Real-time scene reconstruction and triangle mesh generation using multiple RGB-D cameras

Siim Meerits, Vincent Nozick, Hideo Saito

Real-time scene reconstruction and triangle mesh generation using multiple RGB-D cameras

Siim Meerits, Vincent Nozick & Hideo Saito

Journal of Real-Time Image Processing

ISSN 1861-8200

J Real-Time Image Proc
DOI 10.1007/s11554-017-0736-x

Springer

J Real-Time Image Proc
https://doi.org/10.1007/s11554-017-0736-x

ORIGINAL RESEARCH PAPER

Real-time scene reconstruction and triangle mesh generation using multiple RGB-D cameras

Siim Meerits[1] · **Vincent Nozick**[2,3] · **Hideo Saito**[1]

Received: 3 April 2017 / Accepted: 26 September 2017

Abstract We present a novel 3D reconstruction system that can generate a stable triangle mesh using data from multiple RGB-D sensors in real time for dynamic scenes. The first part of the system uses moving least squares (MLS) point set surfaces to smooth and filter point clouds acquired from RGB-D sensors. The second part of the system generates triangle meshes from point clouds. The whole pipeline is executed on the GPU and is tailored to scale linearly with the size of the input data. Our contributions include changes to the MLS method for improving meshing, a fast triangle mesh generation method and GPU implementations of all parts of the pipeline.

Keywords 3D reconstruction · Meshing · Mesh zippering · RGB-D cameras · GPU

1 Introduction

3D surface reconstruction and meshing methods have been researched for decades in the computer vision and computer graphics fields. Its applications are numerous and have practical uses in fields such as archeology [42], cinematography [45] and robotics [31]. A majority of the works produced so far have focused on static scenes. However, many interesting applications, such as telepresence [13, 37, 44], require 3D reconstruction in a dynamic environment, i.e. in a scene where geometric and colorimetric properties are not constant over time.

For most 3D reconstruction systems, the process can be divided conceptually into three stages:

1. *Data acquisition* Traditionally, acquiring a real-time 3D structure of an environment using stereo or multi-view stereo algorithms has been a challenging and computationally expensive stage. The advent of consumer RGB-D sensors has enabled the 3D reconstruction process to become truly real-time, but RGB-D devices still have their own drawbacks. The generated depth maps tend to be noisy, and the scene coverage is restricted due to the sensor's limited focal length, making the following process stages more difficult to achieve. Many reconstruction systems start with a preprocessing stage to reduce some noise inherent to RGB-D sensors.
2. *Surface reconstruction* The surface reconstruction consolidates available 3D information to a single consistent surface.
3. *Geometry extraction* After a surface has been defined, it should be converted to a geometric representation that is useful for a particular application. Some commonly used formats are point clouds, triangle meshes and depth maps.

Our work involves both surface reconstruction and triangle mesh generation. We enhance a moving least squares (MLS)-based surface reconstruction method to fit our needs. We generate triangle meshes in the geometry

✉ Siim Meerits
meerits@hvrl.ics.keio.ac.jp

Vincent Nozick
vincent.nozick@u-pem.fr

Hideo Saito
saito@hvrl.ics.keio.ac.jp

[1] Department of Information and Computer Science, Keio University, Yokohama, Japan

[2] Institut Gaspard Monge, Université Paris-Est Marne-la-Vallée, Champs-sur-Marne, France

[3] Japanese-French Laboratory for Informatics, Tokyo, Japan

Published online: 18 November 2017

extraction stage, since they are the most commonly used representation in computer graphics and are view independent.

1.1 Related works

We constrain the related literature section to 3D reconstruction methods that can work with range image data from RGB-D sensors and provide explicit surface geometry outputs, such as triangle meshes. As such, view-dependent methods are not discussed.

1.1.1 Visibility methods

Visual hull methods, as introduced in Laurentini [34], reconstruct models using an intersection of object silhouettes from multiple viewpoints. Polyhedral geometry [20, 39] has become a popular representation of hull structure. To speed up the process, Li et al. [36], Duckworth and Roberts [17] developed GPU-accelerated reconstruction methods. The general drawback with those approaches is that they need to extract objects from an image background using silhouettes. In a cluttered and open scene, this is difficult to do. We also may wish to include backgrounds in reconstructions, but this is generally not supported.

Curless and Levoy [15] use a visual hull concept together with range images and define a space carving method. Thanks to the range images, background segmentation becomes a simple task. However, space carving is designed to operate on top of volumetric methods, which means that issues with volumetric methods also apply here. Using multiple depth sensors or multiple scans of scenes can introduce range image misalignments. Zach et al. [52] counters these misalignments by merging scans with a regularization procedure, albeit with even higher memory consumption. When a scene is covered with scans at different scales, fine resolution surface details can be lost. Fuhrmann and Goesele [21] developed a hierarchical volume approach to retain the maximum amount of details. However, the method is designed to combine a very high number of viewpoints for off-line processing; as such, it is unsuited to real-time use.

1.1.2 Volumetric methods

Volumetric reconstruction methods represent 3D data as grids of voxels. Each volume element can contain space occupancy data [12, 14] or samples of continuous functions [15]. After commodity RGB-D sensors became available, the work of Izadi et al. [28] spawned a whole family of 3D reconstruction algorithms based on truncated signed distance function volumes. The strength of these methods is their ability to integrate noisy input data in real time to produce high-quality scene models. However, a major drawback is their high memory consumption. Whelan et al. [50] and Chen et al. [10] propose out-of-core approaches where reconstruction volume is moved around in space to lower system memory requirements. However, these methods cannot capture dynamic scenes. Newcombe et al. [40], Dou et al. [16] and Innmann et al. [27] support changes to the scene by deforming the reconstructed volume. These methods expect accurate object tracking, which can fail under complex or fast movement.

Variational volumetric methods, such as those of Kazhdan et al. [30] and Zach [51], reconstruct surfaces by solving an optimization task under specified constraints. Recently, the method of Kazhdan and Hoppe [29] has become a popular choice with Collet et al. [13] further developing it for use in real-time 3D reconstruction, albeit utilizing an incredible amount of computational power. Indeed, the global nature of the optimization comes with a great computational cost, making it infeasible in most situations with consumer-grade hardware.

1.1.3 Point-based methods

MLS methods have a long history in data science as a tool for smoothing noisy data. Alexa et al. [2] used this concept in computer visualization to define point set surfaces (PSS). These surfaces are implicitly defined and allow points to be refined by reprojecting to them. Since then, a wide variety of methods based on PSS have appeared—see Cheng et al. [11] for a partial summary. In the classical formulation, Levin [35] and Alexa et al. [3] approximate local surfaces around a point as a low-degree polynomial. Alexa and Adamson [1] simplify the approach by formulating a signed distance field of the scene from oriented normals. To increase result stability, Guennebaud and Gross [22] formulate higher-order surface approximation while Fleishman et al. [19] and Wang et al. [49] add detail-preserving MLS methods. Kuster et al. [33] introduce temporally stable MLS for use in dynamic scenes.

Most works to date utilize splatting [53] for visualizing MLS point clouds. While fast, this approach cannot handle texturing without blurring, so it is not as well supported in computer graphics as traditional triangle meshes are. It has been considered difficult to generate meshes on top of MLS processed point clouds. Regarding MLS, Berger et al. [7] note that "it is nontrivial to explicitly construct a continuous representation, for instance an implicit function or a triangle mesh". Scheidegger et al. [46] and Schreiner et al. [47] propose advancing front methods to generate triangles on the basis of MLS point clouds. These methods can achieve good results, but they come with high computational costs and are hard to parallelize. Plüss et al. [44]

directly generate triangle meshes on top of refined points. However, the result generates multiple disjointed meshes and does not deal with mesh stability.

1.1.4 Triangulation methods

Point clouds can be meshed directly using Delaunay triangulation and its variations [9]. These approaches are subject to noise and uneven distances between points. Amenta and Bern [5] and Amenta et al. [6] propose robust variants with the drawback of being slow to compute.

Maimone and Fuchs [37] create triangle meshes for multiple cameras separately by connecting neighboring range image pixels to form triangles. The meshes are not merged, but first rendered. Then, the images are merged. Alexiadis et al. [4] take the idea further and merge triangle meshes before rendering. While those methods can achieve high frame rates, the output quality could be improved.

1.2 Proposed system

Direct triangle mesh generation [24, 25, 32, 43] from point clouds has been popular in 3D reconstruction systems [4, 37]. Its strengths are its exceedingly fast operation and simplicity. However, it can be problematic to use when the input point cloud is not smooth or when there is more than one range image to merge. This paper tackles both problems by providing point cloud smoothing and direct mesh generation processes that can work together on a GPU.

Algorithm 1 High-level algorithm of the system

1: **for** every RGB-D camera i **do**
2: $\quad \boldsymbol{p}^i_{\text{raw}} = \text{GetPointsFromCamera}(i)$ ▷ Sec. 2
3: $\quad \boldsymbol{n}^i_{\text{raw}} = \text{NormalEstimation}(\boldsymbol{p}^i_{\text{raw}})$ ▷ Sec. 3
4: **end for**
5: $\boldsymbol{p}_{\text{MLS}}, \boldsymbol{n}_{\text{MLS}} = \text{SurfaceReconstruction}(\boldsymbol{p}_{\text{raw}}, \boldsymbol{n}_{\text{raw}})$ ▷ Sec. 4
6: **for** every RGB-D camera i **do**
7: $\quad M^i_{\text{I}} = \text{InitialMeshGeneration}(\boldsymbol{p}_{\text{MLS}})$ ▷ Sec. 5.1
8: $\quad M^i_{\text{E}} = \text{MeshErosion}(M^i_{\text{I}})$ ▷ Sec. 5.2
9: $\quad M^i_{\text{M}} = \text{MeshMerging}(M^i_{\text{E}})$ ▷ Sec. 5.3
10: **end for**
11: $M_{\text{F}} = \text{FinalMeshGeneration}(M_{\text{M}})$ ▷ Sec. 5.4
12: $\text{Rendering}(M_{\text{F}})$

MLS methods constitute an efficient way to smooth point clouds. Their memory requirements are dependent on the number of input points and not on a reconstruction volume. Additionally, the process is fully parallelizable, making it an excellent target for GPU acceleration [23, 33].

The issue here is that direct meshing methods expect point clouds to have a regular grid-like structure when points are projected to camera. Unfortunately, the traditional MLS smoothing process loses that structure. Therefore, we need to change part of the MLS method to make the output conform to meshing requirements.

Direct meshing of the input range images results in separate meshes for each range image. To reduce rendering costs and obtain a watertight surface, the meshes should be merged. Mesh zippering [38, 48] is a well-known method of doing that. However, this method is not particularly GPU-friendly. As such, we develop our own approach inspired by mesh zippering.

Figure 1 outlines the major parts of our system. Point normals are calculated separately for each RGB-D device range image (Sect. 3). We use an MLS-based method to jointly reduce the noise of all input point clouds and to reconcile data from different RGB-D cameras (Sect. 4). It also provides temporal stability. Next, meshing the point clouds is performed by multiple steps: (Sect. 5) generating a triangular mesh for each camera separately (Sect. 5.1), removing duplicate mesh areas (Sect. 5.2), merging meshes (Sect. 5.3) and finally outputting a single refined mesh (Sect. 5.4). The result is sent to rendering or to further processing as per the targeted application. This whole procedure is also summarized in Algorithm 1.

In summary, our contributions in this work are changing the MLS projection process to make it suitable for meshing and providing a new method of merging multiple triangle meshes. The whole system is designed with parallelization in mind and runs on commodity GPUs.

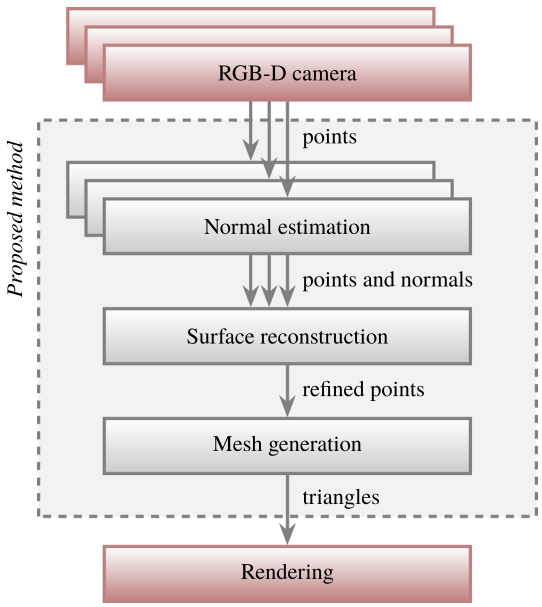

Fig. 1 System overview and data flow. The dashed gray box marks the proposed method

2 System setup

Using our system makes sense in cases where there are at least two RGB-D devices (see Fig. 2). All cameras generate a noisy and incomplete depth map of their surroundings.

Our 3D reconstruction method expects camera intrinsic and extrinsic calibration parameters to be known at all times. The cameras are allowed to move in scene as long as their position is known in a global coordinate system. In cases where calibration is inaccurate, an iterative closest point method [8] was deemed sufficient to align point clouds to the required precision.

We consider the scene to be fully dynamic, i.e. all objects can move freely. As output, we require a single triangle mesh that is temporally stable.

3 Normal estimation

Our pipeline starts with initial surface normals estimation at every input point location, much like previous work does [41, 43]. This is a prerequisite for the MLS process. Normals are calculated separately for each depth map. In a nutshell, we look at a gradient of points in a local neighborhood to get a normal estimate. More precisely, we calculate gradients for every range image coordinate (x, y).

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

and

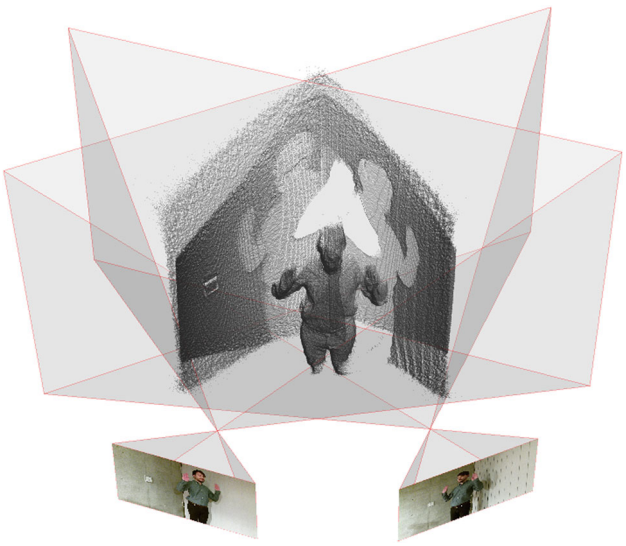

Fig. 2 Example of a system setup with two RGB-D cameras. The shaded pyramids with red edges show the camera's field of view and range

$$g_y(x, y) = p(x, y + 1) - p(x, y - 1), \tag{2}$$

where $p(x, y)$ is the range image pixel's 3D point location in a global coordinate system, g_x is the horizontal gradient, and g_y is the vertical gradient.

In practice, some gradient calculations may include points with invalid data from an RGB-D sensor (i.e. a hole in the depth map). In that case, the gradients g_x and g_y are marked as invalid. Another issue is that the points used in gradient calculation might be part of different surfaces. We detect such situations by checking whether the distance between points is more than a constant value d. In these cases, both the g_x and g_y gradients are marked as invalid for the particular coordinate.

Next, the unnormalized normal in global coordinates is calculated as a cross-product,

$$u(x, y) = \sum_{i,j} g_x(i, j) \times \sum_{i,j} g_y(i, j)\,, \tag{3}$$

where the sums are taken over a local neighborhood of points around (x, y). Any gradients marked invalid should be excluded from the sums. Finally, we normalize $u(x, y)$ so that

$$n(x, y) = \frac{u(x, y)}{||u(x, y)||}\,, \tag{4}$$

which is the surface normal result.

The neighborhood area in the sum of Eq. 3 is typically very small, e.g., 3×3. This area is insufficient for computing high-quality normals. However, in a later surface reconstruction phase of our pipeline, we use weighted averaging of the normals. This is done over a much larger support area, e.g., 9×9, which results in good normals.

Figure 3 shows a comparison of principal component analysis (PCA)-based normal estimation [26] and our selected gradient method. A similar estimation radius was

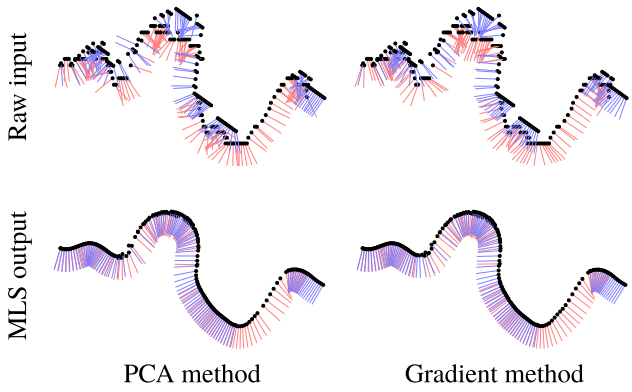

Fig. 3 Comparison of normal calculation methods on real data. Red and blue lines denote normals of points from different RGB-D cameras. Note that the gradient method gives similar results compared to PCA, but with much faster computation

selected for both methods. While there are differences in the initial normals, the final result after MLS smoothing is practically identical. It thus makes sense to use the faster gradient-based normal estimation.

4 Moving least squares surface reconstruction

MLS methods are designed to smooth point clouds to reduce noise that may have been introduced by RGB-D sensors. They require a point cloud with normals as input and produce a new point cloud with refined points and normals.

We choose to follow a group of MLS methods that approximates surfaces around a point $\boldsymbol{x}$ in space as an implicit function $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ representing the algebraic distance from a 3D point to the surface. This method is an iterative process consisting of two main components: an estimation of the implicit function f (and its gradient if necessary) and an optimization method to project points to the implicit surface defined by f. We first use a well-established method to estimate an implicit surface from the point cloud and then project the same points to this surface with our own projection approach.

4.1 Surface estimation

Following Alexa et al. [3], we compute the average point location a and normal n at point $\boldsymbol{x}$ as

$$a(\boldsymbol{x}) = \frac{\sum_i w\big(||\boldsymbol{x} - \boldsymbol{x_i}||\big)\boldsymbol{x_i}}{\sum_i w\big(||\boldsymbol{x} - \boldsymbol{x_i}||\big)} \tag{5}$$

and

$$n(\boldsymbol{x}) = \frac{\sum_i w\big(||\boldsymbol{x} - \boldsymbol{x_i}||\big)\boldsymbol{n_i}}{\sum_i w\big(||\boldsymbol{x} - \boldsymbol{x_i}||\big)}, \tag{6}$$

where $w(r)$ is a spatial weighting function and $\boldsymbol{n_i}$ are the normals calculated in Sect. 3. As in Guennebaud and Gross [22], we use a fast Gaussian function approximation for the weight function, defined as

$$w(r) = \left(1 - \left(\frac{r}{h}\right)^2\right)^4, \tag{7}$$

where h is a constant smoothing factor. Finally, the implicit surface function is obtained as

$$f(\boldsymbol{x}) = n(\boldsymbol{x})^T(\boldsymbol{x} - a(\boldsymbol{x})). \tag{8}$$

The sums in Eqs. 5–6 are taken over all points in vicinity of $\boldsymbol{x}$. Due to the cutoff range of the weighting function $w(r)$, considering points in the radius of h around $\boldsymbol{x}$ is sufficient. Traditionally, points $\boldsymbol{x_i}$ and normals $\boldsymbol{n_i}$ are stored in spatial data structures such as k-d tree or octree. While fast, the spatial lookups still constitute the biggest impact on MLS performance. Kuster et al. [33] propose storing points and normals data as two-dimensional arrays similar to range images. In that case, a lookup operation would consist of projecting search points to every camera and retrieving an $s \times s$ block of points around projected coordinates, where s is known as window size. This allows for very fast lookups and is cache friendly.

To achieve temporal stability, we follow Kuster et al. [33] who propose extending $\boldsymbol{x}$ to a 4-dimensional vector that contains not only spatial coordinates but also a time value. Every frame received from a camera has a timestamp that is assigned to the fourth coordinate of all points in the frame. Hence, it is now possible to measure the spatial and temporal distance of any two points. This allows us to use multiple consecutive depth frames in a single MLS calculation. The weighting function $w(r)$ guarantees that points from newer frames have more impact on reconstruction, while older frames have less. In our system, the number of frames used is a fixed parameter f_{num} and is selected experimentally. Also note that the time value should be scaled to achieve desired temporal smoothing.

4.2 Projecting to surface

Alexa and Adamson [1] present multiple ways of projecting points to the implicit surface. One core concept of this work is that the implicit function $f(\boldsymbol{x})$ can be understood as a distance from an approximate surface tangent frame defined by point $a(\boldsymbol{x})$ and normal $n(\boldsymbol{x})$. This means that we can project a point $\boldsymbol{x}$ to this tangent frame along the normal vector $\boldsymbol{n}$ using

$$\boldsymbol{x}' = \boldsymbol{x} - f\boldsymbol{n}. \tag{9}$$

We call this the *simple projection*. Since the tangent frame is only approximate, the procedure needs to be repeated. On each iteration, the surface tangent frame estimation becomes more accurate as a consequence of the spatial weighting function $w(r)$. Another option is to propagate points along the $f(\boldsymbol{x})$ gradient. This is called *orthogonal projection*.

Instead of following normal vectors or an $f(\boldsymbol{x})$ gradient to the surface, we constrain the iterative optimization to a line between the initial point location and the camera's viewpoint. Given a point $\boldsymbol{x}$ to be projected to a surface and camera viewpoint $\boldsymbol{v}$, the projection will follow vector $\boldsymbol{d}$ defined as

$$\boldsymbol{d} = \frac{\boldsymbol{v} - \boldsymbol{x}}{||\boldsymbol{v} - \boldsymbol{x}||}. \tag{10}$$

Our novel *viewpoint projection* operator projects a point to the tangent frame in direction $\boldsymbol{d}$ instead of $\boldsymbol{n}$ as in Eq. 9.

With the use of some trigonometry, the projection operator becomes

$$\boldsymbol{x}' = \boldsymbol{x} - \frac{f\boldsymbol{d}}{\boldsymbol{n}^T\boldsymbol{d}}. \tag{11}$$

Algorithm 2 Point to surface projection

$\boldsymbol{d} = (\boldsymbol{v} - \boldsymbol{x}) / \|\boldsymbol{v} - \boldsymbol{x}\|$
$i = 0$
repeat
  $\boldsymbol{a} = a(\boldsymbol{x})$
  $\boldsymbol{n} = n(\boldsymbol{x})$
  $f = \boldsymbol{n}^T(\boldsymbol{x} - \boldsymbol{a})$
  if $f > 0$ **then**
    $f = \min\left(h, f/(\boldsymbol{n} \cdot \boldsymbol{d})\right)$
  else
    $f = \max\left(-h, f/(\boldsymbol{n} \cdot \boldsymbol{d})\right)$
  end if
  $\boldsymbol{x} = \boldsymbol{x} - f\boldsymbol{d}$
  $i = i + 1$
until $f < \epsilon$ or $i > i_{max}$

This operator works similarly to simple projection when $\boldsymbol{d}$ is close to $\boldsymbol{n}$ in value. However, this optimization cannot easily converge when $\boldsymbol{n}$ and $\boldsymbol{d}$ are close to a right angle. Conceptually, the closest surface is in a direction where we do not allow the point to move. Dividing by $\boldsymbol{n}^T\boldsymbol{d}$ may propel the point extremely far, well beyond the local area captured by the implicit function f. We thus limit each projection step to distance h (also used in the spatial weighting function 7). This results in a search through space to find the closest acceptable surface.

If a point does not converge to a surface after a fixed number of iterations $i_{\max}$, we consider the projection to have failed and the point is discarded. This is a desired behavior and indicates that a particular point is not required. The pseudocode for the projection method is listed in Algorithm 2.

Figure 4 shows the visualization of different projection methods. Figure 5 shows them in action (except for orthogonal projection, which is computationally more expensive). Our method results in a more regular grid of points on a surface than the normals-based simple projection. This process is crucial in making the final mesh temporally stable. Moreover, the stability of the distances between points is a key condition to compute the mesh connectivity in the next section.

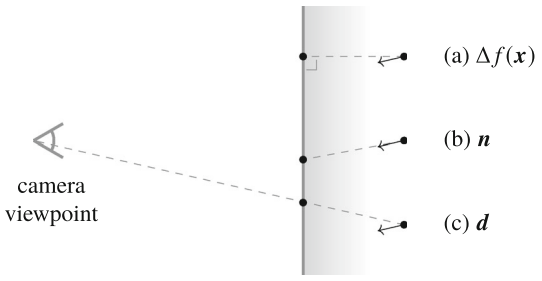

Fig. 4 Visualization of different surface projection methods. a Orthogonal projection, b simple projection, c viewpoint projection (ours)

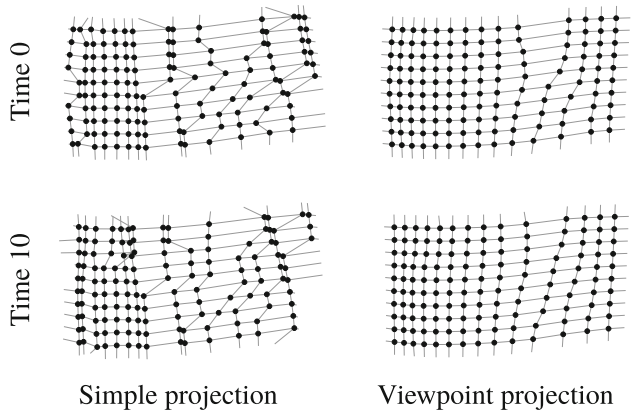

Fig. 5 Comparison of simple projection (left) and our viewpoint projection (right) for the same depth map patch. The latter shows excellent temporal stability

5 Mesh generation

The purpose of mesh generation is to take refined points produced by MLS and turn them into a single consistent mesh of triangles. The approach is to first generate initial triangle meshes for every RGB-D camera separately and then join those meshes to get a final result.

Our proposed method is inspired by a mesh zippering method pioneered by Turk and Levoy [48]. This method was further developed by Marras et al. [38] to enhance output quality and remove some edge-case meshing errors. Both zippering methods accept initial triangle meshes as input and produce a single consistent mesh as output. Conceptually, they work in three phases:

1. *Erosion* remove triangles from meshes so that overlapped mesh areas are minimized.
2. *Clipping* in areas where two meshes meet and slightly overlap, clip triangles of one mesh against triangles of another mesh so that overlapping is completely eliminated.
3. *Cleaning* retriangulate areas where different meshes connect to increase mesh quality.

Prior zippering work did not consider the parallelization of these processes. As such, we need to modify the approach to be suitable for GPU execution.

The mesh erosion process of zippering utilizes a global list of triangles. The main operation in this phase is deleting triangles. If parallelized, the triangle list would need to be locked during deletions to avoid data corruption.

For this reason, we need to introduce a new data structure where triangles are not deleted, only updated to reflect a new state. This allows for completely lockless processing on GPUs. A similar issue arises with mesh clipping, as it would require locking access to multiple triangles to carry out clipping. To counter this, we replace mesh clipping with a process we call mesh merging. It updates only one triangle at a time and thus does not requiring locking. The last step of our mesh generation process is to turn our custom data structures back into a traditional triangle list for rendering or other processing. We call this final mesh generation. This step also assumes the use of mesh cleaning as seen in previous works. Our mesh generation consists of following steps:

1. *Initial mesh generation* creates a separate triangle mesh for every RGB-D camera depth map.
2. *Erosion* detects areas where two or more meshes overlap (but do not delete triangles like in zippering).
3. *Merging* locates points where meshes are joined.
4. *Final mesh generation* extracts a single merged mesh.

The next sections discuss each of these points in detail.

5.1 Initial mesh generation

The first step of our meshing process is to generate a triangle mesh for each depth map separately. In practice, we join neighboring pixels in the depth map to form the triangle mesh. The idea was proposed in Hilton et al. [24] and has widespread uses. We follow Holz and Behnke [25] to generate triangles *adaptively*.

Triangles can be formed inside a *cell* which is made out of four neighboring depth map points (henceforth called vertices). A cell at depth map coordinates (x, y) consists of vertices v_{00} at (x, y), v_{10} at $(x+1, y)$, v_{01} at $(x, y+1)$ and v_{11} at $(x+1, y+1)$. Edges are formed between vertices as follows: e_u between v_{00} and v_{10}, e_r between v_{10} and v_{11}, e_b between v_{01} and v_{11}, e_l between v_{00} and v_{01}, e_z between v_{10} and v_{01}, and e_x between v_{00} and v_{11}. An edge is valid only if both its vertices are valid and their distance is below a constant value d. The maximum edge length restriction acts as a simple mesh segmentation method, e.g., to ensure that two objects at different depths are not connected by a mesh.

Connected loops made out of edges form triangle faces. A cell can have six different triangle formulations as illustrated in Fig. 6. For example, the type 1 form is made out of edges e_u, e_z, e_l. However, ambiguity can arise when all possible cell edges are valid. In this situation, we select either type 3 or type 6 depending on whether edge e_x or e_z is shorter. Since the triangles for a single cell can be stored in just one byte, this representation is highly compact.

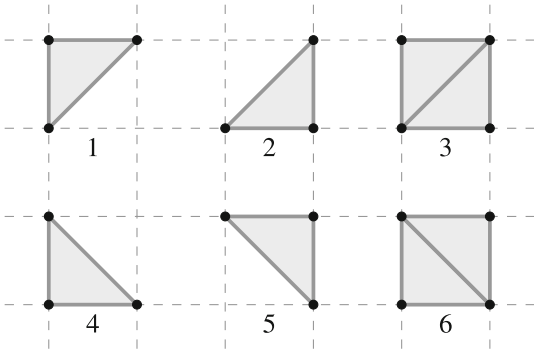

Fig. 6 Forming triangles adaptively between vertices. Each number indicates the triangle formation type. Type 0, which represents an empty cell, is not shown

5.2 Erosion

The initially generated meshes often cover the same surface area twice or more due to the overlap of RGB-D camera views. Mesh erosion detects redundant triangles in those areas; more specifically, erosion labels all initial meshes to *visible* and *shadow* mesh parts. This labeling is based on the principle that overlapping areas should only contain one mesh that is marked visible. The remaining meshes are categorized as shadow meshes. In the previous mesh zippering methods, redundant triangles were simply deleted or clipped. In our method, we keep those triangles in shadow meshes for later use in the mesh merging step.

To segment a mesh into visible and shaded parts, we start from the basic building block of a mesh: a vertex. All vertices are categorized as visible or as a shadow by projecting them onto other meshes. Next, if an initial mesh edge consists of at least one shadow vertex, the edge is considered a shadow edge. Finally, if a triangle face has a shadow edge, it is a shadow triangle.

Note that if we were to project each vertex onto every other mesh, we would end up only with shadow meshes and no visible meshes at overlap areas. Therefore, one mesh should remain visible. For this purpose, we project vertices only to meshes with lower indices. For example, a vertex in mesh i will only be projected to mesh j if $i > j$.

Algorithm 3 Erosion process

```
for every vertex v in meshes i ∈ {1, 2, ..., n} do        ▷ parallelized
    Initially label v as visible vertex
    for every mesh j ∈ {1, 2, ..., i − 1} do
        p = ProjectVertexToMeshSurface(v, j)
        if IsPointInsideTriangle(p, j) then
            Label v as shadow vertex
        end if
    end for
end for
```

Algorithm 3 sums up the erosion algorithm. The ProjectVertexToMeshSurface $(\boldsymbol{v}, j)$ function projects a vertex $\boldsymbol{v}$ to mesh j surface. This is possible because initial meshes are stored as 2D arrays in camera image coordinates. As such, the function simply projects the vertex to a corresponding camera image plane. IsPointInsideTriangle(p, j) checks whether coordinate p falls inside any triangle of mesh j. Figure 7 shows an example of labeling a mesh into visible and shaded parts. There are visible gaps between the two meshes, but this issue will be rectified in the next meshing stages.

5.3 Mesh merging

The task of mesh merging is to find transitions from one mesh to another. For simplicity, consider that meshes A and B were to merge. If mesh A has a shadow mesh that extends over mesh B, we have transition from A to B. Such a situation can be seen on the left side of Fig. 8, where A would be the red mesh and B the blue mesh. In terms of notation, visible vertices are marked with V and shadow vertices are marked with S, e.g., v_S^A denotes the A mesh's shadow vertex.

We begin merging by going through all shadow vertices v_S^A. If we find a vertex that is joined by an edge to a visible vertex v_V^A, then this edge covers a transition area between two meshes. Such edges are depicted as dashed lines on the left side of Fig. 8.

Having located the correct shadow vertices v_S^A, our next task is to merge them with the v_V^B vertices so that the two meshes are connected. The end result of this is illustrated on the right side of Fig. 8. A more primitive approach of locating the nearest v_V^B to v_S^A would not work well, since the closest vertices v_V^B are not necessarily on the mesh boundary. Instead, we trace an edge from v_V^A to v_S^A until we hit the first B mesh triangle. The closest triangle vertex, v_V^B to v_V^A, will be selected as a match. Since meshes are stored as two-dimensional arrays, we can use a simple drawing algorithm, such as a digital differential analyzer, to trace from v_V^A to v_S^A.

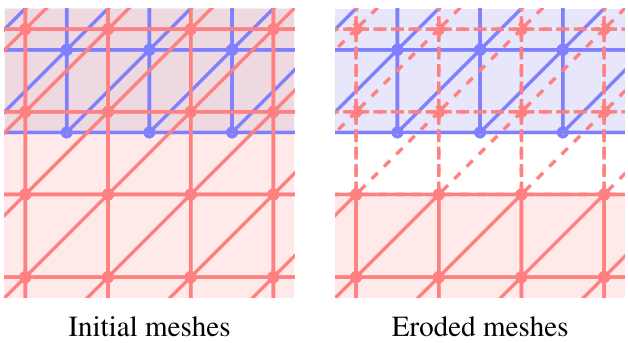

Fig. 7 Illustration of mesh erosion. Initially, two meshes (one red and one blue) overlap. After erosion, the red mesh in the overlap area becomes a shadow mesh, denoted by dashed lines

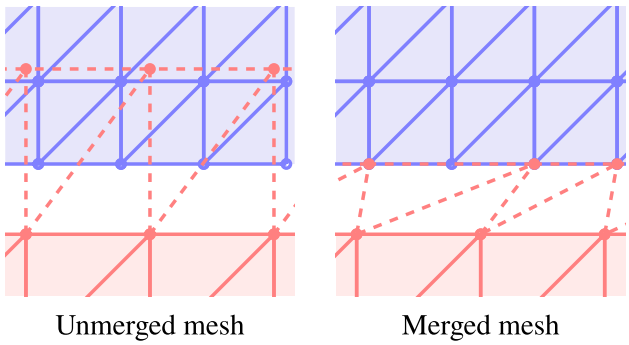

Fig. 8 Illustration of mesh merging. Shadow mesh points are projected to other mesh (left) and then traced to the closest triangles for merging (right)

After the mesh merging procedure, we found edges connecting the two meshes. However, triangles have yet to be generated. This is addressed in the next and final mesh generation section.

5.4 Final mesh generation

The last part of our meshing method collects all data from previous stages and outputs a single properly connected mesh. Handling triangles made out of visible vertices is simple, since they can simply be copied to output. However, transitions from one mesh to another require an extra processing step.

For simplicity's sake, we will once again examine two meshes, A and B, using notation introduced in Sect. 5.3. All the triangles in transition areas consist of one or two shadow vertices v_S^A, with the rest being visible vertices v_V^A. Triangles with just one shadow vertex can be copied to the final mesh without modifications. Triangles with two shadow vertices, however, are a special case. The problem lies in connecting the two consecutive v_S^A vertices with an edge. This situation is illustrated on the left side of Fig. 9. Specifically, the red mesh A's top shadow edge does not coincide with mesh B's edges. Therefore, we create a polygon that traces through B's mesh vertices v_V^B. To reiterate, the vertices of the polygon will be starting point v_V^A, the first shadow vertex v_S^A, mesh B's vertices v_V^B, the second shadow vertex v_S^A and the starting point v_V^A. This polygon is broken up into triangles, as illustrated on the right side of Fig. 9. Note that the polygon is not necessarily convex, but in practice, nonconvex polygons tend to be rare and may be ignored for performance gains if the application permits small meshing errors.

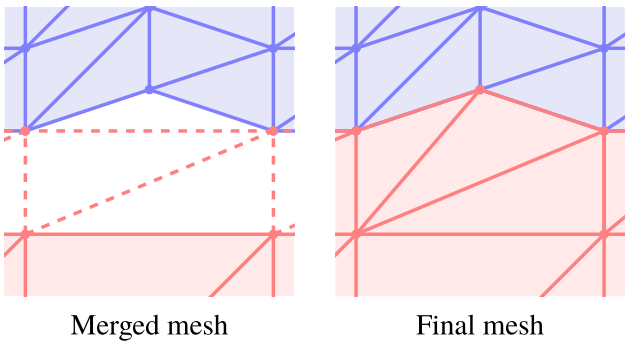

Fig. 9 Illustration of final mesh generation

This concludes the stages of mesh generation. The results can be used in rendering or for any other application.

6 Implementation and results

Our system has been implemented on a platform with GPUs running OpenGL 4.5. All experiments were carried out on a consumer PC with an Intel Core i7-5930 K 3.5 GHz processor, 64 GB of RAM and an Nvidia GeForce GTX 780 graphics card. We used OpenGL compute shaders for executing code. As all of our point cloud and mesh data are organized as two-dimensional arrays, we utilized OpenGL textures for storage.

Table 1 gives an overview of the time spent on different system processes. The measurements were taken with OpenGL query timers to get precise GPU time info. The experiment used two RGB-D cameras, both producing 640 $\times$ 480 resolution depth maps, resulting in up to 614k points per frame. We ran the test with parameters given in Table 2.

An overwhelming majority of processing time is spent on surface reconstruction. This is due to fetching a large number of points and normals from GPU memory. Nevertheless, as the data are retrieved in $s \times s$ square blocks from textures, the GPU cache is well utilized. We also implemented surface reconstruction on the CPU for comparison reasons. The execution has been parallelized across 6 processor cores using OpenMP. The average runtime was 1.6 s per frame on the test dataset. This means that using a GPU gives us roughly 10$\times$ the performance benefit over a CPU.

Our mesh generation method boasts better performance than competing mesh zippering-based methods [38, 48]. We generate two initial meshes in Sect. 5.1 for the test dataset and compare the process in Sects. 5.2–5.4 with two previous methods. The Turk and Levoy [48] implementation takes 48 s, while the Marras et al. [38] implementation takes over 9 minutes of execution time. These methods were originally designed for off-line use on static scenes and thus focused on mesh quality rather than execution speed. These implementations are single-threaded CPU processes and cannot be easily parallelized due to algorithmic constraints outlined in Sect. 5.

Another major reason for the timing differences in zippering [38, 48] is the speed of point lookups. Previous methods are more general and can accept arbitrary meshes as input; they use tree structures such as k-d tree for indexing mesh vertices. Our method arranges meshes similarly to RGB-D camera depth maps. This allows for spatial point lookups by projecting a point to the camera image plane. This is much faster than traversing a tree.

Figure 10 shows a comparison of meshes using different MLS projection methods. The simple viewpoint projection method produces very noisy meshes, and no grid-like regularity is observed. Our viewpoint projection method, however, can organize points to a grid-like structure, making the result much higher quality.

Figure 11 shows comparisons with previous mesh zippering research. Due to differences in the erosion process, the merger areas of meshes may end up in radically different places depending on method used. Therefore, we applied our erosion method to force mesh mergers to appear in the same places for comparison. Turk and Levoy [48] produce meshes with similar quality to ours. Marras et al. [38] reference implementation tends to produce a high number of triangles in merger areas regardless of configuration parameters. An issue with this method is that its intended use is to fill holes in a mesh using another

Table 1 System performance

Process	Avg. time (ms)	Max. time (ms)
Normal estimation	4	6
Surface reconstruction	157	159
Mesh generation	2	2
Initial mesh	0.33	0.34
Erosion	0.41	0.42
Merging	0.18	0.19
Final mesh	0.77	0.79
Total	163	167

Table 2 System parameters and recommended values

Parameter	Explanation
$h = 3$ cm	MLS spatial smoothing factor
$s = 9$	MLS window size
$i_{\max} = 3$	MLS maximum number of iterations
$f_{\text{num}} = 4$	MLS number of camera frames used
$d = 3$ cm	Maximum allowed triangle edge length

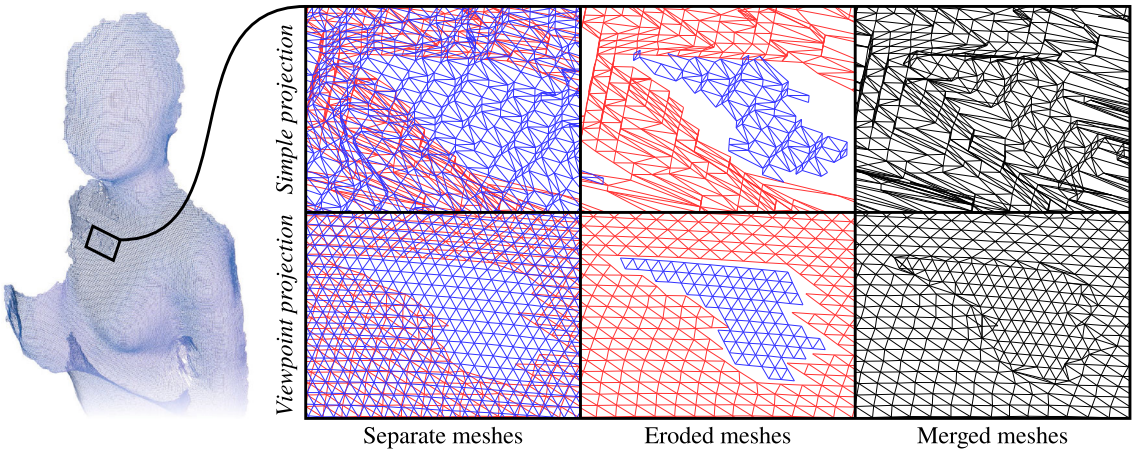

Fig. 10 A mesh merging example. Our viewpoint projection method (lower row) can produce much higher-quality meshes than simple projection (upper row). The columns from left to right show the merging process stages for two meshes (red and blue)

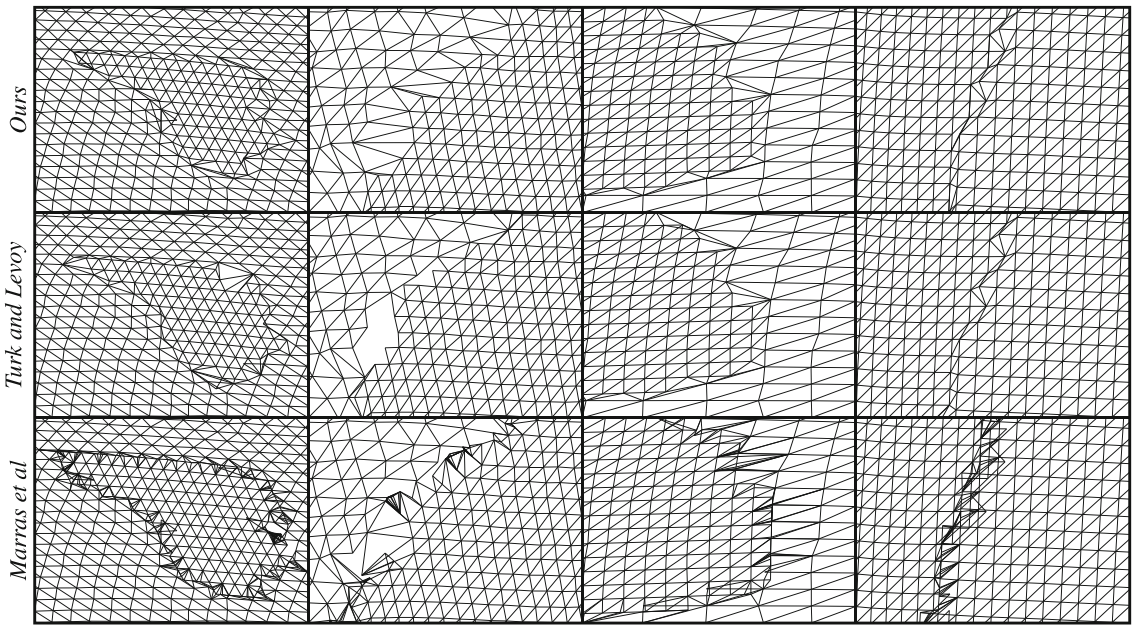

Fig. 11 A comparison of mesh zippering methods at various randomly chosen locations in a real scene. Turk and Levoy [48] show comparable meshing quality to our method, but is CPU-based method and has limited speed. Marras et al. [38] implementation produces excessive amounts of triangles in merger regions

mesh. Our scene is open which does not satisfy this requirement.

Figure 12 shows our method compared to two popular 3D reconstruction methods: volumetric TSDF-based reconstruction [28] and Poisson reconstruction [29]. TSDF method was not designed for dynamic scenes; for this reason, we reconstruct the scene separately on each frame. This results in a mesh that is not completely stable in time. The Poisson method, however, can fill in missing areas of the scene. This might be a good feature if the holes in the model are small, but would cause problems in our case. Large filled-in areas would be inaccurate and tend to flicker.

While there are plenty of RGB-D camera datasets available [18], almost none of them use multiple sensors simultaneously in a dynamic scene. Thus, we used both a dataset created by Kuster et al. [33] and our own recorded data. Figure 12 shows them in use. Scene A, courtesy of Kuster et al. [33], uses Asus Xtion cameras based on structured light technology. Scene B, created by us, uses Microsoft Kinect 2 cameras based on time-of-flight technology. While the type of noise and distortion differs

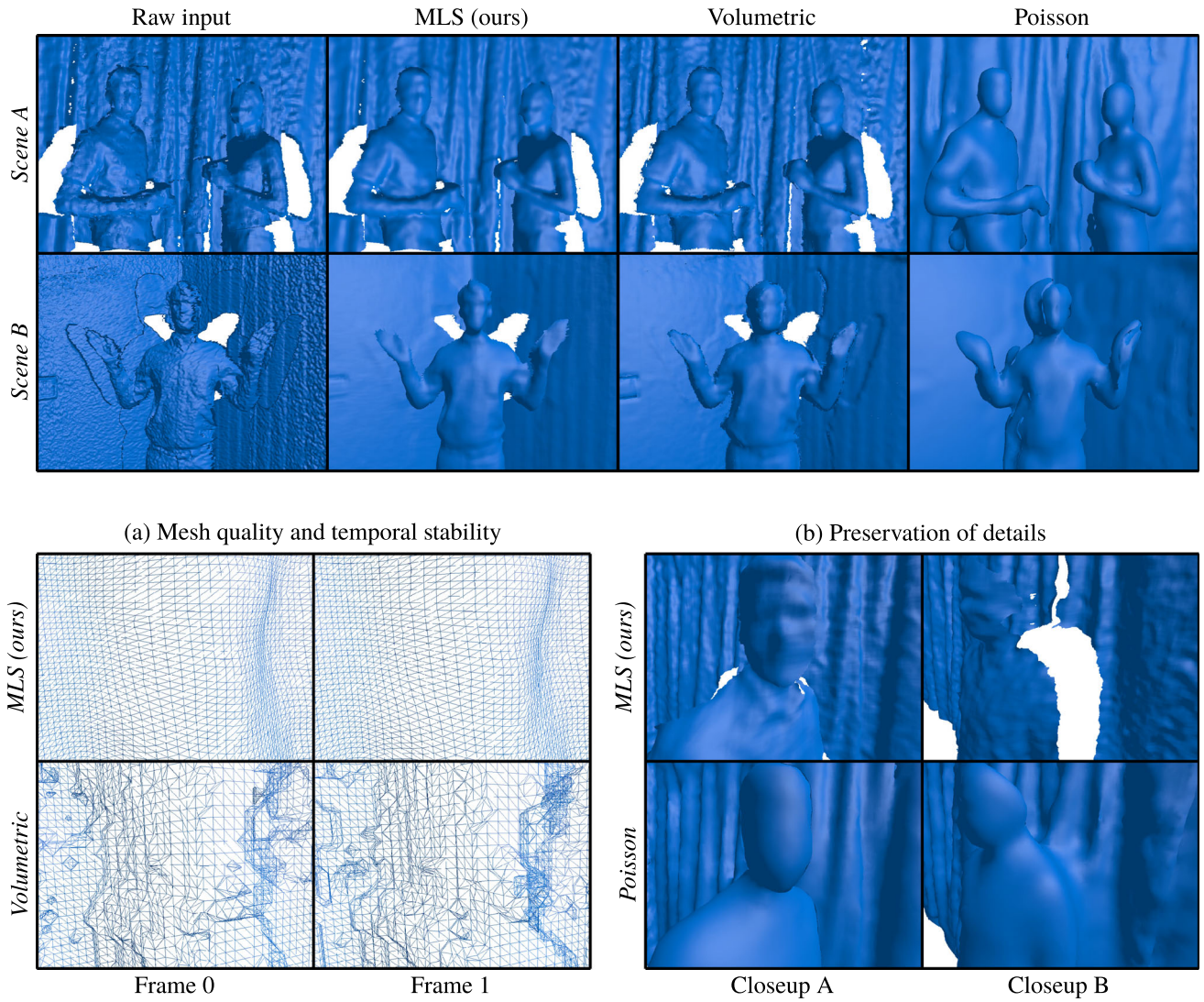

Fig. 12 A comparison with popular 3D reconstruction methods using marching cubes triangularization: a truncated surface distance function (TSDF)-based volumetric reconstruction [28] and Poisson reconstruction [30]. **a** Our method produces higher-quality meshes with better stability than marching cubes based triangularization. **b** Poisson reconstruction tends to oversmooth when using fast processing settings (reconstruction depth <8) and incorrectly generate surfaces in occluded areas. Scene A is courtesy of Kuster et al. [33], whereas scene B is ours

between devices, the results show that our system is general enough to achieve high-quality meshes in both cases.

7 Conclusion

In this paper, we proposed a real-time 3D reconstruction method that can turn data from RGB-D cameras into consistent triangle meshes. The system has two core elements: an MLS-based method to smooth depth camera data and a mesh zippering-inspired mesh generation and merging method for GPUs.

Based on the results, our viewpoint projection method for MLS greatly assists in generating high-quality meshes. We also show that our proposed multiple mesh merging system can generate consistent meshes and is much faster than state-of-the-art methods. The implemented system is completely GPU-based and designed to scale linearly with input data, making it a promising solution for future large-scale real-time 3D reconstruction methods.

References

1. Alexa, M., Adamson, A.: On normals and projection operators for surfaces defined by point sets. SPBG **4**, 149–155 (2004)
2. Alexa, M., Behr, J., Cohen-Or, D., Fleishman, S., Levin, D., Silva, CT.: Point set surfaces. In: Proceedings of the Conference on Visualization '01, IEEE Computer Society, Washington, DC, USA, VIS '01, pp. 21–28 (2001)
3. Alexa, M., Behr, J., Cohen-Or, D., Fleishman, S., Levin, D., Silva, C.T.: Computing and rendering point set surfaces. IEEE Trans. Vis. Comput. Graph. **9**(1), 3–15 (2003)
4. Alexiadis, D.S., Zarpalas, D., Daras, P.: Real-time, full 3-d reconstruction of moving foreground objects from multiple consumer depth cameras. IEEE Trans. Multimed. **15**(2), 339–358 (2013)
5. Amenta, N., Bern, M.: Surface reconstruction by voronoi filtering. Discrete Comput. Geom. **22**(4), 481–504 (1999)
6. Amenta, N., Choi, S., Kolluri, R.K.: The power crust. In: Proceedings of the sixth ACM symposium on solid modeling and applications, ACM, pp. 249–266 (2001)
7. Berger, M., Tagliasacchi, A., Seversky, L.M., Alliez, P., Levine, J.A., Sharf, A., Silva, C.T.: State of the art in surface reconstruction from point clouds. In: Lefebvre, S., Spagnuolo, M. (eds.) Eurographics 2014—State of the Art Reports, 1(1), pp. 161–185 (2014). https://doi.org/10.2312/egst.20141040
8. Besl, P.J., McKay, N.D.: A method for registration of 3-d shapes. IEEE Trans. Pattern Anal. Mach. Intell. **14**(2), 239–256 (1992). https://doi.org/10.1109/34.121791
9. Cazals, F., Giesen, J.: Delaunay triangulation based surface reconstruction: Ideas and algorithms. In: Effective Computational Geometry for Curves and Surfaces, Springer, pp. 231–273 (2006)
10. Chen, J., Bautembach, D., Izadi, S.: Scalable real-time volumetric surface reconstruction. ACM Trans. Graph. (TOG) **32**(4), 113 (2013)
11. Cheng, Z.Q., Wang, Y.Z., Li, B., Xu, K., Dang, G., Jin, S.Y.: A survey of methods for moving least squares surfaces. In: Hege, H.C., Laidlaw, D., Pajarola, R., Staadt, O. (eds.) Proceedings of the Fifth Eurographics / IEEE VGTC Conference on Point-Based Graphics (SPBG'08). pp. 9–23. Eurographics Association (2008). https://doi.org/10.2312/VG/VG-PBG08/009-023
12. Chien, C., Sim, Y., Aggarwal, J.: Generation of volume/surface octree from range data. In: Computer Society Conference on Computer Vision and Pattern Recognition Proceedings of CVPR'88, IEEE, pp. 254–260 (1988)
13. 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 Trans. Graph. (TOG) **34**(4), 69 (2015)
14. Connolly, C.: Cumulative generation of octree models from range data. In: 1984 IEEE International Conference on Robotics and Automation Proceedings, IEEE, vol. 1, pp. 25–32 (1984)
15. 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, ACM, pp. 303–312 (1996)
16. Dou, M., Khamis, S., Degtyarev, Y., Davidson, P., Fanello, S.R., Kowdle, A., Escolano, S.O., Rhemann, C., Kim, D., Taylor, J., et al.: Fusion4d: Real-time performance capture of challenging scenes. ACM Trans. Gr. (TOG) **35**(4), 114 (2016)
17. Duckworth, T., Roberts, D.J.: Parallel processing for real-time 3d reconstruction from video streams. J. Real Time Image Process. **9**(3), 427–445 (2014). https://doi.org/10.1007/s11554-012-0306-1
18. Firman, M.: RGBD datasets: past, present and future. In: CVPR Workshop on Large Scale 3D Data: Acquisition, Modelling and Analysis (2016)
19. Fleishman, S., Cohen-Or, D., Silva, C.T.: Robust moving least-squares fitting with sharp features. ACM Trans. Graph **24**(3), 544–552 (2005)
20. Franco, J.S., Boyer, E.: Exact polyhedral visual hulls. In: British Machine Vision Conference (BMVC'03), Norwich, United Kingdom, vol. 1, pp. 329–338 (2003)
21. Fuhrmann, S., Goesele, M.: Fusion of depth maps with multiple scales. In: Proceedings of the 2011 SIGGRAPH Asia Conference, ACM, New York, NY, USA, SA '11, pp. 148:1–148:8 (2011). https://doi.org/10.1145/2024156.2024182
22. Guennebaud, G., Gross, M.: Algebraic point set surfaces. ACM Trans. Graph **26**(3) (2007). https://doi.org/10.1145/1276377.1276406
23. Guennebaud, G., Germann, M., Gross, M.: Dynamic sampling and rendering of algebraic point set surfaces. Comput. Gr. Forum **27**(2), 653–662 (2008). https://doi.org/10.1111/j.1467-8659.2008.01163.x
24. Hilton, A., Stoddart, A.J., Illingworth, J., Windeatt, T.: Reliable Surface Reconstruction from Multiple Range Images, pp. 117–126. Springer, Berlin (1996). https://doi.org/10.1007/BFb0015528
25. Holz, D., Behnke, S.: Fast range image segmentation and smoothing using approximate surface reconstruction and region growing. In: Lee, S., Cho, H., Yoon, KJ., Lee, J. (eds.) Intelligent Autonomous Systems 12. Advances in Intelligent Systems and Computing, vol. 194, pp. 61–73. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-33932-5_7
26. Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., Stuetzle, W.: Surface reconstruction from unorganized points. SIGGRAPH Comput. Gr. **26**(2), 71–78 (1992). https://doi.org/10.1145/142920.134011
27. Innmann, M., Zollhöfer, M., Nießner, M., Theobalt, C., Stamminger, M.: VolumeDeform: Real Time Volumetric Non-rigid Reconstruction, pp. 362–379. Springer, Berlin (2016). https://doi.org/10.1007/978-3-319-46484-822
28. Izadi, S., Kim, D., Hilliges, O., Molyneaux, D., Newcombe, R., Kohli, P., Shotton, J., Hodges, S., Freeman, D., Davison, A, et al.: Kinectfusion: real-time 3d reconstruction and interaction using a moving depth camera. In: Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology, ACM, pp. 559–568 (2011)
29. Kazhdan, M., Hoppe, H.: Screened poisson surface reconstruction. ACM Trans. Gr. (TOG) **32**(3), 29 (2013)
30. Kazhdan, M., Bolitho, M., Hoppe, H.: Poisson surface reconstruction. In: Sheffer, A., Polthier, K. (eds.) Symposium on Geometry Processing, The Eurographics Association (2006). https://doi.org/10.2312/SGP/SGP06/061-070
31. Kostavelis, I., Gasteratos, A.: Semantic mapping for mobile robotics tasks. Robot Auton. Syst. **66**(C), 86–103 (2015). https://doi.org/10.1016/j.robot.2014.12.006
32. Kriegel, S., Rink, C., Bodenmüller, T., Suppa, M.: Efficient next-best-scan planning for autonomous 3d surface reconstruction of unknown objects. J. Real Time Image Process. **10**(4), 611–631 (2015). https://doi.org/10.1007/s11554-013-0386-6
33. 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. Comput. Gr. Forum **33**(2), 1–10 (2014). https://doi.org/10.1111/cgf.12285
34. Laurentini, A.: The visual hull concept for silhouette-based image understanding. IEEE Trans. Pattern Anal. Mach. Intell. **16**(2), 150–162 (1994). https://doi.org/10.1109/34.273735
35. Levin, D.: Mesh-independent surface interpolation. In: Brunnett, G., Hamann, B., Müller, H., Linsen, L. (eds.) Geometric Modeling for Scientific Visualization. Mathematics and Visualization, pp. 37–49. Springer, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-662-07443-5_3

36. Li, M., Schirmacher, H., Magnor, M., Siedel, HP.: Combining stereo and visual hull information for on-line reconstruction and rendering of dynamic scenes. In: 2002 IEEE Workshop on Multimedia Signal Processing, pp. 9–12, (2002). https://doi.org/10.1109/MMSP.2002.1203235
37. Maimone, A., Fuchs, H.: Encumbrance-free telepresence system with real-time 3d capture and display using commodity depth cameras. In: 2011 10th IEEE International Symposium on Mixed and Augmented Reality (ISMAR), IEEE, pp. 137–146 (2011)
38. Marras, S., Ganovelli, F., Cignoni, P., Scateni, R., Scopigno, R.: Controlled and adaptive mesh zippering. In: Proceedings of the International Conference on Computer Graphics Theory and Applications, pp. 104–109 (2010)
39. Matusik, W., Buehler, C., McMillan, L.: Polyhedral Visual Hulls for Real-Time Rendering, pp. 115–125. Springer, Vienna (2001). https://doi.org/10.1007/978-3-7091-6242-2-11
40. Newcombe, R.A., Fox, D., Seitz, SM.: Dynamicfusion: Reconstruction and tracking of non-rigid scenes in real-time. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 343–352 (2015)
41. Olesen, S.M., Lyder, S., Kraft, D., Krüger, N., Jessen, J.B.: Real-time extraction of surface patches with associated uncertainties by means of kinect cameras. J. Real Time Image Process. **10**(1), 105–118 (2015). https://doi.org/10.1007/s11554-012-0261-x
42. Oliveira, A., Oliveira, J.F., Pereira, J.M., de Araújo, B.R., Boavida, J.: 3d modelling of laser scanned and photogrammetric data for digital documentation: the mosteiro da batalha case study. J. Real Time Image Process. **9**(4), 673–688 (2014). https://doi.org/10.1007/s11554-012-0242-0
43. Orts-Escolano, S., Morell, V., Garcia-Rodriguez, J., Cazorla, M., Fisher, R.B.: Real-time 3d semi-local surface patch extraction using gpgpu. J. Real Time Image Process. **10**(4), 647–666 (2015). https://doi.org/10.1007/s11554-013-0385-7
44. Plüss, C. (Kuster), Ranieri, N., Bazin, J.C., Martin, T., Laffont, P.Y., Popa, T., Gross, M.: An immersive bidirectional system for life-size 3d communication. In: Proceedings of the 29th International Conference on Computer Animation and Social Agents, ACM, New York, NY, USA, CASA '16, pp. 89–96 (2016). https://doi.org/10.1145/2915926.2915931
45. Ronfard, R., Taubin, G.: Image and Geometry Processing for 3-D Cinematography, Geometry and Computing, vol. 5. Springer, Berlin (2010). https://doi.org/10.1007/978-3-642-12392-4
46. Scheidegger, C.E., Fleishman, S., Silva, C.T.: Triangulating point set surfaces with bounded error. In: Proceedings of the Third Eurographics Symposium on Geometry Processing, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, SGP '05 (2005)
47. Schreiner, J., Scheidegger, C.E., Fleishman, S., Silva, C.T.: Direct (Re)meshing for efficient surface processing. Comput. Gr. Forum (2006). https://doi.org/10.1111/j.1467-8659.2006.00972.x
48. Turk, G., Levoy, M.: Zippered polygon meshes from range images. In: Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques, ACM, pp. 311–318 (1994)
49. Wang, J., Yu, Z., Zhu, W., Cao, J.: Feature-preserving surface reconstruction from unoriented, noisy point data. Comput. Gr. Forum **32**(1), 164–176 (2013). https://doi.org/10.1111/cgf.12006
50. Whelan, T., McDonald, J.B., Kaess, M., Fallon, M.F., Johannsson, H., Leonard, J.J.: Kintinuous: Spatially extended KinectFusion. In: RSS Workshop on RGB-D: Advanced Reasoning with Depth Cameras, Sydney, Australia (2012)
51. Zach, C.: Fast and high quality fusion of depth maps. In: Proceedings of the International Symposium on 3D Data Processing, Visualization and Transmission (3DPVT), Citeseer, vol. 1 (2008)
52. Zach, C., Pock, T., Bischof, H.: A globally optimal algorithm for robust tv-l1 range image integration. In: 2007 IEEE 11th International Conference on Computer Vision, pp. 1–8 (2007). https://doi.org/10.1109/ICCV.2007.4408983
53. Zwicker, M., Pfister, H., van Baar, J., Gross, M.: Surface splatting. In: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, ACM, New York, NY, USA, SIGGRAPH '01, pp. 371–378 (2001). https://doi.org/10.1145/383259.383300

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

V. Nozick received his Ph.D. degree in computer sciences in 2006 from Universite Paris-Est Marne-la- Vallee, France. In 2006, he was the laureate of a Lavoisier fellowship for a post doc position at Prof. Hideo Saito laboratory, Keio University. From 2008, he has been hired as a tenured "maitre de conferences" (assistance/associate professor) at Universite Paris-Est Marne-la-Vallee, France. He served as the headmaster of the Imac engineering school from 2011 to 2013. He got a "delegation CNRS" position from 2016 to 2017 at Japanese French Laboratory for Informatics (JFLI), at Tokyo University, NII and Keio University. In addition to computer vision applications, his research interests include both digital image forensics and geometric algebra.

H. 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, 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, a 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.