Large-Scale Point-Cloud Visualization through Localized Textured Surface Reconstruction

Murat Arikan, Reinhold Preiner, Claus Scheiblauer, Stefan Jeschke and Michael Wimmer

Abstract—In this paper, we introduce a novel scene representation for the visualization of large-scale point clouds accompanied by a set of high-resolution photographs. Many real-world applications deal with very densely sampled point-cloud data, which are augmented with photographs that often reveal lighting variations and inaccuracies in registration. Consequently, the high-quality representation of the captured data, i.e., both point clouds and photographs together, is a challenging and time-consuming task. We propose a two-phase approach, in which the first (preprocessing) phase generates multiple overlapping surface patches and handles the problem of seamless texture generation locally for each patch. The second phase stitches these patches at render-time to produce a high-quality visualization of the data. As a result of the proposed localization of the global texturing problem, our algorithm is more than an order of magnitude faster than equivalent mesh-based texturing techniques. Furthermore, since our preprocessing phase requires only a minor fraction of the whole dataset at once, we provide maximum flexibility when dealing with growing datasets.

Index Terms—Image-based rendering, surface representation, color, large-scale models, segmentation

1 Introduction

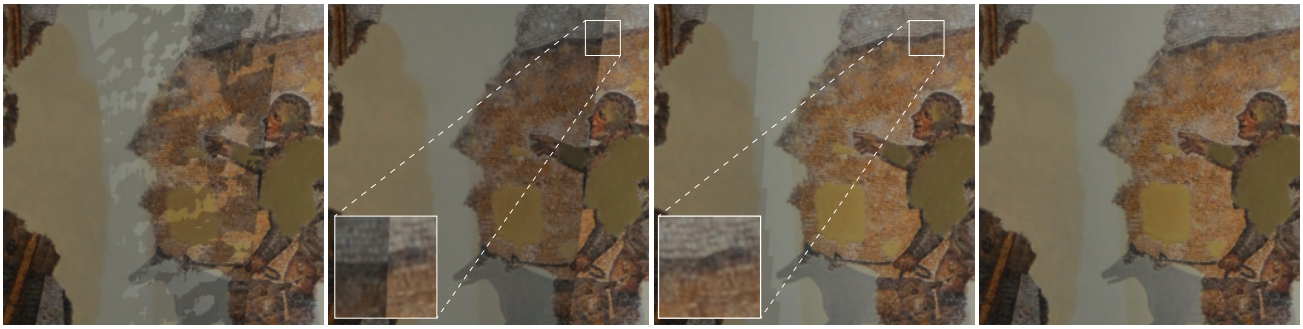
The high-quality visualization of point-cloud data gathered from laser scans or photogrammetric approaches is a fundamental task in many scientific and non-scientific applications, like preservation in cultural heritage, digitalization of museums, documentation of archaeological excavations, virtual reality in archaeology, urban planning, architecture, industrial site management, and many others. An essential component for the quality of the resulting visualization is the use of registered high-resolution images (photographs) taken at the site to represent surface material, paintings etc. These images typically overlap, exhibit varying lighting conditions, and reveal inaccuracies in registration. Consequently, for high-quality visualizations, the individual images have to be *consolidated* to provide a common, homogeneous representation of the scene.

One way to display these data is to directly render point-based surfaces texture-mapped with the images [1], [2]. These methods are flexible but cause visible artifacts, and are therefore not suitable for high-quality visualization requirements (see Section 7).

In the traditional visualization pipeline, on the other hand, a mesh surface is reconstructed from the 3D points and textured by the registered images. Optimization-based methods have been developed to produce a high-resolution texture over the mesh surface while simultaneously minimizing the visibility of seams, typically using graph-cuts [3], [4]. One problem is that such algorithms are global in nature and thus assume that both the mesh and the images fit into main memory. However, many real-world applications deal with large-scale input datasets, which not only pose a problem of scalability for texturing techniques, but for which it is extremely time-consuming to construct a consistent mesh in the first place. Even if a mesh is available, changing or extending the dataset with new data proves nontrivial. In such scenarios, mesh-based techniques would require first an out-of-core meshing of the dataset, second, a robust out-of-core texturing algorithm, and third, the maintenance of the mesh topology and an expensive re-texturing every time the dataset changes. This raises a maintenance overhead, which makes current mesh-based methods unsuitable for certain data complexities and applications.

One important observation about the texturing problem is that it is "semi-global": at each location of the scene, only a small part of the geometric and image data is required to provide a good visualization. In this paper, we therefore propose a new *semi-global scene representation* that abstracts from the deficiencies of both point- and mesh-based techniques: it provides the flexibility and ease of use of point-based data combined with the quality of mesh-based reconstruction. The main idea of our system is to reconstruct many smaller textured surface patches *as seen by the image cameras*. This leads to a collection of patches, one for each image camera, that we stitch together at render-time to produce a high-quality visualization of the data. We therefore avoid the need for the reconstruction and maintenance of the whole surface

- *M. Arikan, R. Preiner, C. Scheiblauer and M. Wimmer are with the Institute of Computer Graphics and Algorithms, Vienna University of Technology, Austria.*
  E-mail: marikan@cg.tuwien.ac.at
- *S. Jeschke is with the Institute of Science and Technology, Austria.*



(a) (b) (c) (d)

Fig. 1. (a) Penetration artifacts caused by rendering two meshes with z-buffering. (b) Any overwriting order of the meshes in the overlap area resolves the penetration artifacts. However, heavy artifacts occur at the transition of the meshes. (c) Our novel image-space stitching solution, and (d) an intensity levelling post-process, assures a seamless surface representation from multiple meshes.

at once, allowing for both an *efficient data representation* and an *easy extension* of the dataset by new images or points if new scans are acquired in a scanning campaign, for example.

The main challenge when working with multiple textured patches is to avoid stitching artifacts and visible seams at their transitions (see Fig. 1 (a) and (b) respectively). For this, we propose as our main technical contribution a novel efficient image-space stitching method that computes a smooth transition between individual patches (Fig. 1 (c) and (d)).

2 Related Work

Point-Based Rendering: To convey the appearance of a closed surface, several methods [5], [6], [7] render splats, i.e., small discs in 3D, instead of simple one-pixel points. Botsch et al. [5] propose a splat-filtering technique by averaging colors of overlapping splats. Instead of using a constant color for each splat, Sibbing et al. [2] extend this splatting approach by blending textures of overlapping splats. Another work [1] assigns several images to each splat and blends between them in a view-dependent manner.

Surface Reconstruction: As an alternative scene representation, a mesh surface can be reconstructed from a point cloud, e.g., using methods such as the Poisson surface reconstruction [8] and its screened variant [9]. However, these methods are not suited to large-scale datasets. Bolitho et al. [10] address the out-of-core reconstruction of surfaces from large point clouds. However, texturing a mesh surface consisting of millions of triangles from a collection of high-resolution images remains a time-consuming and tedious task. In another work, Fuhrmann and Goesele [11] fuse multiple depth maps into an adaptive mesh with coarse as well as highly detailed regions. Turk and Levoy [12] remove redundant border faces of two overlapping patches and glue them together by connecting their pruned borders. Marras et al. [13] take this idea further by allowing to merge meshes with very different granularity.

Texturing: To generate a high-quality texture over a mesh surface from multiple images, several previous works [3], [4] apply a graph-cut based optimization that incorporates certain criteria to select for each surface part a portion of a single source image. These methods reduce the visibility of seams between areas textured by different images.

Another option is to perform a weighted blending of all the images over the whole surface [14], [15], [16]. However, the blending approach produces undesirable ghosting artifacts in the presence of misregistrations.

Our approach builds on the former group and extends these methods to consider several overlapping surface patches.

Optical Flow: Optical-flow techniques [17], [18] have proven useful to correct small inaccuracies introduced in the image-to-geometry registration. We do not explicitly correct misaligned features along seams, although an optical-flow strategy can be integrated as a post-process into our pipeline. The aim of our method is to achieve a smooth transition between surface patches without requiring expensive computation of warp fields in order to produce an accurate color mapping of a set of images onto a 3D model.

3 Problem Analysis

Motivation for Multi-Mesh Approach: We consider the problem of generating a textured surface representation from captured real-world data, given by a point cloud together with a set of registered images $\mathbf{I} = \{I_1, \ldots, I_n\}$. The traditional way is to reconstruct a single mesh M from the point cloud and choose for each triangle $\mathbf{t} \in M$ an image $I(\mathbf{t}) \in \mathbf{I}$ that should be mapped onto it. This image-to-triangle assignment problem considers both local quality criteria (i.e., detail provided by an image), as well as

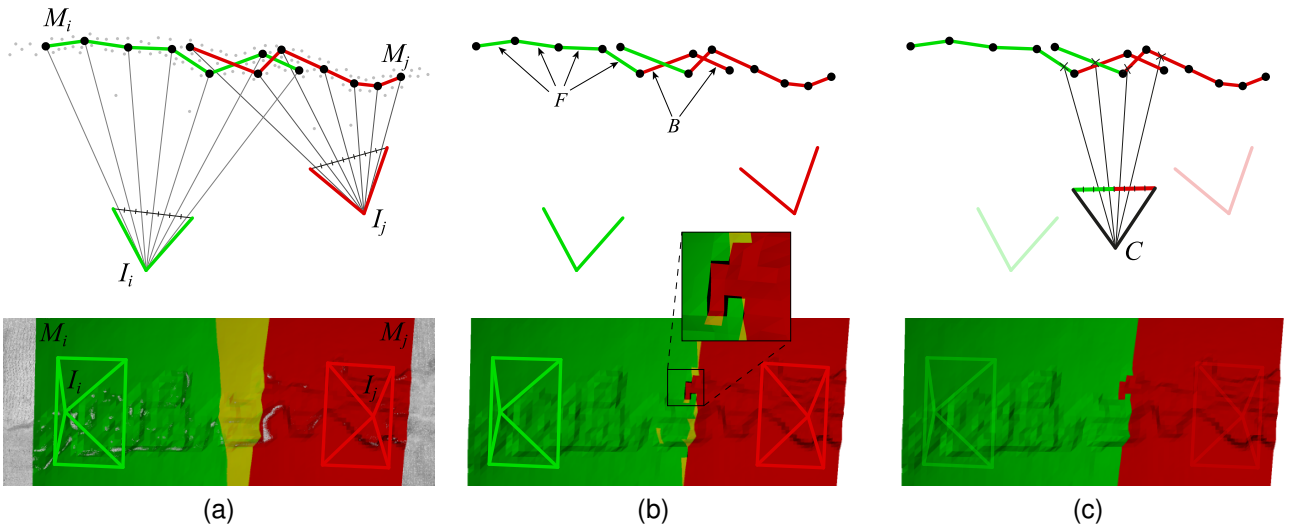


Fig. 2. Overview of our pipeline. (a) Meshes are generated by rendering depth maps from image cameras. The meshes M_i and M_j are color-coded by their respective images and additively blended in the overlap area (yellow). (b) Each of the meshes is textured by all the input images. Besides, each mesh face is equipped with a binary label: *foreground* (*F*), if the face is assigned to its respective image during the texturing, and *background* (*B*) otherwise. As we will show in Section 6.1, it's beneficial to reconstruct the scene by using foregrounds. This provides in the major part of the overlap area a deterministic solution, however, some minor ambiguities remain such as overlaps (yellow) and thin cracks (black) between foreground faces (see inset). (c) In order to resolve these ambiguities, we render entire meshes and decide for each screen pixel, based on faces' binary labels, which of the overlapping fragments to display.

continuity in the mapping (i.e., avoiding visible seams between areas represented by different images), and is commonly solved with a graph-cut based optimization [3], [4]. The mesh is then displayed with each triangle $\mathbf{t}$ textured by $I(\mathbf{t})$. Both the optimization, usually carried out in a preprocess, as well as the visualization are straightforward. However, as discussed above, generating M and solving the optimization problem for large datasets is intractable.

In this paper, we observe that the problem is semi-global, i.e., each part of the sampled surface is only covered by a small number of images. Therefore, instead of solving the mapping problem *globally* for a *single* mesh M, we provide a surface representation consisting of multiple meshes, and solve the mapping problem locally for each mesh. This provides a strong localization of the problem, since each individual mesh represents only a small part of the scene. Unfortunately, while this *multi-mesh* approach makes the mapping problem tractable, it is not straightforward anymore, as we will discuss now.

Setup of the Multi-Mesh Approach: We will use the following setup, illustrated in Fig. 2: from the given point cloud we reconstruct a set of meshes $\mathbf{M} = \{M_1, \ldots, M_n\}$. Each mesh M_i corresponds to an image $I_i \in \mathbf{I}$, in the sense that it represents the scene from the same viewpoint and with the same camera parameters as the image camera (Fig. 2 (a)). However, the triangles of M_i can be textured by any image, not only by I_i. Thus, in the following we examine various ways how to determine the mapping of images to individual triangles of each mesh (Fig. 2 (b)). This concludes the *preprocessing phase* of our algorithm. Furthermore, we also have to deal with the fact that a representation by a collection of meshes is not a unique representation of the surface anymore. In particular, there will be regions where multiple meshes will overlap. As discussed, seamlessly stitching the meshes in the preprocess is intractable. Therefore, we need to send all the visible meshes to the GPU in their entirety, and devise a rendering algorithm that decides for each screen pixel in an overlap region which of the overlapping mesh triangles to display. This constitutes the *visualization phase* of our algorithm (Fig. 2 (c)).

Challenges of the Multi-Mesh Approach: Let us first look at *visualization*: the simplest approach is to display all meshes using the standard rendering pipeline, and resolve any overlaps using z-buffering. However, this leads to heavy rendering artifacts, because the individual meshes exhibit geometric variations (see Fig. 1 (a)).

Even if z-buffering artifacts can be avoided by prescribing an overwriting order of meshes (Fig. 1 (b)), *texturing*, i.e., solving the image-to-triangle assignment problem, is not straightforward. Let us first look at the very simple image assignment, i.e., $I(\mathbf{t}) = I_i$ for all $\mathbf{t} \in M_i$. This has the obvious problem that image

I_i is usually not the best choice for every triangle $\mathbf{t} \in M_i$. This problem could be solved by applying the single-mesh approach to each mesh M_i separately, always using all images $\mathbf{I}$. For each mesh individually, this would provide an optimal solution. However, single-mesh texturing does not take mesh borders into account, so two problematic cases occur, illustrated in Fig. 3 (a) and (b):

First (Fig. 3 (a)), assume M_i is rendered on top of M_j, and at the border of M_i, I_i happens to be the best choice after the optimization of M_i. However, I_i is not defined beyond the border of the overlap region. Therefore, the visible part of mesh M_j is textured with an image $I_j \neq I_i$, which usually leads to color discontinuities (from misalignments and lighting variations) at the mesh border due to camera misregistrations, geometric variations of the two meshes in their overlapping region, and different lighting conditions of the images. These color discontinuities are particularly visible when they go through salient color features (see also Fig. 1 (b)).

Second (Fig. 3 (b)), still assuming M_i is rendered on top of M_j, but I_i is *not* the best choice at the border of M_i. In this case, no color discontinuity due to the use of different images will appear at the mesh border. However, geometric variations of the meshes will still cause visible misalignments if the mesh border passes through a salient color feature.

This shows that the idea of applying the single-mesh approach to each mesh M_i separately is not sufficient for a good visualization. While the optimization can find optimal image seams on each individual mesh, the mesh borders lead to transitions that are not under the control of this optimization. Therefore, it is essential to also adapt the mesh transitions so that visible artifacts along these are minimal.

Making the Multi-Mesh Approach Work: We start by creating individual meshes and solving an image-to-triangle assignment problem on each mesh separately in a *preprocess*. However, in contrast to the simple approach described before, we also take mesh borders into account. Since at mesh borders, multiple meshes are involved, in order to keep the locality of the approach during the preprocessing phase, we shift some of the burden of the optimization to the *visualization* phase. In particular, we determine at runtime optimal transitions for mesh overlap regions.

This is done in the following way for the two cases discussed above: if I_i is dominant at the border of M_i (Fig. 3 (a)), we shift both the image and the mesh transition away from the mesh border, so that the artifacts along these transitions are less apparent (Fig. 3 (c)). If I_i is not assigned to the border of M_i (Fig. 3 (b)), we adjust the mesh transition so that it follows an image seam in the overlap region, since that is where it is least likely to be visible (Fig. 3 (d), and see Fig. 1 (c) for another example). Finally, we also postprocess the overlap regions in order to reduce the remaining intensity differences among the images (Fig. 1 (d)).

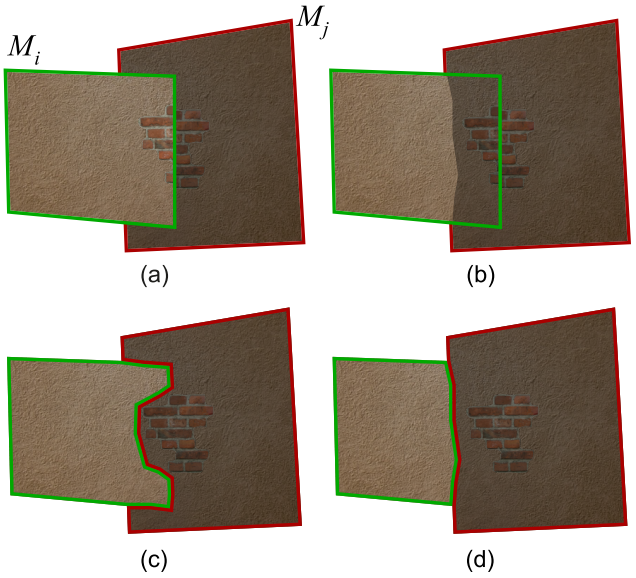


Fig. 3. Illustration of misalignments at mesh borders and our solution strategy. (a) and (b) show two cases of a graph-cut optimization of M_i. (a) I_i (i.e., the corresponding image of M_i) is assigned to the border of M_i. (b) I_i is *not* assigned to the border of M_i. In both cases, rendering M_i on top of M_j leads to misalignments along the mesh border. To avoid the case from (a), our graph-cut optimization (c) excludes I_i from the border of M_i, and our stitching solution (c), (d) pushes the mesh transition towards the image seam, where the transition is not as visible.

4 Algorithm Overview

Preprocess – Textured Multi-Mesh Generation: In a preprocessing phase (Section 5), we reconstruct textured meshes. The first step is to generate the meshes themselves. For each image, we create a depth map by rendering the scene as seen by the image's camera. These depth maps (stored as 2D textures) can be interpreted and rendered as triangular meshes, so-called *depth meshes* [19]. For our purposes, this has the advantage that texturing a mesh with its corresponding image is guaranteed to have no occlusion artifacts, and the accuracy of the representation can be easily tuned through the depth-map resolution.

The second step is to texture the generated meshes while taking into account the problematic overlap regions, in particular mesh borders. Following the reasoning in Section 3, we first carry out a graph-cut optimization (also referred to as *labeling* in the following) with the set of candidate images for each individual mesh. The candidate set consists of *only* the images whose camera can see the mesh. To avoid the case from Fig. 3 (a), we will show in Section 5.2 that excluding image I_i from the border pixels in the

labeling of mesh M_i (Fig. 4) will later push mesh transitions towards image seams where the transition is not as visible.

Visualization – Image-Space Stitching: In the visualization phase (Section 6), the generated meshes are *visually stitched* together to provide a high-quality textured surface representation. Note that we do not perform an actual mesh stitching, but rather resolve conflicting fragments of overlapping meshes on a per-pixel basis at render time. This works by rendering all meshes in the overlap region and choosing an appropriate mesh for each overlap pixel at runtime (Fig. 2 (c)), based on the mesh labelings calculated in the preprocessing phase.

Since the input images can differ arbitrarily in viewing parameters, the generated depth maps may represent the observed surface at different sampling rates. Our approach of visually stitching meshes with varying sampling rates at render-time is closely related to the work of Fuhrmann and Goesele [11]. However, our method also addresses texturing issues, and most importantly, the runtime stitching of the meshes makes the easy handling of large datasets possible.

5 Textured Multi-Mesh Generation

5.1 Depth-Map Generation

For the generation of the depth maps, the input point cloud is converted into splats with normal vectors and spatial extents. Here we can optionally use available input normals, or apply an in-situ-reconstruction at depth-map rendering time [20] if no normals are given. Then these small discs are used as rendering primitives to generate depth maps. For each image camera, we render the scene as seen by this image camera using Gauss-splatting [7]. However, since we deal with large-scale datasets, the input points do not necessarily fit into main or video memory. We therefore use an out-of-core data structure to store the points and stream only those parts seen by an image camera into video memory (our implementation is based on the work of Scheiblauer and Wimmer [21], but any other out-of-core point rendering system can be used). The out-of-core data structure also provides the splat radii, derived from the density of the points at the current detail level [21].

In order to reduce storage and processing costs, the user can choose to create the depth maps at a lower resolution than the input images. For the examples in this paper, we used a resolution of 256×171 for 12 MPixel images (see Table 2). While depth-map generation is rather straightforward, our work focuses on generating a high-quality surface representation from them as described in the following.

5.2 Multi-Mesh Labeling

The second step of the preprocessing phase is the computation of a graph-cut based labeling with the set of candidate images for each depth mesh. Compared to traditional single-mesh labeling approaches, we also take mesh borders and overlap regions into account. We further equip each mesh triangle with a binary value (foreground/background), which is then utilized at runtime by our image-space stitching to determine the overwriting order of meshes on a per-pixel basis (Fig. 2 (c)).

5.2.1 Foreground and Background Segmentation

The optimization step, described shortly, will assign each mesh triangle to an image. We call the set of all triangles of mesh M_i that is assigned to its respective image *foreground* (denoted by F_i), i.e., $\mathbf{t} \in F_i \subseteq M_i$ iff $I(\mathbf{t}) = I_i$. The remaining part that is assigned to an image $I_j \neq I_i$ is called *background* (denoted by B_i). In the rest of this paper, we further use the notation $B_{ik} \subseteq B_i$ to distinguish between background triangles assigned to different images, i.e., $\mathbf{t} \in B_{ik} \subseteq M_i$ iff $I(\mathbf{t}) = I_k$.

5.2.2 Candidate Labels Selection

The set of candidate images (labels) for each mesh triangle is formed by those images whose camera can see the triangle. The visibility of a triangle is determined based on an in-frustum test and an occlusion test (given the computed depth maps) of each individual triangle vertex.

However, reconsider the situation (illustrated in Fig. 3 (a)) where the mesh borders lead to transitions between images that are not under the control of the optimization of individual meshes. In order to account for this problem, we constrain the candidate labels of *outer* mesh borders (i.e., those that are not inner borders due to holes in the input datasets). Our strategy is to exclude the foreground label (i.e., the index of the respective image) from the candidate label set of each outer border triangle. By setting such a hard constraint, these triangles are forced to be labeled as background even if best suited as foreground. Through the shrinking of the potential foreground region, we intentionally insert image seams at mesh borders (Fig. 4 (a)). The smoothness requirement of the labeling, described shortly, will ensure that these seams are shifted away from mesh borders to regions where the transition is not as visible (Fig. 4 (b)).

5.2.3 Optimization Step

For each depth mesh M, we compute a labeling $\mathcal{L}$, i.e., a mapping from mesh triangles to image indices (labels). This labeling has two goals: (1) maximizing back-projection quality (high resolution, low anisotropy, etc.) of images onto triangles, and (2) penalizing the visibility of seams between areas

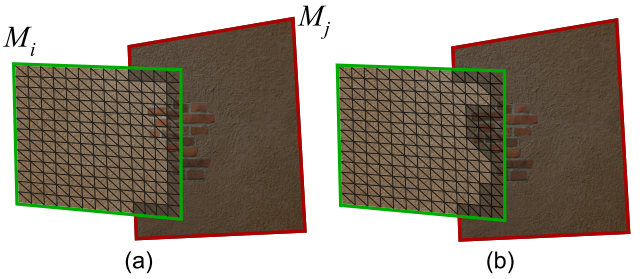


Fig. 4. Handling of mesh borders. (a) Excluding the respective image from the border faces in the labeling of mesh M_i introduces an image seam at the mesh border. The example in (b) illustrates how the image seam is pushed by the labeling towards more homogeneous color regions where artifacts caused due to camera misregistrations are less apparent.

mapped from different images. The labeling problem can typically be formulated in terms of an energy minimization (see Kolmogorov and Zabih [22], and Boykov and Kolmogorov [23] for a comprehensive discussion of vision problems expressed in terms of an energy minimization). The labeling $\mathcal{L}$ is computed as the minimizer of the energy

$$E(\mathcal{L}) = E_d(\mathcal{L}) + \lambda_s E_s(\mathcal{L}), \tag{1}$$

which is composed of a data term E_d and a smoothness term E_s.

Data Term: For each triangle $\mathbf{t}$, the data term should favor the image with the highest "quality". The quality was previously computed by counting the number of texels covered by the projection of a triangle into an image [24]. This was later extended by taking color variation of the covered pixels into account, favoring more detailed images [4]. However, color variation can also be caused by noise in the image, so we prefer a purely geometric term. Furthermore, while the number of covered texels combines both distance from the image and orientation into one term, we found that in the case of depth meshes generated from noisy point clouds, it is beneficial to allow more control over the relative importance of these two factors. In particular, since the selected image also determines the mesh to be rendered, and since meshes generated from orthogonal views have better quality, we usually give more weight to the orientation factor.

We therefore propose the following data term:

$$E_d = \sigma_t\big(\lambda_{dist} E_{dist} + \lambda_{or} E_{or}\big), \tag{2}$$

with the distance factor

$$E_{dist} = \sum_{\mathbf{t}\in M} \frac{1}{h} \min(\|\mathbf{t}_c - \mathbf{c}_{i_t}\|, h), \tag{3}$$

where $\mathbf{t}_c$ is the triangle center and $\mathbf{c}_{i_t}$ the center of projection of image I_{i_t}. The orientation factor

$$E_{or} = \sum_{\mathbf{t}\in M} 1 - \left|\mathbf{n}_t \cdot \frac{\mathbf{t}_c - \mathbf{c}_{i_t}}{\|\mathbf{t}_c - \mathbf{c}_{i_t}\|}\right| \tag{4}$$

uses the triangle normal $\mathbf{n}_t$.

In order to make the data term comparable between different meshes with differently sized triangles (e.g., in overlap areas), it is scaled by the world-space area σ_t of the triangle. The parameter h allows adjusting the relative influence of E_{dist} and E_{or}: the distance term is clamped to a maximum of h and then normalized to one, so that for all images at a distance h or larger, the distance penalty matches the orientation penalty of an image with normal deviation of $\pi/2$ degrees.

Smoothness Term: As in Gal et al. [4], we use the smoothness term

$$E_s = \sum_{(\mathbf{t},\mathbf{t}')\in\mathcal{N}} \int_{\mathbf{e}_{tt'}} \big\|\Phi_{i_t}(x) - \Phi_{i_{t'}}(x)\big\|\, dx, \tag{5}$$

where Φ_i is the projection operator into image I_i, $\mathbf{e}_{tt'}$ is the edge between neighboring faces $\mathbf{t}$ and $\mathbf{t}'$, and the integral is evaluated by regular sampling along the edge. The smoothness term penalizes color differences due to label changes at triangle edges, and therefore biases the method towards solutions where the transitions from one image to another are less visible.

Energy Minimization: For the minimization of the objective function of Eq. 1, we apply an α-expansion graph-cut algorithm [25]. Further, the choice of the parameter λ_s reflects the trade-off between the local quality provided by the images and the color smoothness.

5.2.4 Summary

Through the particular handling of border triangles during the candidate labels selection, the presented optimization routine accounts for optimal image transitions in overlap regions. However, visible artifacts at transitions from one mesh to another remain (e.g., see Fig. 4 (b)). In Section 6, we will show our stitching solution that uses the binary labels to also adapt mesh transitions so that these coincide with image seams.

6 Image-Space Stitching

6.1 Overview

A depth mesh represents the scene from the same viewpoint and with the same viewing parameters as its corresponding image camera. Therefore, the quality of projection of an image onto a surface region on the one hand and the sampling rate of this region by the image's depth mesh on the other hand are inherently coupled. As we have shown in Section 5.2.3, the data term of the graph-cut optimization always favors the image with the highest quality. Thus, for

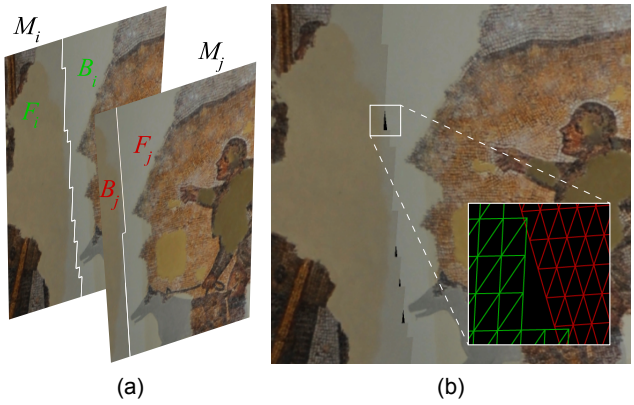


Fig. 5. (a) Two meshes M_i and M_j are binary segmented into foreground and background regions. (b) However, rendering only foreground-labeled regions causes holes due to different mesh discretizations.

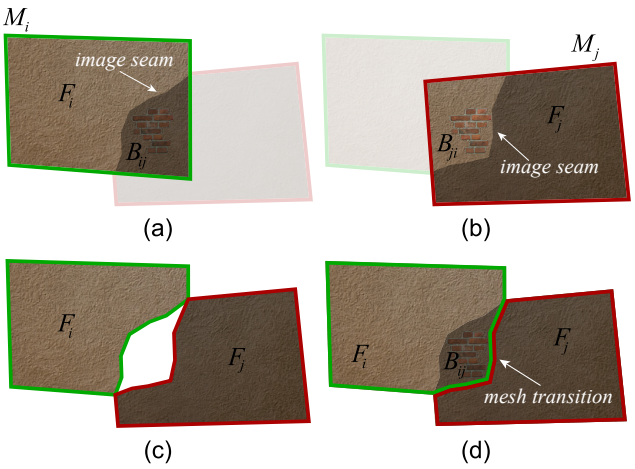


Fig. 6. Illustration of our stitching idea. (a) and (b) show the labelings of the meshes M_i and M_j, respectively. (c) Rendering only foreground regions F_i and F_j does not cover the complete object, due to the different seams in the overlap area. (d) The rendering algorithm resorts to the background-labeled region B_{ij}, where no foreground label is available.

rendering a region of several overlapping meshes, it is preferable to also render the corresponding mesh, i.e., the mesh labeled as foreground, as this may imply an increase of the sampling rate of the overlap area (for an example, compare Fig. 10 (a) and (b), respectively). Note, however, that foreground regions do not always have higher sampling rates, since the optimization has to consider color continuity as well.

Ideally, the label assignment of mesh regions is consistent between meshes in overlap regions, so all foreground regions together could theoretically cover the whole surface. However, in practice this will not work due to two reasons:

1) Triangles of different meshes sharing a spot of the scene typically differ in size and rotation, which makes a "perfect" alignment of foreground borders impossible (see Fig. 5).
2) Since each individual mesh is optimized separately, the local minimization of the objective function of Eq. 1 for different meshes can produce different seams in their overlapping region, as in the example of Fig. 6 (a) and (b).

The union of all resulting foregrounds is thus not guaranteed to cover the whole surface (Fig. 6 (c)). Our stitching approach, therefore, is to reconstruct the scene by preferably drawing foreground regions, while still being able to fall back to background-labeled regions where no foreground label is available (Fig. 6 (d)). In practice, this is implemented by assigning mesh faces a depth value according to their binary label (foreground/background) and employing z-buffering to perform the corresponding overwriting decisions.

6.2 Rendering Pipeline

The rendering pipeline consists of five steps, which are described in the following:

Visibility Pass: In this first pass, all meshes intersecting the view frustum are rendered with z-buffering, writing to a *depth buffer*, a *mesh-index buffer*, and a *triangle-label buffer*. We will use the triangle-label buffer in the image-management step to determine which images are required for texturing the currently visible meshes, and to cache those images to the GPU if necessary. Similarly, the mesh-index buffer contains the indices of all the visible meshes that will be rendered in the stitching pass. Other meshes are considered to be occluded. The depth buffer will be used in the stitching pass to determine which triangles belong to the front surface and need to be resolved.

Image Management: Since all the high-resolution images do not fit into video memory, we employ an out-of-core streaming technique for continuously caching the currently required images into a GPU texture array. Due to the limited size of this texture array, it is our goal to always have the most relevant images available on the GPU. We measure this relevance by the frequency of occurrence of an image's label in the triangle-label buffer. In this step, we therefore compute a label histogram from the triangle-label buffer, based on which we can always cache the currently most relevant images onto the GPU. If the texture array is already full, the least relevant images are dropped to make space for more relevant images. For performance reasons, we currently restrict the number of image loads to only one high-resolution image per frame. In case of the unavailability of images that are required for the texturing, we alternatively use precomputed per-vertex colors.

Stitching Pass: In this pass, the visible meshes are rendered again, and visually stitched together in

image space in order to produce a coherent representation of the observed surface. The visible meshes are determined from the mesh-index buffer during the computation of the label histogram in the image-management step.

We resolve conflicting fragments of overlapping meshes on a per-pixel basis, using the binary labeling of the meshes. One or more triangle fragments are said to be *overlapping* at a pixel position p if their depth values differ by less than ε from the front-most depth value of p, stored in the depth buffer which is bound as a texture. Triangle fragments beyond this threshold are considered to be occluded and discarded. For the overlapping triangle fragments, our stitching chooses the fragment from the best-suited mesh based on their binary labels. To this end, we equip each mesh triangle with a depth according to its binary label (foreground: 0, background: 1). In this way, for each pixel p of an overlapping region, z-buffering automatically picks one mesh whose label at p is foreground, and chooses a background-labeled mesh if no foreground candidate is available. This pass outputs two textures that store positions and labels of the chosen triangle fragments.

Similar to surface splatting [5], [7], visibility is not resolved for features smaller than epsilon. Except for small concave features, the resulting artifacts can be avoided by using back-face culling using the mesh orientations given by the generating camera positions.

Texturing Pass: We render a full-screen quad to retrieve the color of each chosen triangle fragment by projecting it onto its assigned image based on the position and label retrieved from the output textures of the previous stitching pass.

Levelling: The outcome of the texturing pass is a rendering of the current view (see Fig. 7 (a)), which typically reveals visible seams caused by lighting variations among the input images. To minimize the visibility of such seams, we apply as a rendering post-process an adapted version of the seamless cloning method of Jeschke et al. [26], which we briefly illustrate in Fig. 7. The goal of this method is to compute a *levelling texture* (denoted by L) that can be added to the output of the texturing pass to locally alleviate lighting variations. This is done by minimizing the Laplacian $\nabla^2 L$ of a texture, where border pixels of uniformly labeled regions (Fig. 7 (b) and (c)) impose boundary constraints on that optimization problem. Moreover, our levelling is geometry-aware, which means that we prevent levelling over non-connected parts of the surface. This is easily achieved by introducing depth-discontinuity border pixels as additional boundary constraints.

Before applying the levelling texture, we average it with the textures of previous frames in order to avoid flickering during animations. For this, we follow the idea of temporal smoothing presented in Scherzer et al. [27]. We first determine for each currently rendered

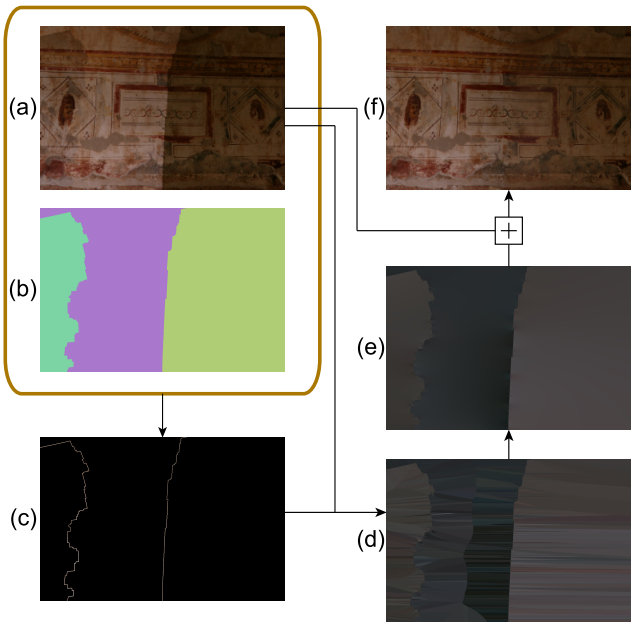


Fig. 7. Levelling pipeline. (a) Output of the texturing pass. (b) Output texture of the stitching pass storing fragment labels. (c) A one-pixel wide border around each connected pixel region of the same label is detected. The color of each border pixel is fixed to the average of its own and that of its direct neighbors on the other side of the border. Additionally, we fix colors of depth-discontinuity border pixels. All these fixed border pixels impose color constraints on the minimization of the Laplacian of the levelling texture. The difference between the fixed and original colors of the closest border pixels is the initial guess (d) of the levelling texture (e). (a) is added to (e) to produce the final result (f) with the locally levelled intensities.

fragment at position (x_c, y_c) its corresponding pixel position (x_p, y_p) in the previous frame. Then, we compute the smoothed levelling value as $l_c(x_c, y_c) = wL(x_c, y_c) + (1-w)l_p(x_p, y_p)$, where l_p stores levelling values of the previous frame. For all new fragments (x_{new}, y_{new}) that are not present or occluded in the previous frame, we use the non-averaged levelling value $l_c(x_{new}, y_{new}) = L(x_{new}, y_{new})$.

7 Results

We have tested our method on four large-scale point datasets acquired by a laser scanner, accompanied by a set of high-resolution images (Table 1, see Fig. 14 for various results and the accompanying video for a walkthrough of the *Hh2 We1* model). In the following, we give a detailed analysis of the performance, the memory consumption, the reconstruction and rendering quality, and the extensibility of our proposed system in comparison to a point-based and a single-mesh approach.

As a comparable point-based method involving texturing, we adapted the work of Sibbing et al. [2]

TABLE 1
Model statistics.

Model	#Points	#Scans	Point cloud memory consumption	$\|\mathbf{I}\|$	Image resolution	Images memory consumption
Hanghaus 2 Wohneinheit 1 *(Hh2 We1)*	682.6M	46	16.4GB	276	4258 × 2832 (36MB)	9.9GB
Hanghaus 2 Wohneinheit 6 *(Hh2 We6)*	34.7M	25	0.8GB	188	4032 × 2674 (32MB)	6GB
Siebenschläfer *(7schläfer)*	907.4M	44	21.8GB	254	4258 × 2832 (36MB)	9.1GB
Centcelles	1091.3M	42	26.2GB	161	4258 × 2832 (36MB)	5.8GB

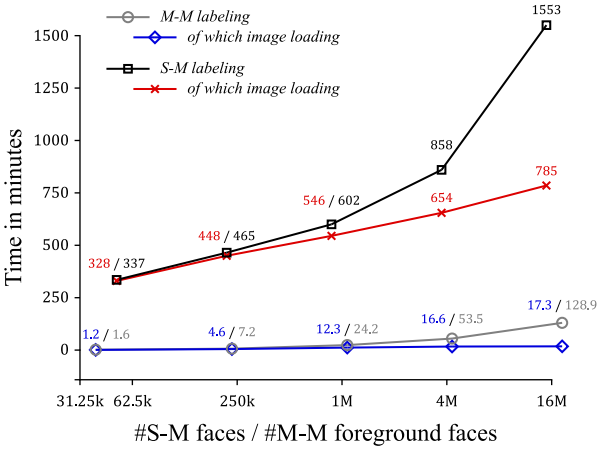


Fig. 8. Plot of the labeling timings of the single-mesh and our multi-mesh approach, applied to the *Hh2 We1* dataset, for increasing mesh sizes.

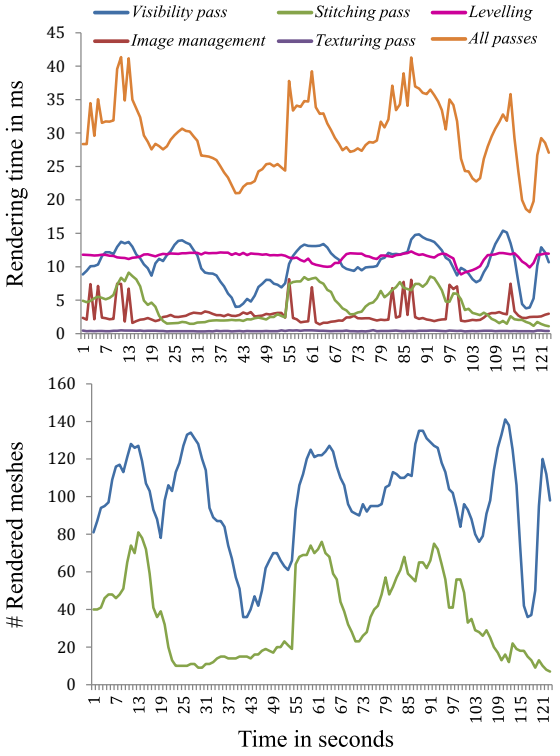


Fig. 9. Rendering timings and numbers of rendered meshes measured during a walkthrough of the *Hh2 We1* model.

to handle large-scale point clouds.

As a comparable surface-reconstruction technique, we used the out-of-core Poisson-based approach proposed by Bolitho et al. [10]. To texture the meshes, we employed Lempitsky and Ivanov's work [3], which most closely matches our approach. For the minimization of their energy function, we used the α-expansion algorithm in Boykov et al. [25], as in our case.

7.1 Performance and Memory Consumption

All results in this paper were produced on a standard PC with an Intel i7 2600K 3.40 GHz CPU, 16 GB RAM and NVIDIA GeForce GTX580 GPU.

The reference single-mesh reconstruction- and labeling approach was applied to the datasets *Hh2 We1* and *7schläfer*. Table 2 compares the resulting mesh sizes and timings with those of our proposed multi-mesh system for one particular configuration. A direct comparison of the timings at the same reconstruction accuracy is non-trivial due to our runtime surface representation. However, we observed that during rendering, the majority of the overlap areas is represented by foreground regions (see also Fig. 2). Therefore, to obtain an expressive comparison, we chose the resolution of our depth maps in a way that the total number of all foreground faces approximately matches the number of single-mesh faces (see Table 2). In this configuration, our approach is faster by an order of magnitude or even more.

Fig. 8 analyzes just the labeling times for increasing mesh sizes. In both approaches, image loading alone already requires a significant amount of total labeling time. This is due to the image requests during the computation of the smoothness costs. Since not all of the high-resolution input images fit into main memory, images are loaded on demand. We reserved 3GB ($\sim$80 images) of main memory for them. Images in memory that are not required for a longer time are dropped according to the *least recently used* paradigm [28]. Table 2 also indicates that in the multi-mesh case, much fewer images are loaded, for reasons explained in Section 7.3. However, even ignoring image load time, our method is significantly faster.

Fig. 9 shows the performance of our rendering approach for a walkthrough of the *Hh2 We1* model, rendered at a resolution of 1280 × 720 (see also the

TABLE 2
Timings of the single-mesh (in hours) and our multi-mesh approach (in minutes), and in both cases, the mesh sizes.

	Model	*Hh2 We1*	*7schläfer*
Single-mesh	Meshing time	3h	8h
	#Vertices	1.9M	3.95M
	#Faces	3.8M	7.9M
	Memory consumption	68.4MB	142.3MB
	Labeling time	14.3h	102.95h
	of which image loading/count	10.9h/171k	96.9h/1592k
	Cycles[1]	2	2
	Total preprocess time	17.3h	110.95h
Multi-mesh	Depth maps generation time	4m	5.3m
	Depth-map resolution[2]	256×171	256×171
	#Faces	17M	16.7M
	Memory consumption	48.4MB	44.4MB
	Labeling time	53.5m	30m
	of which image loading/count	16.6m/7.5k	6.8m/3.3k
	Cycles[1]	2	2
	Candidate images min/aver/max	2/46/122	4/39/93
	#Foreground faces	4.6M	7.2M
	Overlap ratio[3] (o_r)	0.73	0.57
	Total preprocess time	57.5m	35.3m

[1] Each cycle performs an iteration for every label (expansion algorithm).
[2] The depth-map aspect ratio matches the image aspect ratio.
[3] Overlap ratio is the ratio of the number of background faces to the total number of multi-mesh faces.

TABLE 3
Parameters used for the rendering performance analysis.

Labeling				Image man.	Stitching	Levelling	
λ_s	λ_{dist}	λ_{or}	h[1]	tex. array size	ε[1]	#iter.	w
0.2	1	2	10	25 (∼0.9GB)	0.1	8	0.4

[1] in meters

accompanying video). It also gives numbers of depth meshes rendered during the visibility and stitching render passes. For this performance analysis, the depth maps are generated at a resolution of 256×171 (Table 2). These depth maps and auxiliary textures storing triangle labels and vertex colors were stored on the GPU. In total, these textures require ∼180MB of video memory. Table 3 shows the parameters used for the labeling and during the rendering. The minimum, average and maximum frame rates obtained during the walkthrough are 24 fps, 34 fps and 55 fps, respectively. For the same walkthrough, rendering of the single mesh (with 3.8M faces) and textured splats took 42 and 23 fps on average, respectively. For single-mesh rendering, we employed OpenSceneGraph [29] and a virtual-texturing algorithm [30]. For textured splatting, we used an out-of-core point renderer [21] to stream visible parts of the point cloud to the GPU. We restricted the renderer to stream a maximum amount of 10M points to the GPU each frame.

The lower memory consumption of depth meshes (despite the much higher total number of faces) results

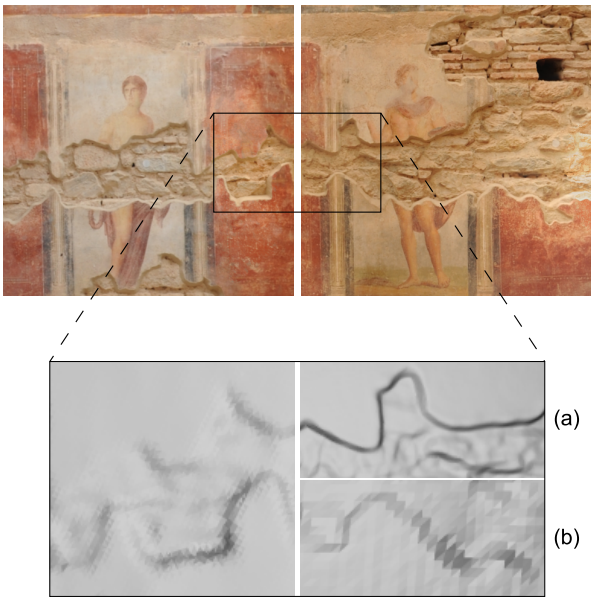


Fig. 10. Comparison of the single-mesh reconstruction (left) and our multi-mesh approach (right, (a)) applied to a part of the *Hh2 We1* dataset. For a fair comparison, the number of single-mesh faces approximately equals the number of multi-mesh foreground faces (∼1M). In the presence of high-resolution depth maps of this particular region, the multi-mesh approach produces a very accurate surface representation as it chooses a foreground for rendering (a). (b) shows one of many backgrounds of this region, which are not considered.

from the fact that each depth-map pixel stores one float compared to three floats per single-mesh vertex and three integers per single-mesh triangle. On the other hand, while mesh-based visualization can avoid further storage of the input points, textured splatting requires points and their normal vectors for rendering, which results in a vast memory consumption (see Table 1).

7.2 Reconstruction and Rendering Quality

As stated in Fuhrmann and Goesele [11], a Poisson-based reconstruction technique [8] does not account for input data at different scales. In contrast, the multi-mesh approach seamlessly combines overlapping depth meshes at various scales, and in overlap areas, prefers meshes with higher sampling rates for rendering. Thus, a multi-mesh representation uses more triangles in regions where high-resolution depth maps are present. Our approach therefore can represent particular surface regions more accurately than the reference single-mesh reconstruction (for an example, compare Fig. 10 left and right, respectively).

However, a direct comparison of the reconstruction accuracy of the single-mesh and our multi-mesh technique is not straightforward, since the consideration of the smoothness also affects the accuracy of a multi-mesh representation. Fig. 11 shows a close-up view

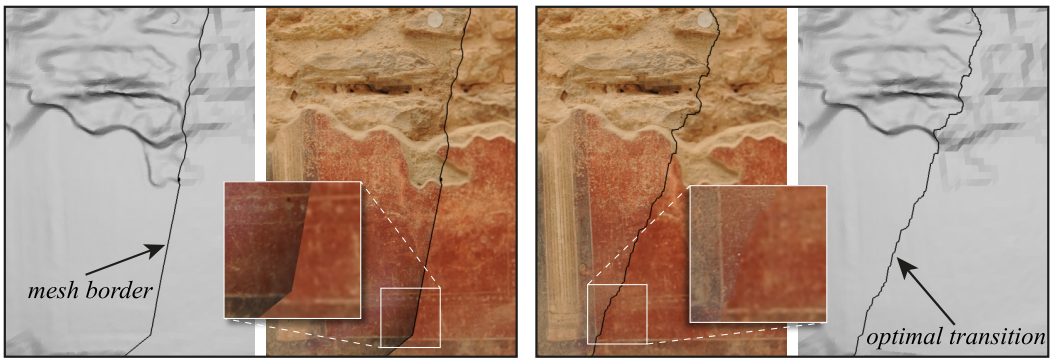


Fig. 11. A close-up view of our rendering without (a) and with (b) the color-smoothness constraint during the computation of the labelings. (b) Artifacts due to lighting variations and registration errors are barely visible along the transition of the meshes, but this comes at a price of an overall lower detail.

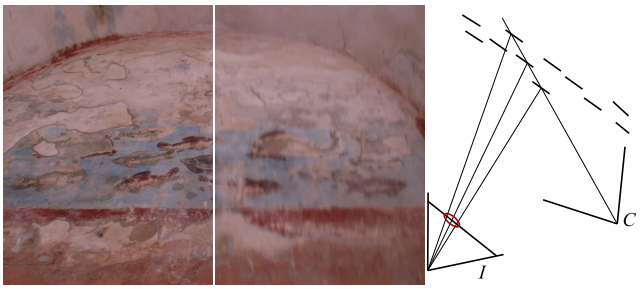


Fig. 12. Side-by-side comparison of our approach (a) and textured splatting (b). (c) demonstrates the dependency of the blurriness of the images generated by textured splatting on the noise and the view direction. C denotes the user camera and I an input image camera.

of a multi-mesh result with (b) and without (a) the consideration of color smoothness. The trade-off of both the texture resolution and the geometric detail for less noticeable artifacts along the seam is clearly visible.

Comparing to textured surface splatting, Fig. 12 demonstrates the superior quality of our multi-mesh approach in terms of visual quality. It can be clearly seen that in contrast to mesh rendering, blending texture-mapped splats can cause heavy blurring artifacts depending on the viewing angle and the degree of the noise present in the point datasets.

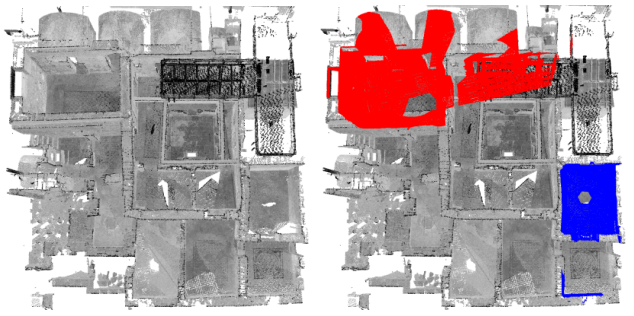
7.3 Asymptotic Analysis

In this section we give an asymptotic argument why the multi-mesh approach is significantly faster in the labeling step. We first consider the upper bound of the number of smoothness computations: In the single-mesh case, each iteration of the expansion algorithm accounts for $\mathcal{O}(e_S * |\mathbf{I}|^2)$ smoothness computations, where e_S and $|\mathbf{I}|$ denote the number of mesh edges and the number of labels, respectively. A number of $|\mathbf{I}|$ iterations results in $\mathcal{O}(|\mathbf{I}| * e_S * |\mathbf{I}|^2)$ total computations. In the multi-mesh case, the upper bound for the number of smoothness computations per mesh is $\mathcal{O}(k * e_M * k^2)$, where e_M is the number of edges in a single depth mesh and k denotes the average number of candidate images considered for the labeling of each depth mesh (Table 2). The labeling of all the $|\mathbf{I}|$ meshes results in $\mathcal{O}(|\mathbf{I}| * k * e_M * k^2)$ smoothness computations. Without loss of generality, to obtain an expressive comparison, let's choose the resolution of – and thus the number of edges e_M in – each depth map in a way that in total, all foregrounds contain the same amount of edges as the single mesh, i.e., $(1 - o_r) * |\mathbf{I}| * e_M = e_S$. Then the upper bound of the multi-mesh case can be reformulated as $\mathcal{O}(e_S * k^3/(1 - o_r))$, whereas the single-mesh bound is $\mathcal{O}(e_S * |\mathbf{I}|^3)$. Thus, the computation of the smoothness costs in our multi-mesh approach is generally faster by a factor of $(|\mathbf{I}|/k)^3 * (1 - o_r)$. In general, $k << |\mathbf{I}|$, so we obtain a significant speed up in comparison to the single-mesh case.

In a similar way, upper bounds of the number of data-term computations can be formulated: In the single mesh case, each iteration accounts for $\mathcal{O}(f_S * |\mathbf{I}|)$ computations, where f_S denotes the number of mesh faces. A number of $|\mathbf{I}|$ iterations results in $\mathcal{O}(f_S * |\mathbf{I}|^2)$ computations. In the multi-mesh case, the upper bound is given by $\mathcal{O}(f_M * k^2)$. Then, the labeling of all meshes results in $\mathcal{O}(|\mathbf{I}| * f_M * k^2)$ total data-term computations. Analogous to above, by setting $(1 - o_r) * |\mathbf{I}| * f_M = f_S$ we get a speed-up factor of $(|\mathbf{I}|/k)^2 * (1 - o_r)$.

7.4 Extensibility

Our system is designed to provide maximum flexibility and extensibility when dealing with large-scale datasets, which might be extended over time. In the single-mesh case, adding new images would require an expensive global relabeling of the model incorporating all previous and new images. On the other



(a) (b)

Fig. 13. Point and image data acquired from new scan positions can be easily integrated into an existing multi-mesh representation without requiring a complete re-computation. To evaluate this, we generated a multi-mesh representation from (a) 44 scan positions of the *Hh2 We1* model and then (b) added two new scan positions (red and blue). In this example, a recomputation for 63 and 11 of 264 previous meshes was necessary.

hand, given our proposed multi-mesh representation of the model, adding a new photograph of the scene involves the generation and labeling of one new depth mesh, and a relabeling of (on average) only the k meshes that overlap with the new one.

A more interesting scenario is to extend a model by new point and image data acquired from a new scan position. In this case, we first generate depth maps for the new images. Then, we determine all the previous images whose camera can see the new point data, since the corresponding depth maps have to be regenerated. For this purpose, we perform for each image an occlusion query for the new points by rendering them as seen by the image's camera, where the corresponding depth mesh serves as an occluder. If the new point data is visible, i.e., if at least one point is drawn, we regenerate the depth map incorporating all previous and new point data. Finally, we compute for all new and regenerated depth meshes the labeling.

To evaluate the effectiveness of the proposed extensibility method, we first used 44 scan positions (Fig. 13 (a)) and corresponding 264 images to generate the multi-mesh representation of the *Hh2 We1* model and then subsequently added the remaining 2 scan positions (shown in Fig. 13 (b) red and blue, respectively) and 12 images. To a major extent, both scan positions are contained in the existing model, which means that the corresponding points add detail to the point data acquired from the previous 44 positions. Adding these two new scans caused a regeneration and relabeling of 63 and 11 previous meshes, respectively.

8 Conclusion

In this paper, we proposed a novel two-phase approach for providing high-quality visualizations of large-scale point clouds accompanied by registered images. In a preprocess, we generate multiple overlapping meshes and solve the image-to-geometry mapping problem locally for each mesh. In the visualization phase, we seamlessly stitch these meshes to a high-quality surface representation.

We have shown that the localization of the global mapping problem provides a huge performance gain in the preprocessing phase, an overall lower memory consumption, and a higher flexibility and extensibility of large-scale datasets, in exchange for a slightly higher rendering complexity (due to the rasterization of multiple meshes in overlap regions) and minor stitching artifacts at object concavities. Similar to Fuhrmann and Goesele [11], our approach produces an adaptive surface representation with coarse as well as highly detailed regions. Additionally, our multi-mesh method addresses texturing issues.

9 Limitations and Future Work

The most time-consuming steps of the preprocessing phase are the computation of the smoothness costs and the minimization of the objective function of Eq. 1, since these are currently performed on the CPU. However, the 4-connected grid neighborhood of triangle pairs could be further exploited to transfer these tasks to the GPU. Note that we perform the remaining operations of the labeling step (i.e., the computation of candidate labels and data costs) on the GPU.

In order to keep the image-management step simple, we currently do not support mipmapping. We use the high-resolution images for texturing, and resort to per-vertex colors if images are not available on the GPU. However, a more sophisticated method for so-called virtual texturing [30] can be easily integrated into our system to alleviate this.

Another limitation is that our stitching currently makes a random decision between overlapping background fragments. This can lead to low-resolution geometry filling small gaps between foreground regions. By favoring fragments with a lower data penalty, the resolution provided by backgrounds could be improved. Unfortunately, such a per-pixel decision would not consider label continuity over neighboring pixels. We thus plan to investigate further possibilities to make a more elaborate choice between overlapping backgrounds.

Acknowledgments

We wish to express our thanks to the reviewers for their insightful comments. We also thank Norbert Zimmermann for providing us with the datasets and Michael Birsak for his assistance in the performance analysis.

This research was supported by the Austrian Research Promotion Agency (FFG) project REPLICATE

Fig. 14. Two views of each input point cloud, and the corresponding multi-mesh representations of (from top to bottom) *Hh2 We1*, *Hh2 We6*, *Centcelles* and *7schläfer* datasets. The point clouds are displayed with the colors acquired by a scanner. In the last example, points are color-coded according to their normals.

(no. 835948), the EU FP7 project HARVEST4D (no. 323567), and the Austrian Science Fund (FWF) project DEEP PICTURES (no. P24352-N23).

REFERENCES

[1] R. Yang, D. Guinnip, and L. Wang, "View-dependent textured splatting," *The Visual Computer*, vol. 22, no. 7, pp. 456–467, 2006.

[2] D. Sibbing, T. Sattler, B. Leibe, and L. Kobbelt, "Sift-realistic rendering," in *Proc. the 2013 International Conf. 3D Vision (3DV 13)*, 2013, pp. 56–63.

[3] V. Lempitsky and D. Ivanov, "Seamless mosaicing of image-based texture maps," in *Computer Vision and Pattern Recognition (CVPR 07), IEEE*, June 2007, pp. 1–6.

[4] R. Gal, Y. Wexler, E. Ofek, H. Hoppe, and D. Cohen-Or, "Seamless montage for texturing models," *Computer Graphics Forum*, vol. 29, no. 2, pp. 479–486, 2010.

[5] M. Botsch and L. Kobbelt, "High-quality point-based rendering on modern gpus," in *Proc. the 11th Pacific Conf. Computer Graphics and Applications (PG 03)*, 2003, pp. 335–343.

[6] M. Botsch, M. Spernat, and L. Kobbelt, "Phong splatting," in *Proc. the 1st Eurographics Symp. Point-Based Graphics (SPBG 04)*, 2004, pp. 25–32.

[7] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt, "High-quality surface splatting on today's gpus," in *Proc. the 2nd Eurographics / IEEE VGTC Symp. Point-Based Graphics (SPBG 05)*, 2005, pp. 17–24.

[8] M. Kazhdan, M. Bolitho, and H. Hoppe, "Poisson surface reconstruction," in *Proc. the 4th Eurographics Symp. Geometry Processing (SGP 06)*, 2006, pp. 61–70.

[9] M. Kazhdan and H. Hoppe, "Screened poisson surface reconstruction," *ACM Trans. Graph.*, vol. 32, no. 3, pp. 29:1–29:13, June 2013.

[10] M. Bolitho, M. Kazhdan, R. Burns, and H. Hoppe, "Multilevel streaming for out-of-core surface reconstruction," in *Proc. the 5th Eurographics Symp. Geometry Processing (SGP 07)*, 2007, pp. 69–78.

[11] S. Fuhrmann and M. Goesele, "Fusion of depth maps with multiple scales," in *Proc. the 2011 SIGGRAPH Asia Conf. (SA 11)*, 2011, pp. 148:1–148:8.
[12] G. Turk and M. Levoy, "Zippered polygon meshes from range images," in *Proc. the 21st Annual Conf. Computer Graphics and Interactive Techniques (SIGGRAPH 94)*, 1994, pp. 311–318.
[13] S. Marras, F. Ganovelli, P. Cignoni, R. Scateni, and R. Scopigno, "Controlled and adaptive mesh zippering," in *GRAPP*, 2010, pp. 104–109.
[14] F. Bernardini, I. M. Martin, and H. Rushmeier, "High-quality texture reconstruction from multiple scans," *IEEE Transactions on Visualization and Computer Graphics*, vol. 7, no. 4, pp. 318–332, Oct. 2001.
[15] A. Baumberg, "Blending images for texturing 3d models," in *Proc. the British Machine Vision Conference (BMVC 02)*, 2002, pp. 38.1–38.10.
[16] M. Callieri, P. Cignoni, M. Corsini, and R. Scopigno, "Masked photo blending: mapping dense photographic dataset on high-resolution 3d models," *Computers and Graphics*, vol. 32, no. 4, pp. 464–473, Aug 2008.
[17] M. Eisemann, B. De Decker, M. Magnor, P. Bekaert, E. de Aguiar, N. Ahmed, C. Theobalt, and A. Sellent, "Floating textures," *Computer Graphics Forum*, vol. 27, no. 2, pp. 409–418, Apr. 2008.
[18] M. Dellepiane, R. Marroquim, M. Callieri, P. Cignoni, and R. Scopigno, "Flow-based local optimization for image-to-geometry projection," *IEEE Transaction on Visualization and Computer Graphics*, vol. 18, no. 3, pp. 463–474, Mar 2012.
[19] D. Aliaga, J. Cohen, A. Wilson, E. Baker, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stuerzlinger, R. Bastos, M. Whitton, F. Brooks, and D. Manocha, "Mmr: an interactive massive model rendering system using geometric and image-based acceleration," in *Proc. the 1999 Symp. on Interactive 3D Graphics (I3D 99)*, 1999, pp. 199–206.
[20] R. Preiner, S. Jeschke, and M. Wimmer, "Auto splats: Dynamic point cloud visualization on the gpu," in *Proc. Eurographics Symp. on Parallel Graphics and Visualization (EGPGV 12)*, May 2012, pp. 139–148.
[21] C. Scheiblauer and M. Wimmer, "Out-of-core selection and editing of huge point clouds," *Computers and Graphics*, vol. 35, no. 2, pp. 342–351, Apr. 2011.
[22] V. Kolmogorov and R. Zabih, "What energy functions can be minimized via graph cuts?" in *Proc. the 7th European Conf. Computer Vision-Part III (ECCV 02)*, 2002, pp. 65–81.
[23] Y. Boykov and V. Kolmogorov, "An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 9, pp. 1124–1137, Sep. 2004.
[24] C. Allene, J.-P. Pons, and R. Keriven, "Seamless image-based texture atlases using multi-band blending," in *Proc. 19th International Conf. Pattern Recognition (ICPR 08)*, 2008, pp. 1–4.
[25] Y. Boykov, O. Veksler, and R. Zabih, "Fast approximate energy minimization via graph cuts," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 23, no. 11, pp. 1222–1239, Nov. 2001.
[26] S. Jeschke, D. Cline, and P. Wonka, "A gpu laplacian solver for diffusion curves and poisson image editing," *ACM Trans. Graph.*, vol. 28, no. 5, pp. 116:1–116:8, Dec. 2009.
[27] D. Scherzer, S. Jeschke, and M. Wimmer, "Pixel-correct shadow maps with temporal reprojection and shadow test confidence," in *Proc. the 18th Eurographics Conf. Rendering (EGSR 07)*, 2007, pp. 45–50.
[28] P. J. Denning, "The working set model for program behavior," *Comm. ACM*, vol. 11, no. 5, pp. 323–333, May 1968.
[29] D. Burns and R. Osfield, "Open scene graph a: Introduction, b: Examples and applications," in *Proc. the IEEE Virtual Reality (VR 04)*, 2004, pp. 265–.
[30] M. Mittring and C. GmbH, "Advanced virtual texture topics," in *ACM SIGGRAPH 2008 Games (SIGGRAPH 08)*, 2008, pp. 23–51.

Murat Arikan is a Ph.D. student at the Institute of Computer Graphics and Algorithms of the Vienna University of Technology. He received his M.Sc. degree in Mathematics from Vienna University of Technology in 2008. His current research interests are real-time rendering, point-based rendering, and interactive modeling.

Reinhold Preiner received his B.Sc. degree in Computer Science from Graz University in 2008 and his M.Sc. degree in Computer Science from Vienna University of Technology in 2010. His research interests include reconstruction, geometry processing, and interactive global illumination. He is now an assistant professor and doctoral researcher at the Institute of Computer Graphics and Algorithms of the Vienna University of Technology.

Claus Scheiblauer is a Ph.D. student at the Institute of Computer Graphics and Algorithms of the Vienna University of Technology, where he received an M.Sc. in 2006. His current research interests are real-time rendering, point-based rendering, and out-of-core processing.

Stefan Jeschke is a scientist at IST Austria. He received an M.Sc. in 2001 and a Ph.D. in 2005, both in computer science from the University of Rostock, Germany. His research interest includes modeling and display of vectorized image representations, applications and solvers for PDEs, as well as modeling and rendering complex natural phenomena.

Michael Wimmer is an associate professor at the Institute of Computer Graphics and Algorithms of the Vienna University of Technology, where he received an M.Sc. in 1997 and a Ph.D. in 2001. His current research interests are real-time rendering, computer games, real-time visualization of urban environments, point-based rendering and procedural modeling. He has coauthored many papers in these fields, and was papers co-chair of EGSR 2008 and Pacific Graphics 2012, and is associate editor of Computers & Graphics.