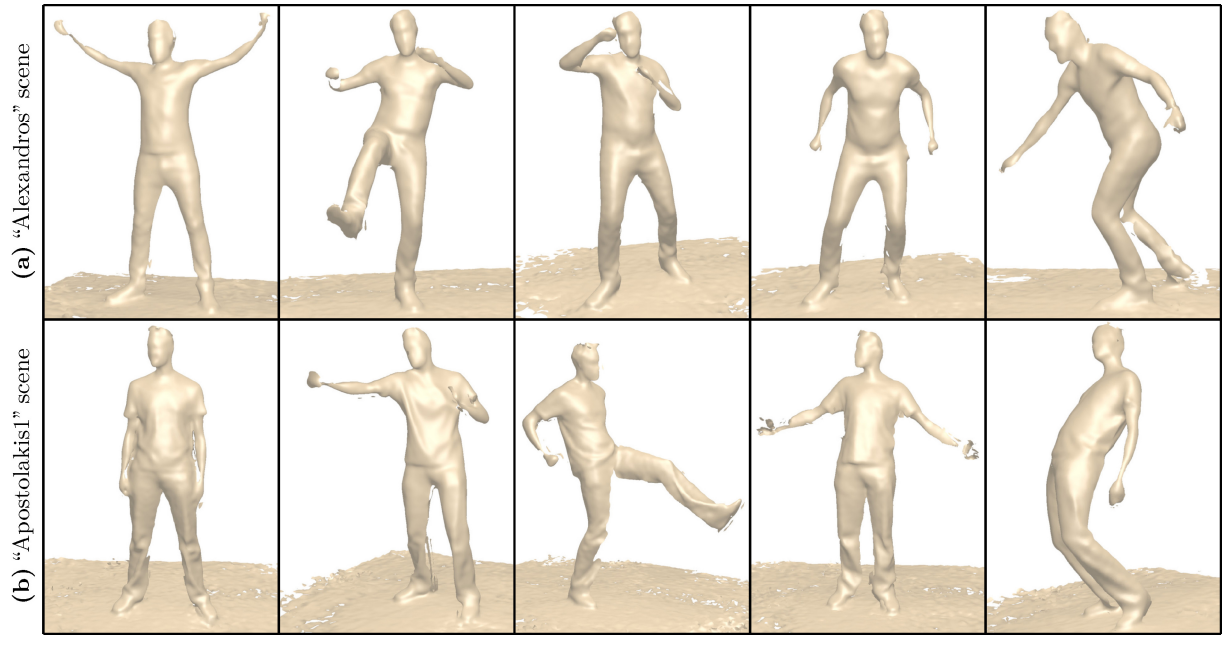


Fig. 10 Reconstruction of human actors in various poses. Capture data is courtesy of Alexiadis et al. [21]. Above: part of "Alexandros" scene. Below: part of "Apostolakis1" scene. In general, the models are correctly reconstructed. However, some smaller details are missing due to lack of visibility from cameras.

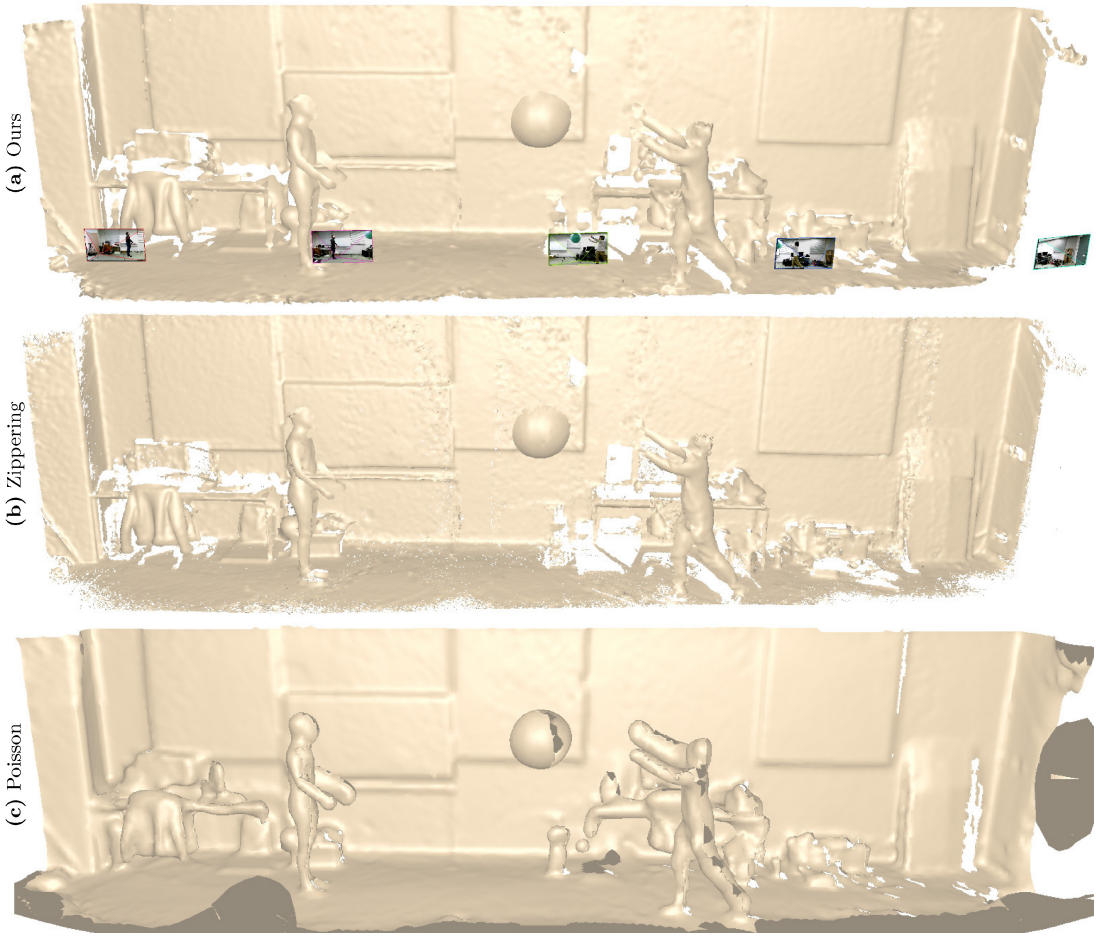


Fig. 11 Reconstructing a fast ball throw in a large scene with five RGB-D cameras: (a) shows our method together with camera positions, (b) uses MLS-based mesh zippering [24], and (c) shows Poisson reconstruction method [19].

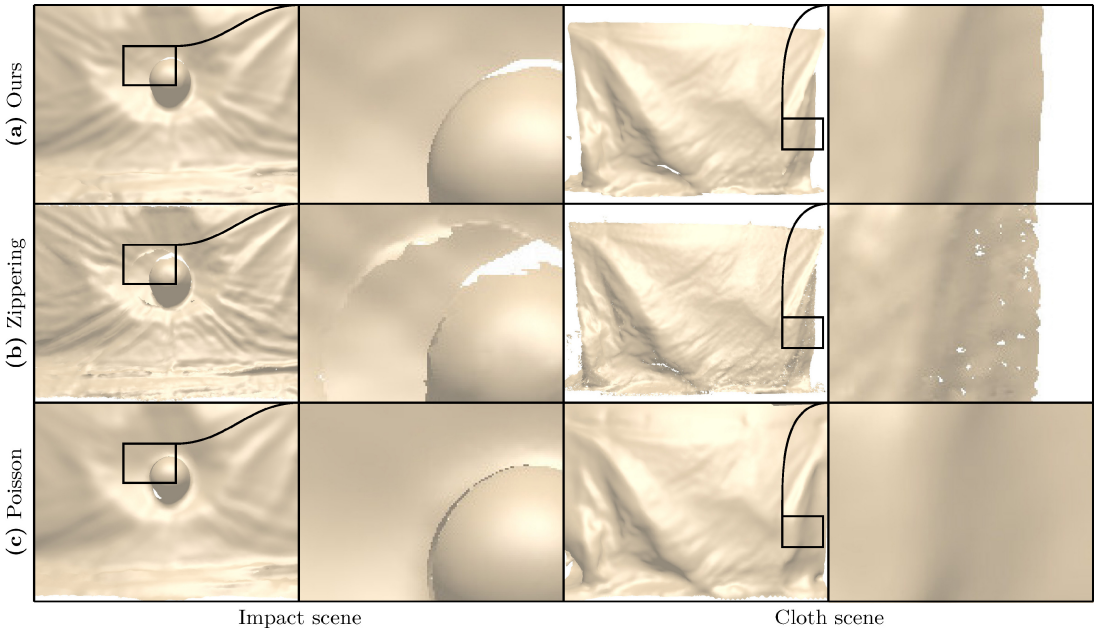


Fig. 12 Comparison of 3D reconstruction methods: (a) shows our method, (b) uses MLS-based mesh zippering [24], and (c) shows the Poisson reconstruction method [19]. Zippering shows bad edge quality and occasionally has incorrect triangles protruding from surfaces. The Poisson method tends to over-smooth areas and incorrectly handles open scenes.

8 Conclusions

In this paper we demonstrated a novel 3D reconstruction system that can reconstruct highly dynamic, large scenes in real time. We solve the problem of combining data from multiple consumer RGB-D cameras lacking synchronized shutters. Our reconstruction method is designed to be memory efficient and provides real-time performance.

The proposed method may have uses in teleconferencing, virtual reality, or free-viewpoint television applications. It allows the use of consumer RGB-D devices for scene capture rather than requiring custom-made camera rigs. Finally, the synthetic slow-motion playback could be useful in performance-capture applications.

Electronic Supplementary Material Supplementary material demonstrating our method in both highly dynamic and large scenes is available in the online version of this article at `https://doi.org/10.1007/s41095-018-0121-0`.

References

[1] De Reu, J.; Plets, G.; Verhoeven, G.; Smedt, P. D.; Bats, M.; Cherretté, B.; Maeyer, W. D.; Deconynck, J.; Herremans, D.; Laloo, P.; Meirvenne, M. V.; Clercq, W. D. Towards a three-dimensional cost-effective registration of the archaeological heritage. *Journal of Archaeological Science* Vol. 40, No. 2, 1108–1121, 2013.

[2] Rong, Y.; Zheng, Y.; Shao, T.; Yang, Y.; Zhou, K. An interactive approach for functional prototype recovery from a single RGBD image. *Computational Visual Media* Vol. 2, No. 1, 87–96, 2016.

[3] Chen, K.; Lai, Y.-K. ; Hu, S.-M. 3D indoor scene modeling from RGB-D data: A survey. *Computational Visual Media* Vol. 1, No. 4, 267–278, 2015.

[4] Orts-Escolano, S.; Rhemann, C.; Fanello, S.; Chang, W.; Kowdle, A.; Degtyarev, Y.; Kim, D.; Davidson, P. L.; Khamis, S.; Dou, M.; Tankovich, V.; Loop, C.; Cai, Q.; Chou, P. A.; Mennicken, S.; Valentin, J. ; Pradeep, V.; Wang, S.; Kang, S. B.; Kohli, P.; Lutchyn, Y.; Keskin, C.; Izadi, S. Holoportation: Virtual 3D teleportation in real-time. In: Proceedings of the 29th Annual Symposium on User Interface Software and Technology, 741–754, 2016.

[5] Zollhöfer, M.; Nießner, M.; Izadi, S.; Rehmann, C.; Zach, C.; Fisher, M.; Wu, C.; Fitzgibbon, A.; Loop, C.; Theobalt, C.; Stamminger, M. Real-time non-rigid reconstruction using an RGB-D camera. *ACM Transactions on Graphics* Vol. 33, No. 4, Article No. 156, 2014.

[6] Newcombe, R. A.; Fox, D.; Seitz, S. M. DynamicFusion: Reconstruction and tracking of non-rigid scenes in real-time. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 343–352, 2015.

[7] Innmann, M.; Zollhöfer, M.; Nießner, M.; Theobalt, C.; Stamminger, M. VolumeDeform: Real-time volumetric non-rigid reconstruction. In: *Computer Vision–ECCV 2016. Lecture Notes in Computer Science, Vol. 9912.* Leibe, B.; Matas, J.; Sebe, N.; Welling, M. Eds. Springer Cham, 362–379, 2016.

[8] Yu, T.; Guo, K.; Xu, F.; Dong, Y.; Su, Z.; Zhao, J.; Li, J.; Dai, Q.; Liu, Y. BodyFusion: Real-time capture of human motion and surface geometry using a single depth camera. In: Proceedings of the IEEE International Conference on Computer Vision, 910–919, 2017.

[9] Collet, A.; Chuang, M.; Sweeney, P.; Gillett, D.; Evseev, D.; Calabrese, D.; Hoppe, H.; Kirk, A.; Sullivan, S. High-quality streamable free-viewpoint video. *ACM Transactions on Graphics* Vol. 34, No. 4, Article No. 69, 2015.

[10] Dou, M.; Khamis, S.; Degtyarev, Y.; Davidson, P.; Fanello, S. R.; Kowdle, A.; Escolano, S. O.; Rhemann, C.; Kim, D.; Taylor, J.; Kohli, P.; Tankovich, V.; Izadi, S. Fusion4D: Real-time performance capture of challenging scenes. *ACM Transactions on Graphics* Vol. 35, No. 4, Article No. 114, 2016.

[11] Dou, M.; Davidson, P.; Fanello, S. R.; Khamis, S.; Kowdle, A.; Rhemann, C.; Tankovich, V.; Izadi, S. Motion2fusion: Real-time volumetric performance capture. *ACM Transactions on Graphics* Vol. 36, No. 6, Article No. 246, 2017.

[12] Nießner, M.; Zollhöfer, M.; Izadi, S.; Stamminger, M. Real-time 3D reconstruction at scale using voxel hashing. *ACM Transactions on Graphics* Vol. 32, No. 6, Article No. 169, 2013.

[13] Berger, M.; Tagliasacchi, A.; Seversky, L. M.; Alliez, P.; Guennebaud, G.; Levine, J. A.; Sharf, A.; Silva, C. T. A survey of surface reconstruction from point clouds. *Computer Graphics Forum* Vol. 36, No. 1, 301–329, 2017.

[14] Li, Z.; Ji, Y.; Yang, W.; Ye, J.; Yu, J. Robust 3D human motion reconstruction via dynamic template construction. In: Proceedings of the International Conference on 3D Vision, 496–505, 2017.

[15] Wang, K.; Zhang, G.; Xia, S. Templateless non-rigid reconstruction and motion tracking with a single RGBD camera. *IEEE Transactions on Image Processing* Vol. 26, No. 12, 5966–5979, 2017.

[16] Curless, B.; Levoy, M. A volumetric method for building complex models from range images. In: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, 303–312, 1996.

[17] Guo, K.; Xu, F.; Yu, T.; Liu, X.; Dai, Q.; Liu, Y. Real-time geometry, albedo, and motion reconstruction using a single RGB-D camera. *ACM Transactions on Graphics* Vol. 36, No. 3, Article No. 32, 2017.

[18] Zhang, H.; Xu, F. MixedFusion: Real-time reconstruction of an indoor scene with dynamic objects. *IEEE Transactions on Visualization and Computer Graphics* DOI: 10.1109/TVCG.2017.2786233, 2018.

[19] Kazhdan, M.; Hoppe, H. Screened Poisson surface reconstruction. *ACM Transactions on Graphics* Vol. 32, No. 3, Article No. 29, 2013.

[20] Wang, R.; Wei, L.; Vouga, E.; Huang, Q.; Ceylan, D.; Medioni, G.; Li, H. Capturing dynamic textured surfaces of moving targets. In: *Computer Vision–ECCV 2016*. Leibe, B.; Matas, J.; Sebe, N.; Welling, M. Eds. Springer Cham, 271–288, 2016.

[21] Alexiadis, D. S.; Zioulis, N.; Zarpalas, D.; Daras P. Fast deformable model-based human performance capture and FVV using consumer-grade RGB-D sensors. *Pattern Recognition* Vol. 79, 260–278, 2018.

[22] Alexiadis, D. S.; Chatzitofis, A.; Zioulis, N.; Zoidi, O.; Louizis, G.; Zarpalas, D.; Daras, P. An integrated platform for live 3D human reconstruction and motion capturing. *IEEE Transactions on Circuits and Systems for Video Technology* Vol. 27, No. 4, 798–813, 2017.

[23] Kuster, C.; Bazin, J.-C.; Öztireli, C.; Deng, T.; Martin, T.; Popa, T.; Gross, M. Spatio-temporal geometry fusion for multiple hybrid cameras using moving least squares surfaces. *Computer Graphics Forum* Vol. 33, No. 2, 1–10, 2014.

[24] Meerits, S.; Nozick, V.; Saito, H. Real-time scene reconstruction and triangle mesh generation using multiple RGB-D cameras. *Journal of Real-Time Image Processing* DOI: 10.1007/s11554-017-0736-x, 2017.

[25] Turk, G.; Levoy, M. Zippered polygon meshes from range images. In: Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques, 311–318, 1994.

[26] Yan, Z.; Xiang, X. Scene flow estimation: A survey. *arXiv preprint* arXiv:1612.02590, 2016.

[27] Li, L.; Xiang, S.; Yang, Y.; Yu, L. Multi-camera interference cancellation of time-of-flight (TOF) cameras. In: Proceedings of the IEEE International Conference on Image Processing, 556–560, 2015.

[28] Guennebaud, G.; Gross, M. Algebraic point set surfaces. *ACM Transactions on Graphics* Vol. 26, No. 3, Article No. 23, 2007.

[29] Chen, J.; Bautembach, D.; Izadi, S. Scalable real-time volumetric surface reconstruction. *ACM Transactions on Graphics* Vol. 32, No. 4, Article No. 113, 2013.

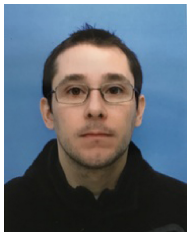
[30] Steinbrücker, F.; Sturm, J.; Cremers, D. Volumetric 3D mapping in real-time on a CPU. In: Proceedings of the IEEE International Conference on Robotics and Automation, 2021–2028, 2014.

[31] Alexa, M.; Adamson, A. On normals and projection operators for surfaces defined by point sets. In: Proceedings of the First Eurographics Conference on Point-Based Graphics, 149–155, 2004.

[32] Lorensen, W. E.; Cline, H. E. Marching cubes: A high resolution 3D surface construction algorithm. *ACM SIGGRAPH Computer Graphics* Vol. 21, No. 4, 163–169, 1987.

[33] Everitt, C. Interactive order-independent transparency. *White paper, nVIDIA* Vol. 2, No. 6, 7, 2001.

Siim Meerits received his B.Sc. degree in physics from Tartu University, Estonia, in 2010. He continued at Keio University, Japan, receiving his M.Sc.Eng. degree in computer science in 2015. Currently he is in the Ph.D. program at the same institution. His research interests include computer vision, particularly 3D reconstruction, and augmented reality.



Diego Thomas received his master degree in informatics and applied mathematics from the Ecole Nationale Superieure d'Informatique et de Mathematiques Appliquees de Grenoble, France, in 2008. He received his Ph.D. degree from the Graduate University for Advanced Studies in 2012. After two years as a JSPS postdoc at Kyushu University, he is now assistant professor at the Laboratory for Image and Media Understanding at Kyushu University, Fukuoka, Japan. His research interests include 3D image registration, 3D reconstruction, and photometric analysis.

Vincent Nozick received his Ph.D. degree in computer sciences in 2006 from Université Paris-Est Marne-la-Vallée, France. In 2006, he was laureate of a Lavoisier fellowship for a post-doc position in the laboratory of Prof. Hideo Saito, Keio University. Since 2008, he has been a tenured "maître de conférences" (assistance/associate professor) at Université Paris-Est Marne-la-Vallée, France. He served as a headmaster of the Imac Engineering School from 2011 to 2013. He held a

"délégation CNRS" position from 2016 to 2018 at the Japanese French Laboratory for Informatics (JFLI), at Keio University, NII and The University of Tokyo, Japan. In addition to computer vision applications, his research interests include digital image forensics and geometric algebra.

Hideo Saito received his Ph.D. degree in electrical engineering from Keio University, Japan, in 1992. Since then, he has been on the Faculty of Science and Technology, Keio University. From 1997 to 1999, he joined the Virtualized Reality Project in the Robotics Institute at Carnegie Mellon University as a visiting researcher. Since 2006, he has been a full professor in the Department of Information and Computer Science, Keio University. His recent activities for academic conferences include being Program Chair of ACCV2014, General Chair of ISMAR2015, and a Program Chair of ISMAR2016. His research interests include computer vision and pattern recognition and their applications to augmented reality, virtual reality, and human–robotics interaction.