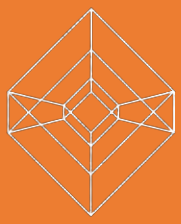


HARVEST4D

HARVESTING DYNAMIC 3D WORLDS FROM COMMODITY SENSOR CLOUDS

Publications for Task 8.1

Deliverable 8.11

Date:	6.7.2015
Grant Agreement number:	EU 323567
Project acronym:	HARVEST4D
Project title:	Harvesting Dynamic 3D Worlds from Commodity Sensor Clouds

Document Information

Deliverable number	D8.11
Deliverable name	Publications for Task 8.1
Version	0.1
Date	2015-07-06
WP Number	8
Lead Beneficiary	PARISTEC
Nature	R
Dissemination level	PU
Status	Final
Author(s)	VUT

Revision History

Rev.	Date	Author	Org.	Description
0.1	2015-07-06	Michael Wimmer	VUT	Draft

Statement of originality

This deliverable, although of public dissemination level, may at the time of delivery still contain original unpublished work (e.g., accepted papers that are not public yet, or papers under revision). Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both.

TABLE OF CONTENTS

1 EXECUTIVE SUMMARY

1.1 INTRODUCTION

This deliverable describes the publications that resulted from Task 8.1, and how they fit into the work plan of the project.

The objective of Task 8.1 is to develop new visualization strategies for large time-dependent data that combine efficiency and generality. Several publications in this direction have been produced, focusing on various aspects of rendering. They include rendering of large-scale data with dynamic annotations, rendering of point clouds with images, rendering of models with enhanced reflectance models (BSVTF), enhancing rendering using global illumination effects, as well as physical rendering.

1.2 PUBLICATIONS

The following 7 publications are mainly associated with Task 8.1 and can be found in the appendix of this deliverable:

- Christian Kehl, Tim Tutenel and Elmar Eisemann
  Smooth, Interactive Rendering Techniques on Large-Scale, Geospatial Data in Flood Visualisations
  ICT Open, 2013
- Murat Arikan, Reinhold Preiner, Claus Scheiblauer, Stefan Jeschke, Michael Wimmer
  Large-Scale Point-Cloud Visualization through Localized Textured Surface Reconstruction
  IEEE Transactions on Visualization & Computer Graphics, 20(9):1280-1292, September 2014
- Murat Arikan, Reinhold Preiner, Michael Wimmer
  Multi-Depth-Map Raytracing for Efficient Large-Scene Reconstruction
  IEEE Transactions on Visualization & Computer Graphics, preprints, 2015
- Christopher Schwartz, Roland Ruiters, and Reinhard Klein
  Level-of-Detail Streaming and Rendering using Bidirectional Sparse Virtual Texture Functions
  Computer Graphics Forum (Proc. of Pacific Graphics), 32(7):345-354, Oct. 2013
- Quintijn Hendrickx, Leonardo Scandolo, Martin Eisemann , Elmar Eisemann
  Adaptively Layered Statistical Volumetric Obscurance
  In *Proceedings of High-Performance Graphics*, 2015
- Thomas Kroes, Martin Eisemann, and Elmar Eisemann.
  Visibility sweeps for joint-hierarchical importance sampling of direct lighting for stochastic volume rendering.

In *Proceedings of the 41st Graphics Interface Conference* (GI '15). Canada, p 97-104, 2015.
BEST STUDENT PAPER AWARD!

- Paolo Cignoni, Nico Pietroni, Luigi Malomo, Roberto Scopigno
  Field-Aligned Mesh Joinery
  ACM Transactions on Graphics (TOG), Volume 33 (1), 2014

Several other papers are related to Task 8.1. They can be found in the deliverables they mainly contribute to:

- Mohamed Radwan, Stefan Ohrhallinger, Michael Wimmer
  Efficient Collision Detection While Rendering Dynamic Point Clouds
  Graphics Interface, p 25-33. 2014
  (Task 3.2)

- Johannes G. Leskens, Christian Kehl, Tim Tutenel, Timothy Kol, Gerwin de Haan, Guus Stelling, Elmar Eisemann
  An interactive simulation and visualization tool for flood analysis usable for practitioners
  Mitigation and Adaptation Strategies for Global Change, May 2015
  (Task 8.2)

- Tim Tutenel, Christian Kehl, Elmar Eisemann
  Interactive visual analysis of flood scenarios using large-scale LiDAR point clouds
  Geospatial World Forum 2013, May 2013
  (Task 8.2)

2 DESCRIPTION OF PUBLICATIONS

2.1 OVERVIEW

The main objective of this task is to develop new methods for efficient visualization of large multimodal and/or time-dependent data. The publications in this deliverable focus on several aspects:

- Out-of-core rendering of very large data sets (i.e., floods) [Kehl et al. 2013]
- Out-of-core rendering of large point clouds with images [Arikan et al. 2014, Arikan et al. 2015]. We explored two methods, one based on mesh rendering and another on ray tracing the point-based model directly.
- Out-of-core rendering of realistic surface descriptions [Schwartz et al. 2013], in particular, focusing on Bidirectional Sparse Virtual Texture Functions.
- Screen-space enhancement of rendering using ambient occlusion [Hendrickx et al. 2015].
- Enhancing volume rendering using advanced illumination effects [Kroes et al. 2015]. We explored both direct lighting of volume rendering, as well as rendering ambient-occlusion effects in volumes.

- Physical rendering [Cignoni et al. 2014]

These contributions cover a range of application scenarios in this task and Harvest4D. The publications on **out-of-core rendering**, in particular, show different tradeoffs between data-set size and visualization quality and accuracy.

The first paper [Kehl et al. 2013] deals with very large data sets (several Terabytes), which occur in flood visualizations. Here the focus is on a strong out-of-core strategy, and methods to interactively modify the visualization, introducing time-dependency. The interactive modifications work for the application scenario of 2.5D data, e.g., floods.

The two publications on rendering points with images [Arikan et al. 2014, Arikan et al. 2015] deal with smaller (but still large) data sets, but introduce higher quality due to the direct use of photographs taken of the scene. This allows a reconstruction with very high quality both for geometry and textures. This method also allows the fast integration of new data – not in real time, but much faster than a whole rebuild of the data structure would take. Both geometry and image data are handled out-of-core in this method

Finally, the publication on realistic surface descriptions [Schwartz et al. 2013] handles normal-sized models, but adds another level of quality by introducing a highly realistic surface description, Bidirectional Sparse Virtual Texture Functions. Here, the reflectance data is handled out-of-core due to its large size.

While these publications all deal with out-of-core rendering, there are further contributions that focus on **other aspects of rendering**. The first method [Hendrickx et al. 2015] adds ambient occlusion effects and applies to any rendering method that produces a depth buffer, since it works entirely in screen space. Thus, it also applies to the rendering methods discussed so far. The second one [Kroes et al. 2015] deals with a modality that has so far not been treated in Harvest4D, namely volume data sets. Here, an efficient importance-sampling method is developed for high-quality illumination in volume datasets. We will investigate whether this method could also be generalized to other data sets that involve very dense samples, like the methods working on large point-based data.

Finally, we also explore the option of breaking out of the screen entirely and moving towards a **physical representation** of data using an illustrative arrangement of geometry to represent a model [Cignoni et al. 2014].

2.2 SMOOTH, INTERACTIVE RENDERING TECHNIQUES ON LARGE-SCALE, GEOSPATIAL DATA IN FLOOD VISUALISATIONS

Figure 1: Point-cloud of a coastal area with a dyke that has been interactively added to the data set using our system.

In this paper [Kehl et al. 2013], we present new approaches to render and interact with detail-varying LiDAR point sets, which, due to the enormous data size, cannot currently be rendered interactively without significantly compromising quality. Furthermore, our approach allows the attachment of large-scale geospatial meta information and the modification of point attributes on the fly. The core of our algorithm is a dynamic GPU-based hierarchical tree data structure that is used in conjunction with an out-of-core, Level-of-Detail Point-based Rendering algorithm to modify data on the fly. This combination makes it possible to augment the original data with dynamic context information that can be used to highlight features (e.g., routes, marked areas) or to reshape the entire data set in real-time.

We showcase the usefulness of our algorithm in the context of disaster management and illustrate how decision makers can discuss a flood scenario covering a large area (spanning 300 km2) and discuss hazards, as well as related protection measures, interactively. One of our presented reference point sets includes parts of the AHN2 data set (14 TB of LiDAR data in total) (see Figure 1). Previous rendering algorithms relied on a long offline preprocessing (several hours) to ensure a quick data display. This step made any changes to the data impossible. With our new approach, we can modify point sets without requiring a new preprocessing run.

2.3 LARGE-SCALE POINT-CLOUD VISUALIZATION THROUGH LOCALIZED TEXTURED SURFACE RECONSTRUCTION

Figure 2: A large-scale reconstructed 3D scene colored via photos shot from a few positions (see left). A segmentation algorithm associates a photo to each location, while optimizing for visual fidelity.

In Harvest4D, a broad variety of inputs such as image data or points are fused into a high-quality data representation. To handle their massive size, these data sets usually have to be treated out-of-core. One example is our work on high-quality textured archaeological data sets, where high-resolution images are used to texture a detailed mesh to achieve a high-quality reconstruction [Arikan et al. 2014]. Our solution relies on a resampling of the input point cloud using depth meshes created from the input cameras. For each mesh part, the best image is chosen using a graphcut optimization that takes geometric and image-based criteria into account. Since this optimization can be performed on a single depth mesh at a time, the algorithm is localized and can run out of core. It automatically generates a texture atlas in significantly less time than existing solutions and allows us to apply online virtual texturing to obtain a streaming out-of-core high-quality visualization system (Figure 2).

2.4 MULTI-DEPTH-MAP RAYTRACING FOR EFFICIENT LARGE-SCENE RECONSTRUCTION

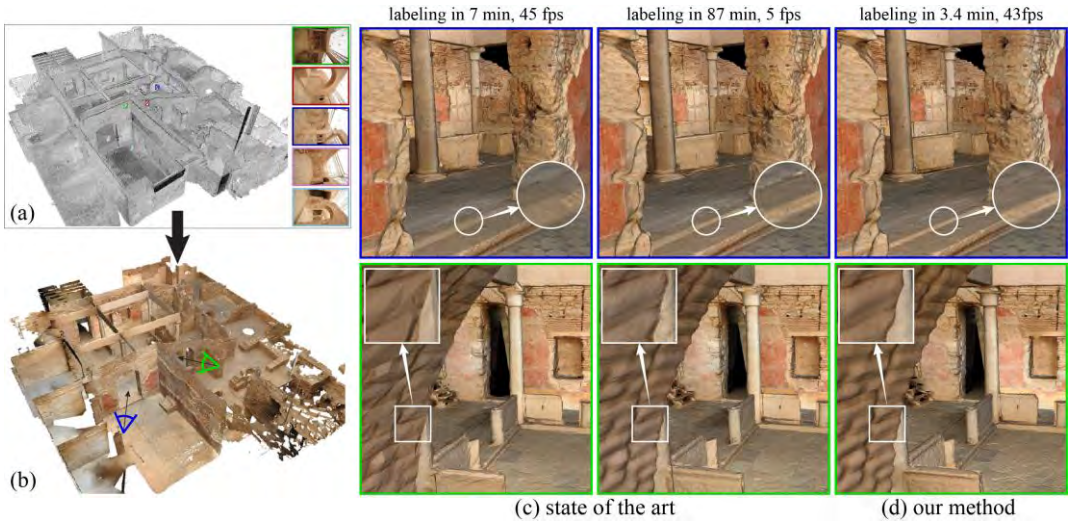


Figure 3: Equal-time and equal-quality comparisons for our method for the scene shown on the left.

In this work [Arikan et al. 2015], we revisit the problem of rendering large point clouds with images. In particular, we found that the rendering performance of original method [Arikan et al. 2014] is strongly dependent on the number of depth maps and their resolution. Moreover, for the proposed scene representation, every single depth map has to be textured by the images, which in practice heavily increases processing costs. In this paper, we thus present a novel method to break these dependencies by introducing an efficient raytracing of multiple depth maps. In a preprocessing phase, we first generate high-resolution textured depth maps by rendering the input points from image cameras and then perform a graphcut based optimization to assign a small subset of these points to the images. At runtime, we use the resulting point-to-image assignments (1) to identify for each view ray which depth map contains the closest ray-surface intersection and (2) to efficiently compute this intersection point. The resulting algorithm accelerates both the texturing and the rendering of the depth maps by an order of magnitude. An equal-time/equal-quality comparison can be found in Figure 3.

2.5 LEVEL-OF-DETAIL STREAMING AND RENDERING USING BIDIRECTIONAL SPARSE VIRTUAL TEXTURE FUNCTIONS

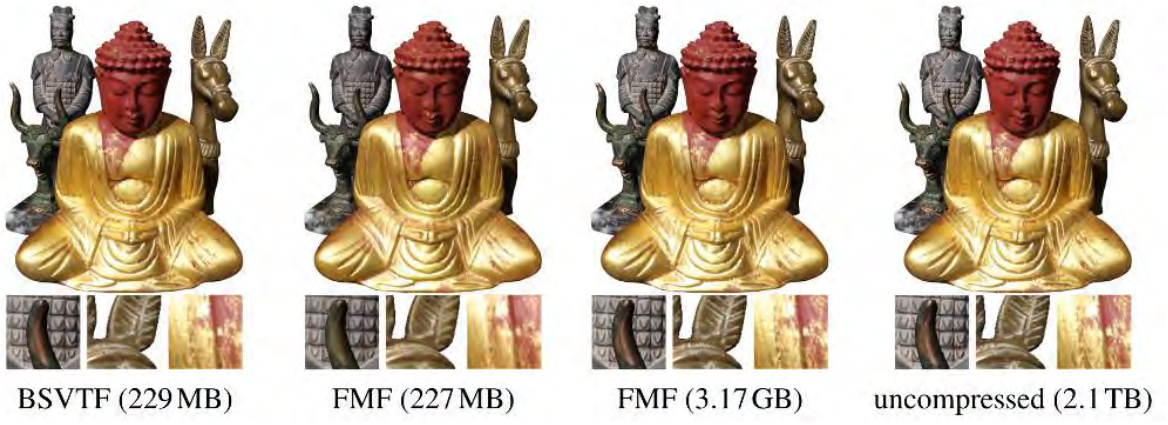


Figure 4: Rendering quality comparison of our new BSVTF technique.

In this work, we move towards a higher-quality material representation, thus establishing a link between WP8 and WP7, i.e., we combine high-quality reflectance information with out-of-core techniques. In particular, Bidirectional Texture Functions (BTFs) are among the highest quality material representations available today and thus well suited whenever an exact reproduction of the appearance of a material or complete object is required. BTFs are usually measured from real-world samples and easily consist of tens or hundreds of gigabytes. By using data-driven compression schemes, such as matrix or tensor factorization, a more compact but still faithful representation can be derived. This way, BTFs can be employed for real-time rendering of photo-realistic materials on the GPU. However, scenes containing multiple BTFs or even single objects with high-resolution BTFs easily exceed available GPU memory on today's consumer graphics cards unless quality is drastically reduced by the compression.

In this publication [Schwartz et al. 2013], we therefore propose the Bidirectional Sparse Virtual Texture Function, a hierarchical level-of-detail approach for the real-time rendering of large BTFs that requires only a small amount of GPU memory. More importantly, for larger numbers or higher resolutions, the GPU and CPU memory demand grows only marginally and the GPU workload remains constant. For this, we extend the concept of sparse virtual textures by choosing an appropriate prioritization, finding a trade off between factorization components and spatial resolution. Besides GPU memory, the high demand on bandwidth poses a serious limitation for the deployment of conventional BTFs. We show that our proposed representation can be combined with an additional transmission compression and then be employed for streaming the BTF data to the GPU from local storage media or over the Internet. In combination with the introduced prioritization, this allows for the fast visualization of relevant content in the user's field of view and a consecutive progressive refinement. See Figure 4 for a comparison.

2.6 ADAPTIVELY LAYERED STATISTICAL VOLUMETRIC OBSCURANCE

(a) *17 point samples (2.0ms)* (b) *12 line samples (1.7ms)* (c) *StatVO (1.6ms)*

Figure 5: Statistical volumetric obscurance (c) achieves higher quality than point sampling (a) or line sampling (b) at similar computational time.

In this work [Hendrickx et al. 2015], we aim to improve the rendering quality of images obtained using other techniques in this task. The idea is to apply a screen-space postprocessing pass that adds shading effects based on the local geometry. In particular, we accelerate volumetric obscurance, a variant of ambient occlusion, and solve undersampling artifacts, such as banding, noise or blurring, that screen-space techniques traditionally suffer from. We make use of an efficient statistical model to evaluate the occlusion factor in screen-space using a single sample. Overestimations and halos are reduced by an adaptive layering of the visible geometry. Bias at tilted surfaces is avoided by projecting and evaluating the volumetric obscurance in tangent space of each surface point. We compare our approach to several traditional screen-space ambient obscurance techniques and show its competitive qualitative and quantitative performance (see Figure 5). Our algorithm maps well to graphics hardware, does not require the traditional bilateral blur step of previous approaches, and avoids typical screen-space related artifacts such as temporal instability due to undersampling.

2.7 VISIBILITY SWEEPS FOR JOINT-HIERARCHICAL IMPORTANCE SAMPLING OF DIRECT LIGHTING FOR STOCHASTIC VOLUME RENDERING

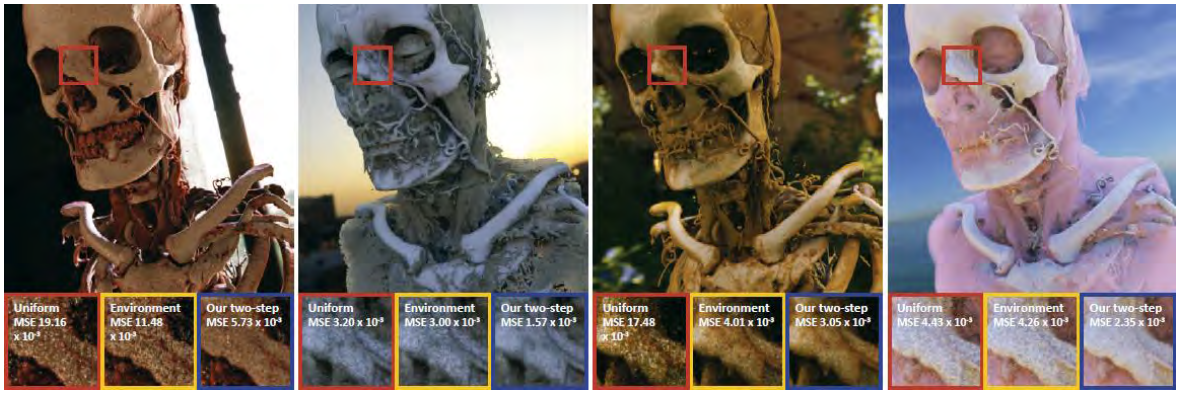


Figure 6: Comparison of our technique (blue) to uniform sampling (red) and importance sampling of the environment map (yellow), showing the faster convergence with less samples.

In this work [Kroes et al. 2015], we investigate high-quality rendering for a modality not handled so far in Harvest4D, namely volumes. Physically based light transport in heterogeneous volumetric

data is computationally expensive because the rendering integral (particularly visibility) has to be stochastically solved. We present a visibility estimation method in concert with an importance-sampling technique for efficient and unbiased stochastic volume rendering. Our solution relies on a joint strategy, which involves the environmental illumination and visibility inside of the volume. A major contribution of our method is a fast sweeping-plane algorithm to progressively estimate partial occlusions at discrete locations, where we store the result using an octahedral representation. We then rely on a quadtree-based hierarchy to perform a joint importance sampling. Our technique is unbiased, requires little precomputation, is highly parallelizable, and is applicable to a various volume data sets, dynamic transfer functions, and changing environmental lighting. Figure 6 shows a comparison to previous sampling techniques.

2.8 FIELD-ALIGNED MESH JOINERY

Figure 7: An example of an illustrative representation of a 3D model that was physically fabricated by means of a set of interlocking planar shapes that are able to convey the overall shape of the object..

Finding new ways to illustrate geometric data is one the objectives of this WP. In this context we have developed a new approach for the illustrative visualization of geometric data that relies on physical fabrication. In Mesh joinery, we have achieved an innovative method to produce complex fabricable structures in an efficient and visually pleasing manner. We represent an input geometry as a set of planar pieces, which are arranged to compose a rigid structure by exploiting an efficient slit mechanism [Cignoni et al. 2013] (see Figure 7)

3 APPENDIX

The following pages contain all the publications that are directly associated with this deliverable. Other publications referenced in this deliverable can be found in the public Harvest4D webpage (for already published papers), or in the restricted section of the webpage (for papers under submission, conditionally accepted papers, etc.).

Field-Aligned Mesh Joinery

PAOLO CIGNONI and NICO PIETRONI
CNR - ISTI
LUIGI MALOMO
University of Pisa
and
ROBERTO SCOPIGNO
CNR - ISTI

Mesh joinery is an innovative method to produce illustrative shape approximations suitable for fabrication. Mesh joinery is capable of producing complex fabricable structures in an efficient and visually pleasing manner. We represent an input geometry as a set of planar pieces arranged to compose a rigid structure, by exploiting an efficient slit mechanism. Since slices are planar, to fabricate them a standard 2D cutting system is enough.

We automatically arrange slices according to a smooth cross-field defined over the surface. Cross-fields allow representing global features that characterize the appearance of the shape. Slice placement conforms to specific manufacturing constraints.

Categories and Subject Descriptors: I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*Physically based modeling*

General Terms: Algorithms, Design

Additional Key Words and Phrases: Geometry processing, object fabrication, manufacturing

ACM Reference Format:

Paolo Cignoni, Nico Pietroni, Luigi Malomo, and Roberto Scopigno. 2014. Field-aligned mesh joinery. ACM Trans. Graph 33, 1. Article 11 (January 2014) 12 pages.
DOI: http://dx.doi.org/10.1145/2537852

1. INTRODUCTION

In this article we introduce mesh joinery, a novel and practical approach to fabricate artistic illustrative shape approximations made up of several interlocked planar pieces, called slices. Such slices can be easily fabricated using any 2D cutting device and then assembled through a sequence of manual operations.

Compared to previous approaches (such as McCrae et al. [2011], Hildebrand et al. [2012], and Schwartzburg and Pauly [2012]) we oriented the slices according to a given cross-field defined on the surface. As most of the recent quadrangulation papers have shown [Ray et al. 2006; Kälberer et al. 2007; Bommes et al. 2009, 2012; Pietroni et al. 2011], cross-fields are an excellent instrument for capturing the global structure of a given shape.

We provide a novel formalism to design a slice-to-slice interlocking system. This formalism provides enough degrees of freedom to follow complex cross-fields and, consequently, to efficiently approximate the global structure that characterizes the input shape. Additionally, we ensure a sufficient degree of physical stability of the final structure along with the sequence of manual operations required for the assembly procedure.

Our approach provides limited but low-cost solutions due to the simple cutting technologies employed and the relatively inexpensive material used (such as cardboard). Although the proposed slice structure approximates, to some extent, the original geometry, it cannot be considered as a "physical copy". Nevertheless, we believe that our approach could be attractive in specific markets, such as in artistic or illustrative contexts, in puzzles or toys, and where assembly is a key part of user experience.

1.1 Motivation

Rapid prototyping [Dimitrov et al. 2006] has been developed over the last decade to support the manufacturing process, especially for the production-quality parts in relatively small numbers. It exploits a wide variety of basic technologies to create real-world tangible reproductions from 3D digital models. While initially the range of materials was very limited, modern technologies enable a wide range of materials (plastic, glued gypsum, steel, ceramic, stone, wood, etc.) to be used. At the same time, the printing resolution has improved substantially and, consequently, accuracy in terms of reproduction has reached high standards. Nevertheless, rapid prototyping is still perceived as being too expensive for the mass market. Moreover, the input geometry has to satisfy certain geometric characteristics (manifoldness, watertightness, etc.) and static mechanical properties, in order to produce a compact, high-quality, fabricated model that is free of artifacts.

A few years ago radically new paradigms for shape fabrication were proposed [Mitani and Suzuki 2004; Shatz et al. 2006; Massarwi et al. 2007; Mori and Igarashi 2007; Li et al. 2010]. The main idea was to drastically simplify the overall printing procedure by fabricating a plausible representation of the digital model, instead of its exact copy. This class of methods relies on a simple

The research leading to these results was partially funded by EU FP7 project ICT FET Harvest4D (http://www.harvest4d.org/, G.A. no. 323567).
Authors' addresses: P. Cignoni (corresponding author) and N. Pietroni, Visual Computing Lab, CNR-ISTI, Italy; email: cignoni@isti.cnr.it; L. Malomo, Computer Science Department, University of Pisa, Italy; R. Scopigno, Visual Computing Lab, CNR-ISTI, Italy.

DOI: http://dx.doi.org/10.1145/2537852

Fig. 1. Given a 3D shape with a smooth cross-field, we generate a set of planar slices that can be interlocked in a self-supporting structure.

concept: approximating an object does not necessarily mean that there will be a visual deficit.

A recent approach proposed approximating the surface using an orthogonal arrangement of planar pieces [Hildebrand et al. 2012]. The slices are plugged into each other to compose a rigid shape.

1.2 Contributions

We redesigned the traditional slice interlocking approach in order to approximate generic 3D surfaces with greater flexibility. We focused on building arrangements composed of shallow ribbon-shaped pieces which follow a cross-field defined on the surface. These structures are made up of planar pieces that interlock with each other using an extended slit mechanism. Specifically, our contributions are as follows.

—We propose a novel strategy to fabricate illustrative shape approximations based on ribbon-shaped planar slices. Compared to classical planar sections [Hildebrand et al. 2012], ribbon-shaped slices reduce the physical constraints involved in the assembling procedure, allowing for more complex structures.

—We extend the classical slit mechanism [Hildebrand et al. 2012] by providing additional structural degrees of freedom. In particular, we consider insertion movements that are not orthogonal to slices. In addition, we formulated nonorthogonal slice placement [McCrae et al. 2011; Schwartzburg and Pauly 2012] in a novel, structurally sound perspective. We have demonstrated how these additional degrees of freedom can be exploited to efficiently represent complex models.

—We propose a novel, efficient strategy to approximate a surface with a set of slices. Slice placement is driven by an input cross-field (such as Hertzmann and Zorin [2000], Bommes et al. [2009], and Ray et al. [2009]). It provides a set of appealing, uniformly distributed polylines lying on the surface of a mesh. In addition, the method also takes into account slice insertion constraints and, while it does not theoretically guarantee that the mounting sequence is collision free, it yields arrangements that are practically assemblable and that exhibit a sufficiently robust slice structure. Our method may also take advantage of field symmetrization techniques, such as Panozzo et al. [2012] (see Figure 2) for a better perception of the global structure of the generated structure.

—We propose an automatic procedure to ensure that the slice structure is physically achievable. First, it improves the final rigidity, acting upon the slit interlocking mechanism. Second, it ensures that the slice structure conforms to the physical constraints required by the manual assembling procedure. This procedure is specifically designed to deal with our extended slit mechanism.

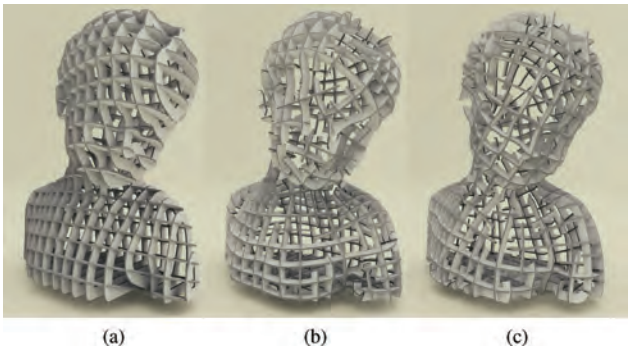


Fig. 2. (a) The classical waffle approach modeling technique (with axis-aligned slices); (b) our method applied to a cross-field calculated with Bommes et al. [2009]; (c) field symmetrization techniques [Panozzo et al. 2012] increase the visual appeal of the final result. The total length of the polylines for each method is approximately the same.

2. RELATED WORK

Fabricating tangible models from a digital 3D shape is fundamental in many industrial production processes. The majority of current applications require a high level of accuracy, that is, the printed model needs to be a highly accurate physical copy of the digital shape. For example, several applications require this level of accuracy for aesthetic purposes or for performing functional tests. However, different contexts (toys, artistic reproductions) do not require the same level of accuracy, or even prefer the production of an illustrative version of the digital model.

On the basis of accuracy and reproduction we can classify the various methods into two broad categories.

—*Accurate*. Modern devices enable almost exact copies of a given shape to be reproduced. To guarantee high reproduction accuracy, the printer and the reproduction material may both be expensive.

—*Illustrative*. These methodologies fabricate approximate copies of a given object, usually by relying on standard and inexpensive printing technologies.

In both categories, the model can be fabricated as a single piece or it can be split into a set of separate pieces and assembled afterwards.

2.1 Accurate Methods

Rapid prototyping techniques [Dimitrov et al. 2006] have been created to support the design industry. Usually the digital model needs to be represented as a closed, piecewise, manifold mesh. Due to the physical properties of the material employed and the production procedure, specific mechanical constraints must be satisfied. These constraints guarantee that the model is kept physically compact throughout the printing procedure.

Recent research has focused on how to acquire the physical properties of a real object to transplant onto the fabricated model. For example, Bickel et al. [2010] proposed a technique to match the elastic properties of a given object. Other papers focus on appearance properties: Cignoni et al. [2008] proposed a technique to enhance colors for rapid prototyping; Weyrich et al. [2009] and Matusik et al. [2009] reported a method for the improved reproducibility of surface reflectance properties by adding microgeometry; and Hašan et al. [2010] and Dong et al. [2010] proposed a technique to print specific subsurface scattering characteristics.

One common strategy is to divide up the original shape into different components, which are fabricated separately but assembled together to produce the desired shape. One example is architectural modeling, where the original shape is subdivided into a finite set of triangular [Singh and Schaefer 2010] or quadrilateral [Fu et al. 2010; Eigensatz et al. 2010] basic panels. A method to fit a freeform shape with a set of single direction bendable panels (like wooden panels) is proposed in Pottmann et al. [2010]. To further improve the smoothness of freeform surfaces in architectural design, Bo et al. [2011] introduced the so-called circular arc structures.

In architecture, the decomposition of an object is usually mandatory, and depends on the dimensions of the fabricated shape. Conversely, generic shapes were deliberately decomposed into small pieces to create a puzzle-like structure in Lo et al. [2009] and Xin et al. [2011].

2.2 Illustrative Methods

The aim of illustrative methods is to fabricate an illustrative approximation of an input digital model.

Illustrative methods are generally designed to employ materials and devices that are very popular and inexpensive. Since the fabrication process does not require a sophisticated device, a number of inexpensive, accessible servicing companies have recently flourished. The interest in these technologies is testified by the recent release of software tools devoted to planar slice fabrication procedures (such as Autodesk 123DMake [Autodesk 2013]).

For example, Mori and Igarashi [2007] proposed a sketching interface to design plush toys. Li et al. [2010, 2011] put forward a strategy to automatically fabricate pop-up models made of paper. Pop-up models can remain in two different states: open (showing the modeled shape) and closed (reduced to a simple sheet of paper). A method to fabricate a three-dimensional shape illustrated through a stack of colored slices was reported by Holroyd et al. [2011]. Finally, several methods [Mitani and Suzuki 2004; Shatz et al. 2006; Massarwi et al. 2007] represent the input model through a set of foldable strips (usually made of paper), which can be glued together to create a layered 3D representation.

McCrae et al. [2011] create shape abstractions arranging planar slices to optimize the perception of the original object. This method allows nonorthogonal slices, however, it is not designed for the fabrication of tangible objects and problems of the assembly of these slices have not been investigated.

Recently, Hildebrand et al. [2012] proposed a method to semi-automatically fabricate objects made up of planar slices. Altough this method produces a wide range of visually appealing results, unfortunately, it does not fit well with complex geometries (models with a high degree of asymmetry or even complex topology) and it favors arrangements of orthogonal slices. Similarly, Schwartzburg and Pauly [2012] allow nonorthogonal slices, but their method tries to retain the simplicity of orthogonally intersecting pieces. Recently Schwartzburg and Pauly [2013] extended their approach to provide a more detailed formulation on the assembly of nonorthogonal slices by dealing with rigidity constraints. Given a set of predefined intersecting slices, Schwartzburg and Pauly [2013] optimize slice positions to restrict the possible movement of each slice, thus maximizing the rigidity of the resulting structure.

However, as demonstrated by the results, our method is capable of automatically sampling planar slices in a visually appealing manner. Our approach captures and represents the global structure of complex objects, providing, at the same time, a fabrication strategy that meets the physical rigidity constraints.

3. AN OVERVIEW OF THE COMPLETE PIPELINE

Our fabrication pipeline, as shown in Figure 3, has the following steps.

(1) As input, we get a triangle mesh with a cross-field defined on its surface (see Figure 3(a)). We obtained the cross-field using the method proposed in Bommes et al. [2009] with the symmetrization of Panozzo et al. [2012].
(2) We sample a set of planar polylines that lies on the original surface (see Figure 3(b)). These polylines need to be oriented consistently with the cross-field and uniformly distributed on the surface of the object. At the same time, the polylines need to conform to specific constraints thus ensuring the stability of the final structure. This step is detailed in Section 5.
(3) The polylines are transformed into a set of ribbon-shaped slices (see Figure 3(c)). These profiles are obtained through a sequence of boolean operations performed in a 2D space (using ClipperLib [Johnson 2013]).
(4) We derive the interlocking mechanism to produce a physically stable structure. At the same time we provide the sequence of inserting gestures that make up the assembly procedure. This step requires some slices to be split/carved (highlighted by the close-up in Figure 3(d)). This step is detailed in Section 6.
(5) Each slice is then converted to a vectorial representation and organized into sheets ready for automatic laser cutting (see Figure 3(e)).
(6) Finally the slices are assembled by following the sequence specified by our system (see Figure 3(f)). The derivation of the assembling sequence is detailed in Section 7.

4. INTERLOCKING PLANAR SLICES

In this section we provide an overview of the basic concepts regarding interlocking mechanisms between planar slices. For a more general discussion on interlocking shapes, see Séquin [2012].

For the sake of simplicity, consider the simple situation of two perpendicular slices fitting together (see Figure 4). One slice moves along a line parallel to the intersection between the two slices, to fit with the other one which is fixed (this is the typical configuration of waffle meshes). For each piece we create a rectangular slit at the

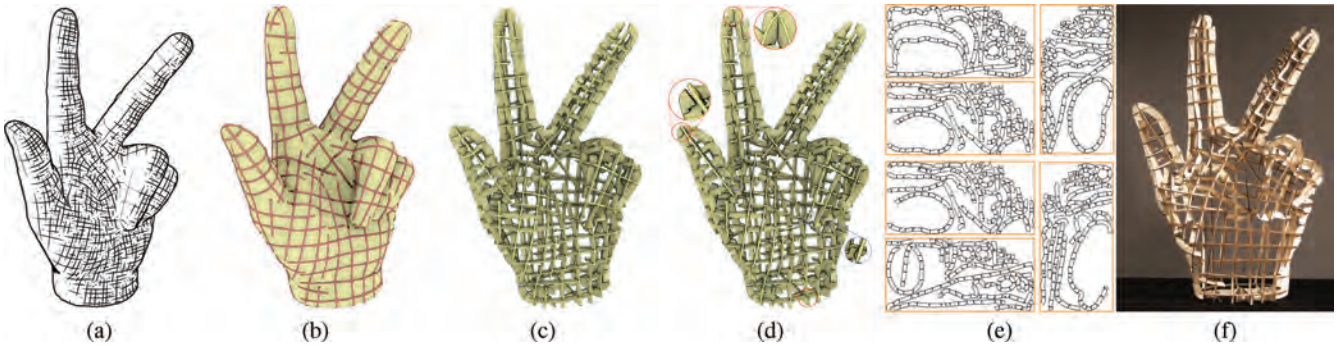


Fig. 3. A complete overview of our fabrication pipeline: (a) We get as input a triangle mesh and an associated smooth (possibly symmetric) cross-field; (b) we sample a set of well-distributed field-oriented planar polylines; (c) the polylines are transformed into ribbon-shaped slices; (d) the slice structure is modified to ensure that the final structure is physically achievable; (e) the slices are transformed into 2D vectorial profiles that are laser cut; (f) the pieces are assembled manually by following the instructions.

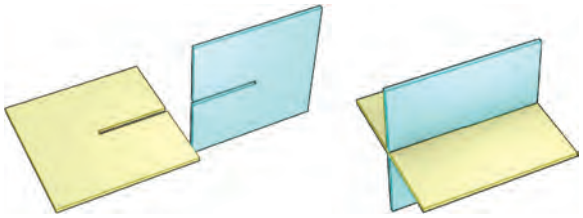


Fig. 4. The classical situation of two connected slices: for each piece we create a rectangular slit in correspondence with the intersection line.

intersection line. The width of the slit must be equal to the width of the material used to create the slicing structure.

This classical, well-known configuration is built on two hard constraints.

Orthogonality constraint. The angle between each pair of intersecting slices must be a right angle.

Parallelism constraint. For each pair of intersecting slices, the insertion movement is parallel to the segment defined by their intersection.

Conforming to these constraints means that the slice arrangement is mostly arranged as an axis-aligned grid, the well-known *waffle*-shaped configuration.

Unfortunately, orthogonality and parallelism constraints have several modeling limitations. These limitations produce serious artifacts, especially for an input shape with a low degree of axis alignment. Obviously, this reduces the range of possible shapes to which this method can be applied.

To overcome this problem (instead of increasing the sampling rate) we explicitly relax these two constraints.

4.1 Relaxing the Orthogonality Constraint

The traditional slit insertion forces the two slices to be orthogonal to each other. This assembling mechanism is solid and strong because it relies on a tight grip of the slits around the slices, which ensures a firm interlock of the two pieces. If the two slices are not orthogonal, the slit has to be widened by the factor λ

$$\lambda = (|\tan(\pi/2 - \alpha)| + 1) \cdot \tau, \tag{1}$$

where τ is the slice thickness and α is the angle between the two slice planes.

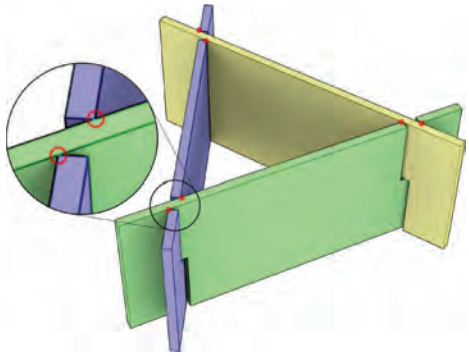


Fig. 5. Three interlocked slices are rigid and tightly connected, although the slices are not orthogonal and the wide slits are not tightly fitted onto the surface of the other slice. The red dots denote where the slices are pressed/forced against each other, such that the resulting friction ensures the stability of the structure.

On the other hand, if we consider arrangements consisting of multiple slices, the solidity of the grip can be guaranteed by a simple triangular arrangement (see Figure 5) or, alternatively, by four slices interlocked together with nonparallel intersections (see Figure 7). In the latter case, the rigidity derives from the fact that a nonorthogonal slit is like a hinge and the four connected slices form a four-bar linkage [McCarthy and Soh 2000]. Any spatial linkage formed by four links and four hinged joints, when in general position, is a highly constrained (rigid) mechanical system. Section 5 outlines how we exploit this mechanism to ensure stability in the final structure.

4.2 Relaxing the Parallelism Constraint

Just allowing the angle between slice planes to deviate from 90° is not sufficient to deal with all the possible real scenarios. Indeed, as illustrated in Figure 6, when a slice (the green one) has to be inserted over four existing nonparallel slices (the blue ones), the direction of insertion will definitely not be parallel to some of the intersections. In these cases the slit has to be enlarged so that it can accommodate the insertion movement. The size and shape of the widened slit (trapezoidally-shaped) depend on the chosen direction for the insertion.

Guaranteeing that the inserted piece has a firm grip is important, so an insertion direction that is parallel with at least one of the

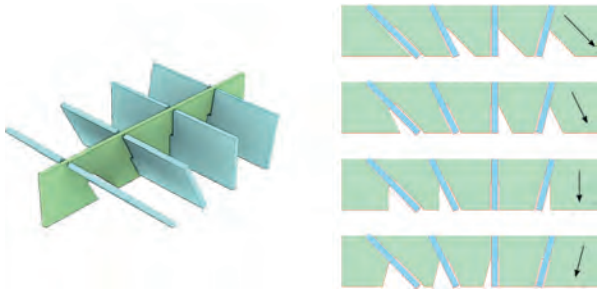


Fig. 6. The shape of the slit widening depends on the insertion direction. The divergence of the green slice is the maximum angle between the various intersection segments when the best insertion direction is chosen. On the right we show how the slit widening varies when different insert directions are chosen.

intersection segments is required, so that at least one of the slits holds the other piece steadily.

To increase the overall rigidity, arrangements that limit the slit widenings are clearly preferable. The size of the slit widening also depends on the order in which we insert the slices. In the example shown in Figure 6, we could have avoided any widening by simply placing the slices in a different order: for example, by inserting the four blue slices one at a time on the green slice. An even more complex example is shown in Figure 7 where four slices are interlocked together. Note that, given the ordering shown in the figure, just a single slit widening is enough to assemble the structure. To quantify how well a slice can be inserted over a set of existing slices we introduce the concept of *divergence*. Given a slice s that is inserted over a set of slices $s_1, \dots s_n$, let $\ell_i = s \cap s_i$ be the intersection segment formed between the slice s and the i-th slice; we define the divergence Λ of the slice s with respect to $s_1, \dots s_n$ as

$$\Lambda(s) = \min_i \left(\max_{j \neq i} \textsc{Angle}(\ell_i, \ell_j) \right). \tag{2}$$

In practice $\Lambda(s)$ denotes the maximum slit widening that we are forced to make even when the best slice for the perfect slit is chosen. For the example in Figure 6, the divergence of the green slice is the angle indicated in the second row of the right part of the figure.

4.3 Exploiting Oblique Slice-to-Slice Arrangement

By relaxing the orthogonal and insertion constraints we considerably increase the resulting expressive power. However, this additional degree of freedom needs to be carefully tuned to ensure that the final structure is physically stable. This entails optimizing the overall structure. Thus:

—the physical stability for a given slice arrangement is influenced by the shape of the slits. As the slits become larger, there is less friction between the pieces, thus reducing their physical stability. When the slit between two pieces is not enlarged, then we have a *perfect plug*.

—the shape of the slit is directly related both to the position of the slice and its insertion direction. As the slices become less and less perpendicular and, likewise, as the divergence between the insertion direction and intersection segment increases, the slit increases in size.

Our framework must be general enough to guarantee a correct slice structure for a given, arbitrary placement. This means that the absolute position of slices must be maintained constant, though the insertion directions can be changed.

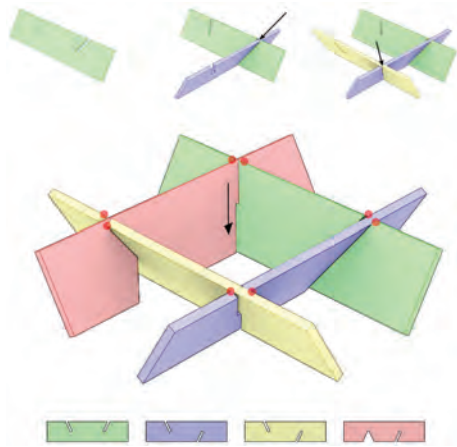


Fig. 7. Four interlocked slices that are rigidly and tightly connected, even though the slices are neither orthogonal nor inserted along a direction parallel to the intersections. Starting from the green slice, the blue and yellow slices are inserted one by one onto the previous slice along the intersection line (no slit widening needed). The last pink slice is inserted over two nonparallel slices, so widening is required. The red dots denote contact points.

From an overall purely aesthetic perspective, the final slice structure does not depend on the sequence of gestures needed to assemble it. We only have to ensure the existence of a valid mounting sequence. Then, for a given set of slices, we optimize the insertion direction in order to increase the overall stability of the structure.

4.4 Ribbon-Shaped Slices

In our framework, we shaped the slices into ribbons, that is, the slices are not solid but they only define the main silhouette of the object. This kind of shape has particularly appealing visual results. Since it is possible to see through the slices, this provides a complete vision of the overall structure. Ribbon-shaped slices have additional advantages in terms of fabrication: there are considerable savings in terms of material and it is very uncommon for three slices to intersect at the same point.

Having three slices intersecting at the same point is, indeed, the standard situation of the approaches based on solid slices (such as Hildebrand et al. [2012]). The solution to these cases consists in decomposing the slices hierarchically using a BSP tree. Unfortunately, this approach means that the slices are excessively fragmented as the sampling resolution is incremented.

This situation may also arise in our approach, especially in a high curvature region, where ribbons degenerate into solid sections of the mesh. In this case, we follow a heuristic similar to Hildebrand et al. [2012]: we remove one intersecting slice by splitting the ribbon that has the smallest area.

5. FIELD-ALIGNED SLICE DISTRIBUTION

We define a set of *ribbons* by inflating *planar* polylines that lie on the surface of the input object.

As mentioned in Section 1 we exploit a smooth feature-aligned cross-field defined over the original surface. Given a manifold, single-connected component mesh and a cross-field, we automatically provide a set of polylines, on the original surface, which conform to the following characteristics.

Cross-Field Alignment. The polylines should be as aligned as possible to the input cross-field. In general, since gradient lines of a cross-field are not planar, it is impossible to provide a perfect alignment (unless we rely on tiny polylines). We must then make a trade-off between length and alignment.

Uniform Distribution. Polylines must sample the original surface as uniformly as possible. Since polylines intersect each other, then the intersection points must also be distributed uniformly on the original surface. This makes the overall shape seem more "regular".

Stability. Once assembled, the fabricated structure must be rigid. As explained in Section 4.1, stability can be ensured locally by the orthogonality of the slices or, globally, by mutual interlocking.

5.1 Alignment to Cross-Field

We designed a simple procedure to trace field-aligned planar polylines. For each face and for each direction, we iteratively trace a polyline, called a *separatrix*, which follows the orientation of the field. Since the cross-field is invariant to 90° rotations, at each tracing step the separatrix follows one of four possible directions which has the smallest angle with the previous direction. At each tracing step, we also fit a plane to the current separatrix (the plane is constrained to lie on the initial face). We perform tracing steps iteratively while the maximum distance between the separatrix and its fitting plane stays below a certain threshold. Additionally, we may also stop the iterative tracing if the separatrix self-intersects.

The final set of *planar* polylines, which we call *traces*, is defined as the intersection between the mesh and the fitting planes. The extremes of each trace are chosen according to the extremes of the generating separatrix.

5.2 Distribution Constraints

We formalized a set of constraints between slices to distribute them uniformly on the surface of the object. Given a disk radius r, we sample a set of traces $\Sigma = \{t_0, t_1, \ldots, t_n\}$ generating a set C of intersections c_j such that:

—for each $c_i, c_j \in C$: $D(c_i, c_j) > r$;

—for each $x_i \in t_i, x_j \in t_j$:
$D(x_i, x_j) < r \rightarrow$
$\exists\, c_k \in t_i, t_j : D(x_i, c_k) < r \bigvee D(x_j, c_k) < r$,

where $D()$ is the geodesic distance on the original surface. In practice, we search for traces whose intersections are well spaced and so that the geodesic distance between traces is larger than r (except in a neighborhood of the intersections). An example of the uniform distribution of polylines on the surface is shown in Figure 8.

Figure 9 shows a mesh sampled at different radius resolutions. Obviously the higher resolution (small values of r) increases the details of the final model.

5.3 Stability Constraints

In order to keep the final structure stable, the slice arrangement must be a single-connected component.

Moreover, the slices should be almost orthogonal to each other. Indeed, orthogonality provides a good grip for the interlocking mechanism, by minimizing the slit widening.

We consider a slice *stable* if:

—it is the first slice placed on the structure;

—or it has a perfect fit with at least one other stable slice. We consider two slices to be in a perfect fit if the intersection between their planes is in between $[\pi/2 - \delta, \pi/2 + \delta]$;

Fig. 8. The constraint used to guarantee an even distribution of the traces. Gray disks represent intersection distances, while the red disks show the distances between points that are far from the intersections.

—or the slice is interlocked in a rigid substructure (see Section 4.1, following the intuition of the triangular configuration in Figure 5).

5.4 The Sampling Strategy

We designed a simple algorithm to produce a slice arrangement that conforms to the constraints we mentioned before.

We build a candidate set by collecting two traces for each face (corresponding to each orthogonal direction of the cross-field). We then assign a priority value to each candidate trace. The priority of a candidate trace is the maximum length without violating the distribution constraints.

Initially we place the longest trace, and since it is the first one it is consequently stable. Then, we iteratively search for the longer trace which, when placed, would become stable.

By following this simple greedy strategy, we add candidates one by one, until no further trace can be inserted.

5.5 Global Regularization

Finally, we improve the distribution of the traces with a global regularization step in order to balance the space between slice intersections.

Given a trace with its intersection points, we evaluate the *optimal position* of each intersection point. Given an intersection point p_{int} its optimal position is the one that minimizes the squared sum of distances with the surrounding intersections (or endpoints). After we have calculated the optimal points, each trace is slightly moved to approach the optimal points. This operation is executed only if distribution and stability constraints are not violated.

We repeatedly execute optimization operations until the trace displacements become lower than a certain threshold.

A sequence showing the placement and optimization of slices is shown in Figure 10.

6. FROM RIBBONS TO ASSEMBLABLE SLICES

The planar polylines defined over the surface in the previous sections can be easily transformed into ribbons by simple extrusion.

However, if we consider a set of generic intersecting slices, there are several situations where physical assembly is impossible. For example, it is impossible to interlock two closed rings without opening at least one of them. In relation to this specific problem, Figure 11 shows a typical situation: three orthogonal ribbons, each one intersecting the other two in two different points. In this case the slices must be decomposed into at least four pieces leaving only one annular ribbon. We refer to the situation where two ribbons intersect in two different points as *multiple intersections*.

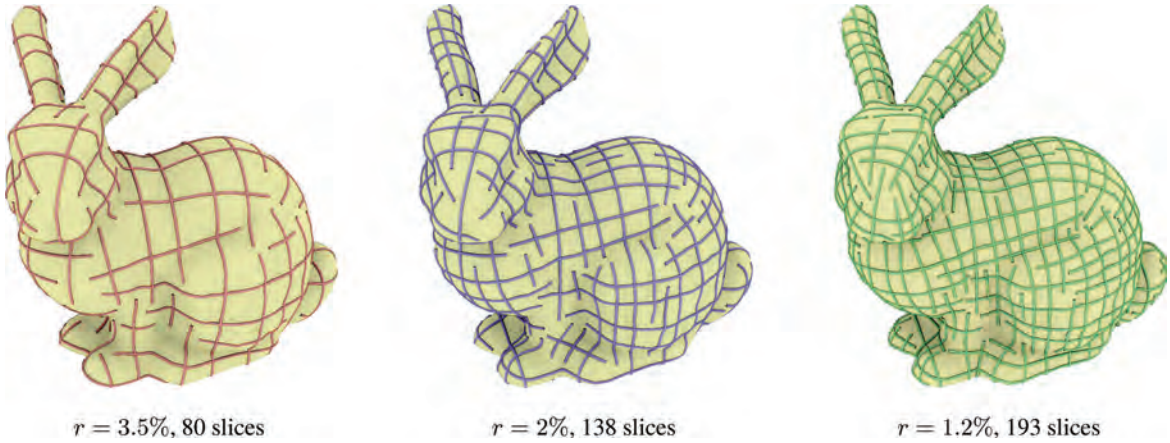


Fig. 9. The Bunny model sampled at different radius resolutions. Sampling radius r is given as a percentage of the diagonal of the model's bounding box.

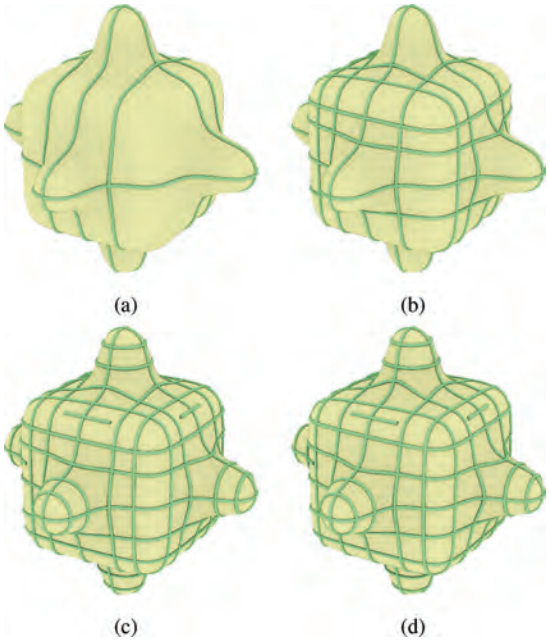


Fig. 10. A sequence of the slice sampling procedure: (a); (b) show two intermediate steps of the slice sampling procedure, composed of 6 and 12 slices respectively; (c) the final slice structure composed of 33 slices and its global regularization (d).

Let us assume that we have a set $S = s_0, \ldots, s_n$ of planar ribbons that approximates a given 3D surface M. We aim to transform S into a set $S' = s_0, \ldots, s_m$ of ribbons such that:

(1) for each pair of ribbons s_1 s_2, the intersection $s_1 \cap s_2$ is a proper segment ℓ with exactly one of the two endpoints lying over the surface M;
(2) we have a proper assembly sequence, such that the resulting *divergence* is lower than a given threshold.

Under the aforesaid constraints, we are able to create the slit mechanisms described in Section 4 and, in order to fulfill them, we use the following two-step procedure which:

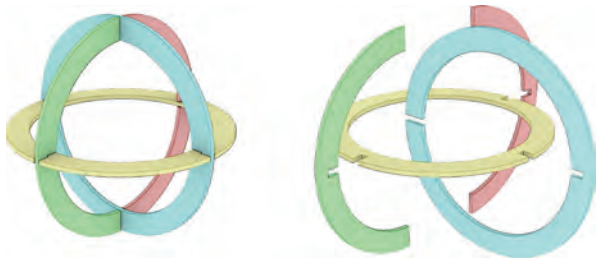


Fig. 11. Three interlocked looping ribbons must be split into four pieces so that they can be untangled.

—removes multiple intersections that limit the assembly procedure;
—minimizes the *divergence* by shuffling the slice order or if necessary by splitting some of the ribbons.

In the following sections we first introduce all the basic concepts behind the process, and then provide a more detailed description of each step.

6.1 Slice Graph

We model the relations between slices in the arrangement structure using a directed graph. Each node s_i of this graph represents a slice. Each arc corresponds to a physical intersection between two slices (and has to be transformed into a slit mechanism). The direction of each arc represents the priority in the partial ordering of the assembly sequence, for example, the arc $s_i \rightarrow s_j$ means that the piece s_i must be plugged into s_j, which should already have been assembled.

Three simple examples of slice graphs with the corresponding slice arrangements are shown in Figure 12.

A valid slice graph must be acyclic. A cycle in the slice graph involves plugging one slice onto another slice that still needs to be inserted (in some geometric cases this may still be feasible by assembling all the pieces simultaneously), but this is obviously not desirable.

The orientation of the arcs in the *slice graph* can significantly affect the shape of the slit widenings, as described in Section 4 and shown in the last two rows of Figure 12 where the different arc orientations generate different slit widenings; the configuration in the middle row needs two slit widenings, while the bottom row needs only one.

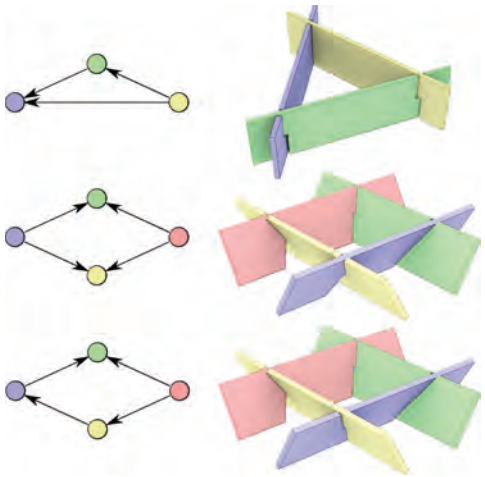


Fig. 12. The two slice graphs corresponding to the slice arrangements shown in Figures 5 and 7. The last two rows show two different arc orientations for the same slice arrangement: the slit widenings are affected by the orientation.

6.1.1 *Finding a Good Sink Set.* Initially we must select a *sink* set, that is, the initial set of disconnected, independent slices into which the remaining slices are inserted one by one. Intuitively, the sink set of a slice graph represents the *ribs* of the whole structure which we try to preserve in the various processing steps. More formally, we search for the sink set that is composed of a maximal independent set of nodes and exhibits the maximum number of arcs/relations. Unfortunately finding this optimal sink set is closely related to the problem of finding the maximum independent set of nodes in a graph: an NP-hard problem. For practical purposes, we verified that it is sufficient to randomize the procedure in order to build a maximal independent set (we randomly add nodes until the set is maximal), repeat it for a limited time, and then pick the best candidate. We found that for a typical set of slices (100 pieces), 10k to 100k attempts (a few hundred msecs of computing time) are sufficient to get a stable sink set.

6.1.2 *Optimizing the Graph.* Once the sink set has been defined, we need to sort all the remaining nodes. In order to provide a good initial order, we sort all the nonsink nodes according to their maximal divergence between each pair of intersection segments. The idea is to minimize the variance of the insertion directions and their divergence once the arcs have been oriented.

Starting from this initial ordering, we swap the direction of each arc if this reduces the divergence between the insertion direction and intersection segment. We follow a greedy approach by swapping the arc that produces the greatest divergence improvement. Simultaneously, we reject any swap operation that would introduce cycles into the graph. The result of the optimization process is shown in Table I, which highlights how the graph optimization process improves the quality of the interlocking between slices. The table reports the number of slices that are perfect fits (i.e., slices with a divergence equal to zero) and the number of slices with a significant divergence (i.e., larger than 45 degrees).

6.2 Intersection Graph

Given a set of ribbons during the process of making it physically achievable, we need to control the degree of solidity of the assembled structure.

Table I. Slice Graph Optimization Results

The slice graph optimization allows us to increase the number of slices that make perfect fits (all the insertion directions are parallel) and to reduce the slices whose divergence is higher than a given threshold.

Model	Slice	Perf. Fit after	Perf. Fit before	>45 after	>45 before
Man	112	71	54	6	17
Hand	123	82	68	0	26
Bimba	196	134	110	4	22
Ico	90	70	58	0	0

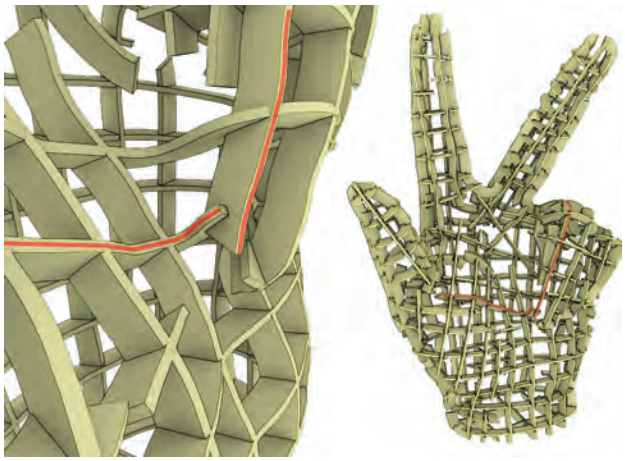


Fig. 13. A close-up of an improper intersection in the Hand model. The two ribbons marked in red have an intersection that does not touch the original surface.

For this purpose let us consider the *intersection graph*. Each node represents a ribbon intersection and an arc represents a slice that embeds two adjacent intersections.

We exploit the concept of *isoperimetric number* [Bobkov et al. 2000] (or Cheeger constant) $h(G)$ of a graph $G = \{V, E\}$, a common measure of the presence of bottlenecks in a graph. The isoperimetric number $h(G)$ is defined as

$$h(G) = \min_{0<|G|\leq \frac{n}{2}} \frac{|\partial(U)|}{|U|}, \tag{3}$$

where the minimum is over all nonempty sets $U \subset V$ of at most $n/2$ vertices and $\partial(U)$ is the *edge boundary* of S, that is, the set of edges with exactly one endpoint in U. In practice $h(G)$ becomes small when a significant portion of the graph is connected to the rest of the graph by just a few arcs.

6.3 Splitting a Ribbon

Given two slices s_1, s_2 with intersection segments $\ell_1, \ldots, \ell_k$, we can improve the set of ribbons by using a *split* operation *Split*(s_1, ℓ_j) which modifies s_1 so that it no longer intersects s_2 along ℓ_j. The splitting operation *Split*(s_1, ℓ_j) is performed by carving out from s_1 all the points at a distance lower than λ from ℓ_j (e.g., taking into account the relative orientation between s_1 and s_2, as specified by Eq. (1)). This operation may split a slice into two separate components or, if the ribbon is a loop, it may open it.

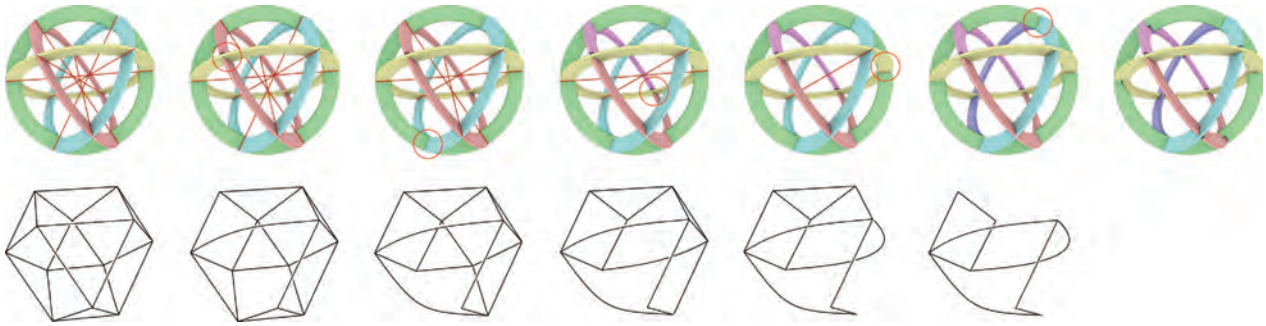


Fig. 14. An arrangement containing multiple double intersections (indicated by red lines) is corrected by means of repeated split operations (indicated with red circles). In the bottom row we show the intersection graph at each step of the process. The top-right image shows the arrangement when all the remaining six intersections are transformed into slit mechanisms.

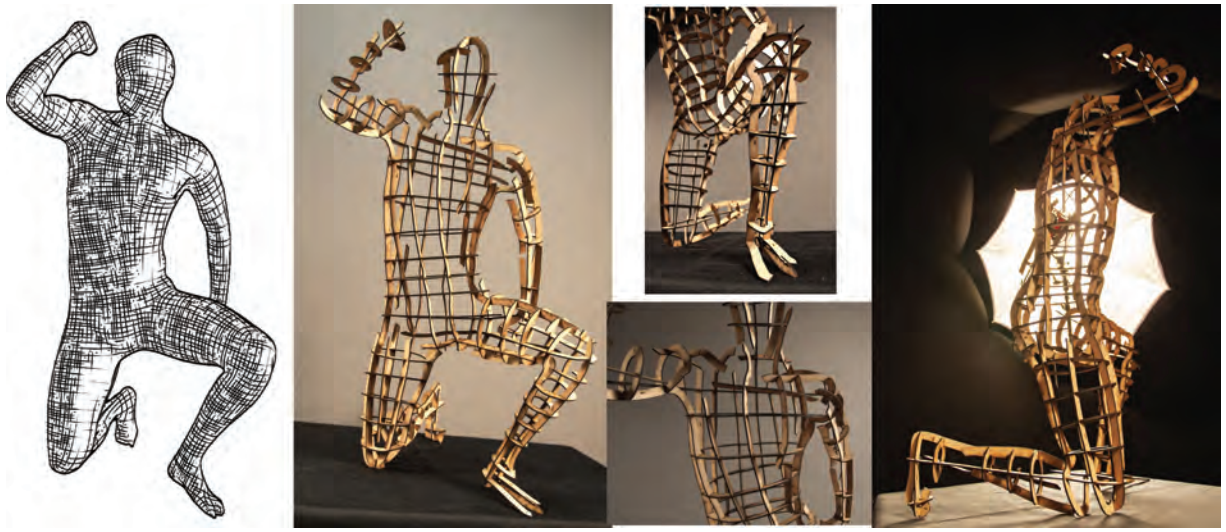


Fig. 15. The Kneeling Human model. The model is composed of 140 slices.

6.4 Removing Improper Intersections

At the very beginning of the process we clean out all the improper intersections from S, for example, all the intersection segments ℓ between two slices s_1, s_2 that do not intersect the surface of M. These intersections do not correspond to any intersections of the generating polyline and are caused only by the intersections of the inner extrusion of the polylines. We simply remove all of them by applying two split operations for both the involved slices $Split(s_1, \ell)$, $Split(s_2, \ell)$. In all the encountered examples there are only a few of these improper intersections and, once removed, we ignore their contribution for the rest of the process. In Figure 3(d) the two blue circles highlight the ribbons that were processed for removal of improper intersections. Figure 13 shows a close-up of one of these improper intersections: the two ribbons marked in red have an intersection that does not touch the original surface and therefore does not correspond to an intersection between the originating traces.

6.5 Removing Double Intersections

There are two main reasons for splitting a ribbon:

—to remove double intersections;

—to lower slice divergence.

First, we remove all the double intersections, that is, pairs of slices s_i, s_j whose intersection is not a single segment ℓ, but it is composed of two (or more) segments. A typical situation is depicted in Figure 11.

To clean out a double intersection, we have to carve out a portion of the slice from one of the two slices around the intersection. There is generally a choice of four different carvings (one for each slice/intersection pair). We opt for the split operation that maximizes the resulting isoperimetric number. If there are many slice splittings that lead to the same isoperimetric number, we split the nonsink slice that has the largest number of intersections with other slices.

We keep the slices in the sink intact because they were chosen specifically to increase the rigidity of the structure. Similarly, of the nonsink slices, we pick the one that will remain connected as much as possible with other slices.

Figure 14 shows an example of this process for a small arrangement made up of nonorthogonal looping ribbons on a sphere. The top row of the figure shows how the arrangement evolves during the process. The red circle highlights the result of the last split operation. The red lines highlight the double intersections that are still present in the arrangement. The last image in the top row shows the slice arrangement after transforming the remaining six intersections into slit mechanisms (machining tolerances are exaggerated

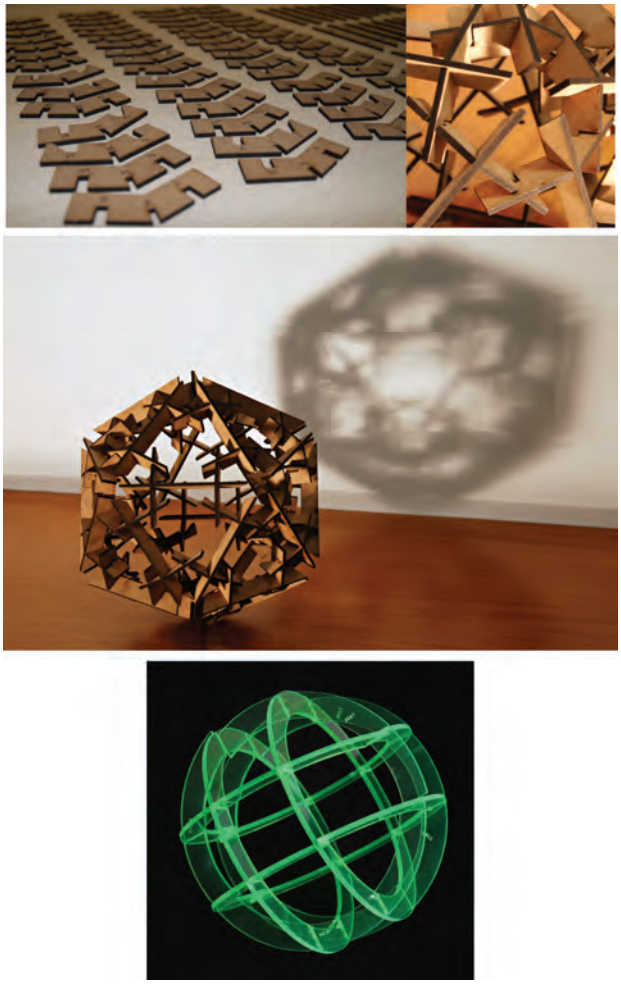


Fig. 16. A simpler slice arrangement (rather than following a cross-field) has been tested to assemble an icosahedroan and a sphere (which has been built using plexiglass).

for sake of image readability). At the beginning the first sink set has just one random ribbon (in this case the yellow one). Each ribbon intersects every other ribbon in two points, so there are six double intersections. The intersection graph corresponding to each step of the process is shown in the bottom row of the figure. At the beginning, the intersection graph is equivalent to the edges of a cuboctahedron and its isoperimetric number is 8/6, that is, the most fragile set of intersections has six intersections from which there are eight connections to other intersections.

We start with a sequence of five split operations and we remove the double intersections. Then the only slice that remains untouched is the original sink, two of the other ribbons have been split twice thus generating four ribbons and the last one has been split only once, thus remaining a connected component. At this point in the process there are no more double intersections and the whole structure is still rigid (see Section 4.1: each slice is involved in a four-cycle of nonparallel intersections).

6.6 Lowering Divergence by Splitting a Slice

Once all the double intersections have been removed and the slice graph has been optimized, we can still improve the overall arrangement by splitting those slices with a high divergence which could cause huge slit widenings. In general, when we have a slice with high divergence we can split it along one of its intersection segments. Of all the possible splitting operations that significantly minimize the divergence, we pick the operation that maximizes the resulting isoperimetric number.

Fig. 17. Our algorithm applied to the Hand model. The arrangement is composed of 122 pieces.

Looking again at the final arrangement in Figure 14 there is a slice with a high divergence which causes slit widening. We could remove this widening by splitting the slice, but this would lead to significant loss of rigidity. In fact, with another split, we would fail to satisfy the rigidity conditions described in Section 4.1.

In Figure 3(d) the three red circles highlight some of the split operations that were performed in order to remove double intersections (the two top red circles) and to lower the divergence (bottom red circle).

7. ASSEMBLING PROCEDURE

To facilitate the assembly procedure we provide basic references: all the slices and slits are labeled so that matching between pieces is unambiguous. We derive an appropriate assembling sequence as follows.

The *slice graph* optimization steps described in Section 6.1 generate a partial ordering which is tailored to minimize the divergence of the slices. Starting from this relation we want to generate a total ordering that is easy to assemble in the real world. We thus use a greedy procedure which, starting from the fully assembled slice arrangement, removes at each step the slice s_i that satisfies the following conditions.

(1) the isoperimetric number of the *intersection graph* of $S \setminus s_i$ is maximum (i.e., we remove the slice that leaves the structure as robust as possible);
(2) of all the slices with the minimal $h()$, s_i has the smallest number $o(s_i)$ of outgoing arcs in the *slice graph*;
(3) of all the slices with the minimal $h()$ and $o()$, s_i is the closest (in terms of Euclidean distance) slice to s_{i-1}.

In practice, given the fact that we consider $h(S)$ as a measure of the robustness of the structure, we try to find an assembly order that

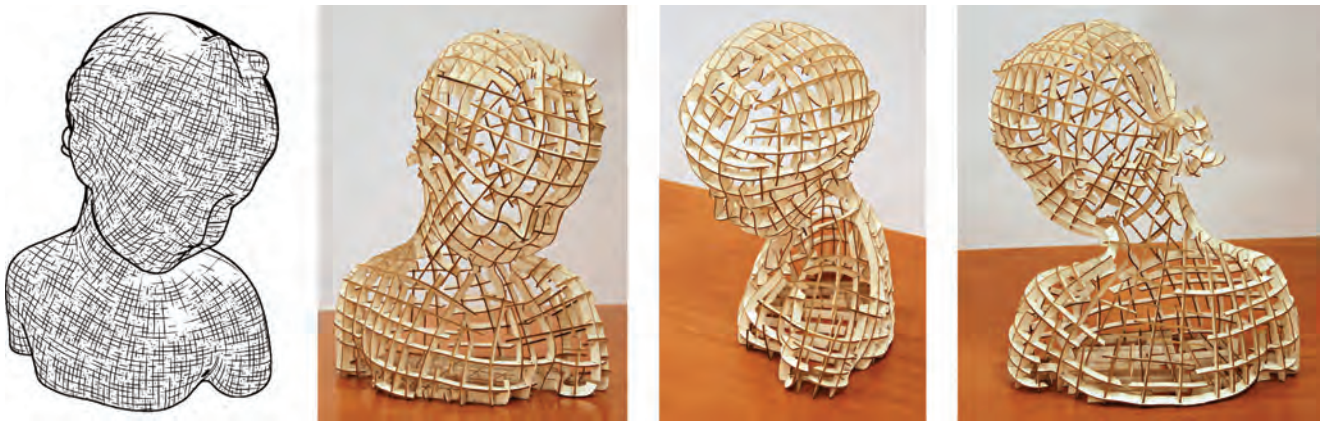


Fig. 18. Our algorithm applied to the Bimba model. The arrangement is composed of 178 pieces.

keeps the structure reasonably solid at each step, and in ambiguous cases, we proceed by adding the slice that has the most intersections with the already assembled structure and if possible close to the previous slices. This ordering is used to label both slices and slits.

8. RESULTS

We tested our method with several models from the Stanford 3D Scanning Repository (Bunny) and the AIM@SHAPE Repository (Hand, Bimba, and Kneeling Human). All the results presented in this article have been generated automatically.

If a cross-field is not available we may simply arrange slices procedurally. As an example, two configurations approximating an icosahedron and a sphere are illustrated in Figure 16.

We successfully applied the entire pipeline described in Section 3 to approximate input geometries with an associated feature-aligned cross-field as input. These structures are shown in Figures 15, 17, and 18. It took from about one to three hours to manually assemble each final model, with most of the time spent searching for the next slice. Once assembled, the resulting models were physically stable. Exploiting an input cross-field has several advantages over axis-aligned approaches, such as Hildebrand et al. [2012] (this comparison is shown in Figure 2). In addition, the cross-field can be further optimized in a preprocessing step to increase the quality of the results (see Figure 2).

Although the entire process is completely automatic, users can perform some simple editing operations to obtain a more visually pleasing result at the end of the process. Users can suggest which slice should be inserted in the sink set and force the split of a particular slice. We used the first option in the Bunny, preferring a vertical orientation of the sink slices, which is much easier to assemble.

9. CONCLUSIONS AND FUTURE WORK

We have proposed a novel method for the automatic fabrication of an illustrative representation of a given geometry made up of interlocked planar slices. We have shown the effectiveness of our method both in terms of illustrative quality and physical stability. To the best of our knowledge, no existing fabrication paradigms are able to represent such complex objects.

Our method is particularly efficient in terms of production costs. In fact, the production costs scale with the surface of the object since slices are sampled almost uniformly over the surface. In addition, due to the slice decomposition, mesh joinery is also suitable for the production of medium-scale objects.

A useful extension of our framework would be to automatically generate effective instructions to simplify the manual assembly procedure, for example, a packing strategy that could preserve the partial ordering of the model to facilitate the search for the next piece.

9.1 Limitations

Although the range of shapes that we can efficiently approximate is wide, our method suffers from minor limitations. We did not account for the presence of other slices that could obstruct a straight insertion. However, in our experience, due to the ribbon shape of the slices, this never constitutes a serious limitation.

Moreover, we did not consider the physical issues regarding gravity and the position of the barycenter and the resulting stress acting on each individual slice. Again, in our experience, given the rigidity of the material, we had no stability problems for any of the assembled structures shown in the article.

ACKNOWLEDGMENTS

We also thanks Giuliano Kraft and Tv@Area of the CNR Research Area of Pisa for the support in the of the production of the paper videos.

REFERENCES

Autodesk. 2013. 123D make. http://www.123dapp.com/make/.

B. Bickel, M. Bacher, M. A. Otaduy, H. R. Lee, H. Pfister, M. Gross, and W. Matusik. 2010. Design and fabrication of materials with desired deformation behavior. *ACM Trans. Graph.* 29, 3, 63:1–63:10.

P. Bo, H. Pottmann, M. Kilian, W. Wang, and J. Wallner. 2011. Circular arc structures. *ACM Trans. Graph.* 30, 101, 1–11.

S. Bobkov, C. Houdr, and P. Tetali. 2000. Lambda and infinity, vertex isoperimetry and concentration. *Combinatorica* 20, 2, 153–172.

D. Bommes, B. Levy, N. Pietroni, E. Puppo, C. Silva, M. Tarini, and D. Zorin. 2012. State of the art in quad meshing. In *EG'12 State of the Art Reports*, M.-P. Cani and F. Ganovelli, Eds., EuroGraphics Association.

D. Bommes, H. Zimmer, and L. Kobbelt. 2009. Mixed-integer quadrangulation. *ACM Trans. Graph.* 28, 3, 77:1–77:10.

P. Cignoni, E. Gobbetti, R. Pintus, and R. Scopigno. 2008. Color enhancement for rapid prototyping. In *Proceedings of the 9th International Symposium on Virtual Reality, Archaeology and Cultural Heritage (VAST'08)*. EuroGraphics Association, 9–16.

D. Dimitrov, K. Schreve, and N. De Beer. 2006. Advances in three dimensional printing state of the art and future perspectives. *Rapid Prototyp. J. 12*, 136–147.

Y. Dong, J. Wang, F. Pellacini, X. Tong, and B. Guo. 2010. Fabricating spatially-varying subsurface scattering. *ACM Trans. Graph.* 29, 62:1–62:10.

M. Eigensatz, M. Kilian, A. Schiftner, N. J. Mitra, H. Pottmann, and M. Pauly. 2010. Paneling architectural freeform surfaces. *ACM Trans. Graph.* 29, 4, 45:1–45:10.

C.-W. Fu, C.-F. Lai, Y. He, and D. Cohen-Or. 2010. K-set tilable surfaces. *ACM Trans. Graph.* 29, 4, 44:1–44:6.

M. Hasan, M. Fuchs, W. Matusik, H. Pfister, and S. Rusinkiewicz. 2010. Physical reproduction of materials with specified subsurface scattering. *ACM Trans. Graph.* 29, 4, 61:1–61:10.

A. Hertzmann and D. Zorin. 2000. Illustrating smooth surfaces. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'00)*. ACM Press/Addison-Wesley, New York, 517–526.

K. Hildebrand, B. Bickel, and M. Alexa. 2012. Crdbrd: Shape fabrication by sliding planar slices. *Comput. Graph. Forum* 31, 583–592.

M. Holroyd, I. Baran, J. Lawrence, and W. Matusik. 2011. Computing and fabricating multilayer models. *ACM Trans. Graph.* 30, 187:1–187:8.

A. Johnson. 2013. Clipper library 5.1.6- An open source freeware polygon clipping library. http://www.angusj.com/delphi/clipper.php.

F. Kalberer, M. Nieser, and K. Polthier. 2007. Quadcover- Surface parameterization using branched coverings. *Comput. Graph. Forum* 26, 3, 375–384.

X.-Y. Li, T. Ju, Y. Gu, and S.-M. Hu. 2011. A geometric study of v-style pop-ups: Theories and algorithms. *ACM Trans. Graph.* 30, 4, 98:1–98:10.

X.-Y. Li, C.-H. Shen, S.-S. Huang, T. Ju, and S.-M. Hu. 2010. Popup: Automatic paper architectures from 3D models. *ACM Trans. Graph.* 29, 4, 111:1–111:9.

K.-Y. Lo, C.-W. Fu, and H. Li. 2009. 3D polyomino puzzle. *ACM Trans. Graph.* 28, 5, 157:1–157:8.

F. Massarwi, C. Gotsman, and G. Elber. 2007. Papercraft models using generalized cylinders. In *Proceedings of the 15th Pacific Conference on Computer Graphics and Applications*. IEEE Computer Society, 148–157.

W. Matusik, B. Ajdin, J. Gu, J. Lawrence, H. P. A. Lensch, F. Pellacini, and S. Rusinkiewicz. 2009. Printing spatially-varying reflectance. *ACM Trans. Graph.* 28, 5, 128:1–128:9.

J. M. McCarthy and G. S. Soh. 2000. *Geometric Design of Linkages*, Vol. 11. Springer.

J. McCrae, K. Singh, and N. J. Mitra. 2011. Slices: A shape-proxy based on planar sections. *ACM Trans. Graph.* 30, 6, 168:1–168:12.

J. Mitani and H. Suzuki. 2004. Making papercraft toys from meshes using strip-based approximate unfolding. *ACM Trans. Graph.* 23, 3, 259–263.

Y. Mori and T. Igarashi. 2007. Plushie: An interactive design system for plush toys. *ACM Trans. Graph.* 26, 45:1–45:8.

D. Panozzo, Y. Lipman, E. Puppo, and D. Zorin. 2012. Fields on symmetric surfaces. *ACM Trans. Graph.* 31, 4, 111:1–111:12.

N. Pietroni, M. Tarini, O. Sorkine, and D. Zorin. 2011. Global parameterization of range image sets. *ACM Trans. Graph.* 30, 6, 149:1–149:10.

H. Pottmann, Q. Huang, B. Deng, A. Schiftner, M. Kilian, L. Guibas, and J. Wallner. 2010. Geodesic patterns. *ACM Trans. Graph.* 29, 4, 43:1–43:10.

N. Ray, W. C. Li, B. Levy, A. Sheffer, and P. Alliez. 2006. Periodic global parameterization. *ACM Trans. Graph.* 25, 1460–1485.

N. Ray, B. Vallet, L. Alonso, and B. Levy. 2009. Geometry aware direction field processing. *ACM Trans. Graph.* 29, 1:1–1:11.

Y. Schwartzburg and M. Pauly. 2012. Design and optimization of orthogonally intersecting planar surfaces. In *Computational Design Modeling*, C. Gengnagel, A. Kilian, N. Palz, and F. Scheurer, Eds., Springer, Berlin, 191–199.

Y. Schwartzburg and M. Pauly. 2013. Fabrication-aware design with intersecting planar pieces. *Comput. Graph. Forum* 32, 2pt3, 317–326.

C. H. Sequin. 2012. Prototyping dissection puzzles with layered manufacturing. In *Proceedings of the Fabrication and Sculpture Track, Shape Modeling International Conference.*

I. Shatz, A. Tal, and G. Leifman. 2006. Paper craft models from meshes. *Vis. Comput.* 22, 825–834.

M. Singh and S. Schaefer. 2010. Triangle surfaces with discrete equivalence classes. *ACM Trans. Graph.* 29, 4, 46:1–46:7.

T. Weyrich, P. Peers, W. Matusik, and S. Rusinkiewicz. 2009. Fabricating microgeometry for custom surface reflectance. *ACM Trans. Graph.* 28, 3, 32:1–32:6.

S. Xin, C.-F. Lai, C.-W. Fu, T.-T. Wong, Y. He, and D. Cohen-Or. 2011. Making burr puzzles from 3d models. *ACM Trans. Graph.* 30, 4, 97:1–97:8.

Received May 2013; revised October 2013; accepted October 2013

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.*

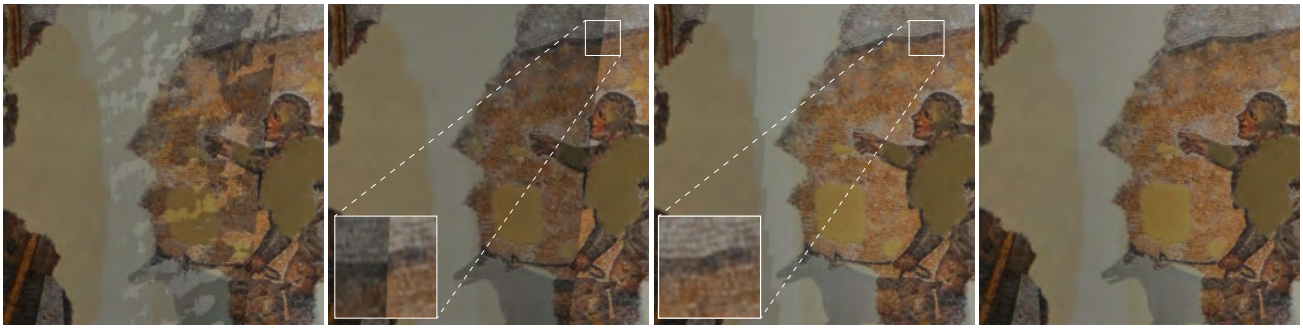


Fig. 1. (a) Penetration artifacts caused by rendering two meshes with z-buffering. (b) Any overwriting order of the meshes in the overlap area resolves the penetration artifacts. However, heavy artifacts occur at the transition of the meshes. (c) Our novel image-space stitching solution, and (d) an intensity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

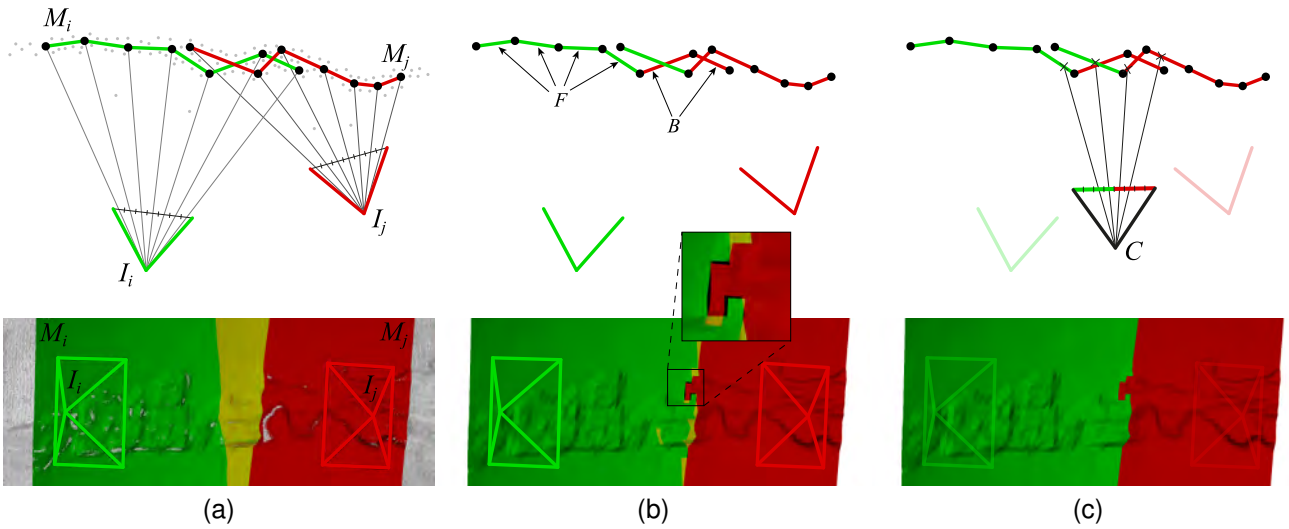


Fig. 2. Overview of our pipeline. (a) Meshes are generated by rendering depth maps from image cameras. The meshes M_i and M_j are color-coded by their respective images and additively blended in the overlap area (yellow). (b) Each of the meshes is textured by all the input images. Besides, each mesh 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)).

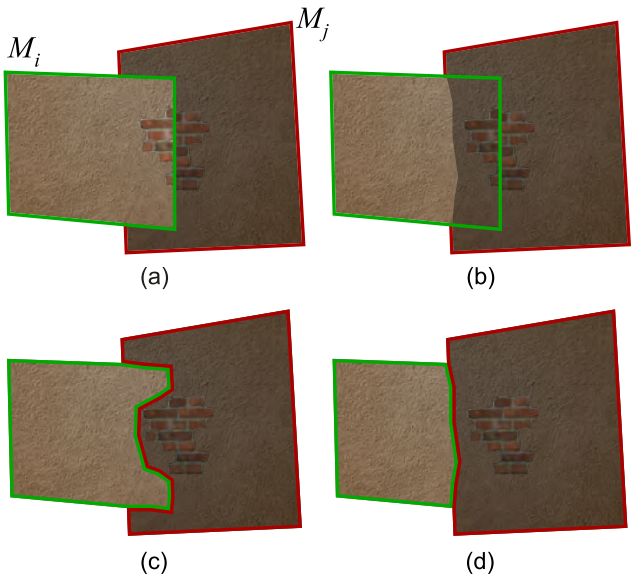


Fig. 3. Illustration of misalignments at mesh borders and our solution strategy. (a) and (b) show two cases of a graph-cut optimization of M_i. (a) I_i (i.e., the corresponding image of M_i) is assigned to the border of M_i. (b) I_i is *not* assigned to the border of M_i. In both cases, rendering M_i 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

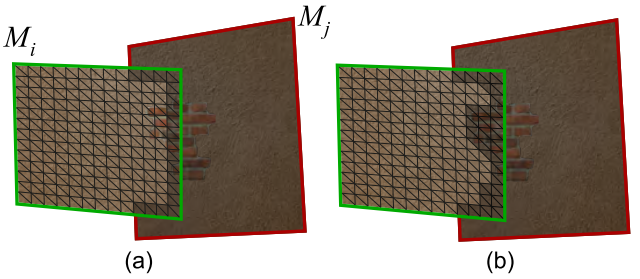


Fig. 4. Handling of mesh borders. (a) Excluding the respective image from the border faces in the labeling of mesh M_i introduces an image seam at the mesh border. The example in (b) illustrates how the image seam is pushed by the labeling towards more homogeneous color regions where artifacts caused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

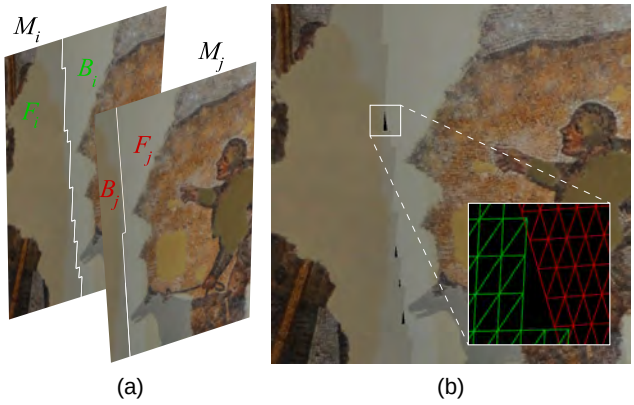


Fig. 5. (a) Two meshes M_i and M_j are binary segmented into foreground and background regions. (b) However, rendering only foreground-labeled regions causes holes due to different mesh discretizations.

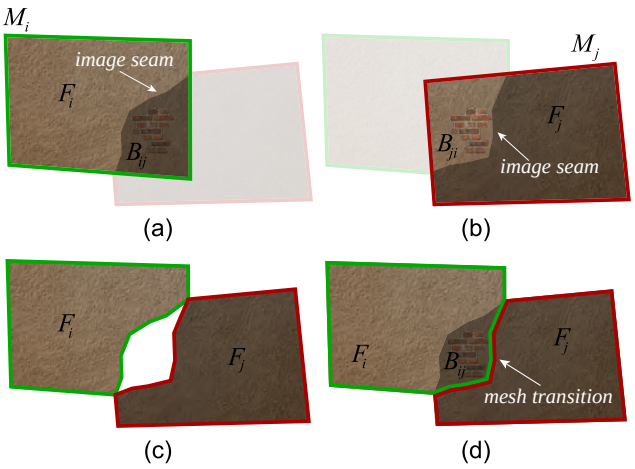


Fig. 6. Illustration of our stitching idea. (a) and (b) show the labelings of the meshes M_i and M_j, respectively. (c) Rendering only foreground regions F_i and F_j does not cover the complete object, due to the different seams in the overlap area. (d) The rendering algorithm resorts to the background-labeled 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

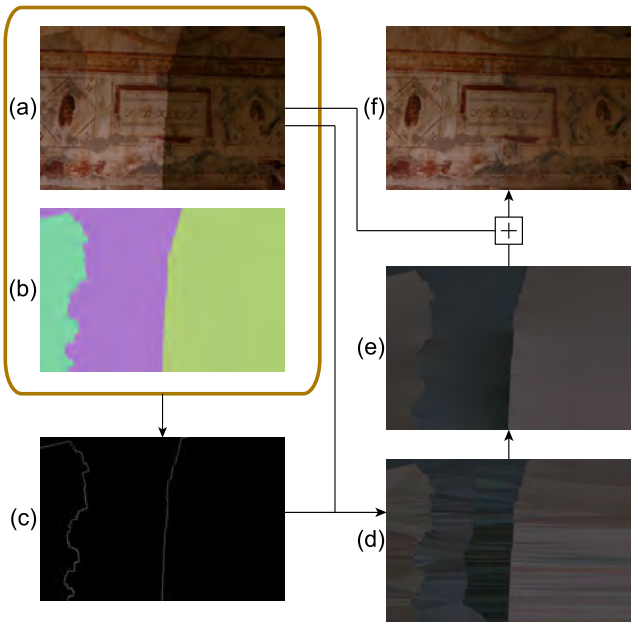


Fig. 7. Levelling pipeline. (a) Output of the texturing pass. (b) Output texture of the stitching pass storing fragment labels. (c) A one-pixel wide border around each connected pixel region of the same label is detected. The color of each border pixel is fixed to the average of its own and that of its 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

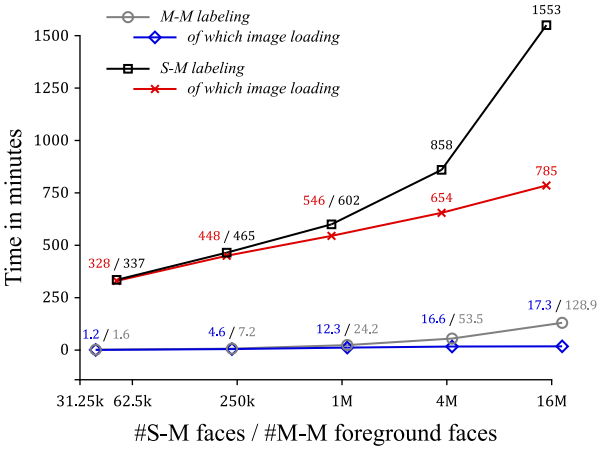


Fig. 8. Plot of the labeling timings of the single-mesh and our multi-mesh approach, applied to the *Hh2 We1* dataset, for increasing mesh sizes.

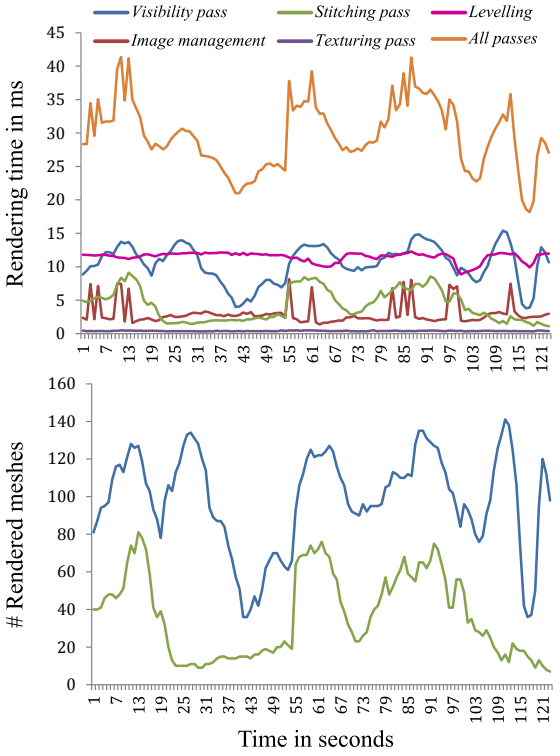


Fig. 9. Rendering timings and numbers of rendered meshes measured during a walkthrough of the *Hh2 We1* model.

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

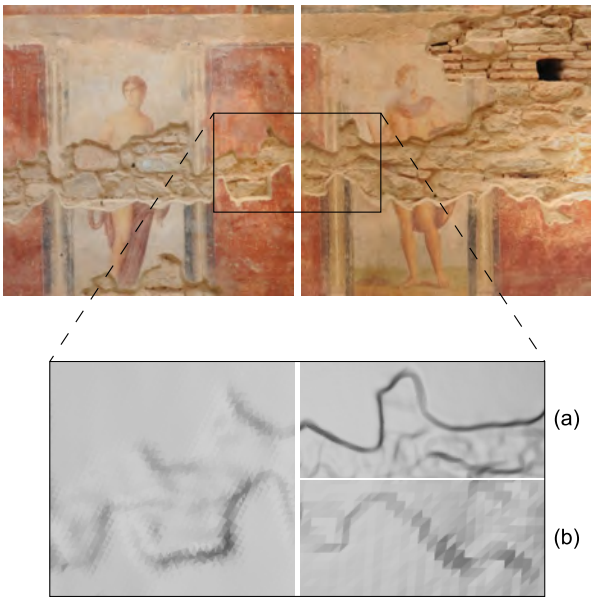


Fig. 10. Comparison of the single-mesh reconstruction (left) and our multi-mesh approach (right, (a)) applied to a part of the *Hh2 We1* dataset. For a fair comparison, the number of single-mesh faces approximately equals the number of multi-mesh foreground faces (∼1M). In the presence of high-resolution 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

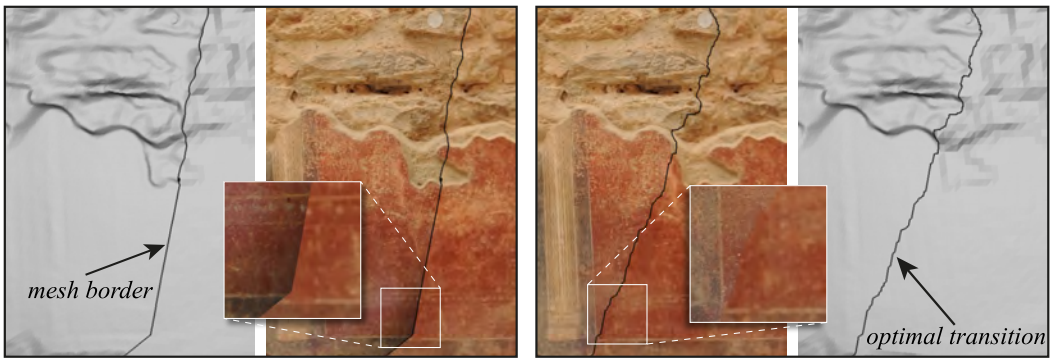


Fig. 11. A close-up view of our rendering without (a) and with (b) the color-smoothness constraint during the computation of the labelings. (b) Artifacts due to lighting variations and registration errors are barely visible along the transition of the meshes, but this comes at a price of an overall lower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

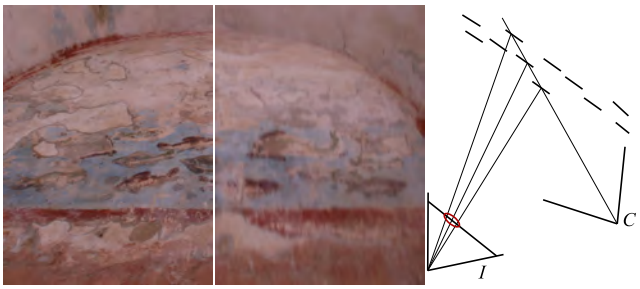


Fig. 12. Side-by-side comparison of our approach (a) and textured splatting (b). (c) demonstrates the dependency of the blurriness of the images generated by textured splatting on the noise and the view direction. C denotes the user camera and I an input image camera.

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.

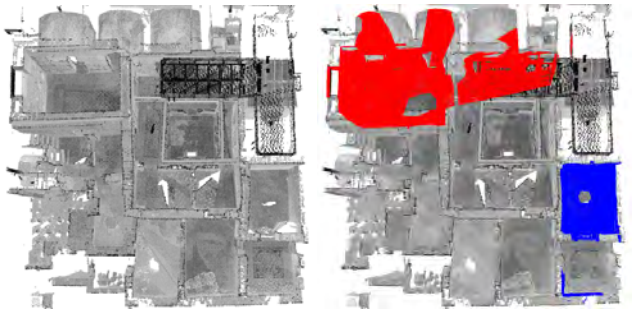
7.3 Asymptotic Analysis

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 recomputation. 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.

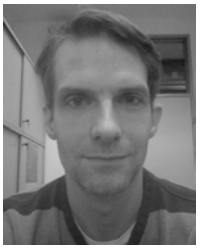
[9] M. Kazhdan and H. Hoppe, "Screened poisson surface reconstruction," *ACM Trans. Graph.*, vol. 32, no. 3, pp. 29:1–29:13, June 2013.

[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.

Multi-Depth-Map Raytracing for Efficient Large-Scene Reconstruction

Murat Arikan, Reinhold Preiner and Michael Wimmer

Abstract—With the enormous advances of the acquisition technology over the last years, fast processing and high-quality visualization of large point clouds have gained increasing attention. Commonly, a mesh surface is reconstructed from the point cloud and a high-resolution texture is generated over the mesh from the images taken at the site to represent surface materials. However, this global reconstruction and texturing approach becomes impractical with increasing data sizes. Recently, due to its potential for scalability and extensibility, a method for texturing a set of depth maps in a preprocessing and stitching them at runtime has been proposed to represent large scenes. However, the rendering performance of this method is strongly dependent on the number of depth maps and their resolution. Moreover, for the proposed scene representation, every single depth map has to be textured by the images, which in practice heavily increases processing costs. In this paper, we present a novel method to break these dependencies by introducing an efficient raytracing of multiple depth maps. In a preprocessing phase, we first generate high-resolution textured depth maps by rendering the input points from image cameras and then perform a graph-cut based optimization to assign a small subset of these points to the images. At runtime, we use the resulting point-to-image assignments (1) to identify for each view ray which depth map contains the closest ray-surface intersection and (2) to efficiently compute this intersection point. The resulting algorithm accelerates both the texturing and the rendering of the depth maps by an order of magnitude.

Index Terms—Point-based rendering, raytracing depth maps, large-scale models

1 Introduction

The high-quality reconstruction and visualization of large scenes from huge amounts of raw sensor data is an important and particularly challenging task in many application areas, ranging from digitization and preservation of cultural heritage, over virtual reality and games, to planning and visualization for architecture and industry. To virtually recreate such scenes, geometry is reconstructed from scanned 3D point-cloud data and commonly textured from registered high-resolution photographs taken at the original site.

In practice, computing a high-quality texturing from such images is a non-trivial task due to image overlaps, varying lighting conditions, different sampling rates and image misregistrations. One potential workflow represents the geometry as a point cloud again and directly texture-maps the resulting point-based surface [1], [2]. However, this approach can exhibit visible artifacts like illumination seams and texture misalignments, which heavily degenerate the visual quality of the result. A more common approach is to convert the point data into a mesh once [3], [4] and then render the scene as a textured mesh, reducing both memory and bandwidth consumption. In order to obtain the required texturing, an image-to-triangle assignment (also called *labeling*) problem has to be solved, for which state-of-the-art methods [5], [6] use a graph-cut based optimization, which provides a homogeneous and high-quality solution. In large-scale scenarios, this is done once in an expensive preprocessing phase, and the resulting textured mesh is then used for efficient rendering. However, these methods are not very flexible – any change or addition in the geometry or image data requires an expensive relabeling of the mesh – and do not scale well due to the time complexity of the global labeling. Moreover, large-scale scenarios require an out-of-core computation of the mesh [7] and its texturing, imposing an additional maintenance overhead.

State of the art: To break down the problem complexity and accelerate the reconstruction and labeling preprocessing, Arikan et al. [8] introduced a *localized* textured surface reconstruction and visualization approach. They employ a set of *Textured Depth Maps* to represent the scene as a collection of *surface patches*, avoiding the reconstruction and maintenance of the whole surface and significantly reducing the optimization costs by labeling only a set of small depth maps instead of a large out-of-core mesh. These patches are triangulated and stitched at runtime, trading a minor increase in rendering time against a huge decrease in preprocessing time. Moreover, the patch-based representation offers both more flexibility and better scalability, since new patches can be added and textured easily without recomputing the whole surface. However, the rendering performance heavily depends on the number of depth maps and their resolution. This introduces a natural bound on the depth-

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

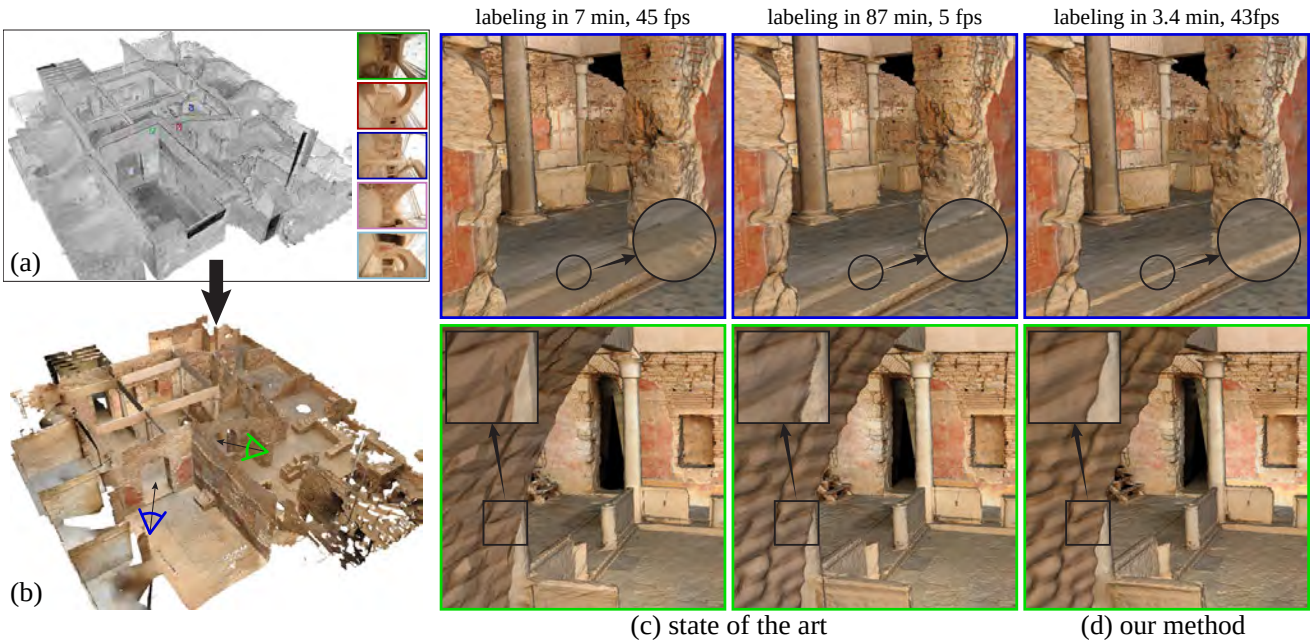


Fig. 1. (a) Point-cloud and image data acquired by a scanner. The data set consists of 682M points and 192 images. (b) Scene overview rendered by our method. (c) and (d) compare the state of the art [8] and our method in terms of performance and quality. The previous approach has to settle with a significantly lower geometric resolution in order to reach the performance of our new method.

map resolution usable to be rendered interactively, thus limiting the achievable geometric quality in the rendered image.

Solution approach: We introduce an *output-sensitive* visualization technique of such a patch-based surface representation. Instead of stitching high-resolution depth maps, which is expensive, we perform a *multi-depth-map raytracing* approach, which efficiently identifies for each view ray the depth map that contains the closest valid ray-surface intersection, and then finds this intersection point. Our method also avoids the labeling of every single depth map in the preprocessing, but instead labels a strongly reduced subset of the original point cloud, which in practice accelerates the labeling process by over an order of magnitude. To obtain high-quality per-pixel labels for texturing, this coarse point set is projected to the screen and its labels are upsampled using a geometry-aware Voronoi decomposition of the depth buffer at runtime.

As our main **contribution over the state of the art**, we propose a novel raytracing approach whose performance is independent of the number and resolution of the depth maps, therefore allowing for a high-quality real-time visualization of large scenes at much higher geometric resolution than the previous approach [8] (Fig. 1).

2 Related Work

The problem of textured scene reconstruction and visualization from large point clouds and photographs has been addressed by several authors.

Point-based rendering techniques like *surface splatting* [9], [10], [11], [12] render the input points as elliptical surface primitives *(splats)*, which are blended to obtain a smooth continuous surface. These methods have been coupled with texturing [1], [2] to obtain a textured point-based visualization of a scene. Texture mapping point-based surfaces avoids a costly large-scale mesh reconstruction, but does not produce optimal point-to-texture assignments. This can produce visible artifacts like texture misalignments and illumination seams.

Mesh-based textured reconstruction techniques achieve a continuous high-quality texturing of the scene by performing a global, graph-cut based optimization of the triangle-to-texture assignments on a single huge mesh [5], [6]. These methods produce high-quality visualizations of large scenes, but require a time-expensive preprocessing for the mesh reconstruction and labeling as well as a large maintenance overhead, making it inflexible to changes and extensions in the data set.

Therefore, Arikan et al. [8] recently proposed a **patch-based reconstruction** approach, which breaks down the meshing and labeling complexity by representing the scene by several surface patches, allowing for both a more efficient preprocessing and a more flexible and scalable data management. Their method generates a set of *textured depth maps* in a preprocessing and stitches them at runtime, which strongly couples the rendering performance with the number and resolution of these depth maps.

Our method builds upon this localized approach for the data representation, but alleviates its performance limitations by introducing a solution for an efficient **raytracing of multiple depth maps**. Finding ray intersections with surfaces represented by two-dimensional range maps has various applications, like rendering soft shadows [13], [14] or reflections [15].

Previous methods for raytracing large-scale scenes depend on the use of spatial acceleration data structures. Reshetov et al. [16] employs a spatial kd-tree to detect scene parts that are guaranteed not to intersect with a collection of view rays. Agrawala et al. [13] proposed a hierarchical ray traversal to skip over large sections of a ray that cannot possibly intersect the scene. Xie et al. [14] raytraces a *multi-layer* depth map to reduce shadowing artifacts. To cope with the additional overhead of searching an intersection point in multiple layers, they introduced a hierarchical intersection test against a quadtree, where each node contains the minimum and maximum depth value of the four child nodes in the layer below. In contrast, we use multiple *single-layer* depth maps covering a scene and employ a labeled coarse subset of the original point cloud to *directly* determine the depth map that is first intersected by a view ray. This is done by splatting the label information of these points into the screen, and *upsampling* their labels to obtain per-pixel labels. The resulting label of a pixel then indicates the depth map to be intersected by the pixel's corresponding view ray.

In the following, we give an overview of our preprocessing and rendering pipeline, and then describe each step of our reconstruction and texturing system in detail.

3 Overview

Our method takes as input a high-density 3D point cloud (denoted by P_{HD}), for example from a laser scanner, and a set of high-resolution photographs $\{I_j\}$ with known camera registrations. We propose a two-phase solution for an efficient high-quality visualization of the data.

In the *preprocessing* phase, we generate high-resolution depth maps by rendering the input point cloud P_{HD} from image cameras (Fig. 2a, Section 4.1), and compute an image-to-point assignment (referred to as *labeling*) only for a small subset $P_{LD} \subseteq P_{HD}$ (Fig. 2b, Sections 4.2 and 4.3), which we will call *proxy points*.

At *runtime*, we reconstruct a high-resolution depth buffer, which stores depth values of the scene as viewed from the user's camera. This is done by first splatting proxy points, and then raytracing the precomputed depth maps, starting from coarse splat positions (Fig. 2c, Section 5.1). In a second step, the labels of P_{LD} are used to obtain an upsampled depth-buffer labeling, which is required for texturing the final output image (Section 5.2).

4 Preprocessing

4.1 Generating the Depth Maps

For each image I_i, we generate a depth map D_i by rendering the original point cloud P_{HD} from the same viewpoint and with the same viewing parameters as I_i. For rendering, we use oriented circular splats as rendering primitives and employ an out-of-core octree data structure [17] to store P_{HD} and stream visible points to the GPU. If point normals are not available, we compute them by fitting a least-square plane to a neighborhood of each point. The splat radii are determined from the density of the rendered points [17].

4.2 Generating the Proxy Points

The proxy points P_{LD} are obtained by sub-sampling P_{HD}. To this end, the octree storing P_{HD} is pruned to contain only its k top-most levels, which correspond to the k lowest levels of detail of P_{HD}. As we will show in Section 6.1, the choice of k is a trade-off between performance and rendering quality. We will also demonstrate that using only a small subset of the original point cloud as proxy points strongly accelerates the subsequent labeling stage, but is still sufficient for a high-quality textured reconstruction from the depth maps at render time.

4.3 Labeling

To obtain a point-to-image assignment, first a set of candidate images of each point $\mathbf{p} \in P_{LD}$ is determined. The image I_i is a candidate of $\mathbf{p}$ if $\mathbf{p}$ is not occluded from the camera view of I_i. In the second step, we pick for each point $\mathbf{p}$ its best-suited candidate image I_j for texturing, i.e., $\mathbf{p}$ is *labeled* with the index j.

This assignment has to consider the quality of the image-to-geometry mapping as well as continuity in the texturing (i.e., avoiding visible artifacts between areas labeled by different images). We solve this problem by a graph-cut based optimization, where the quality and continuity criteria are addressed by a data and a smoothness term, respectively. However, instead of operating on triangles as done in previous approaches, we use the $k$nn-graph built upon the points as input graph for the optimization. We use the same data and smoothness term as in Arikan et al. [8]: For the points, the data term favors orthogonal and close image views. In contrast, the smoothness term penalizes label changes with strong color differences along edges between neighboring points.

5 Multi-Depth-Map Raytracing

In this section, we describe how the precomputed data is used at runtime to obtain a high-quality visualization of the scene. We perform two major steps, *surface*

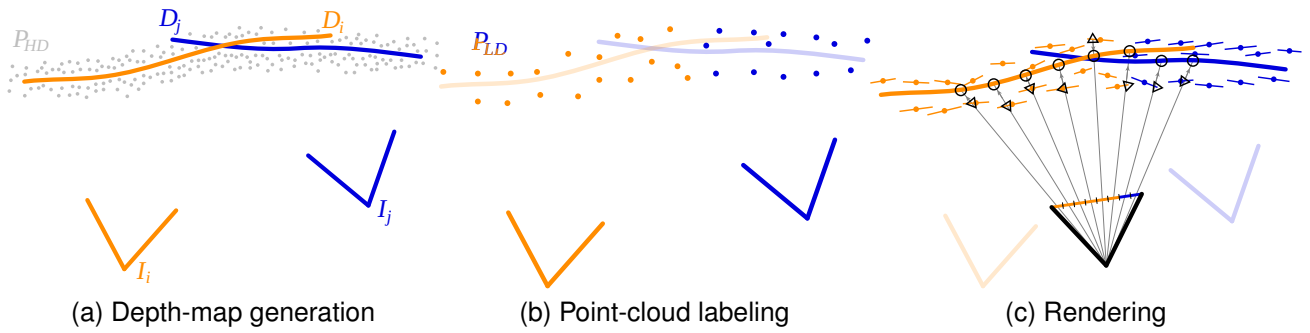


Fig. 2. Overview of our pipeline. (a) High-resolution depth maps are generated by rendering the high-density input point cloud P_{HD} from image cameras. The depth maps D_i and D_j, lifted to 3D, are color-coded by their corresponding images I_i and I_j, respectively. (b) P_{HD} is subsampled, and the resulting low-density point cloud P_{LD} is labeled by the input images, i.e., each point of P_{LD} is assigned to an input image. This concludes the preprocessing phase. (c) Coarse surface positions (marked with $\triangle$) that are equipped with labels are efficiently obtained by splatting points of P_{LD}. Then, starting from these positions, raytracing the respective depth maps yields high-resolution surface positions (marked with $\bigcirc$).

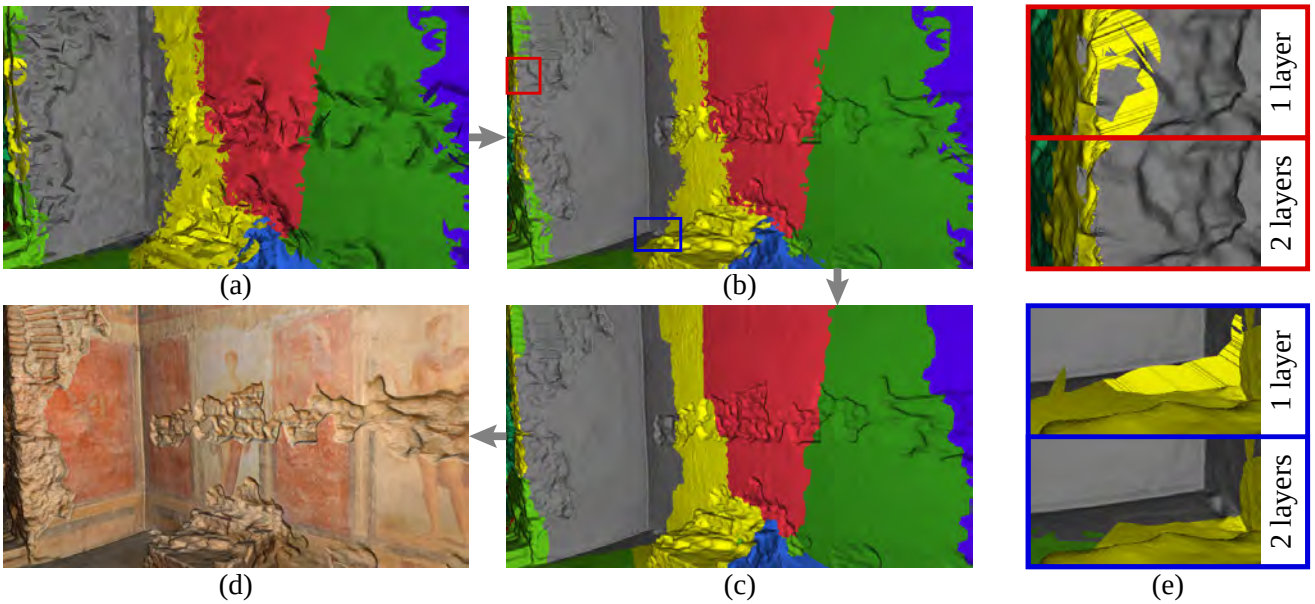


Fig. 3. (a)-(d) Rendering pipeline. (a) Splatting proxy points P_{LD} (color coded according to labels). (b) Raytracing high-resolution depth maps. (c) Per-pixel labeling to be used for texturing. (d) Textured and shaded surface. (e) shows invalid intersections with discontinuity triangles that can occur when raytracing a single depth-map layer along each view ray.

generation and *color mapping*, to render a textured surface.

The surface-generation step first renders P_{LD} as splats to create a depth buffer representing coarse surface positions and a corresponding label buffer (Fig. 3a, Section 5.1.1). For rendering, we employ the same out-of-core data structure [17] that we used to generate the depth maps. Then, starting from these coarse positions, for each pixel the depth map indicated by the label buffer is raytraced in a full-screen rendering pass to produce a high-resolution depth buffer (Fig. 3b, Section 5.1.2).

The following color-mapping step splats P_{LD} again to generate a high-resolution label buffer by upsampling the labels that were output in the first pass (Fig. 3c, Section 5.2.1).

Finally, high-resolution images relevant for texturing are cached to the GPU (Section 5.2.2), and the color of each pixel is retrieved in a full-screen pass by projecting it onto its assigned image based on the depth and label retrieved from the high-resolution depth and label buffers (Fig. 3d, Section 5.2.3).

In the following, we will describe the individual steps of our rendering pipeline in more detail.

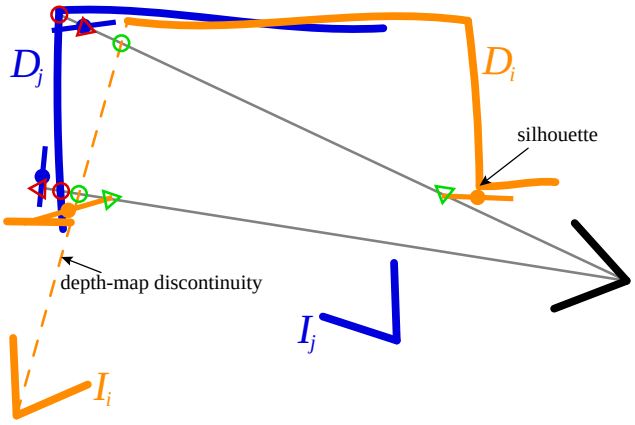


Fig. 4. Multiple label layers for raytracing. Invalid intersections (◯, green) can be caused by initializing raytracing with the front-most splat position (△, green) and its label i. In this case, starting from second-layer positions (△, red) with label j, raytracing D_j produces valid intersection points (◯, red).

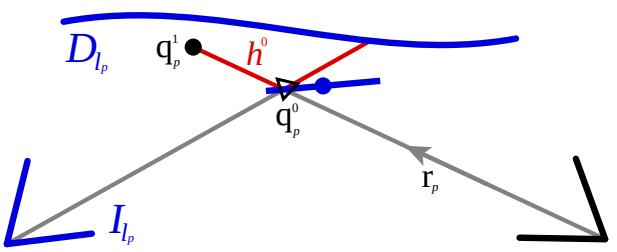


Fig. 5. A single iteration of the iterative search, taking a step of h^0 on $\mathbf{r}_p$. The start position $\mathbf{q}_p^0$ and its label index l_p are retrieved from the closest ray-splat intersection.

5.1 Surface Generation

5.1.1 Visibility Stage

In the first pass, P_{LD} is rendered with z-buffering, writing to a *depth buffer* B_d and a *label buffer* B_l. The generated buffers represent the front-most *label layer*, which will be used in the raytracing pass to compute the intersection points of view rays with depth maps. In particular, a ray cast from the viewpoint through the pixel position $p = (x_p, y_p)$ will intersect the depth map indexed by label $l_p = B_l(x_p, y_p)$, and the intersection search will start at the 3D position $\mathbf{q}_p^0$ corresponding to the depth value $d_p = B_d(x_p, y_p)$.

This fast, *direct* selection technique gives the correct depth map for the vast majority of the view rays in the screen. However, in some cases, the labels in B_l will not correspond to a depth map that contains a valid ray intersection. This mostly happens for proxy points splatted very close to depth-map discontinuities and silhouettes (Figs. 3e and 4). In such a case, we retrieve the depth-map label for the intersection test from the next closer proxy point splat along the ray with a different label. For this, we have to store a second label layer to look up the next depth map for raytracing if no valid intersection point is found in the first depth map (Figs. 3e and 4). To extract this second label layer, P_{LD} is rendered again with z-buffering, and at each pixel p, fragments with label l_p or depth values less than d_p are discarded. The resulting depth and label values are written into two additional buffers. We then extend this approach to multiple layers computed in a depth-peeling fashion [18].

5.1.2 Raytracing Pass

We render a full-screen quad and perform for each screen-space pixel p an iterative search in the high-resolution depth map D_{l_p}, followed by a binary search. The iterative search starts at $\mathbf{q}_p^0$ and uses a stepsize that adapts to the current estimated distance to the intersection. The next point on the ray is computed as follows:

$$\mathbf{q}_p^i = \mathbf{q}_p^{i-1} + h^{i-1} * \mathbf{r}_p, \tag{1}$$

where $\mathbf{r}_p$ is the normalized ray direction. The adaptive stepsize h^{i-1} is calculated as the signed distance of $\mathbf{q}_p^{i-1}$ to D_{l_p} along the line to the center of projection of I_{l_p} (Fig. 5). The distance is signed since the low-resolution depth-buffer value used as initialization can lie in front or behind the high-resolution depth map.

Since $\mathbf{q}_p^0$ provides a sufficiently good initialization, only a few iterations are required (except at oblique angles) to find a pair of points $\mathbf{q}_p^{k-1}$ and $\mathbf{q}_p^k$ enclosing an intersection. In a second step, the interval $[\mathbf{q}_p^{k-1}, \mathbf{q}_p^k]$ is refined by a binary search to find a more accurate approximation $\hat{\mathbf{q}}_p$ of the intersection point.

We then check whether $\hat{\mathbf{q}}_p$ lies on a depth discontinuity of D_{l_p}. For this, we detect the four texels of D_{l_p} (yielding two triangles in 3D) that are nearest to the projection of $\hat{\mathbf{q}}_p$ into D_{l_p}, and assume a discontinuity if the depth disparity between any two triangle vertices is above a user-defined threshold (20cm in our examples). Averaging the two triangle normals also provides us with per-pixel normals, which can be optionally used for lighting effects. In case of a depth discontinuity, raytracing is re-performed to find an intersection with the depth map retrieved from the next label layer (Fig. 4).

The results of the raytracing pass basically refine for each pixel the depth value and – in case of a discontinuity – the label value originally obtained from splatting P_{LD}.

5.2 Color Mapping

5.2.1 Labeling Pass

The aim of this rendering pass is to equip the high-resolution depth data from the previous pass with labels that are suitable for texturing. Unfortunately, we cannot use the label buffer B_l created in the visibility stage as is, since due to the low resolution of P_{LD}, this buffer exhibits non-regular borders between

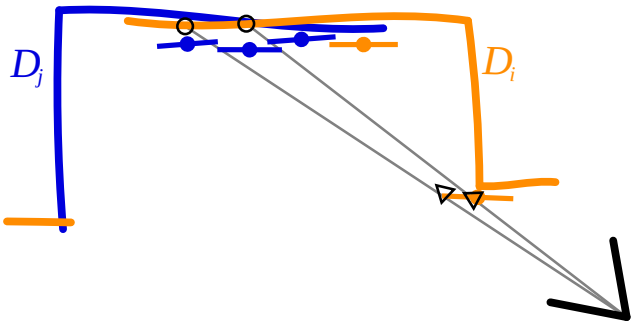


Fig. 6. Illustration of false labels near silhouettes. View rays through the splat at the silhouette have valid intersections with D_i. Therefore, this splat projects its label i to the background, causing corresponding pixels of that region to be assigned the label i instead of j.

differently labeled regions (Fig. 3b) and false labels near silhouettes (Figs. 6 and 8c).

Instead, we compute a Voronoi decomposition of the screen space into equally labeled regions. The seeds of this decomposition are specified by the projection of the points $\mathbf{c}_j \in P_{LD}$ into screen space, and distances between pixels and seed points are measured by the Euclidean distances of the respective points $\hat{\mathbf{q}}_p$ and $\mathbf{c}_j$ in 3D. This way, each pixel will be assigned the label of its closest seed $\mathbf{c}_j$. This results in a high-resolution label buffer with per-pixel labels upsampled from the sparse labeling information in P_{LD}.

In practice, this is implemented by rendering P_{LD} as splats using z-buffering, with the depth value of a splat at pixel p manually set to the 3D Euclidean distance $d(\mathbf{c}_j, \hat{\mathbf{q}}_p)$ between the splat center $\mathbf{c}_j$ and the point $\hat{\mathbf{q}}_p$. This pass stores at each pixel p (corresponding to the surface point $\hat{\mathbf{q}}_p$) the label of $\mathbf{c}_j$ with $j = \text{argmin}_j\, d(\mathbf{c}_j, \hat{\mathbf{q}}_p)$.

5.2.2 Image Management

In this step, we employ an out-of-core streaming technique [8] for continuously caching the currently most relevant images into a GPU texture array, where the relevance of an image is measured by the frequency of occurrence of its label in the updated label buffer.

5.2.3 Texturing Pass

A full-screen quad is rendered to retrieve the color of each pixel p by projecting $\hat{\mathbf{q}}_p$ onto the image indicated by the updated label buffer.

In a last step, we perform an online screen-space leveling method [8] to balance the color intensities between regions textured by different photographs and thus reduce illumination seams in the final output image.

6 Results

We have tested our approach on three different data sets acquired by a laser scanner (Table 1, Fig. 17).

TABLE 1
Scene characteristics.

Model	# Points	# Images
Hanghaus 2 Wohneinheit 6 (Scene 1)	35M	188
Hanghaus 2 Wohneinheit 1 (Scene 2)	682M	192
Centcelles (Scene 3)	1091M	161

Scene 1 and 2 are scans of different building units in terrace house (Hanghaus) 2 in the excavation of ancient Ephesus, while Scene 3 is a scan of the cupola of the Roman villa of Centcelles. In the following, we discuss performance and quality tradeoffs depending on the algorithm's main parameters, and give a detailed analysis of memory consumption, reconstruction error compared to ground truth, and the convergence of the iterative search. Then, we will compare our approach (denoted by DMRT) to the related depth-map triangulation approach (denoted by DMT) in terms of both quality and performance.

All results in this paper were produced on a PC with an Intel i7-4770K 3.50 GHz CPU, 32 GB RAM and NVIDIA GeForce GTX TITAN GPU. A framebuffer resolution of 1280×720 was used in all our experiments and the accompanying video.

6.1 Performance and Quality Tradeoffs

Number of layers. Currently, we extract layers in a depth-peeling fashion [18], which requires a geometry pass for every single layer. Therefore, the choice of the number of layers is a trade-off between rendering performance and quality. Table 2 shows that, even using more than ten layers, DMRT achieves real-time frame rates. For the measurements in this table, we used a proxy point cloud that is sub-sampled from the original point cloud by a factor of 686 (as in Fig. 17). The table also shows a breakdown of the running time of the algorithm by its stages.

Size of proxy point cloud. In our approach, another key criterion for the rendering performance and quality as well as the labeling time is the size of the proxy point cloud. Fig. 7 shows renderings for different parameters. As expected, the number of layers required for a high-quality rendering decreases with growing sizes of proxy point clouds. For the same layer count on the other hand, a DMRT reconstruction with more proxy points results in an increase of the labeling time and a decrease of the rendering performance.

For more performance results, see Section 6.5.

6.2 GPU Memory Consumption

The GPU memory usage of our method is affected by several factors, including the number of input images, the size of the proxy point cloud, the layer count, and the framebuffer resolution.

For all our test scenes, we generated *depth maps* of size 1024×684. Each map consumes 2.8MB of GPU

$\|P_{LD}\|$	4.4M	1M		0.2M	
labeling in	16.4 min	3.4 min		0.8 min	
# layers	3	3	4	4	7
fps	44	54	48	49	36

Fig. 7. Results for different parameters. The red ellipses indicate regions of some artifacts.

TABLE 2
Average performance of DMRT rendering (in ms) for different numbers of layers, measured during a walkthrough of Scene 2.

# layers	1	3	5	7	9	11
visibility stage	2	6	9.8	13.7	17.5	21.2
raytracing pass	1.2	2	2.3	2.7	3	3.3
labeling pass	2	2	2	2	2	2
img. man.	2.5	2.8	2.9	3.1	3.3	3.3
texturing pass	6.3	6.4	6.3	6.3	6.3	6.3
total	14	19.2	23.3	27.8	32.1	36.1
fps	71	52	43	36	31	28

memory (one float per pixel). As described in Section 5.2.2, the *high-resolution input images* are cached in a GPU texture array on demand. We reserved 1GB of GPU memory for them. We resort to *low-resolution images* (of size 256×171) if input images are not available in the texture array. All of these are stored on the GPU, and each requires 0.13MB. As an example, rendering Scene 2 requires 563MB for the 192 depth maps and low-resolution images.

Furthermore, each point of the *proxy point cloud* is represented by six floats for the position and the normal vector, and an integer for the label. A screen-space pixel in a *layer* requires two floats, one for the depth, and the other for the label. Therefore, an optimal DMRT rendering of Scene 2 with $|P_{LD}| = 1$M at a resolution of 1280×720 and five layers (see the accompanying video) occupies an additional 65MB of GPU memory (37MB for the proxy points and 28MB for the layers).

6.3 Ground-Truth Comparison

In order to analyze the reconstruction error of DMRT, we rendered the scene from the viewpoint of one of the image cameras, and compared the color output and depth buffer at different stages of our rendering pipeline to the original image and its corresponding high-resolution depth map, respectively (Fig. 8). This comparison can give a first impression of the

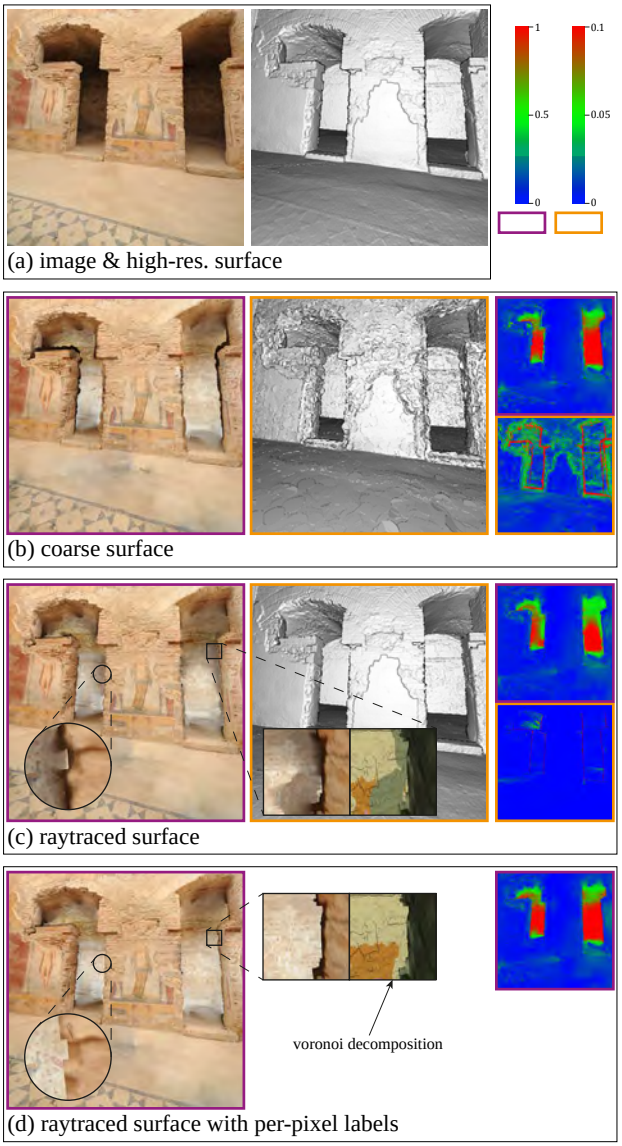


Fig. 8. Analysis of the reconstruction error. The scene is rendered as seen by the image shown in (a). The error is measured as the deviation of the color output and depth buffer at different rendering stages (b)-(d) from the reference image and its corresponding high-resolution depth map, respectively. The color and depth differences are visualized as heat maps shown in the right column.

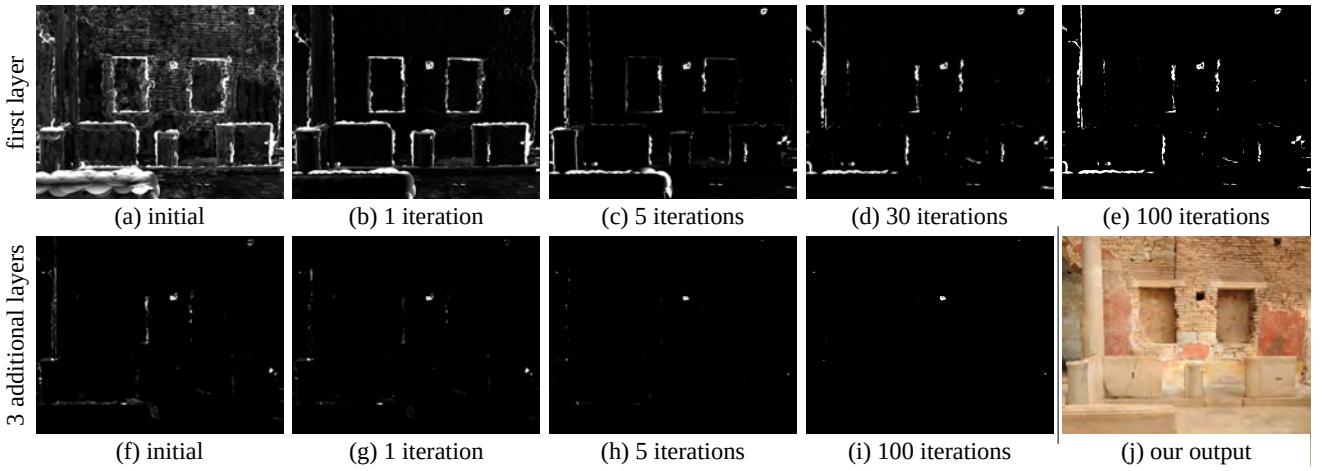


Fig. 9. (a)-(e) show absolute stepsizes of the iterative search in the first layer. For each of the white pixels in (e), where raytracing of the first layer fails, a maximum of three more layers are consecutively searched until a valid intersection is found. (f)-(i) show stepsizes in the layer where raytracing succeeds. Note that for the visualization, all stepsizes are multiplied by 10 and then clamped to the range $[0, 1]$.

reconstruction error, however note that due to different lighting conditions when acquiring the different images, a full match to the "ground truth" is not possible. The rightmost column in Fig. 8 shows color and depth differences as heat maps. For this analysis, we used a sub-sampling factor of 686 ($|P_{LD}| = 1$M) to generate the proxy point cloud, and rendered the scene using four layers. Figs. 8c and 8d show the DMRT reconstruction after a maximum of 100 iterative search and 20 binary search iterations.

A comparison of the heat maps (of depth differences) in Figs. 8b and 8c shows that raytracing reduces the overall depth error. As expected, remaining differences are maximal at oblique angles and silhouettes. However, note that the differences at silhouettes are not generated by our raytracing method. Instead, these occur naturally since the depth map of the image and the raytraced depth maps have different sampling rates of the observed surface, and thus exhibit slight geometric variations at silhouettes.

Interestingly, the overall color error is minimal, except inside the two small rooms. This is because the labeling assigns the points there to images that have better geometric resolution, but were acquired under different lighting conditions than the reference image in Fig. 8a.

Fig. 8c shows that while raytracing resolves the geometry at silhouettes adequately, it generates false labels among these regions by mapping the labels of proxy splats to the background (see also Fig. 6). As we have shown in the accompanying video, these false labels generate ghosting artifacts during animations, and are resolved by our per-pixel labeling step (Fig. 8d).

Fig. 10. Worst-case scenario. (a) shows a poor initialization of the stepsizes of the iterative search, therefore requiring many layers for a high-quality visualization. Our output with four (b) and eleven (c) layers.

6.4 Convergence

In this section, we analyze the convergence of the iterative search with adaptive stepsize, which is responsible for finding a "tight" pair of points enclosing an intersection point to seed the binary search. We also discuss the limits of our rendering method for a synthetically generated scene configuration.

We rendered the scene using the same parameters as in Section 6.3. Figs. 9a-9e show absolute stepsizes of the iterative search in the first label layer. For some pixels, our raytracing failed to find intersections in this layer. These pixels are marked white in Fig. 9e, and for each of them, an intersection point is searched in three additional layers. Figs. 9f-9i show absolute stepsizes in the layer where an intersection point is found.

We perform a total of $c_{total} = \sum_{i=1}^{k} c_i$ iterative search iterations for each pixel, where $1 \leq c_i \leq c_{max}$ is the number of iterations performed in the ith layer. The maximum iteration count in each layer is bound by c_{max} (100 in this example), and k refers to the index of the layer where the intersection is found (or the

TABLE 3
Comparison of the labeling times and rendering performance on Scene 2.

DMT	depth-map res.	256×171	512×342	1024×684
	aver. num. of labels per depth map	46		
	labeling times	7 min	23 min	87 min
	min/avg/max fps	32/45/84	11/17/46	3/5/16
DMRT	depth-map res.	1024×684		
	$\|P_{LD}\|$	1M		
	# labels	192		
	# layers	5		
	labeling times	3.4 min		
	min/avg/max fps	30/43/74		

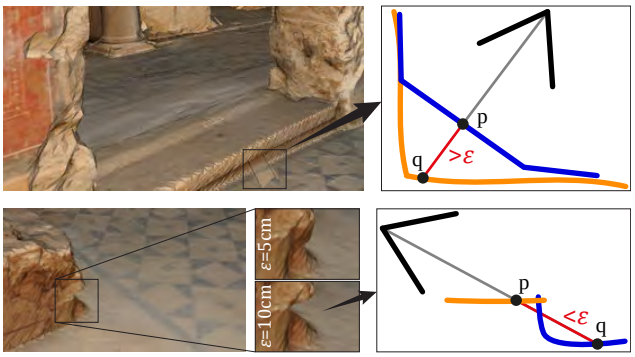


Fig. 11. DMT's stitching artifacts. Top: Due to a small stitching threshold ε, the points $\mathbf{p}$ and $\mathbf{q}$ are considered as non-overlapping by the DMT, leading to the point $\mathbf{p}$ on the low-resolution depth map to be chosen for texturing. Bottom: In DMT, visibility is not resolved for features smaller than the ε threshold. Thus, the invisible point $\mathbf{q}$ can shine through the front surface.

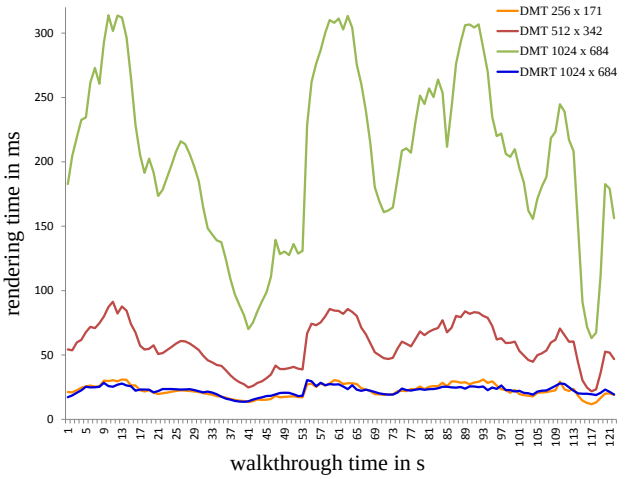


Fig. 12. Comparison of the rendering performance of DMT and our DMRT approach for a walkthrough of Scene 2. Using high-resolution depth maps, our method runs at 43 fps, being on average about an order of magnitude faster than the previous work, which has to settle with a quarter of the resolution to reach this performance.

user-defined maximum layer count).

In practice, iterative search converges in a few iterations to an intersection point, if any. Otherwise, it terminates early if an intersection with a discontinuity triangle is found. In our experiment, the iteration count c_{total} was on average 4.6 over all pixels, and it took the raytracing pass 2.4 ms to complete (including the binary search procedure).

The convergence of the iterative search is only guaranteed if each texel along the projection of the view ray onto the depth map is visited, which is slow if the depth-map resolution is high. The iterative search with adaptive stepsize, on the other hand, proved very efficient in practice to find in a few iterations a pair of points enclosing an intersection point.

In order to see the performance of our raytracing for a poor initialization of the stepsizes (Fig. 10a), we multiplied the splat radii by 2.5, and rendered the scene again. In this scenario, the iterative search required on average $c_{total} = 9.5$ iterations per pixel, and the raytracing pass completed in 4.2 ms. Even though our raytracing was still efficient, four layers was not sufficient to obtain a high-quality result (Fig. 10b). To obtain a comparable result (Fig. 10c) as in Fig. 9j, eleven layers were required, and the raytracing pass performed in 8 ms with $c_{total} = 11$ on average. We see that the most performance-critical part of our rendering pipeline is still the extraction of the layers, while searching for intersections in these layers is quite efficient (see also Table 2).

6.5 Comparison to DMT

Finally, we compare our method to the related depth-map triangulation approach on Scene 2. For this comparison, we used a proxy point cloud of size 1M and five layers for the DMRT approach. Our experiments suggest that this configuration is more than sufficient for a not completely artifact-free, but high-quality DMRT rendering. On the other hand, depending on the chosen stitching threshold, DMT can produce severe artifacts (Fig. 11).

Table 3 compares the labeling times and rendering performance of DMT and DMRT for differently sized depth maps. Since the resolution of the depth maps does not have a direct effect on the performance of DMRT, we used the highest resolution for our approach. The table shows that DMT strongly couples the labeling time and rendering performance to the resolution of the depth maps used to represent the scene. If we aim for an *equal-quality* comparison (Figs. 1c right and 1d), DMT needs to label 192 depth maps of size 1024×684, which takes about 26 times longer (87 min) than labeling the 1M proxy points used by DMRT (3.4 min). While DMT cannot render depth maps of this size in real time anymore (5 fps on average), our new raytracing method is about 9 times

(a) one layer (b) two layers (c) three layers

Fig. 13. (a)-(c) show results of raytracing different number of layers along each view ray. Raytracing a single layer (a) produces severe artifacts (black background pixels), especially near silhouettes. In this example, artifacts produced by raytracing two layers (b) are barely visible. Adding a third layer almost completely removes artifacts.

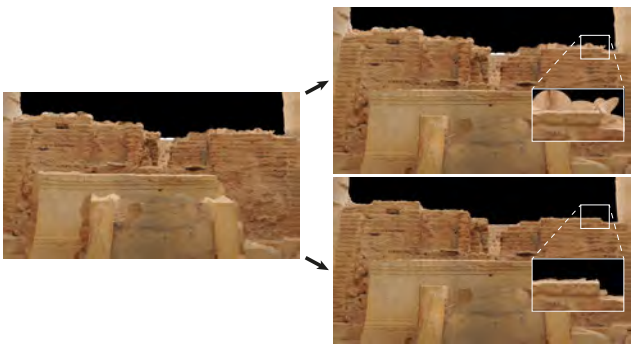


Fig. 14. The left image shows the coarse surface (without raytracing). By using textured coarse positions where raytracing fails, the splats along silhouettes spuriously occlude the background (right top). Therefore, we always discard pixels if no valid intersection with the surface could be found (right bottom).

faster, thus providing a real-time high-quality visualization of the scene (Figs. 1d and 12, see also Table 2). Reducing the depth-map resolution to 256×171 allows DMT to almost match these performance values for labeling and real-time rendering, but noticeably reduces the geometric resolution of the output (Fig. 1c left).

7 Limitations and Future Work

Number of layers We found that extracting a few layers in the visibility stage (Section 5.1.1) is sufficient for high-quality visualizations (Fig. 13). However, in scenes of higher geometric complexity, more layers might be required (e.g., see Fig. 10). At the moment, we use a naive implementation that performs k geometry passes for k layers, which can become inefficient as k increases. In such cases, more elaborate A-Buffer techniques could be incorporated to achieve a multi-layer setup in a single pass [19]. Also, for a few pixels where raytracing fails to find a valid intersection with any of the layers, we show the background color instead of textured coarse surface points (Fig. 14). We

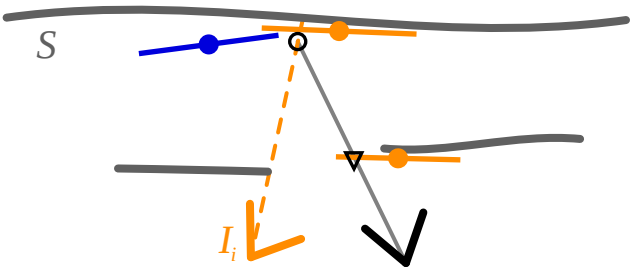


Fig. 15. Sub-sampling issue. The shown view ray intersects a discontinuity edge of I_i. Due to the poor sampling of the surface S by proxy points, there isn't any second layer to search for a valid intersection in this case.

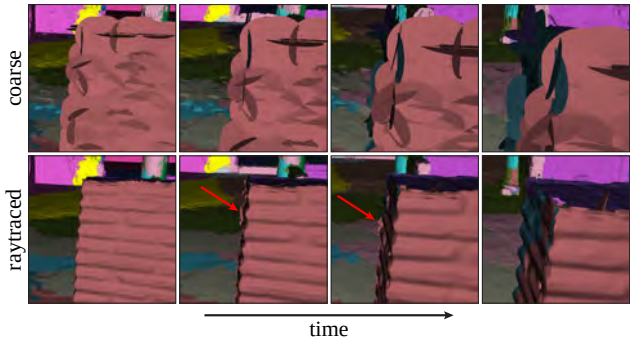


Fig. 16. Label changes under camera motion can lead to view-dependent geometry of silhouettes.

opted for this solution since splats along silhouettes can also occlude the background.

Size of proxy point cloud. As discussed in Section 6.1, the sub-sampling factor is a trade-off between performance and quality. In order to achieve high performance, this factor has to be large enough, but should be small enough to maintain fine surface details. Currently, we discard the highest levels of detail of the input point cloud to obtain proxy points. However, a feature-aware sub-sampling strategy could produce an even better rendering quality, since the generation of the proxy points currently does not take local surface characteristics into account. Fig. 15 illustrates the absence of layers for raytracing, even for a reasonable coverage of the surface by proxy splats.

Motion artifacts. Depth maps can have slightly varying representations of silhouettes based on the viewing angle and distance relative to the observed surface. Thus, label changes under camera motion can lead to raytracing of depth maps with possibly different representations of silhouettes (Fig. 16).

Inherited artifacts. Other rendering artifacts that are inherited from the previous approach [8] are the flickering during animations, and false textures at some silhouettes due to image misregistrations and the noise inherent in point clouds.

Extension. Note that the runtime steps required to

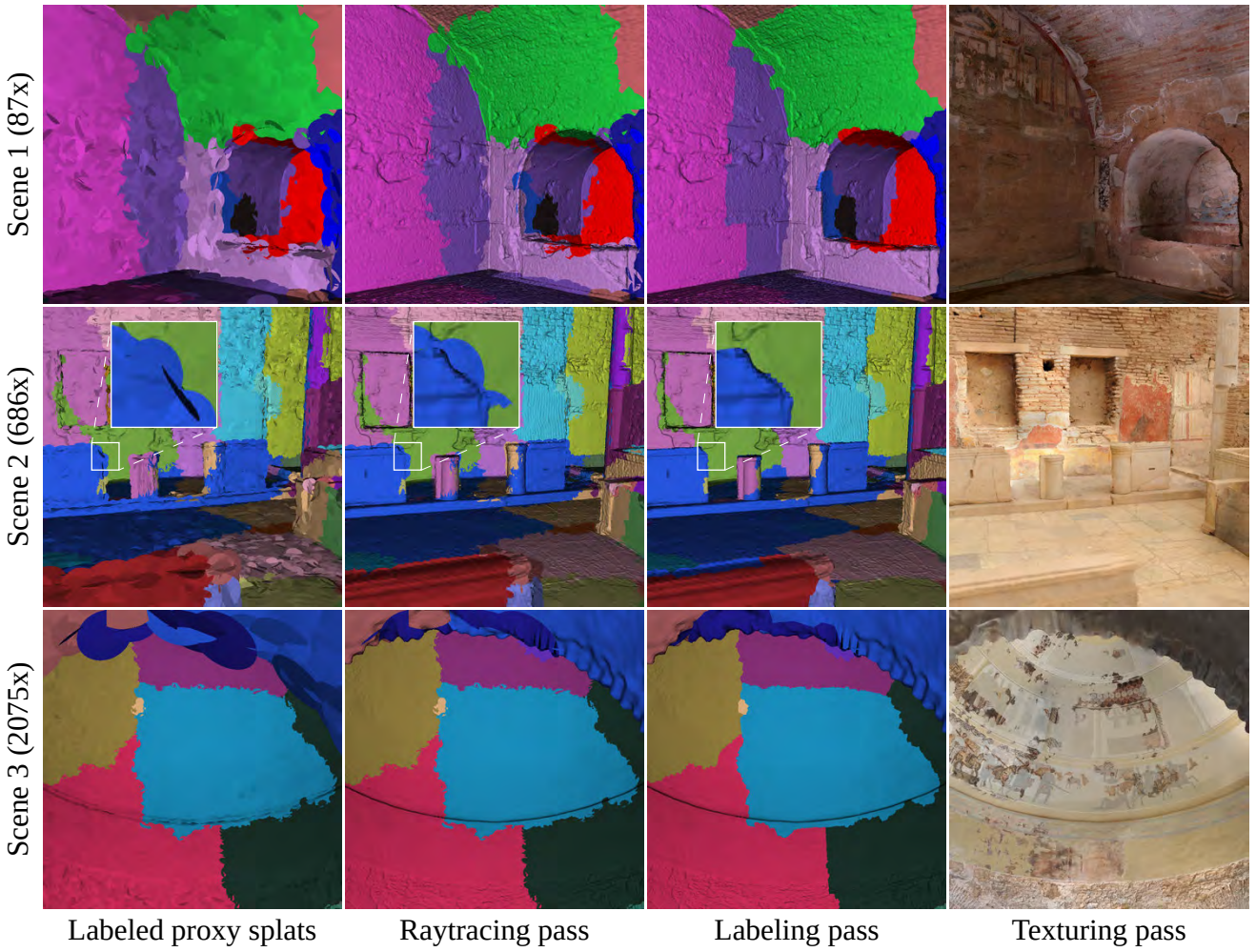


Fig. 17. Results from three data sets. From left to right: Splatted proxy points with (from top to bottom) increasing sub-sampling factors of the original point cloud ranging from $87\times$ up to $2075\times$; raytraced surface without and with per-pixel labels; and textured surface. The insets demonstrate how the labels of the column are mapped to the back wall if the labeling pass is not applied.

create a high-resolution depth buffer (splatting a small number of proxy points (visibility stage) and performing an efficient raytracing in a full-screen pass) are so fast that they could be run twice per frame. This could be used, for example, to create a shadow map for a moving light source, allowing dynamic shadows at interactive frame rates.

8 Conclusion

In this paper, we introduced a novel multi-depth-map raytracing approach for high-quality reconstruction and visualization of large-scale scenes. In a preprocessing, we generate multiple high-resolution depth maps and perform a graph-cut based optimization of the point-to-image assignments (point labels) on a strongly reduced subset of the original point cloud. At runtime, we first reconstruct a high-resolution depth buffer by raytracing these depth maps, where the labels indicate which depth maps to intersect. In a second step, we compute high-quality per-pixel labels from the sparse label information and use these for texturing the depth buffer.

We have shown that our method allows for a real-time visualization of large-scale scenes at much higher geometric resolution than the related state of the art, which is based on rendering and stitching of many depth maps. Our results also indicate a huge performance gain in the labeling step as compared to the previous method.

References

[1] D. T. Guinnip, S. Lai, and R. Yang, "View-dependent textured splatting for rendering live scenes," in *ACM SIGGRAPH 2004 Posters*, ser. SIGGRAPH '04. New York, NY, USA: ACM, 2004, pp. 51–. [Online]. Available: http://doi.acm.org/10.1145/1186415.1186474

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

[4] M. Kazhdan and H. Hoppe, "Screened poisson surface reconstruction," *ACM Trans. Graph.*, vol. 32, no. 3, pp. 29:1–29:13, June 2013.
[5] 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.
[6] 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.
[7] 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.
[8] M. Arikan, R. Preiner, C. Scheiblauer, S. Jeschke, and M. Wimmer, "Large-scale point-cloud visualization through localized textured surface reconstruction," *IEEE Transactions on Visualization and Computer Graphics*, vol. 99, no. PrePrints, p. 1, 2014.
[9] 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.
[10] M. Botsch, M. Spernat, and L. Kobbelt, "Phong splatting," in *Proceedings of the First Eurographics Conference on Point-Based Graphics*, ser. SPBG'04. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2004, pp. 25–32. [Online]. Available: http://dx.doi.org/10.2312/SPBG/SPBG04/025-032
[11] M. Zwicker, J. Räsänen, M. Botsch, C. Dachsbacher, and M. Pauly, "Perspective accurate splatting," in *Proceedings of Graphics Interface 2004*, ser. GI '04. School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada: Canadian Human-Computer Communications Society, 2004, pp. 247–254. [Online]. Available: http://dl.acm.org/citation.cfm?id=1006058.1006088
[12] 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.
[13] M. Agrawala, R. Ramamoorthi, A. Heirich, and L. Moll, "Efficient image-based methods for rendering soft shadows," in *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 375–384. [Online]. Available: http://dx.doi.org/10.1145/344779.344954
[14] F. Xie, E. Tabellion, and A. Pearce, "Soft shadows by ray tracing multilayer transparent shadow maps," in *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, ser. EGSR'07. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2007, pp. 265–276. [Online]. Available: http://dx.doi.org/10.2312/EGWR/EGSR07/265-276
[15] C. Zhang, H.-H. Hsieh, and H.-W. Shen, "Real-time reflections on curved objects using layered depth texture," in *IADIS International Conference Proceedings on Computer Graphics and Visualization*, 2008.
[16] A. Reshetov, A. Soupikov, and J. Hurley, "Multi-level ray tracing algorithm," in *ACM SIGGRAPH 2005 Papers*, ser. SIGGRAPH '05. New York, NY, USA: ACM, 2005, pp. 1176–1185. [Online]. Available: http://doi.acm.org/10.1145/1186822.1073329
[17] 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.
[18] C. Everitt, "Interactive order-independent transparency," NVIDIA, Tech. Rep., 2001.
[19] H. Gruen and N. Thibieroz, "Oit and indirect illumination using dx11 linked lists," in *GDC*, 2010.

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.

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.

Adaptively Layered Statistical Volumetric Obscurance

Quintjin Hendrickx[1] Leonardo Scandolo[1] * Martin Eisemann[1,2] † Elmar Eisemann[1] ‡

[1]Delft University of Technology [2]TH Köln

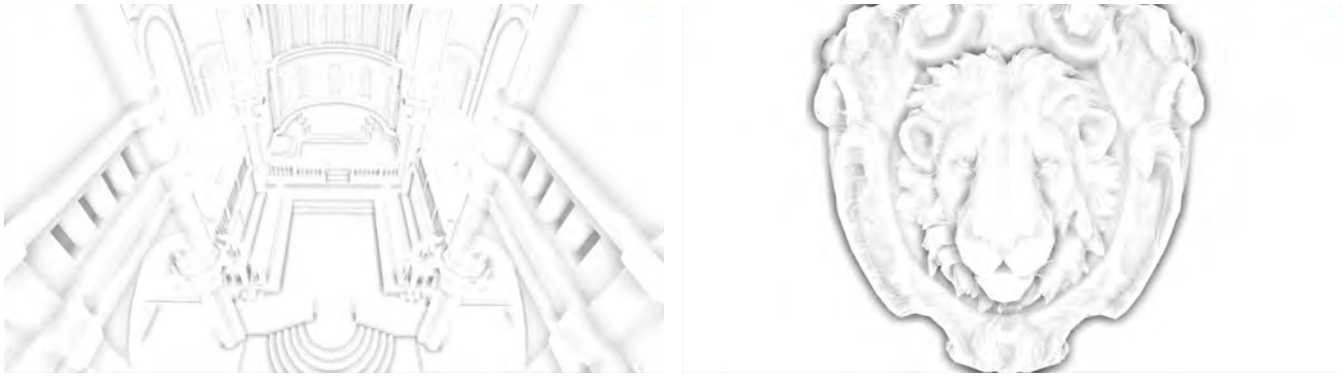


Figure 1: *Ambient Occlusion without shading. We can render images at 320 fps (*1280*x*720 *resolution, 294 MPixels/s) on a GTX 770.*

Abstract

We accelerate volumetric obscurance, a variant of ambient occlusion, and solve undersampling artifacts, such as banding, noise or blurring, that screen-space techniques traditionally suffer from. We make use of an efficient statistical model to evaluate the occlusion factor in screen-space using a single sample. Overestimations and halos are reduced by an adaptive layering of the visible geometry. Bias at tilted surfaces is avoided by projecting and evaluating the volumetric obscurance in tangent space of each surface point. We compare our approach to several traditional screen-space ambient obscurance techniques and show its competitive qualitative and quantitative performance. Our algorithm maps well to graphics hardware, does not require the traditional bilateral blur step of previous approaches, and avoids typical screen-space related artifacts such as temporal instability due to undersampling.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture;

Keywords: SSAO, Summed Area Tables, global illumination

1 Introduction

Efficient computation of global illumination is still one of the hardest problems in computer graphics. In consequence, real-time approximations often make very simplifying assumptions. *Ambient Occlusion* (AO) is an example and focuses on the evaluation of ambient light reaching a point on a surface [Landis 2002] by considering only local geometry as occluders in the scene. Attenuating the ambient light term based on local occlusion creates important contact cues improving overall depth perception.

Historically, AO was first applied in static scenes where its effect could be baked into occlusion maps [Landis 2002]. However, this approach does not work well for dynamic scenes and would need be applied per frame. In recent years, advances in graphics hardware and the development of screen-space approximations have led to real-time implementations of AO [Mittring 2007; Shanmugam and Arikan 2007]. These screen-space ambient occlusion (SSAO) techniques compute the amount of occlusion as a postprocessing pass based on a depth image from the camera's point of view. Traditionally, the occlusion factor is approximately estimated per pixel using a few samples and smoothed using a subsequent bilateral blur step. Most current rendering engines incorporate such solutions.

We aim at an approach that has the low computational complexity of screen-space ambient occlusion (SSAO) approaches, but avoids the usual drawbacks, such as banding, noise or blurriness caused by undersampling. In order to eliminate these artifacts, we have to account for *all* of the local geometry visible in screen-space. To this extent, we reverse the typical order of operations applied in existing SSAO approaches. Instead of taking samples and blurring the result afterwards, we compute a statistical model of the surrounding geometry at a pixel's world position and use it directly for AO computation. Because we do not use traditional sampling there is no need for randomization or blurring of the result [Mittring 2007].

Specifically, our contributions are:

- A screen-space ambient-occlusion approximation, evaluated using a single sample;
- An adaptive depth-slicing technique to efficiently compute this model;
- A GPU-friendly and highly-parallel implementation

In the following, we introduce an approximation for volumetric obscurance and how to compute it efficiently (Sec. 3) and describe how to improve quality via depth slicing (Sec. 4). For acceleration purposes, we introduce an adaptive technique (Sec. 5) and remove bias in the result by incorporating the surface normal into the computation (Sec. 6). We introduce important optimizations, like ap-

*e-mail:l.scandolo@tudelft.nl

†e-mail:martin.eisemann@fh-koeln.de

‡e-mail:e.eisemann@tudelft.nl

proximate summed-area tables (SAT) and differential SAT computation (Sec. 7). We evaluate and compare our approach to common screen-space ambient occlusion and volumetric-obscurance techniques (Sec. 8), before concluding (Sec. 9).

2 Related Work

The idea of approximating ambient illumination to account for local geometry was first described by Zhukov ([Zhukov et al. 1998]) and Landis [Landis 2002] showed the importance of AO in improving depth perception through contact cues and soft shadows. [Luft et al. 2006] employed a similar idea for artistic purposes. AO has since gathered a significant amount of interest resulting in numerous techniques. The algorithms can be divided in roughly two categories, geometry-based and screen-space ambient occlusion.

Geometry-based ambient occlusion incorporates all available geometry into the AO computation. Geometrical data in form of surface elements can be conveniently grouped in a hierarchy based on their distance to evaluate local AO [Bunnell 2005]. Alternatively, AO contributions can be scattered by each primitive via surrounding occlusion volumes [McGuire 2010]. While providing high-quality results the performance depends heavily on the geometrical complexity and AO radius. Density information can also be used for computing AO ([Hernell et al. 2010], [Grottel et al. 2012]) in the context of volume rendering.

Screen-space ambient occlusion techniques compute occlusion based on information in the depth buffer leading to (almost) geometry-independent evaluations. Such an approach was introduced by Crytek [Mittring 2007]. They sampled a sphere around a pixel's world position and reprojected the samples into the depth map to determine if they were occluded by geometry. For real-time performance the approach requires aggressive undersampling, which subsequently leads to noise artifacts that require an additional and costly bilateral blur step [McGuire et al. 2012].

Line sampling [Loos and Sloan 2010] improved upon the Crytek implementation by taking samples in the 2D projection of the sphere and integrating over line segments, thus computing the amount of geometry inside the sample sphere. Concurrently, [Szirmay-Kalos et al. 2010] presented a volumetric approach for estimating ambient occlusion based only on screen space depth values. Horizon-based AO [Bavoil et al. 2008] aims at finding a maximum horizon angle at which light can reach the sample point. Rays in randomized directions are marched and the maximum elevation angle of these are averaged to estimate the occlusion. Line-sweep AO [Timonen 2013] computes oclussion along a set of principal directions and is efficient due to sharing samples between the screen pixels aligned along these directions.

Statistical approaches aim at improving undersampling issues which arise when balancing performance and quality in SSAO-oriented methods. An example is the use of Summed-area tables (SAT), which are an efficient data structure to compute local averages of the depth values per pixel that can be used to approximate AO [Slomp et al. 2010; Díaz et al. 2010]. However, naively applying SATs leads to strong artifacts at depth discontinuities (halos or overestimations). We build upon these approaches and show how to remove such artifacts using adaptive depth layers.

Multiple depth layers have been used to improve AO and global illumination effects. Vardis et al. [2013] use depth information from different views to improve the estimate of ambient occlusion. Deep screen space [Mara et al. 2014] is a technique to create a depth buffer containing non visible fragments that can be used to compute ambient occlusion and indirect illumination effects. Deep G-Buffers [Nalbach et al. 2014], which contain the first two visible layers in the scene, use the enhanced geometrical information to compute global illumination effects. Altough our layering scheme only uses visible surfaces, applying our method to multiple depth layers could be explored in the future.

3 Statistical Volumetric Obscurance

3.1 Background

AO improves upon standard ambient illumination terms in popular shading models by capturing variations due to subtle shadowing caused by surrounding geometry. The amount of ambient occlusion at a point $\mathbf{x}$ on a surface is related to the ratio of outgoing rays that are able to leave a sample volume as opposed to rays that are being blocked by surrounding geometry (Fig. 2a) [Loos and Sloan 2010]. AO is formally defined as:

$$AO(\mathbf{x},\vec{n}) = \frac{1}{\pi}\int_{\Omega}\rho(d(\mathbf{x},\vec{\omega}))\vec{n}\cdot\vec{\omega}\mathrm{d}\vec{\omega} \ , \tag{1}$$

where $\mathbf{x}$ is the position in the scene, and $\vec{n}$ the normal at $\mathbf{x}$. Here, Ω represents the sample directions, usually a surface aligned hemisphere, and d is the distance to the first intersection.

The fall-off function ρ is used to simulate rays with a limited extent to model only local occlusion. In practice a piecewise constant, linear or quadratic fall-off function is used.

While an exact evaluation of AO based on this definition can be computationally costly, different models exist that can approximate the correct result. In particular, Volumetric Obscurance (VO) [Loos and Sloan 2010] estimates the amount of occlusion using the geometric density within a surrounding sample sphere S:

$$VO(\mathbf{x}) = \frac{1}{Vol(S)}\int_{S}\rho(d(\mathbf{x},s))O(s)\mathrm{d}s \ , \tag{2}$$

where the occupancy function O is 0 if s is inside of the geometry and 1 otherwise and $Vol(S)$ is the volume of the sample sphere S.

The assumption used by the VO model is that if a large portion of the sample sphere is inside a closed geometry, it will be less probable for ambient light rays to reach $\mathbf{x}$. While this intuition does not relate to a physical process, results are similar to AO in practice.

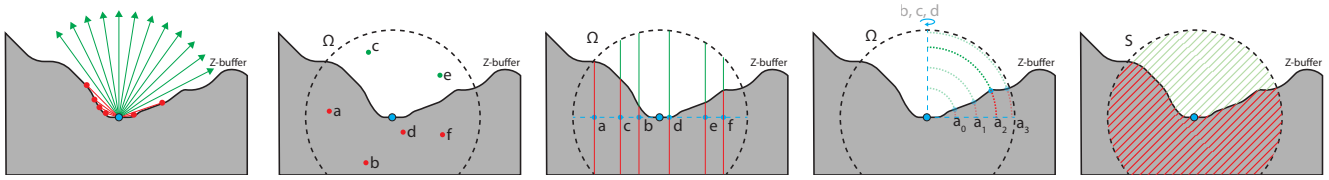
The normal $\vec{n}$, if known, can be used to restrict S to a hemisphere, thus eliminating occlusion caused by geometry below the sample.

3.2 Overview of our method

The method we will present is based on a statistical estimation of volumetric obscurance. For each pixel we define a sampling volume and approximate the amount of space in that volume which is inside the geometry of the scene. We present a model that uses that approximation to compute volumetric obscurance.

Initially we show how to efficiently average the depth of all pixels within a screen aligned sample box centered at each pixel. That sample box may contain pixels representing distant points in 3D space, which will wrongly influence the average depth value. We show how to solve this problem by separating the pixels in different layers and computing a different average per layer.

Finally, we demonstrate how to account for the surface normal by shifting our focus into computing the average of the view space pixel coordinates and projecting the average to the surface normal.



(a) *Screen Space Ambient Occlusion* **(b)** *Point sampling* **(c)** *Line sampling* **(d)** *Horizon-based sampling* **(e)** *Volumetric obscurance*

Figure 2: ***Screen Space Ambient Occlusion:*** *(a) SSAO at a sample point is defined by the ratio of rays that can escape the sample volume. Point sampling (b) and line sampling (c) approximate local ambient occlusion by sampling points within a sample volume. Horizon-based sampling (d) marches in randomized directions to compute the maximum angle at which rays can escape the sample volume. (e) Volumetric obscurance approximates ambient occlusion by computing the percentage of the sample volume that is inside a closed geometry.*

3.3 Our Model

SSAO techniques, such as the one presented in this paper, work solely on the depth map of the rendered image, optionally with an additional normal map. Our solution is based upon the VO model but introduces some important changes that make it more suitable for current graphics hardware and avoids sampling artifacts.

First, we use a sample box (Fig. 3a) instead of a sample sphere to estimate geometric density. The VO assumption is the same, i.e. we determine the percentage of the box which is filled with geometry and assume that it correlates with the amount of occlusion.

Second, we estimate the 3D obscurance function in Eq. (2) with a 2D version as follows. We assume that the geometry in the scene is composed of a single surface whose depth is a continuous function $G : \mathbb{R}^2 \rightarrow \mathbb{R}$ with $G(x,y) = d_{x,y}$ where the values at each pixel position x,y are stored in our Z-buffer. Therefore, every sample point whose depth is greater than what is stored in the Z-buffer is assumed to be occupied, and conversely each sample point whose depth value is less than the corresponding z-value in the depth buffer is deemed unoccupied. The key observation is that we can average the depth function $G(x,y)$ around a sample point and use this value to approximate the occupancy.

The mean value μ of G over a domain V is:

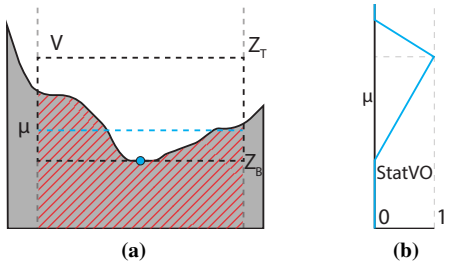
$$\mu = \frac{1}{A_V} \int_V G(x)dx \,, \tag{3}$$

where A_V is the area of the integration domain. The mean value $\mu(x)$ of the screen space depth within a sample box is then used to estimate the geometric occupancy around a pixel (Fig. 3a). Let $z_B(\mathbf{x})$ be the depth value of the bottom face of the sample box and $z_T(\mathbf{x})$ that of the top face. As we are only interested in the relative amount of occupancy, we can cancel A_V out and define our statistical volumetric obscurance (StatVO) model as:

$$StatVO(\mathbf{x}) = \psi\left(\frac{z_B(\mathbf{x}) - \mu(\mathbf{x})}{z_B(\mathbf{x}) - z_T(\mathbf{x})}\right). \tag{4}$$

The function $\psi(x)$ clamps the final value to 0 for negative values of x and behaves linearly for values in $[0,1]$, but will drop off to 0 with a user-defined slope for values greater than 1 (Fig. 3a). In consequence, the VO value results in zero, if the average depth is too far from the surface value, which reflects that the considered sample points fall outside the sample box.

The extent of the sample box in screen-space is computed based on its world position (the screen-space extent of sample boxes should be large nearby and smaller far away). To derive the size of its rectangular projection, we rely on the pixel's linearized depth value. Next, to estimate the occlusion, we need a way to quickly compute the mean depth value inside such screen-aligned rectangle of arbitrary size. To this extent, we make use of *Summed-Area Tables* [Crow 1984]. They allow us to retrieve, at a constant cost, the average in an arbitrary rectangular region around a pixel. In each pixel, they store the sum of all pixels in the upper left quadrant of the texture. The construction can be done using a fast recursive algorithm[Hensley et al. 2005]. To query the average of a rectangular region, the sum of all its interior pixels can be retrieved by combining the values from its four corners.



(a) **(b)**

Figure 3: ***Statistical volumetric obscurance:*** *Our method (a) approach computes the volume integral of a box as an approximation of ambient occlusion. The graph at the right shows how occlusion increases as the average μ rises, after the average leaves the sample box, occlusion falls-off back to zero.*

Approximating the VO via a mean value implies that the fall-off function in Eq. (2) cannot be applied to each sample separately. Furthermore, note that in Eq. (4) the sample box is not restricted in the z coordinate, meaning that sample points outside of the sample box influence μ as well. We would want to reduce the influence of samples that are far from the surface point. In the next section, we explain how to incorporate this idea.

4 Depth Layering

Not applying a fall-off function to each sample when computing volumetric obscurance leads to an overestimation at depth discontinuities, which leads to halos (Fig. 6a). To counteract overestimation, we divide the depth map into m uniformly arranged layers orthogonal to the viewing direction based on the maximum and minimum view-space depths. Each depth pixel is assigned to the layer which overlaps with the corresponding depth value (Fig. 4). Non-

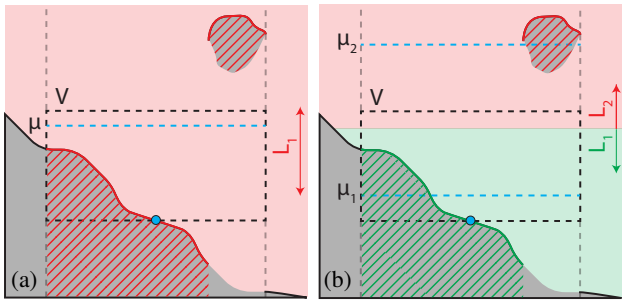


Figure 4: ***Depth Layering:*** *While a single layer will result in a single average over all geometry (a), we can slice our scene in multiple depth layers to obtain averages for each layer separately (b).*

assigned pixels are set to 0. The depth buffer is processed and split among these multiple layers. During the splitting operation, we use an additional (color) channel to keep track of sample/pixel validity, i.e., the channel is one if a depth sample was assigned to the corresponding pixel and zero otherwise. We then generate SAT's for each layer and channel separately. Contributions from each layer L_i overlapping with the sample volume V are weighted by the corresponding sample count n_i (the number of samples assigned to L_i), to account for the missing values, and combined into a final obscurance value $StatVO_{\text{Layered}}(\mathbf{x})$ as follows:

$$StatVO_{\text{Layered}}(\mathbf{x}) = \frac{1}{\sum_{i \in V} n_i(\mathbf{x})} \sum_{i \in V} StatVO_i(\mathbf{x}) n_i(\mathbf{x}) \quad , \qquad (5)$$

where $StatVO_i$ is the statistical volumetric obscurance defined in Eq. (4) computed on layer L_i.

Because of function ψ in Eq. (4), depth slices with depth values significantly different from the surface depth will have no influence on the final computed obscurance. However, the computational effort is linear in the number of layers m and larger scenes require a high number of depth layers, resulting in computation times that are no longer competitive compared to other real-time AO techniques. For example, we found that the Sibenik cathedral requires around 64 layers for good results (Fig. 6e).

5 Adaptive Depth Slicing

To improve upon the linear depth-slicing approach, we propose to use depth layers which adapt to the local geometry. We drew inspiration from higher-dimensional filtering approaches [Gastal and Oliveira 2012] as our technique also builds on a recursive process that partitions the current depth map of a layer into two disjoint sets in each recursion. The intuition behind this step is that as long as pixels with very different depth values are further apart than the screen-space size from the corresponding sample area then they do not influence each other during the obscurance computation.

Our algorithm Fig. 5) works as follows: Initially we have the original depth map and its corresponding SAT. Using this SAT, we can compute the average depth value μ around each pixel, as described in Sec. 3, which amounts to having a smoothed version of the original depth map. We then assign each depth value of the original depth buffer to the upper or lower layer based on its relative depth value compared to μ, hereby often successfully separating locally far and near samples. The intuition behind this approach is that around depth discontinuities, pixels closer to the camera will all have a depth value smaller than the average, and pixels further away will have a depth that is greater than the average. Once each pixel is

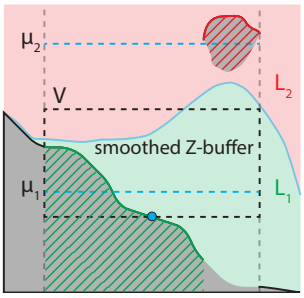
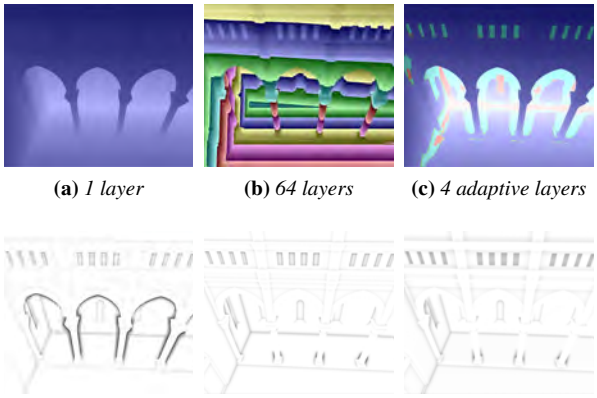


Figure 5: ***Adaptive Depth Slicing:*** *In each recursion a smoothed Z-buffer is constructed, each depth sample is either assigned to the upper (red) or lower layer (green).*



(a) *1 layer* **(b)** *64 layers* **(c)** *4 adaptive layers*

(d) *1 layer, 1 ms* **(e)** *64 layers, 55 ms* **(f)** *4 adaptive layers, 2-3 ms*

Figure 6: ***Comparison****: Using a single layer (a) results in dark halos at depth discontinuities (d). Splitting the scene into multiple layers solves this problem. (b) However, this impacts performance (e). Using adaptive slicing adds additional layers only at depth discontinuities (c). This allows us to eliminate halos and achieve good performance (f).*

assigned to one of the two layers, we compute a new SAT for each layer. As in the uniform depth slicing approach (Sec. 4), we keep track of the amount of active pixels by using an additional channel in these new SATs as well. Once we have the two new SATs, the process can be repeated for each newly created layer in order to further differentiate samples.

During rendering we simply evaluate all layers using Eq. (5). The fall-off function ψ automatically adjusts the influence of each layer, hereby reducing the influence of samples outside the sample box.

The adaptive depth slicing dramatically reduces the number of required layers. As few as four adaptive layers can achieve results that are comparable to the naive 64 uniform layers implementation (Fig. 6e and 6f) for our test scenes.

6 Surface Normal Incorporation

Until now, the sample box was always aligned with the viewing direction. However, when viewing a surface from a grazing angle a bias is introduced into our VO approximation when parts of this surface are hidden in the depth map by pixels closer to the camera (Fig. 7a and b). In Fig. 7a the mean value of layer L_1 (green) is only slightly above the sample position. In Fig. 7b the mean value is

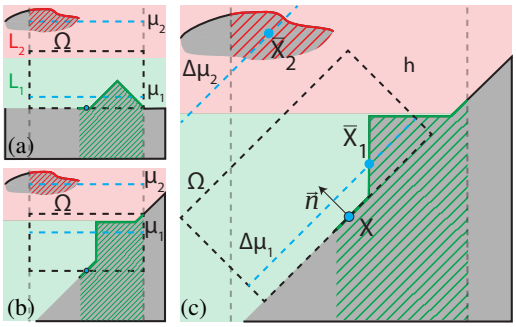


Figure 7: ***Normal Integration:*** *(a) and (b) Considering only the hemisphere/hemibox around a surface point in the viewing direction leads to different occlusion results depending on the slope of the surface. (c) Projection of the mean of all depth samples in 3D space onto the surface normal removes this bias.*

higher since the object in layer L_2 hides a part of the surface underneath. Performing VO computation only for the positive half-space in the direction of the surface normal can enhance the perception of finer scale details [Loos and Sloan 2010]. We make use of the surface normal by extending our approach to 3D and orienting the sample box, so its bottom side is aligned with the surface (Fig. 7c).

After the adaptive layer computation from Sec. 5, we reproject each depth value in each layer into view space to acquire its 3D position. We save the results in RGB maps and compute the corresponding SATs for each. Instead of computing an average depth value, we now compute the average position $\bar{\mathbf{x}}$ of all reprojected depth samples and project it onto the surface normal $\vec{n}$, which conveniently reveals the average height of all samples along the surface normal:

$$\Delta\mu = (\bar{\mathbf{x}} - \mathbf{x}) \cdot \vec{n} \quad , \tag{6}$$

where $\mathbf{x}$ is the surface position. The oriented statistical surface obscurance $StatVO_i$ per layer L_i is then:

$$StatVO_i = \psi\left(\frac{\Delta\mu}{h}\right) \quad , \tag{7}$$

where h is the height of the sample box. Eq. (5) is used to compute the final obscurance value.

Further improvement regarding the approximation quality can be achieved by evaluating four quadrants of the sample region around the pixel separately. The region can be split into four equally-sized parts, for which the result is evaluated independently, by retrieving nine (corners, midpoints, center) instead of four values from the SATs. The final VO value is then computed by averaging the results (Fig. 8). The overall cost increases by roughly 25% to 40%.

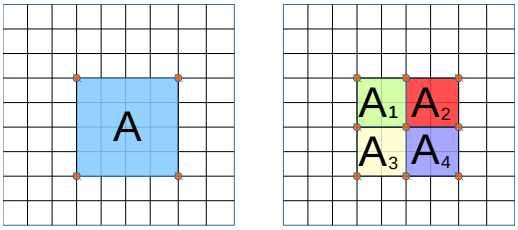
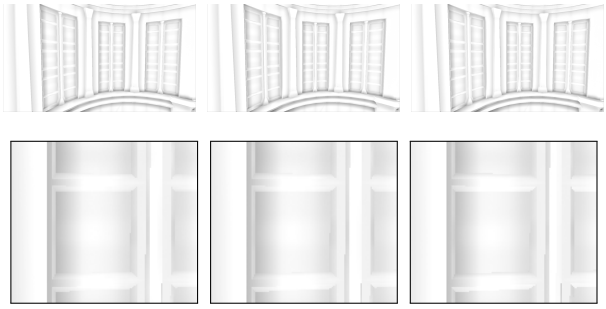


Figure 8: *Left: One region, right: four quadrants. All other images in the paper rely on a single region*



(a) *Full resolution* **(b)** *Half resolution* **(c)** *Quarter resolution*

Figure 9: *Approximate SATs*

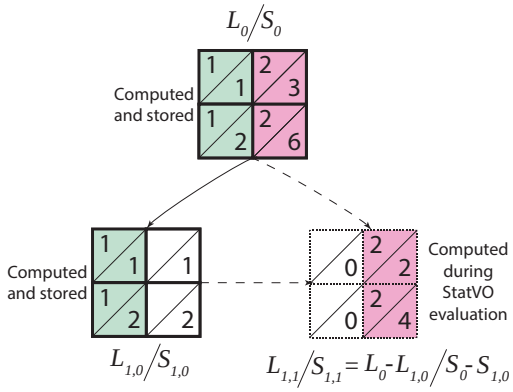


Figure 10: ***Differential SAT Computation:*** *We can spare the computation of one depth layer (L) and one SAT (S) in each partitioning step as they can be reconstructed from their parents and siblings.*

7 Optimizations

We introduce two important performance improvements for our technique; approximate SATs and differential SAT computation.

7.1 Approximate SATs

The most costly computation of our algorithm is the SAT creation. While we experimented with other prefiltering techniques such as Mipmaps, N-Buffers [Décoret 2005] or Y-Maps [Schwarz and Stamminger 2008], SATs provided the highest quality.

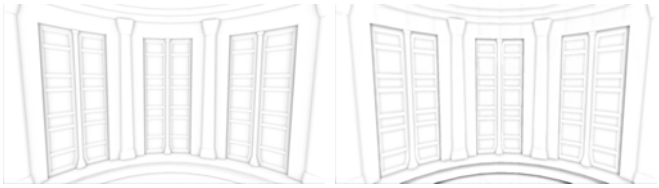
Instead of computing full-resolution SATs, we downsample the input by a factor of 2-4 in both width and height by averaging the depth values, reducing computation times by a factor of 4 to 16 with little impact on quality. We then upscale the low-resolution SATs with linear interpolation to approximate the full resolution input. It is important to note that the additional channel used for sample counting must be handled carefully during downsampling to keep track of the sample count. As linear interpolation is hardware-accelerated, upsampling the SATs is very fast. Fig. 9 shows a comparison between using a full-resolution SAT and downsampled versions. The SAT is queried with sample rectangles, which makes this acceleration suitable for our context. For other sampling strategies, such a solution can be harmful (compare supplementary material).

7.2 Differential SAT Computation

Let L_0 be the (downsampled) original depth map, and $L_{1,0}$ and $L_{1,1}$ be the first two adaptive sublayers resulting from partitioning L_0. Each subdivision of a layer requires the computation of a corresponding SAT S. An important observation here is that while the samples contained in L_0 are distributed among the sublayers $L_{1,0}$ and $L_{1,1}$, their total sum does not change. Thus, subtracting SAT $S_{1,0}$ from S_0 results in $S_{1,1}$ (Fig. 10) removing the need to compute it explicitly. Alg. 1 shows the pseudo-code for the SAT generation routine of four adaptive layers when using differential SATs.

For four adaptive layers, we need to compute only four out of seven SATs explicitly (for more layers the ratio approaches 1:2). During rendering, we compute the value of the missing SATs on-the-fly by subtracting all ancestral and the sibling layer from the root SAT S_0. The pseudo-code in Alg.2 shows how to query the SAT for all four leaf layers $S_{2,0}, S_{2,1}, S_{2,2}$ and $S_{2,3}$ given only $S_0, S_{1,0}, S_{2,0}$ and $S_{2,2}$.

8 Results



(a) *11 line samples (1.47ms)* **(b)** *StatVO (0.93ms)*

Figure 11: ***Comparison to Line Sampling:*** *Both figures evaluate the occlusion at full resolution of 1280×720. Line sampling (a) using 8 samples with a 4×4 randomization kernel with an 8×8 bilateral blur applied. StatVO (b) with 4 adaptive layers with quarter resolution SATs.*

We have implemented our technique using OpenGL/C++. All statistics were measured at 1280×720-pixel resolution on an Intel Core i5 4590 with 8GB of RAM and an NVIDIA GTX 770 graphics card. We implemented the SAT generation algorithm as an OpenGL compute shader [Sellers et al. 2013].

Performance Table 1 shows a detailed performance analysis of our algorithm with four adaptive layers and using full resolution SATs and half resolution in width and height. As the SAT computation is the most costly part of our algorithm, performance increases by a factor of four if width and height are halved. We found that in many scenes, the overall quality loss was small even when reducing the resolution along each axis by a factor of four (Fig. 9). Aggressive downsampling can result in temporal flickering around depth discontinuities due to undersampling when the camera moves.

Algorithm 1 Pseudo-code to compute the needed SATs with 4 adaptive layers.

Given: depth buffer D_0

$S_0 \leftarrow \text{computeSAT}(D_0)$
$D_{1,0} \leftarrow \text{partition}(D_0, S_0)$
$S_{1,0} \leftarrow \text{computeSAT}(D_{1,0})$

$D_{2,0} \leftarrow \text{partition}(D_{1,0}, S_{1,0})$
$D_{2,2} \leftarrow \text{partition}(D_{1,1}, S_{1,1})$

$S_{2,0} \leftarrow \text{computeSAT}(D_{2,0})$
$S_{2,2} \leftarrow \text{computeSAT}(D_{2,2})$

Algorithm 2 Pseudo-code to compute the areas in a two level differential SAT hierarchy

Given: S_0, $S_{1,0}$, $S_{2,0}$, $S_{2,2}$ computed previously

$A_0 \leftarrow \text{sampleSAT}(S_0)$
$A_{1,0} \leftarrow \text{sampleSAT}(S_{1,0})$
$A_{1,1} \leftarrow A_0 - A_{1,0}$
$A_{2,0} \leftarrow \text{sampleSAT}(S_{2,0})$
$A_{2,1} \leftarrow A_{1,0} - A_{2,0}$
$A_{2,2} \leftarrow \text{sampleSAT}(S_{2,2})$
$A_{2,3} \leftarrow A_{1,1} - A_{2,2}$

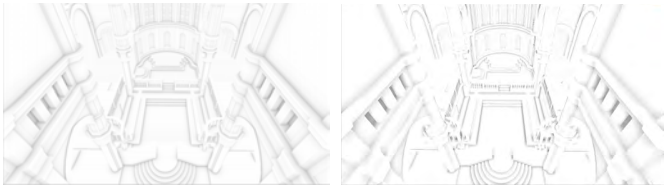
Step	T_{full} (ms)	T_{half} (ms)	Speed-up
Downsample depth buffer	0.13	0.05	×2.6
Compute root SAT	1.40	0.32	×4.4
Compute first level SAT	1.86	0.43	×4.3
Compute second level SATs	3.72	0.84	×4.4
Evaluate StatVO	1.65	0.50	×3.3
Total	8.86	2.14	×4.1

Table 1: ***Performance evaluation:*** *Breakdown of computational cost of our algorithm for full resolution SATs and half resolution SATs when using 4 adaptive layers.*

Memory requirements For our four final SATs, we use four 32bit floating point channels, the first three are used to store view-space coordinates and one is used to mark valid samples in each layer. When computing SATs in full HD, using quarter resolution in width and height, we require a total of 8MB of memory.

Comparison to other techniques We compare our technique to the classic point and line sampling SSAO techniques, which are the most commonly used [Mittring 2007; Loos and Sloan 2010]. By choosing a very high sample count (256 point samples per pixel), we additionally generated a reference image for volumetric obscurance. In Fig. 13, we show that our technique can generate results that are comparable in quality.

In Fig. 12, we compare our technique to point and line sampling. We chose the number of samples so that all approaches produce visually similar quality. Using a quarter resolution SAT our approach

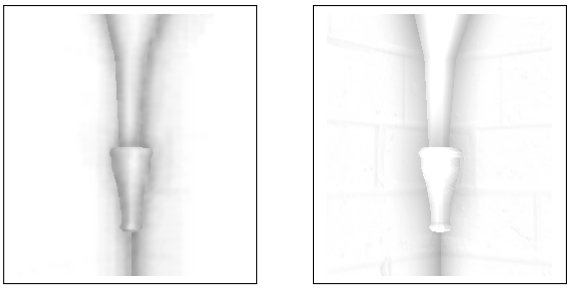


(a) *VO reference* **(b)** *StatVO*

Figure 13: ***Comparison to a Reference VO:*** *(a) Reference from 256 point samples without a randomization kernel or a blur filter. Our method (b) shows comparable results using 4 adaptive layers and full resolution SATs.*

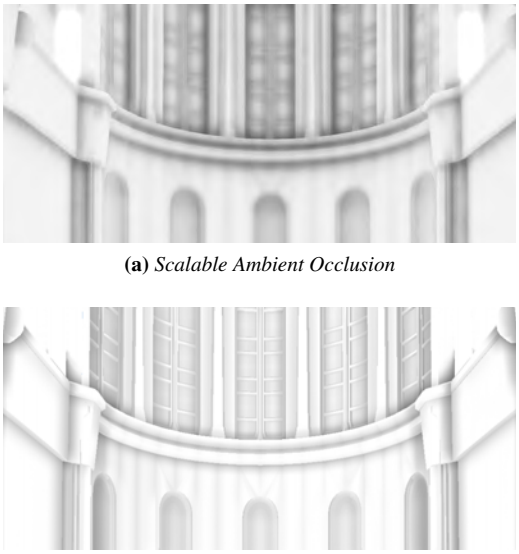
(a) *17 point samples (2.0ms)* **(b)** *12 line samples (1.7ms)* **(c)** *StatVO (1.6ms)*

Figure 12: ***Comparison to Point and Line Sampling:*** *(a) uses a point sampling approach with 17 samples and a 4×4 randomization kernel, occlusion is evaluated at half resolution and a bilateral upsampling with 7× 7 blur kernel is applied. (b) achieves similar results but only uses 12 line samples. In (c) we evaluate occlusion at the full resolution using 4 adaptive layers with quarter resolution SATs.*



(a) *16 point samples (0.77ms)* **(b)** *StatVO (0.65 ms)*

Figure 14: ***Comparison to Point Sampling:****(a) shows a close-up of a point sampling configuration were occlusion is evaluated at half resolution, using 16 samples with a 4×4 randomization kernel. The result is upsampled and an 8×8 bilateral blur filter is applied. (b) StatVO evaluated at full resolution with 4 adaptive layers and quarter resolution SATs.*



(a) *Scalable Ambient Occlusion*

(b) *StatVO*

Figure 15: ***Comparison to Scalable Ambient Occlusion****: (a) shows part of a scene rendered with SAO, (b) shows the same scene rendered with our approach using quarter resolution SATs. Our method does not exhibit blurring despite using downsampled SATs.*

is slightly faster than both. Increasing the AO radius our performance stays the same, whereas the performance of line and point sampling decreases, due to an increase of the required samples and bilateral blur radius, which is mandatory to diminish the increasing undersampling artifacts. This means that our algorithm scales well to higher resolutions compared to point and line sampling (Note that we used a relatively small image resolution of 1024×768 pixels and a similar sample region on a higher resolution image would have to be scaled up).

If only few samples are computed for point or line-sampling (e.g., for very high performance), visible undersampling artifacts appear (Fig. 14). Our approach does not suffer from undersampling and leads to more details, e.g., on the wall.

Fig. 15 shows a comparison of our method to scalable ambient occlusion (SAO) [McGuire et al. 2012], which is currently one of the fastest methods for computing ambient occlusion. SAO exhibits a large amount of blurring due to its usage of mipmapping during sampling and a bilateral blurring step. Although our approach is around 50% slower, it produces crisp results at all depths. Further, our approach does rely on SATs instead of mipmaps, which leads to an additional cost, but proved more stable for animation.

Limitations of screen-space approaches The approach we present is subject to some of the usual limitations associated with screen space AO methods, namely the need for a guard band and the inability to account for surfaces occluded from the view. Due to the statistical nature of our approach, the contribution of thin surfaces parallel to the viewing direction may be underrepresented in the AO computation. Also, as stated previously, downsampling the depth buffer and SATs too aggressively may result in a small amount of temporal flickering. Nonetheless, even when reducing resolution by a factor of 1/4 along each axis, in most areas of the image, there are no visible differences. This result stems from our algorithmic design. An adaptive downsampling strategy could be a promising direction for future research.

9 Conclusion

Statistical Volumetric Obscurance is an alternative to traditional screen-space ambient occlusion, which does not rely on typical sampling. Due to the usage of a sample box, the evaluation of local obscurance is reduced to a simple mean-value computation over the sample area, which is efficiently computed on the GPU using specialized summed-area tables. Previous approaches in this direction suffered from artifacts such as halos. The adaptive depth slicing avoids these and preserves fine-scale features, leading to a quality

similar to previous approaches with many more samples.

For best results the amount of nearby depth discontinuities should be limited. An extreme case, like looking along a row of aligned pillars, breaks this assumption and small halos and dark creases are introduced. Still, our method performs well in these cases (compare accompanying video). Locally adaptive layering would be an interesting future work to address such issues.

As with any SSAO technique, the depth map represents only the visible geometry, whereas important information of the overall scene is lost. Rendering the occluded geometry into the depth layers after they have been created would allow us to incorporate even these occluded parts for more precise results beyond the capabilities of traditional SSAO techniques.

10 Acknowledgements

The work was partially funded by the EU FP7-323567 project Harvest4D, and the Intel VCI at Saarland University. The Sibenik cathedral scene used is a project by Marko Dabrovic and the Sponza atrium scene is freely distributed by Crytek.

References

BAVOIL, L., SAINZ, M., AND DIMITROV, R. 2008. Image-space horizon-based ambient occlusion. In *ACM SIGGRAPH 2008 Talks*, 22:1–22:1.

BUNNELL, M. 2005. Dynamic ambient occlusion and indirect lighting. *Gpu gems 2*, 2, 223–233.

CROW, F. C. 1984. Summed-area tables for texture mapping. *ACM SIGGRAPH computer graphics 18*, 3, 207–212.

DÉCORET, X. 2005. N-buffers for efficient depth map query. In *Computer Graphics Forum*, vol. 24, Wiley Online Library, 393–400.

DÍAZ, J., VÁZQUEZ, P.-P., NAVAZO, I., AND DUGUET, F. 2010. Real-time ambient occlusion and halos with summed area tables. *Computers & Graphics 34*, 4, 337–350.

GASTAL, E. S., AND OLIVEIRA, M. M. 2012. Adaptive manifolds for real-time high-dimensional filtering. *ACM Transactions on Graphics (TOG) 31*, 4, 33.

GROTTEL, S., KRONE, M., SCHARNOWSKI, K., AND ERTL, T. 2012. Object-space ambient occlusion for molecular dynamics. In *Pacific Visualization Symposium (PacificVis), 2012 IEEE*, 209–216.

HENSLEY, J., SCHEUERMANN, T., COOMBE, G., SINGH, M., AND LASTRA, A. 2005. Fast summed-area table generation and its applications. In *Computer Graphics Forum*, vol. 24, Wiley Online Library, 547–555.

HERNELL, F., LJUNG, P., AND YNNERMAN, A. 2010. Local ambient occlusion in direct volume rendering. *Visualization and Computer Graphics, IEEE Transactions on 16*, 4, 548–559.

LANDIS, H. 2002. Production-ready global illumination. *Siggraph course notes 16*, 2002, 11.

LOOS, B. J., AND SLOAN, P.-P. 2010. Volumetric obscurance. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, ACM, 151–156.

LUFT, T., COLDITZ, C., AND DEUSSEN, O. 2006. Image enhancement by unsharp masking the depth buffer. In *ACM SIGGRAPH 2006 Papers*, 1206–1213.

MARA, M., MCGUIRE, M., NOWROUZEZAHRAI, D., AND LUEBKE, D. 2014. Fast global illumination approximations on deep g-buffers. Tech. rep., NVIDIA Corporation.

MCGUIRE, M., MARA, M., AND LUEBKE, D. 2012. Scalable ambient obscurance. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, EGGH-HPG'12, 97–103.

MCGUIRE, M. 2010. Ambient occlusion volumes. In *Proceedings of High Performance Graphics 2010*.

MITTRING, M. 2007. Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 Courses*, ACM, New York, NY, USA, SIGGRAPH '07, 97–121.

NALBACH, O., RITSCHEL, T., AND SEIDEL, H.-P. 2014. Deep screen space. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '14, 79–86.

SCHWARZ, M., AND STAMMINGER, M. 2008. Quality scalability of soft shadow mapping. In *Graphics Interface 2008*, 147–154.

SELGRAD, K., DACHSBACHER, C., MEYER, Q., AND STAMMINGER, M. 2014. Filtering multi-layer shadow maps for accurate soft shadows. In *Computer Graphics Forum*, Wiley Online Library.

SELLERS, G., WRIGHT, R., AND HAEMEL, N. 2013. *OpenGL SuperBible: Comprehensive Tutorial and Reference*, sixth ed. Addison-Wesley Professional.

SHANMUGAM, P., AND ARIKAN, O. 2007. Hardware accelerated ambient occlusion techniques on gpus. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, 73–80.

SLOMP, M., TAMAKI, T., AND KANEDA, K. 2010. Screen-space ambient occlusion through summed-area tables. In *Networking and Computing (ICNC), 2010 First International Conference on*, IEEE, 1–8.

SZIRMAY-KALOS, L., UMENHOFFER, T., TOTH, B., SZECSI, L., AND SBERT, M. 2010. Volumetric ambient occlusion for real-time rendering and games. *Computer Graphics and Applications, IEEE 30*, 2, 70–79.

TIMONEN, V. 2013. Line-sweep ambient obscurance. *Computer Graphics Forum 32*, 4, 97–105.

VARDIS, K., PAPAIOANNOU, G., AND GAITATZES, A. 2013. Multi-view ambient occlusion with importance sampling. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '13, 111–118.

ZHUKOV, S., IONES, A., AND KRONIN, G. 1998. An ambient light illumination model. In *Rendering Techniques '98*. 45–55.

Smooth, Interactive Rendering Techniques on Large-Scale, Geospatial Data in Flood Visualisations

Christian Kehl, Tim Tutenel and Elmar Eisemann
Computer Graphics and Visualisation Group
Delft University of Technology
Email: [c.kehl, t.tutenel, e.eisemann]@tudelft.nl

***Abstract*—Visualising large-scale geospatial data is a demanding challenge that finds applications in many fields, including climatology and hydrology. Due to the enormous data size, it is currently not possible to render full datasets interactively without significantly compromising quality (especially not when information changes over time).**

In this paper, we present new approaches to render and interact with detail-varying Light Detection and Range (LiDAR) point sets. Furthermore, our approach allows the attachment of large-scale geospatial meta information and the modification of point attributes on the fly.

The core of our algorithm is a dynamic GPU-based hierarchical tree data structure that is used in conjunction with an out-of-core, Level-of-Detail (LoD)-Point-based Rendering (PBR) algorithm to modify data on the fly. This combination makes it possible to augment the original data with dynamic context information that can be used to highlight features (e.g., routes, marked areas) or to reshape the entire data set in real-time.

We showcase the usefulness of our algorithm in the context of disaster management and illustrate how decision makers can discuss a flood scenario covering a large area (spanning 300 km^2) and discuss hazards, as well as related protection measures, interactively. One of our presented reference point sets includes parts of the AHN2 data set (14 TB of LiDAR data in total). Previous rendering algorithms relied on a long offline preprocessing (several hours) to ensure a quick data display. This step made any changes to the data impossible. With our new approach, we can modify point sets without requiring a new preprocessing run.

Introduction

An increasing amount of 3D geographic information has recently become available as Digital Elevation Model (DEM), height maps and LiDAR point scans. These data are used in many domains, such as climatology [12], [6] and hydrology [7], [17], [10], to help in the decision making processes. Topographic measurements, geospatial meta data (i.e. area selections) and domain-specific data (e.g. fluid simulation results, water currents and cloud simulation data and measurements) are combined in an illustrative manner.

Recent geographic point data, often captured with terrestrial and aerial LiDAR technology, are too large to be rendered directly. The data neither fit in arendering-dedicated graphics memory, nor in a workstation's main memory. We refer to such data sets as *massive point sets*.

Many tree- and graph-based solutions for rendering large data sets [15], [4], [1], [11], [5], and LiDAR data in particular [9], [8], [3], are already available. Graph-based solutions continuously request new data nodes for the areas in view. The data is often structured in trees in which nodes correspond to a set of points in the corresponding scene area. By limiting the descent in the tree, the point density can be adjusted. We similarly rely on a tree structure that allows us to ensure a *rendering-on-budget* paradigm, such as explored by Goswami et al.[5], to address the rendering of massive, coloured LiDAR datasets. Further, we avoid common popping artifacts that result from adding/removing the points in a node and rather add/remove them progressively, hereby ensuring a smooth appearance.

Besides the rendering, interaction is another key component for using massive point sets in a decision making process. In this context, we refer to "interaction" as the possibility to modify data attributes (e.g. position, colour, visibility) on-the-fly to highlight or adapt information and to get instantaneous visual feedback of the performed operation. Massive point sets are usually modified in an offline process. This procedure is time consuming and not suitable for collaborative decision making. The interactive modification of massive point sets is currently not possible without visual quality loss. Available point set editors [18], [14] can only facilitate interactive modifications for data as far as they fit in the computer's main memory. Wand et al. [15] presented an improved system with the capability to process massive data set in a batch-process manner using a tree hier-

archy. A main assumption of current approaches is that modifying operations are permanently valid. Regarding decision making processes and the goal to provide techniques for collaborative, on-the-fly information exchange, we assume that modifications are only temporarily valid.

The combination of the two presented techniques makes it possible to augment the original data with dynamic context information that can be used to highlight features (e.g., routes, marked areas) or to reshape the entire data set in real-time. We showcase the applicability of our approach in the domain of flood hazard management. Three use cases are presented that demand such large-scale modifications. Our approach is able perform operations interactively that formerly demanded substantial preprocessing. We also briefly present first performance assessment results.

Methods

Rendering-on-Budget

Our initial approach spatially subdivides the point set into tiles of 1000 metre by 1250 metre. Each tiles' points are distributed uniformly along the levels of details [2], [7]. Using this technique, lots of points need to be loaded as a bucket immediately if the user approaches higher detail levels. Our method always loads at least the roughest detail level for each bucket in the dataset. This results in spatial limitations on the presentable extent of LiDAR datasets in practice. This could be solved by employing additional, view-dependent criteria. As formerly discussed [7], this bucket-loading behaviour leads to visual discontinuities that irritate the user.

In our new approach, we apply the rendering-on-budget paradigm to provide a constant rendering speed for large point sets [16]. Hence, The currently available budget depends on the data size and the technical capabilities of the rendering system. The budget is adapted using a Proportional-Integral-Differential (PID) controller, which keeps track of the number of rendered points in relation to total rendering time.

During the rendering, we traverse the LoD tree structure in breadth-first order. For each node in the tree, we render its points as long as rendering budget is still available. We use the local density as a measure to decide which particular should be rendered next. Because the rendering decision is taken on per-point-basis rather than the per-node point bucket of the initial approach, the new technique offers a smooth LoD transition while preserving rendering speed.

On-line Modification of Large-Scale, Geospatial LiDAR point sets

We adapt the point attributes via GPU shaders in an indirect manner. In order to describe the modifications, we use interactively-places, textured 2D polygons. One can imagine that these are defined on a map of the point-cloud region and the changes will affect all points that are covered. They include line-like structures for route descriptions and polygons definitions for areas. These geometric shapes are stored as structured vector lists. In contrast to Scheiblauer and Wimmer [13] the vector format is independent of the point cloud. Further, it is not limited to a uniform grid size (such as geometry, given as Volume Images). Our approach allows us to achieve an accurate rendering at varying scales. Nevertheless, we want to avoid preprocessing the data set, which would result in hours of computation. Instead, we resolve the issue at runtime. For each rendered point, we test whether it falls in one of the polygons and then modify its attributes according to the regions definition.

During the rendering stage, we can not evaluate the bare polygon geometry because the point vertices and the polygon vertices are part of the same global vertex set. Hence, to overcome this drawback, we store the polygon geometry in a texture that can subsequently be evaluated for each point cloud vertex.

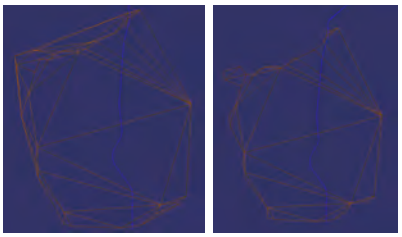
As a first step, we need to find the polygon each point belongs to. In this case, each point needs to be tested with each polygon for an intersection. We need to do this because there is no other reference information available (apart from the point's 3D position) to establish a point-polygon relation. If this localisation succeeds, we can apply our attribute modification. Our initial algorithm presented in Alg. 1.

Calculating the intersection of a point and a polygon is computationally expensive and existing implementations of this procedure do not map well on GPU architectures. It is thus not practical to use a point-in-polygon check for the number ofLiDAR points and annotations we intend to handle. Hence, we simplify Alg. 1 by transforming the polygons into 2D triangular meshes. This exchanges the *PointPolygonIntersection*-test with a *PointTriangleIntersection*-test, which is much simpler to compute and implement on a GPU.

Algorithm 1 Apply point attribute modification

```
function ModifyAttribute(pointlist, polygons)
    for all point ∈ pointlist do
        for all polygon ∈ polygons do
            if PointPolygonIntersect(point, polygon)
is true then
                attribute ← GetAttribute(polygon)
            else
                attribute ← 0
            end if
            ApplyAttribute(point, attribute)
        end for
    end for
end function
```

On the other hand, transforming the 2D polygon to a valid triangular mesh that includes exclusively triangles inside the (possibly concave) polygonal border is another challenge. We approach this problem by first computing a constrained 2D Delaunay Triangulation. This guarantees that the polygonal borders are part of the triangulation. The triangulation generates a convex mesh representation, which means it may also include exterior triangles outside of the polygon constraint. In a second step, we consequently eliminate the unnecessary triangles. We extract the triangles for each polygon separately. Thus, we know which polygon each triangle belongs to. The resulting meshes for an example region are shown in Fig. 1(a)- 1(b).



(a) (b)

Figure 1. polygon-conform constrained 2D Delaunay Triangulation: A straight implementation of the constrained 2D Delaunay Triangulation creates exterior triangles (a), which need to be eliminated. We do this by testing if each triangle is inside the polygon. The result is a polygon-conform 2D mesh (b).

Due to the massive point sets and the numerous triangles, it is still not possible to test each point with each triangle for intersections while maintaining interactive frame rates. We incorporate the triangles in a quadtree data structure in order to reduce the number of triangles that needs to be checked. In the quadtree construction, we start off with a single cell and sequentially add triangles to the quadtree. If the number of triangles in a cell exceeds a limit X ($X = 4$ in our experiments), we split the cell into 4 equal parts. We continue the subdivision up until a certain maximum depth Y ($Y = 5$ in our experiments). Applying this maximum depth limits the tree traversal processed and proved efficient in our implementation.

During runtime, we traverse the tree to determine for each point its respective quadtree node. After extracting a reduced list of candidate triangles (resp. leafs of the final node) for each point, we perform the aforementioned *PointTriangleIntersection*-test to accurately establish a point-triangle relation while maintaining interactive frame rates.

At this point, we have access to the triangle attributes (such as colour and texture coordinates) and their textures. By applying the stored triangle colour or evaluating its texture coordinate inside the texture, we can recover the modification operation for the given point.

Currently, we support the following operations:

- colour a point according to the triangle colour
- discard a point if it is inside the triangle
- colour a point according to the triangle's texture
- displace a point according to a displacement map

The final algorithm is shown in Alg. 2 and 3.

Results

In a first use case, we apply our algorithm together with the discard-operation to create a historic visualisation of the 1953 North Sea flood in the Dutch provinces of South Holland, North Brabant and Zeeland. This visualisation spans an area of 45 km by 60 km, containing around 2 TB of coloured LiDAR data. The problem for a historic visualisation is the amount of landscape- and water protection changes that occurred since 1953, in comparison to the acquired data we currently have. In order to recreate the scenario, we have to adapt the landscape and cut out structures that were added since then. Without the presented approach, this would mean to re-process the 2 TB dataset and discard unavailable points in the process. This process takes several days of computing time. With our current method, we can define the areas we want to exclude from the visualisation, and all point vertices that are covered by the defined areas are discarded by the GPU during rendering. The result is shown on parts of the data set in Fig. 2(a) and 2(b).

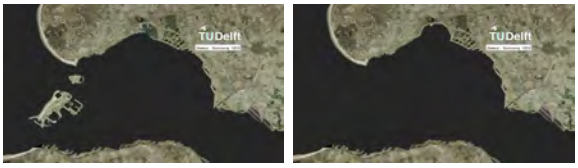
Algorithm 2 triangle texture generation

```
function PolygonToTriangleMesh(polygon)
    ConstrainedDelaunay2D(polygon)
    for all triangles ∈ trianglelist do
        if triangle not inside polygon then
            Delete(triangle,trianglelist)
        end if
    end for
    return trianglelist
end function

function MeshToTexture(trianglelist)
    ▷ converts mesh into a quadtree, then encodes as texture
    create quadtree with root node
    for all triangles ∈ trianglelist do
        locate node in quadtree for triangle
        if depth(node) < Y then
            if length(siblings(node)) > X then
                sibling ← SplitNode(node)
                node ← sibling
            end if
        end if
        AddTriangleLeaf(node,triangle)
    end for
    texture ← Texture1D()
    SerialiseQuadTree(quadtree, texture)
    return texture
end function
```

Algorithm 3 point attribute modification on the GPU

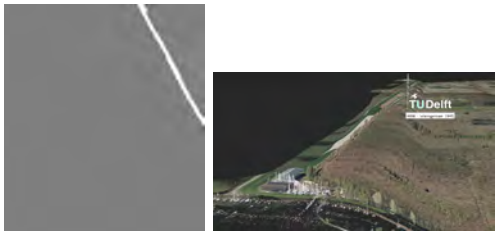
```
for all point ∈ pointlist do
    function GetAttribute(pointlist, texture)
        quadtree ← Texture1D(texture)
        node ← root
        while (depth > 5) ∧ (point ∈ node) do
            node ← Sibling(point, node)
            if node = NULL then
                attribute ← 0
                return attribute
            end if
        end while
        for all triangle ∈ node do
            if PointTriangleIntersect(point, triangle) is true then
                attribute ← GetAttribute(triangle)
                return attribute
            end if
        end for
        attribute ← 0
        return attribute
    end function
    ApplyAttribute(point, attribute)
end for
```



(a) (b)

Figure 2. Visualisation of the 1953 North Sea Flood: originally acquired data (a) and historically adapted data (b)

In a second use case, flood hazard managers can use the system to interactively discuss dyke adaptation strategies. For this case, we generate a displacement map together with the area marking. The displacement can be modified during the runtime to update the point vertex displacement at the dyke. This allows interactive discussions on future adaptation strategies. In combination with the flood simulation visualisation, it allows first estimates of the discussed measures' applicability. The adapted point cloud can be seen in Fig. 3(b), the related displacement map in Fig. 3(a).

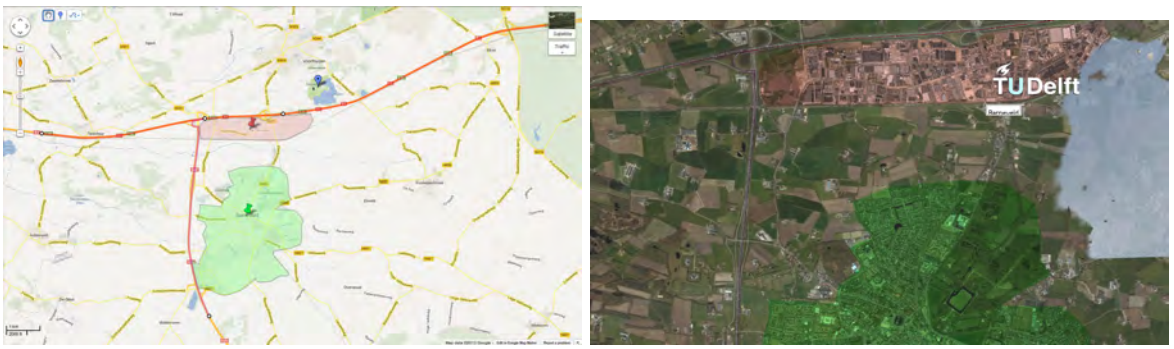


(a) (b)

Figure 3. Dyke adaptation to discuss water protection policy measures, with applied displacement (a) and the resulting point cloud of the dyke (b).

In the final use case, we show an area of 15 km by 20 km near the Dutch city of Barneveld. In this use case, policy managers can discuss protection strategies by marking different areas of importance interactively. For this use case, we colour the point cloud according to polygon colours that are given by a GoogleMaps KML-file. This allows policy managers to use familiar interfaces via mobile devices in order to interact with the point cloud data. In the example shown in Fig. 4(a) and 4(b), we distinguish between an industrial area and the residential parts of the city (marked in red respectively green).

Our initial performance assessment on the interactive point cloud modification reveals a tight correlation between the rendering speed and the geometric complexity of the polygon (i.e. the num-



(a) (b)

Figure 4. Interactive colouring of big LiDAR data (b) using GoogleMaps interfaces (a).

ber of triangles resulting from the triangulation). For the assessment, we have chosen data sets of 3, 10 and 20 areas, including 20, 42, and 298 triangles. Renderings of these data sets are shown in Fig. 5. The speed measurements are presented in Fig. 6. The test system was a workstation with a 6 x 3.2GHz Intel Xeon processors (HT), 16GB of main memory and an NVIDIA GeForce GTX 680 graphics adapter.

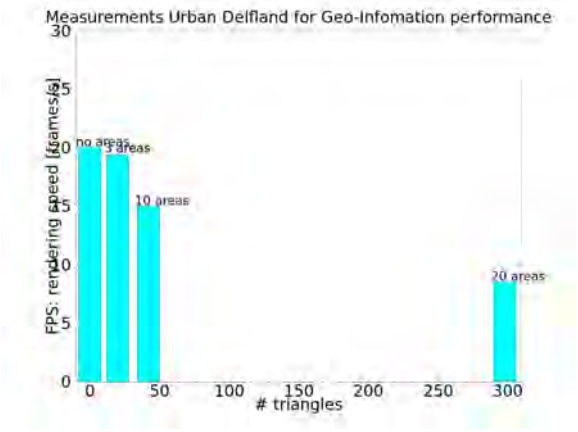


Figure 6. rendering speed measurements for geo-information integration with 3, 10 and 20 areas.

Conclusions

We presented an approach to render and interact with massive aerial LiDAR point sets.

Our rendering-on-budget approach results in a smooth level-of-detail transition at real-time frame rates.

With our on-line modification algorithm and its implementation, we have shown that even out-of-core datasets allow on-the-fly modification without tedious preprocessing procedures. The algorithm is a first step towards more elaborate procedures that allow on-the-fly marker placement and interaction as well as full 3D-modification via volume images or 3D polygons.

Our system, combining all discussed methods, enables decision makers in flood hazard management to collaboratively discuss protection scenarios and devise new protection measures.

Acknowledgements

This project was funded by the Dutch Research Program Knowledge for Climate (KfC) and partially supported by the Intel VCI, and the European Project Harvest4D.

References

[1] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. GigaVoxels: ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, I3D '09, pages 15–22, New York, NY, USA, 2009. ACM. Cited by 0115.

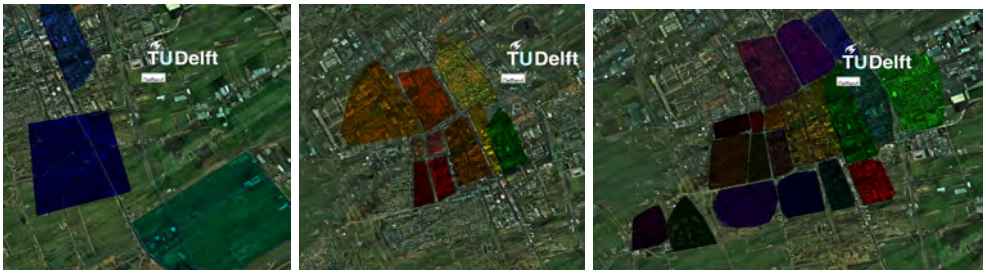
[2] G. de Haan. Scalable visualization of massive point clouds. *Nederlandse Commissie voor Geodesie KNAW*, 49:59–67, 2009.

[3] Zhenzhen Gao, Luciano Nocera, and Ulrich Neumann. Fusing oblique imagery with augmented aerial LiDAR. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '12, pages 426–429, New York, NY, USA, 2012. ACM.

[4] Enrico Gobbetti, Fabio Marton, and JosÃĭ Antonio Iglesias GuitiÃąn. A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24(7–9):797–806, 2008.

[5] Prashant Goswami, Fatih Erol, Rahul Mukhi, Renato Pajarola, and Enrico Gobbetti. An efficient multi-resolution framework for high quality interactive rendering of massive point clouds using multi-way kd-trees. *The Visual Computer*, February 2012.

[6] H. Jänicke, M. Bottinger, and G. Scheuermann. Brushing of attribute clouds for the visualization of multivariate data. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1459–1466, 2008.



(a) (b) (c)

Figure 5. Geo-information integration use cases: 3 (a), 10 (b) and 20 (c) area data set

[7] Christian Kehl and Gerwin de Haan. Interactive simulation and visualisation of realistic flooding scenarios. In *Intelligent Systems for Crisis Management*, 2012.

[8] Bostjan Kovac and Borut Zalik. Visualization of LIDAR datasets using point-based rendering technique. *Computers & Geosciences*, 36(11):1443–1450, November 2010.

[9] Oliver Kreylos, Gerald W. Bawden, and Louise H. Kellogg. Immersive visualization and analysis of LiDAR data. In George Bebis, Richard Boyle, Bahram Parvin, Darko Koracin, Paolo Remagnino, Fatih Porikli, Jörg Peters, James Klosowski, Laura Arns, Yu Ka Chun, Theresa-Marie Rhyne, and Laura Monroe, editors, *Advances in Visual Computing*, number 5358 in Lecture Notes in Computer Science, pages 846–855. Springer Berlin Heidelberg, January 2008.

[10] Wenqing Li, Ge Chen, Qianqian Kong, Zhenzhen Wang, and Chengcheng Qian. A VR-Ocean system for interactive geospatial analysis and 4Dvisualization of the marine environment around antarctica. *Computers & Geosciences*, 37(11):1743–1751, November 2011.

[11] Ruggero Pintus, Enrico Gobbetti, and Marco Agus. Real-time rendering of massive unstructured raw point clouds using screen-space operators. In *Proceedings of the 12th International conference on Virtual Reality, Archaeology and Cultural Heritage*, VAST'11, pages 105–112, Aire-la-Ville, Switzerland, Switzerland, 2011. Eurographics Association.

[12] W. Ribarsky, N. L. Faust, Z. J. Wartell, C. D. Shaw, and J. Jang. Visual query of time-dependent 3D weather in a global geospatial environment. 2002.

[13] Claus Scheiblauer and Michael Wimmer. Out-of-core selection and editing of huge point clouds. *Computers & Graphics*, 35(2):342–351, 2011. <ce:title>Virtual Reality in Brazil</ce:title> <ce:title>Visual Computing in Biology and Medicine</ce:title> <ce:title>Semantic 3D media and content</ce:title> <ce:title>Cultural Heritage</ce:title>.

[14] Michael Wand, Alexander Berner, Martin Bokeloh, Arno Fleck, Mark Hoffmann, Philipp Jenke, Benjamin Maier, Dirk Staneker, and Andreas Schilling. Interactive editing of large point clouds. In *Proceedings of Symposium on Point-Based Graphics (PBG 07)*, pages 37–46, 2007.

[15] Michael Wand, Alexander Berner, Martin Bokeloh, Philipp Jenke, Arno Fleck, Mark Hoffmann, Benjamin Maier, Dirk Staneker, Andreas Schilling, and Hans-Peter Seidel. Processing and interactive editing of huge point clouds from 3d scanners. *Computers & Graphics*, 32(2):204 – 220, 2008.

[16] Berend Wouda. Visualization on a budget for massive lidar point clouds. Master's thesis, Delft University of Technology, 2011.

[17] Izham Mohamad Yusoff, Muhamad Uznir Ujang, and Alias Abdul Rahman. 3D dynamic simulation and visualization for GIS-based infiltration excess overland flow modelling. In Jiyeong Lee and Sisi Zlatanova, editors, *3D Geo-Information Sciences*, Lecture Notes in Geoinformation and Cartography, pages 413–430. Springer Berlin Heidelberg, 2009.

[18] Matthias Zwicker, Mark Pauly, Oliver Knoll, and Markus Gross. Pointshop 3D: an interactive system for point-based surface editing. *ACM Trans. Graph.*, 21(3):322–329, July 2002.

Visibility Sweeps for Joint-Hierarchical Importance Sampling of Direct Lighting for Stochastic Volume Rendering

Thomas Kroes* Delft University of Technology
Martin Eisemann† Delft University of Technology
Elmar Eisemann‡ Delft University of Technology

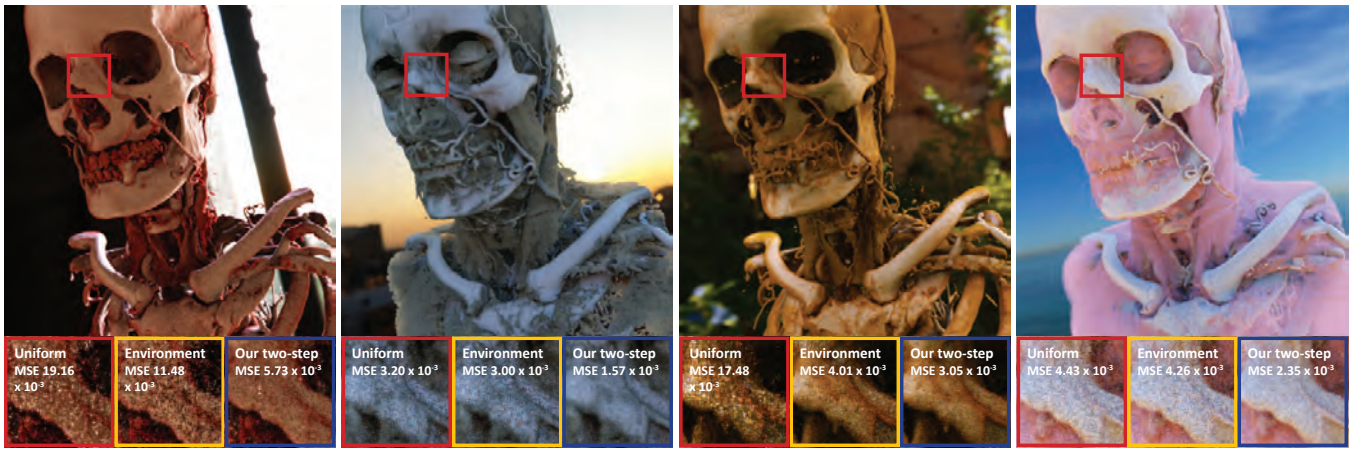


Figure 1: We compute the product of approximated visibility and environment map lighting in a stochastic Monte Carlo volume renderer to steer a joint importance sampling of the direct lighting. Our proposed two-step approach is well suited for dynamic changes in visibility and lighting functions due to a fast sweeping-plane algorithm to estimate visibility. The insets show how our technique (blue) achieves faster convergence with less samples compared to a uniform sampling (red) and importance sampling of the environment map (yellow). Here, 64 samples per pixel have been used. The Manix data set consists of $512 \times 512 \times 460$ voxels.

Abstract

Physically-based light transport in heterogeneous volumetric data is computationally expensive because the rendering integral (particularly visibility) has to be stochastically solved. We present a visibility estimation method in concert with an importance-sampling technique for efficient and unbiased stochastic volume rendering. Our solution relies on a joint strategy, which involves the environmental illumination and visibility inside of the volume. A major contribution of our method is a fast sweeping-plane algorithm to progressively estimate partial occlusions at discrete locations, where we store the result using an octahedral representation. We then rely on a quadtree-based hierarchy to perform a joint importance sampling. Our technique is unbiased, requires little precomputation, is highly parallelizable, and is applicable to a various volume data sets, dynamic transfer functions, and changing environmental lighting.

Index Terms: I.3.3 [Computer Graphics]: Picture/Image Generation—Viewing algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1 Introduction

Stochastic volume rendering is computationally intensive. To evaluate the rendering equation, many samples (rays) are required in order to compute the light distribution within a volume. In practice, rays are sent from the camera through the volume and a scattering event occurs at random positions along the ray based on the current transfer function, which maps the volume's density values to material properties. Each scattering event requires generating one or more sample rays to evaluate the rendering equation via Monte Carlo (MC) integration. These rays are ultimately absorbed or potentially hit a light source, e.g., the environmental light. Using standard sampling techniques at the scattering events can be inefficient, as no knowledge about the volume absorption or light characteristics is used. As a result, many rays might contribute little or nothing to the final image.

Importance-sampling techniques [7, 5, 23] incorporate knowledge about the scene to place more effort on potentially light-carrying paths to accelerate the convergence of the result. Some approaches combine information about the material and light positions. However, one important factor, the (volumetric) scene, and, hence, visibility is not taken into account. Previously, visibility approximations were only used directly in the shading evaluation, resulting in biased images [26]. Furthermore, a brute-force visibility precomputation is costly and transfer-function changes require a complete reevaluation.

In this work, we focus specifically on evaluating direct lighting for a volume data set with arbitrary and interactively changing transfer functions defining varying diffuse materials in the context of an unbiased MC-based stochastic volume renderer. The volume is lit by a natural illumination in the form of environmental lighting.

The key idea of our approach is to use environmental light and visibility represented as a joint probability density function (pdf), Fig. 2. As a result, the sampling process, steered by this pdf, becomes more efficient than in previous work, while keeping the result unbiased. The sampling technique allows us to evaluate the

*e-mail: t.kroes@tudelft.nl
†e-mail: m.eisemann@tudelft.nl
‡e-mail: e.eisemann@tudelft.nl

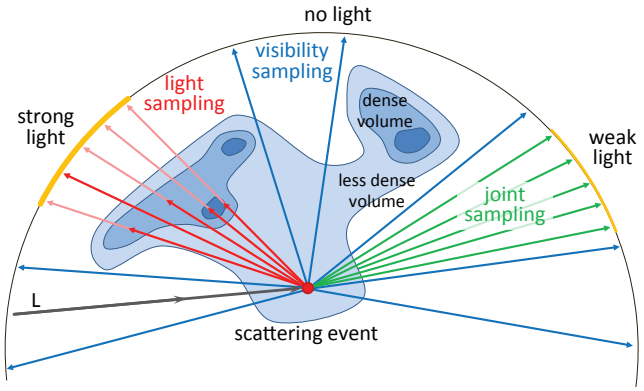


Figure 2: **Problem Statement:** For efficient sampling, samples with both strong light and strong visibility need to be found. Sampling according to the lighting only (red) may give bad results as the samples get absorbed, sampling only according to the visibility (blue) might miss important lights. Product sampling (green) solves the problem. Unfortunately, the visibility is usually unknown beforehand.

direct light at any scattering event within the volume. While our results could be generalized, we illustrate the application to single scattering.

Our contributions are as follows:

- An efficient sweeping-plane algorithm to compute approximate visibility within a 3D volume;
- A product importance sampling solution based on joint environmental light and visibility information;
- A GPU-adapted and highly-parallel implementation.

Our technique is useful for any volumetric renderer with dynamically changing content, such as environmental light, transfer functions, etc., making it an interesting addition to visualization and rendering systems aiming for unbiased results.

2 Related Work

The literature on volumetric-illumination techniques is vast, which is why we will focus only on certain aspects to put our approach in perspective. A recent survey on this topic can be found in [10].

Ambient Occlusion helps to better perceive certain shapes and their relative positions by measuring the light accessibility for each scene point. Luminance is linked to the degree of local occlusion [34]. Multi-resolution variants [16], and even dynamic ambient occlusion variants [28], which allow changes to the transfer function, have been considered. Nonetheless, ambient occlusion computes only a statistical scalar value to approximate the ambient light, which means that directional information is lost. We incorporate full directional support for high-quality unbiased physically-based rendering.

Visibility Approximation for Semi-Transparent Structures are most common in physically-based volume rendering. Opacity shadow maps [11] are an extension of shadow maps [33] using a stack that stores alpha values instead of depth values to support shadow computation for complex, potentially semi-transparent structures. Deep shadow maps [17] are a more compact representation, which store a shadow-function approximation per pixel. They have quickly been adopted for volume rendering [9, 27].

All such techniques are fast but inapplicable in our scenario of stochastic MC volume rendering. First, using approximate visibility directly for shading introduces a bias, which is unacceptable for certain applications. Second, these techniques support only point and directional light sources, whereas we aim for environmental lighting. Third, visibility is costly to compute and even approximating it can usually involve many rays, although not all locations might ultimately contribute to the image. Our approach computes visibility in a coarse 3D grid and uses it only to carefully steer the sample generation. In this way, our result remains unbiased, exact, and supports arbitrary environmental lighting.

Basis-Function Techniques decouple light-source radiance and visibility, which allows for dynamically changing the illumination. Spherical harmonics (SH) are prominent basis functions, used for example for precomputed radiance transfer [30], and were first used in the context of volume rendering to precompute and store isosurface illumination [2]. They have also been used to store visibility for volume rendering under natural illumination [26]. Other research in this area mostly aimed at generalizations to support advanced material properties [15] or reduce memory costs [14].

While SH are well suited to represent low-frequency functions, their direct use for visibility is a strong approximation and introduces bias. Further, only low-frequency illumination is supported, in contrast to our solution.

Image Plane-Sweep Volume Illumination Approaches move a virtual plane through a scene to invoke the shading computations for all positions within the plane in parallel. The parallelism makes these approaches highly applicable to modern architectures, such as the GPU. Using carefully-chosen approximations (e.g., a forward peaked phase function, single point or directional light source), single and forward multiple scattering effects can be simulated at interactive frame rates [31]. We decouple the plane sweep from a particular light source to enable general illumination and efficient sampling in stochastic MC volume rendering.

Recently, iterative convolutions on volume slices have been used to approximate direct lighting [22]. The results are approximate, some parameter settings have to be carefully chosen, and only particular light-source configurations are efficiently supported (e.g., usually Gaussian and behind the observer).

MC Ray Tracing for volume rendering gained attention with the advances of modern GPUs, which made interactive progressive rendering possible. First attempts sacrificed generality for performance [25] and did not support translucent materials. New approaches, such as Exposure Render [13] achieve images of very high realism. They employ all the benefits of physically-based MC techniques: arbitrary natural illumination, real-world cameras with lens and aperture (e.g., for depth-of-field effects). We implemented our approach building upon this open source solution. Only recently, specialized algorithms have been developed to efficiently handle participating media by splitting the evaluation into an analytical and a numerically evaluated part [20].

Importance Sampling is a powerful sampling technique to render objects illuminated by natural or complex lighting [7] under an environmental illumination. An efficient method for non-specular materials is to place pre-integrated directional lights at the brightest locations [1, 12, 21]. These methods work extremely well in the absence of occlusion, but shadowed regions may appear noisy. When materials are increasingly specular, a large number of lights is needed to adequately represent the environment map. Consequently, many physically-based MC techniques sample the environment map directly to avoid any artifacts and its intensity can even be used as a pdf to steer the sampling [23].

If also visibility or material properties are to be included, the pdfs can be combined in a single MC estimator via multiple importance sampling (MIS) [32]. MIS is most efficient if only one of the

sampled functions is complex and will pick the best one. If both are complex, MIS provides little advantage and is likely to waste samples in regions with little influence. Visibility and lighting can both be complex and only a joint sampling of both functions can be efficient (Fig. 2). A first step towards this direction was taken in [3]. Their technique importance samples the environment map to produce a candidate sample. Its probability is then evaluated again using a special pdf involving the BRDF to determine if an evaluation is triggered. Such a sampling can quickly become costly, due to potential high rejections rates (in the order of 90%) [3].

More related to our sampling approach are techniques for joint importance sampling that compute the BRDF/environment-map product [5, 6, 4] and BRDF/visibility/environment-map product [29] to steer sample placement. In the context of participating media, joint importance sampling can also be employed to optimize volumetric paths [8]. In this article, we focus on efficient visibility/environment-map sampling. Nonetheless, we also rely on a quadtree-based product to hierarchically warp samples [4].

3 Overview

In the following, we will describe our algorithm in detail. First, we provide the necessary background knowledge (Sec. 3.1). Then, we describe our actual solution, starting with our data structures and data representations (Sec. 3.2), which are designed with GPU-efficiency in mind. Our visibility-sweep algorithm (Sec. 3.3) is used to compute an approximate visibility within the volume. It is then used in conjunction with the scene illumination to yield a joint sampling technique to steer the MC evaluation (Sec. 3.4). Finally, we describe the necessary implementation details (Sec. 3.5). The benefits for convergence behavior and the support of dynamic lighting and transfer-function changes will be demonstrated in Sec. 4.

3.1 Background and Goal

We adopt the notation from [26] for the emission-absorption volume-rendering equation [18] in an isotropic medium:

$$L = \int_0^\infty A(t)E(\mathbf{x}(t))\mathrm{d}t. \tag{1}$$

It describes the recorded radiance L along a camera-ray position $\mathbf{x}(t)$ parameterized by t, where

$$A(t) = exp\left(-\int_0^t \tau_\alpha(D(\mathbf{x}(s)))\mathrm{d}s\right) \tag{2}$$

$$E(\mathbf{x}(t)) = \int_\Omega \tau_\rho(D(\mathbf{x}(t)))V(\mathbf{x}(t),\omega)L_i(\omega)\mathrm{d}\omega. \tag{3}$$

E is the emission and $A(t)$ is the absorption up to position $\mathbf{x}(t)$. The volume density at location $\mathbf{x}(t)$ is denoted as $D(\mathbf{x}(t))$. The visibility at a position $\mathbf{x}(t)$ in direction ω is denoted as $V(\mathbf{x}(t),\omega)$. The incoming light $L_i(\omega)$ from direction ω, integrated over all possible directions Ω, is assumed to be independent of $\mathbf{x}(t)$, i.e., we assume an environmental light. A transfer function τ maps a density value y to an extinction coefficient $\tau_\alpha(y)$ and scattering albedo $\tau_\rho(y)$. For brevity, we will omit the ray parameter t and write only $\mathbf{x}$ to denote a certain location.

We use stochastic ray marching to solve the integral in Eqs. (1) and (2) based on [13]. To solve Eq. (3) stochastically, MC integration is applied:

$$E(\mathbf{x}) = \frac{1}{N}\sum_{j=1}^{N}\frac{\tau_\rho(D(\mathbf{x}))V(\mathbf{x},\omega_j)L_i(\omega_j)}{p(\mathbf{x},\omega_j)}.$$

Here, p is a pdf that is used to weigh and generate the random sample vectors ω_j. The MC integration can become highly inefficient with a bad choice of the pdf p as it may create many samples ω_j which contribute little to the final result, Fig. 2.

The focus of this paper is on choosing an effective pdf p and its efficient computation. In order to achieve this, we split p into two components

$$p(\mathbf{x},\omega) = \frac{1}{W(\mathbf{x})}p_V(\mathbf{x},\omega)p_{L_i}(\omega).$$

p_V is a pdf based on the visibility, which changes locally throughout the volume based on the location $\mathbf{x}$, p_{L_i} is a pdf based on the position-independent environmental lighting and $W(\mathbf{x}) = \int p_V(\mathbf{x},\omega)p_{L_i}(\omega)\mathrm{d}\omega$ is a normalization factor to produce a valid pdf. p_{L_i} is known and based on the intensity of the environmental lighting, normalized by its overall intensity. The representation of these functions, the computation of $p_V(\mathbf{x},\omega)$, $p(\mathbf{x},\omega)$, and how to draw samples from $p(\mathbf{x},\omega)$ are the core of our method and explained in the following sections. We explain the data structures, then the visibility approximation, which will be used to derive p_V, before combining all the elements.

3.2 Octahedral Representation

Before explaining the algorithmic part of our approach, we will focus on the chosen data structures. They were developed to ensure an efficient evaluation on modern hardware and to simplify generation, sampling, and product computation. These elements will be necessary to drive the MC sampling process.

As we are dealing with potentially semi-transparent media in volume rendering, we assume V to be locally smooth with respect to $\mathbf{x}$ and ω. This allows us to estimate V at discrete positions $\mathbf{x}_d$ and a few discrete directions ω_d. We arrange the locations within a 3D voxel grid of user-defined size encompassing the original volume. These local estimates are then interpolated during rendering to obtain an approximation of the actual visibility in each location.

For a fixed location $\mathbf{x}$ our functions V and L_i and their respective pdfs p_V and p_{L_i} are spherical functions, i.e., they solely depend on a direction vector ω. Given that we only consider piecewise-constant pdfs, we will represent these functions as octahedral maps, which is a discrete image-based area-preserving representation [24] and can be saved/accessed as a 2D texture (Fig. 3). Each texel p in these maps is associated with one direction $\omega_d(p)$ and indicates the accumulated volumetric visibility in direction $\omega_d(p)$ from the maps location. We will refer to these visibility maps for each discrete location $\mathbf{x}_d$ as *visibility voxels* $\mathbf{V}_d$.

3.3 Visibility Approximation

In this section, we describe how to compute the entries of the visibility voxels via our sweeping plane algorithm. The visibility is

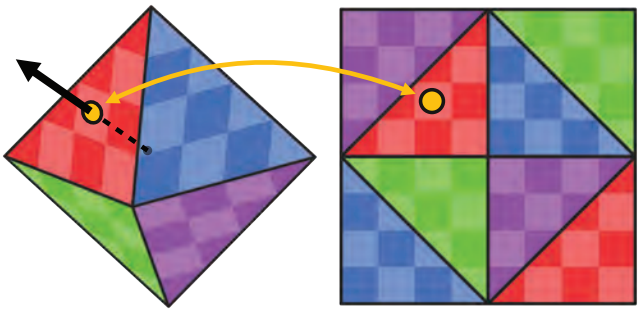


Figure 3: **Octahedral representation:** We present spherical functions using an octahedral representation. (a) 3D representation, (b) unfolded 2D representation.

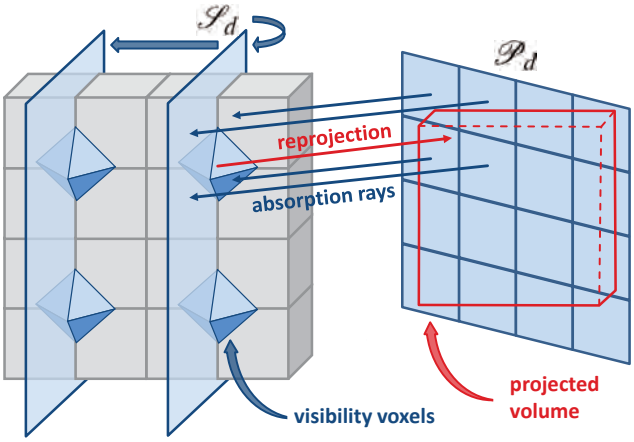


Figure 4: **Visibility Sweeps:** We compute the absorption of sample rays starting at plane n along direction ω_d up to the positions coinciding with a sweeping plane, which is orthogonal to the main component of ω_d. To compute the absorption at a visibility voxel in direction $-\omega_d$ we reproject its position onto $\mathscr{P}_d$ and query the interpolated absorption value. All components can be efficiently computed on the GPU.

computed for one direction $\omega_d(p)$ at the time. In each step, one slice of $\mathbf{V}_d$ is evaluated in parallel. Previous results are reused, making only a few value lookups per step necessary. Therefore, the amortized cost over all $\mathbf{x}_d$ is very low. After all directions were treated, the resulting $\mathbf{V}_d$ is used to derive the pdf p_V, which will guide the sampling process. An illustration of a single sweep step is given in Fig. 4 and described in the following.

Here, we describe the process for one given direction ω_d, for brevity we omit the direction parameter in brackets. First, a plane $\mathscr{P}_d$ with normal ω_d is defined, the orthogonal projection of the data volume's bounding box defines its size. We then create a set of r rays at uniformly distributed positions within this projection having the same direction as ω_d. To coordinate the ray traversal, we introduce a sweeping plane $\mathscr{S}_d$ which is orthogonal to one of the main axes of the original volume data. This axis is chosen based on the main direction of ω_d (defined as the maximum of the absolute values of its three components). $\mathscr{S}_d$ is initialized to intersect the first 2D slice of $\mathbf{V}_d$, so that it coincides with the position of the visibility voxels within this slice. We traverse the volume along the rays starting at $\mathscr{P}$ and accumulate visibility changes until they hit $\mathscr{S}$. This accumulation effectively keeps track of all relevant information that lies behind the ray when reaching a new visibility voxel.

The main loop of the algorithm moves $\mathscr{S}_d$ forward along the main direction by one visibility-voxel slice at a time until all slices are processed. After each step of $\mathscr{S}_d$, the rays advance via ray marching from their previous position until they reach $\mathscr{S}_d$ again. On their way, the absorption values along the ray are accumulated and, when reaching $\mathscr{S}_d$, stored in a 2D texture mapped onto the initial positions on $\mathscr{P}_d$. Please note that the resolution of the visibility voxel grid and the original volume can be different. Next, the visibility voxels coinciding with $\mathscr{S}_d$ (now reached by the rays) are updated by querying the interpolated absorption values produced by the rays. This gathering operation is highly parallelizable and more efficient than a scattering strategy.

After the algorithm finishes and all directions have been processed, we have a discrete approximation $\mathbf{V}_d$ of the visibility within the volume, which, if normalized, results in the pdf p_V. We add a small ε-value beforehand to prevent zero probabilities, which would introduce a bias. The main observation is that this iterative update is more efficient than individual visibility computations per visibility voxel.

3.4 Joint Importance Sampling

At a scattering event during rendering, we want to make use of a joint importance sampling combining visibility and environmental lighting. We have explained how to produce the pdfs for p_V and p_{L_i}. Here, we explain how to combine both. The computation is divided into a preprocess, taking place whenever the environment map or the transfer function changes, and an online process, taking place whenever a scattering event occurs during rendering.

Preprocess For the preprocess, we assume that the environment map is also given as an octahedral map, otherwise we convert it first. As a reminder, p_{L_i} is defined as the normalized intensity value of the environmental lighting, giving higher importance to the brighter parts. In general, the resolution of the octahedral maps of p_{L_i} will be higher than for p_V. To combine both, we first adapt the resolution of p_{L_i}. To simplify explanations, we assume that the resolution in width and height is chosen to be a power of two.

Similar to [4], we create a multiresolution pdf in the form of a quadtree, i.e., each node saves the average of its four child nodes, with the leaves being the individual pixels. To match the resolution between lighting and visibility, we choose a level l in p_{L_i} whose resolution is equal to the directional resolution of a single visibility voxel. We then multiply all entries in $\mathbf{V}_d$ with the respective information in p_{L_i} at level l. The result is an unnormalized joint pdf of the combined product.

Rendering In the rendering phase, we create a final combined pdf p for each scattering event at location $\mathbf{x}$. This pdf is used to draw a single sample, as this strategy is often more efficient in a stochastic volume renderer with semi-transparent media [13]. Nonetheless, the sampling algorithm naturally extends to any number and distribution of initial samples, including quasi-MC methods [19].

To derive the pdf p, we first linearly interpolate the neighboring visibility voxels which now carry the information of both visibility and lighting as described in the preprocess. Initially, this interpolated result is not a pdf. Nevertheless, we do not normalize it right away, but compute a multiresolution representation in the form of a quadtree where each node is the average of its child nodes. Following the hierarchical warping technique [5], we can then transform a uniformly distributed $[0,1)^2$-variable into one that is distributed according to p by passing the sample down in the quadtree according to the local probabilities. In contrast to [5], we need to normalize each 2×2 tile that we encounter during the quadtree sampling but as we only draw a single sample per scattering event the effort is only $O(\log n)$ compared to $O(n)$ if we would create a complete pdf for the interpolated visibility voxel. Here, n is the number of texels in the lowest level of the quadtree.

In case the environment map has a high resolution, we propose to use a *two-step approach* that continues the descent on the remaining quadtree of the higher resolved environment map [6, 4]. This step is especially beneficial for complex high-frequency illumination, which is otherwise not well taken into account during the sampling.

3.5 Implementation Details

We found that using 8×8 maps to represent the visibility at each $\mathbf{x}_d$ is generally a good memory/performance trade-off. The amount of visibility voxels should be based on the scene/feature size and $\mathbf{V}_d$ should be chosen slightly larger to encompass the original volume and ensure a correct boundary sampling. The number of ray-marching steps along the ray should be based on the resolution of the data set and the number of visibility voxels. The maximal step size should be equal to the voxel size in the original data set in order to not miss any details.

For improved cache usage, we actually do not compute all visibility-voxel octahedral maps in advance. Instead, we avoid that rays write to different textures during the ray traversal in the visibility precomputation and store the occlusion values for a direction ω_d in a separate 3D texture. Once the pass for ω_d is completed, we perform the multiplication with the environment map as explained above and keep this texture in memory. During the rendering process, when a scattering event occurs, it might not lie directly on a visibility voxel. We thus retrieve the interpolated values using hardware filtering from these 3D textures and construct the visibility-voxel octahedral map on the fly. Although this might sound costly, it turned out that in practice, this cost is outweighed by the cache advantages due to a better data locality and we avoid constructing visibility voxels in areas where no scattering occurs. Nonetheless, on future hardware a first reconstruction pass might become preferable.

4 Results

We integrated our algorithms into the stochastic CUDA-based volume renderer from [13]. All images have been generated on a 64bit Intel© Core™ i7 920 with 2.67GHz, 12GB of RAM, and an NVIDIA GeForce GTX 760.

We compare the performance and do a qualitative comparison between the existing and our approach. We compute the mean-squared error (MSE) and compared to reference solutions using 8196 samples per pixel with uniform sampling. Our environment maps all had a resolution of 2048×2048 pixels (in the octahedral representation). For all tests, the joint importance pdf p was constructed on the fly for each scattering event via interpolation of eight visibility voxels.

Timings and Parameters The overhead during rendering using our visibility sweeps is low compared to the gain in quality, especially as the sweeping-plane algorithm to update the visibilities in $\mathbf{V}_d$ is evoked only if the transfer function or illumination changes. As standard parameters, we use a 8^2 directional map for each visibility voxel and set one visibility voxel for each 4^3 voxel subset of the original data volume. The overhead during rendering is roughly only 10%, compared to rendering the same number of samples per pixel using plain uniform sampling. This includes interpolating the visibilities, creating the multiresolution 2D pdf representation on the fly and the joint importance sampling itself.

We compared our visibility sweeps approach to a brute-force computation of the visibility where each entry for the visibility voxels is computed exactly using ray marching. Table 1 shows a comparison of the timings for different parameters. For our proposed standard parameters and a reasonable number of absorption rays our approach is approximately $6\times$ faster than the brute-force computation. It is important to note that this factor becomes larger with an increasing number of visibility voxels (up to a factor of 15 in our tests in Table 1). Further, the test scene (Manix) resembles and isosurface due to its very steep transfer function, hence, the brute-force ray marching stops if the ray hits the isosurface. In our sweeping-plane algorithm the rays need to traverse the whole volume. So, we deliberately chose a difficult scenario - the benefit will be even bigger for a higher number of visibility voxels and more transparent volumes.

Additionally, we checked our assumption that we can interpolate the queried visibility during the reprojection step in the sweeping-plane algorithm. It should be pointed out that the results will always remain unbiased, independent of the resolution because the values are only used to guide the sampling process. To this extent, we reduced the number of absorption rays that are traversed through the volume shown in Fig. 5, which is of size $512 \times 512 \times 373$ voxels. Consequently, the visibility voxels will have to rely on an interpolated result. The number of rays influences the result only slightly and no visible errors are introduced. This result indicates that we can rely on a relatively cheap preprocess to approximate the visibility, which reduces the additional overhead in our approach.

Vis. Voxels	256x256x230	128x128x115	64x64x57	32x32x28
Memory	964.7	120.6	15.0	1.8
Sweep (16^2)	3.76	1.93	0.76	0.53
Sweep (32^2)	3.89	1.97	0.79	0.54
Sweep (64^2)	4.05	2.03	0.79	0.55
Sweep (128^2)	4.31	2.40	1.04	0.59
Sweep (256^2)	6.36	3.23	1.63	1.41
Brute-Force	97.35	15.17	2.79	0.57

Table 1: Memory requirements (MB) and timings (seconds) for the visibility sweep algorithm and varying input parameters in comparison to a brute-force visibility computation. We shoot 16^2, 32^2, 64^2, 128^2 and 256^2 absorption rays per sweeping direction. All experiments are performed on the Manix data set ($512 \times 512 \times 460$ voxels).

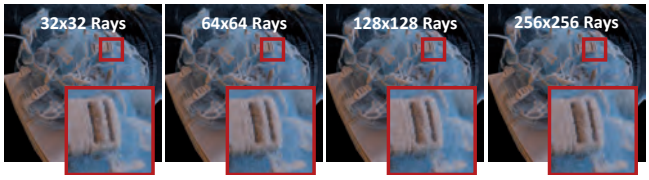


Figure 5: Influence of the visibility sampling precision (number of absorption rays) on the result.

Qualitative Evaluation We compare our approach to uniform sampling, importance sampling of the environment map only, importance sampling of the visibility only, a combined approach, where the visibility pdf is multiplied with the downsampled pdf from the environment map of the same resolution, as well as our combined two-step approach, which makes use of the combined sampling but switches to the full environment-map resolution as soon as a leaf in the combined pdf representation is reached to further support high-frequency lighting.

Fig. 8 shows an equal-time comparison of all the techniques after 10 s render time, excluding the visibility precomputation. For comparison, we show the computed number of samples per pixel (SPP) and the Mean-Squared Error (MSE) for each approach. Though the number of samples is lower, due to the computational overhead induced by the joint sampling, the noise is significantly reduced with our approach. Due to a lower ray coherency the uniform sampling creates less samples per pixel in the same time than most of the other approaches.

Figs. 1 and 6 show an equal sample comparison. Fig. 7 shows an equal quality comparison, where we precomputed the result for various power-of-two number of samples and illustrate the ones closest to the indicated error.

Additionally, we provide error plots for the Statue and Engine Block scene with respect to the number of samples in Figs. 9 and 10. As expected, uniform sampling performs worst. Interestingly, the visibility sampling alone performs better in the beginning but has a worse convergence. We found that the images also contain a lot more firefly artifacts. We, therefore, believe the reason for the convergence behavior lies in the approximate visibility function at occlusion boundaries. If a direction is supposed to be occluded it is sampled with a low probability and therefore a high weighting. If it accidentally hits a bright light source behind it, high energy samples are added to the result which result in the fireflies. As the direction around this occlusion boundary is rarely sampled it takes a lot of

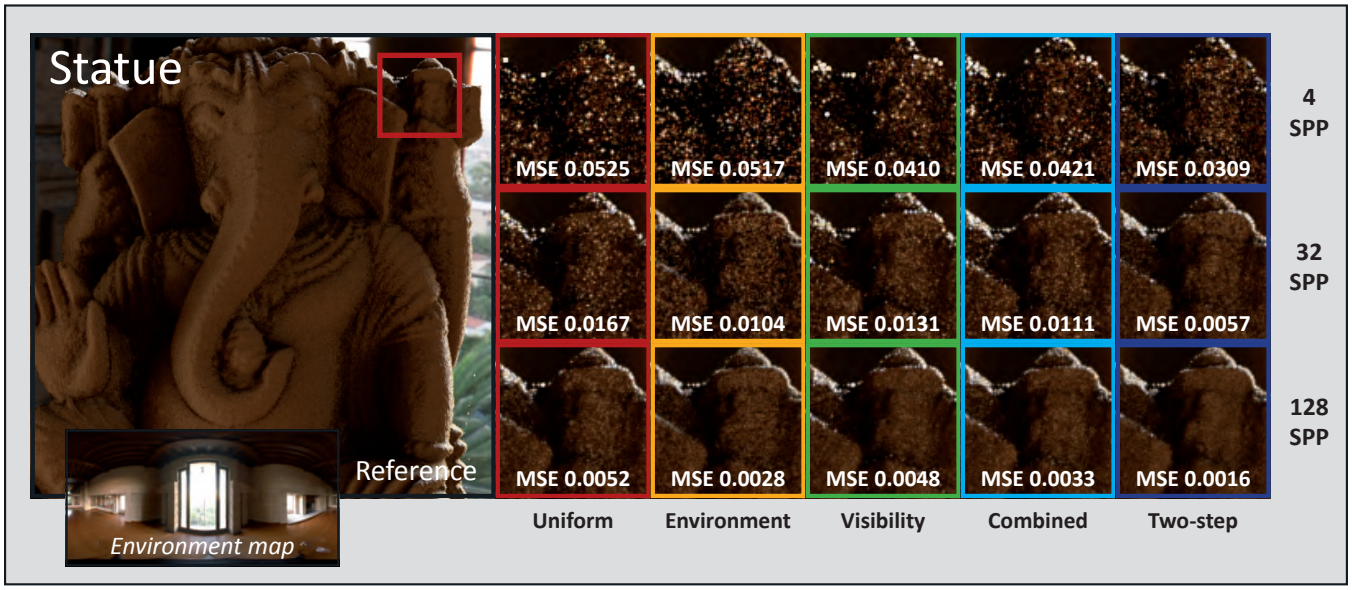


Figure 6: **Equal Sample Comparison:** We compare our proposed two step importance sampling technique (dark blue) using 4, 32 and 128 samples to uniform sampling (red) and importance sampling of the environment map only (yellow), the visibility only (green), and the combined low-resolution product (light blue). All images are unbiased and a reference, as well as the environment map are shown on the left.

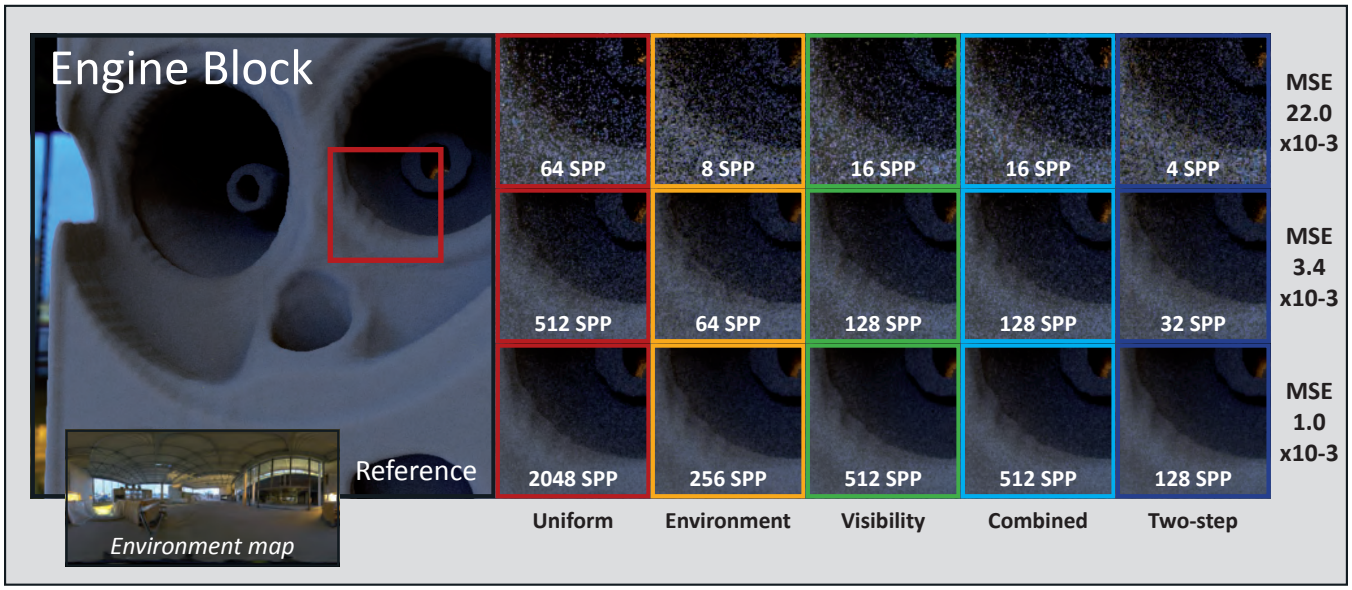


Figure 7: **Equal Quality Comparison:** We compare our proposed two step importance sampling technique (dark blue) to uniform sampling (red) and importance sampling of the environment map only (yellow), the visibility only (green), and the combined low-resolution product (light blue). All images are unbiased and a reference, as well as the environment map are shown on the left. For approximately the same quality, our two-step approach requires significantly less samples.

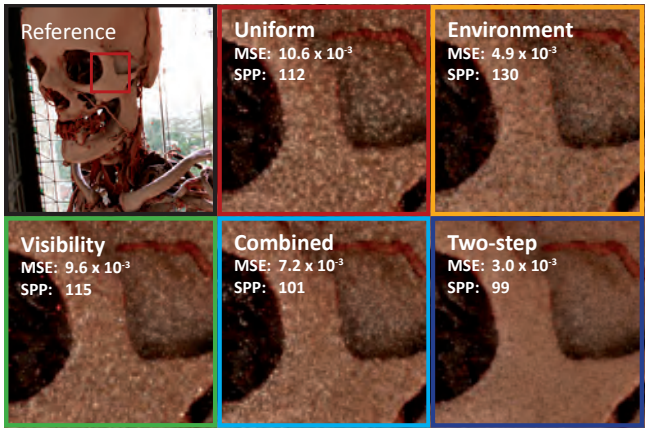


Figure 8: **Equal time comparison:** All images, except the reference image, have been created using 10 seconds of rendering time.

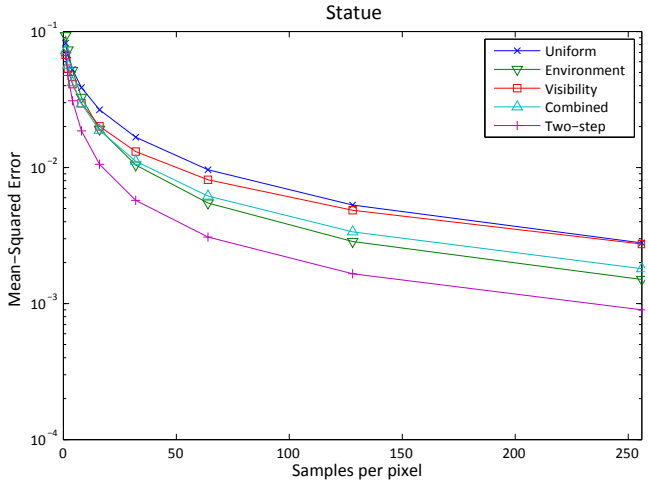


Figure 9: Convergence graphs for the Statue scene (Fig. 6)

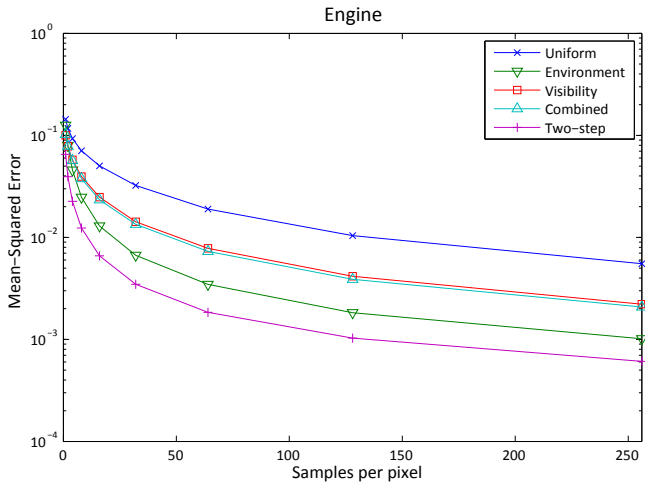


Figure 10: Convergence graphs for the Engine Block scene (Fig. 7).

samples to correct for these errors. If the lighting is incorporated in the pdf, these cases are taken care of sufficiently. Environment map and the low-resolution combined sampling perform almost equally well on the Statue scene. Presumably, importance sampling the light wastes a lot of samples that are absorbed within the volume. The combined sampling approach suffers to some extent from the low resolution of the visibility function and, therefore, the combined pdf is not able to capture the high frequency details of the environment map. This disadvantage is compensated by the two-step approach, which can make use of both the visibility and the high-frequency illumination information and shows better convergence rates even at high sampling rates. The results suggest that it is highly beneficial to incorporate the proposed visibility sweeps and joint sampling in the two-step approach for stochastic MC volume rendering.

5 Conclusion

We presented a joint sampling approach relying on visibility and lighting information within an interactive unbiased stochastic volume renderer. The core of our solution is an efficiently-computed visibility approximation based on a sweep-plane algorithm. Its performance allows us to change environmental lighting and transfer functions dynamically. We carefully designed our algorithm for GPU execution and have demonstrated its applicability to different volume data sets.

Visibility sweeps could prove beneficial for traditional boundary-representation rendering as well. To some extent this is illustrated by using transfer functions, which lead to very sharp features. Our approach usually lowers the amount of needed samples significantly compared to previous solutions at equal quality, which is an important result as the evaluation of samples is a very costly element in most production and rendering contexts.

There are still some options for minor optimizations for the traversal algorithm. First, a pruning of the absorption rays that do not intersect with the volume at all, and second, an early exit strategy for rays that are already fully absorbed, could potentially result in a traversal speed-up at the cost of a more complex algorithm. Though not yet implemented, interactive clipping (slicing) of the volume is naturally supported in our approach, as it simply requires disregarding the intensity values in front of the slicing plane during the visibility computation. A remaining challenge is the incorporation of non-diffuse media. For isotropic media one could precompute several pdfs based on the angle of the incoming and outgoing ray and interpolate these during rendering, but general reflection models remain future work.

Acknowledgements

This work was partially supported by the FP7 European Project Harvest4D and the IVCI at Saarland University.

REFERENCES

[1] S. Agarwal, R. Ramamoorthi, S. Belongie, and H. W. Jensen. Structured importance sampling of environment maps. *ACM Trans. Graph.*, 22(3):605–612, 2003.

[2] K. M. Beason, J. Grant, D. C. Banks, B. Futch, and M. Y. Hussaini. Pre-computed illumination for isosurfaces. In *Conference on Visualization and Data Analysis*, pages 1–11, 2006.

[3] D. Burke. Bidirectional importance sampling for illumination from environment maps. Master's thesis, UBC, 2004.

[4] P. Clarberg and T. Akenine-Möller. Practical product importance sampling for direct illumination. *Computer Graphics Forum (Proceedings of Eurographics)*, 27(2), 2008.

[5] P. Clarberg, W. Jarosz, T. Akenine-Möller, and H. W. Jensen. Wavelet importance sampling: Efficiently evaluating products of complex functions. *ACM Trans. Graph.*, 24(3):1166–1175, 2005.

[6] D. Cline, P. K. Egbert, J. F. Talbot, and D. L. Cardon. Two Stage Importance Sampling for Direct Lighting. In *Eurographics Symposium on Rendering*, 2006.

[7] P. Debevec. Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 189–198. ACM, 1998.

[8] I. Georgiev, J. Křivánek, T. Hachisuka, D. Nowrouzezahrai, and W. Jarosz. Joint importance sampling of low-order volumetric scattering. *ACM Transactions on Graphics (TOG)*, 32(6):164, 2013.

[9] M. Hadwiger, A. Kratz, C. Sigg, and K. Bühler. GPU-accelerated deep shadow maps for direct volume rendering. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '06, pages 49–52. ACM, 2006.

[10] D. Jönsson, E. Sundén, A. Ynnerman, and T. Ropinski. A Survey of Volumetric Illumination Techniques for Interactive Volume Rendering. *Computer Graphics Forum*, 33(1):27–51, 2014.

[11] T.-Y. Kim and U. Neumann. Opacity shadow maps. In *In Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 177–182. Springer-Verlag, 2001.

[12] T. Kollig and A. Keller. Efficient illumination by high dynamic range images. In *Rendering Techniques*, volume 44, pages 45–51. Eurographics Association, 2003.

[13] T. Kroes, F. H. Post, and C. P. Botha. Exposure render: an interactive photo-realistic volume rendering framework. *PLoS ONE*, 7(7), 2012.

[14] J. Kronander, D. Jönsson, J. Löw, P. Ljung, A. Ynnerman, and J. Unger. Efficient Visibility Encoding for Dynamic Illumination in Direct Volume Rendering. *IEEE TVCG*, 18(3):447–462, 2012.

[15] F. Lindemann and T. Ropinski. Advanced light material interaction for direct volume rendering. In *IEEE/EG International Symposium on Volume Graphics*, pages 101–108. Eurographics Association, 2010.

[16] P. Ljung, C. Lundström, and A. Ynnerman. Multiresolution interblock interpolation in direct volume rendering. In *Proc. of EuroVis*, pages 259–266. Eurographics Association, 2006.

[17] T. Lokovic and E. Veach. Deep shadow maps. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 385–392. ACM Press/Addison-Wesley Publishing Co., 2000.

[18] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

[19] H. Niederreiter. *Random Number Generation and quasi-Monte Carlo Methods*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[20] J. Novák, A. Selle, and W. Jarosz. Residual ratio tracking for estimating attenuation in participating media. *ACM Transactions on Graphics (TOG)*, 33(6):179, 2014.

[21] V. Ostromoukhov, C. Donohue, and P.-M. Jodoin. Fast hierarchical importance sampling with blue noise properties. *ACM Trans. Graph.*, 23(3):488–495, 2004.

[22] D. Patel, V. Šoltészová, J. M. Nordbotten, and S. Bruckner. Instant convolution shadows for volumetric detail mapping. *ACM Transactions on Graphics (TOG)*, 32(5):154, 2013.

[23] M. Pharr and G. Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., 2nd edition, 2010.

[24] E. Praun and H. Hoppe. Spherical parametrization and remeshing. *ACM Trans. Graph.*, 22(3):340–349, 2003.

[25] C. Rezk Salama. GPU-based monte-carlo volume raycasting. In *Proc. Pacific Graphics*, pages 411–414, 2007.

[26] T. Ritschel. Fast GPU-based Visibility Computation for Natural Illumination of Volume Data Sets. In P. Cignoni and J. Sochor, editors, *Short Paper Proceedings of Eurographics 2007*, pages 17–20, 2007.

[27] T. Ropinski, J. Kasten, and K. H. Hinrichs. Efficient shadows for GPU-based volume raycasting. In *Proceedings of the 16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG 2008)*, pages 17–24, 2008.

[28] T. Ropinski, J. Meyer-Spradow, S. Diepenbrock, J. Mensmann, and K. Hinrichs. Interactive volume rendering with dynamic ambient occlusion and color bleeding. *Comput. Graph. Forum*, 27(2):567–576, 2008.

[29] F. Rousselle, P. Clarberg, L. Leblanc, V. Ostromoukhov, and P. Poulin. Efficient product sampling using hierarchical thresholding. *The Visual Computer*, 24(7-9):465–474, 2008.

[30] P.-P. Sloan, J. Kautz, and J. Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 527–536. ACM, 2002.

[31] E. Sundén, A. Ynnerman, and T. Ropinski. Image Plane Sweep Volume Illumination. *IEEE TVCG(Vis Proceedings)*, 17(12):2125–2134, 2011.

[32] E. Veach and L. J. Guibas. Optimally combining sampling techniques for monte carlo rendering. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, pages 419–428. ACM, 1995.

[33] L. Williams. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12(3):270–274, 1978.

[34] S. Zhukov, A. Iones, and G. Kronin. An ambient light illumination model. In *Rendering Techniques, Proceedings of the Eurographics Workshop*, pages 45–56. Springer, 1998.

Pacific Graphics 2013
B. Levy, X. Tong, and K. Yin
(Guest Editors)

Volume 32 (2013), Number 7

Level-of-Detail Streaming and Rendering using Bidirectional Sparse Virtual Texture Functions

Christopher Schwartz, Roland Ruiters and Reinhard Klein

University of Bonn, Germany

Abstract

Bidirectional Texture Functions (BTFs) are among the highest quality material representations available today and thus well suited whenever an exact reproduction of the appearance of a material or complete object is required. In recent years, BTFs have started to find application in various industrial settings and there is also a growing interest in the cultural heritage domain. BTFs are usually measured from real-world samples and easily consist of tens or hundreds of gigabytes. By using data-driven compression schemes, such as matrix or tensor factorization, a more compact but still faithful representation can be derived. This way, BTFs can be employed for real-time rendering of photo-realistic materials on the GPU. However, scenes containing multiple BTFs or even single objects with high-resolution BTFs easily exceed available GPU memory on today's consumer graphics cards unless quality is drastically reduced by the compression. In this paper, we propose the Bidirectional Sparse Virtual Texture Function, a hierarchical level-of-detail approach for the real-time rendering of large BTFs that requires only a small amount of GPU memory. More importantly, for larger numbers or higher resolutions, the GPU and CPU memory demand grows only marginally and the GPU workload remains constant. For this, we extend the concept of sparse virtual textures by choosing an appropriate prioritization, finding a trade off between factorization components and spatial resolution. Besides GPU memory, the high demand on bandwidth poses a serious limitation for the deployment of conventional BTFs. We show that our proposed representation can be combined with an additional transmission compression and then be employed for streaming the BTF data to the GPU from from local storage media or over the Internet. In combination with the introduced prioritization this allows for the fast visualization of relevant content in the users field of view and a consecutive progressive refinement.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms I.3.2 [Computer Graphics]: Graphics Systems—Distributed/network graphics

1. Introduction

The realistic rendering of surface materials plays an important role in the generation of photo-realistic as well as predictive synthetic images. The appearance of real-world materials is often the result of complex light scattering interaction within small geometric structures on and under the surface. For a restricted set of materials, such as perfect mirrors, some metals or plastics, a visually pleasing rendering can be achieved by employing physically motivated analytical reflection models. However, the majority of the rich variety of material classes encountered in everyday life cannot as easily be represented by simple analytical models.

In 1997, Dana et al. [DvGNK97] proposed an image-based approach to acquire the appearance of real-world materials, the Bidirectional Texture Function (BTF). The six

Figure 1: *A scene with* 14 *objects, all textured with BTF materials with up to* 2048×2048 *texels, rendered on the GPU at interactive frame-rates using our BSVTF approach.*

Figure 2: *Screenshots of a fly-through animation in a virtual scene with* 100 *different measured BTF material having a size of* 512×512 *texels. The scene renders with* 35 FPS *at* 1280×720 *pixel on a NVIDIA GeForce GTX 680 GPU. While the total amount of the factorized BTF data is* 6.2 GB*, the memory footprint on the GPU using BSVTFs is* 1.7 GB.

dimensional Bidirectional Texture Function $\rho(\mathbf{x}, \omega_l, \omega_v)$ describes the ratio of differential radiance that is scattered at a point $\mathbf{x}$ on a surface into direction ω_v to differential irradiance from direction ω_l. In contrast to the related Spatially Varying Bidirectional Reflectance Distribution Functions (SVBRDF), which are often represented as either spatially varying parameters of an analytical BRDF or spatially varying mixtures of a set of analytical or measured BRDFs, the BTF is characterized by a unique Apparent BRDF (ABRDF) at each point of the surface. Unlike the BRDF, the ABRDF may violate assumptions such as reciprocity or energy conservation and show a more general distribution of the reflected light. This way, the BTF is able to encode position dependent non-local effects, like interreflections, shadows, masking or subsurface scattering, cast from neighboring geometry onto the material surface.

Because of this property, BTFs are an excellent choice for the representation of many real-world materials. In recent years, BTFs have started to find application in industrial settings and in the domain of cultural heritage, as in these cases the exact reproduction of the appearance of materials or complete objects is desired. Although raw BTFs exhibit rather unhandy file-sizes of several tens to hundreds of Gigabytes – recent publications show high resolution BTFs with up to 500 GB – several compression techniques are available to cope with this problem. Among those, matrix factorization based approaches are capable of reducing the file-size up to a factor of 500 while still preserving the unique appearance of the BTF and allowing rendering in real-time.

Therefore, these factorized BTFs are currently the representation of choice for high quality materials in interactive applications as well as real-time graphics. Unfortunately, so far the usefulness of BTFs in real-time graphics is greatly hampered by the still rather large data sizes of up to several hundreds of megabytes per material. An additional entropy coding or lossy compression can be employed to improve the compression ratio over the factorized BTF, e.g. for the fast transmission over the internet. However, the data needs to be unpacked into the factorized representation again to support efficient random access for real-time rendering. In [SRWK11], using a lossy wavelet compression for transmission, factors of 10 for no perceivable up to 60 for a noticeable but still acceptable error were achieved. The authors reported that a BTF with a compressed size of 46.39 MB for the transmission had then to be stored in 2.5 GB of GPU memory. Therefore, rendering even a small scene containing a few objects with high resolution BTF materials on the GPU is simply impossible due to the high memory requirements that can even be hardly met by the latest professional hardware. When considering the trend towards high quality 3D graphic on tablets and mobile phones, which nowadays have performant graphics chips but a drastic shortage of memory, the problem of GPU memory consumption becomes even more severe. However, the familiar problem of rendering very large textures that exceed the available memory has already been successfully handled by employing a technique that is known as Clipmapping [TMJ98] or *Sparse Virtual Texturing* (SVT) [Bar08]. SVT utilizes a level-of-detail hierarchy in the spatial domain to only keep the required parts of the texture in the necessary resolution in GPU memory.

In this paper, we propose the *Bidirectional Sparse Virtual Texture Function* (BSVTF), an adaption of the SVT technique to the context of real-time BTF rendering. In contrast to plain textures, which only have a level-of-detail hierarchy in their spatial resolution, a factorized BTF representation inherently includes a second level-of-detail domain of the ABRDF approximation quality. In this work, we demonstrate that both level-of-detail hierarchies can be combined in a consistent manner by reducing them to a single spatial level-of-detail problem. We show that this way the bottleneck of GPU memory can effectively be circumvented. In contrast to several tens to hundreds of Megabytes per high-resolution BTF, in our case the CPU and GPU memory demand is very moderate. More importantly, the memory demand grows only marginally with higher resolution and with increasing number of materials only additional storage space for the angular part of the factorized matrix is needed. Furthermore, the computational overhead on the GPU introduced by the approach remains constant regardless of the number of BSVTFs, allowing for rich virtual scenes that are textured with several high-resolution materials, such as the scene shown in Fig. 1. We demonstrate that BSVTFs can also be used for the efficient streaming over the Internet, allowing to display scenes with multiple high resolution materials without considerable delay. For this, we apply an additional streaming compression that utilizes the redundancy found in the level-of-detail hierarchy. To facilitate the fast

start of rendering, we interleave the angular factorization components with the transmission of the level-of-detail tiles of the spatial information.

In summary our contributions are

- A hierarchical level-of-detail approach for memory friendly real-time BTF rendering.
- An automatic weighting of the BTF compression approximation error and the spatial level-of-detail error of the SVT by formulating the approximation problem as a unified error minimization.
- A streaming approach utilizing a transmission compression based on the level-of-detail hierarchy, allowing rendering of scenes with BTF materials transmitted over a network without significant loading times.

2. Related Work

To the best of the authors knowledge, there exists no previous literature on a similar level-of-detail application on BTFs. However, there is a large body of related work in the fields of level-of-detail rendering as well as real-time rendering and streaming of BTFs.

Hierarchical level-of-detail: As early as 1976, Clark introduced the concept of hierarchical level-of-detail on geometric models [Cla76]. Here, the problem of considering only that parts of the geometry of a synthetic scene that are actually relevant for rendering the users viewport is solved by using an object hierarchy. The hierarchy holds the geometry of objects in the scene in different levels of detail. A *graphical working set* is built from the hierarchy by choosing exactly those objects that are visible on the screen in a level of detail that is sufficient for the required rendering resolution of the object. Since then, hierarchical level-of-detail has found a lot of application for scene geometries and terrain visualization and also for streaming these types of data over the Internet. More information on these research topics can be found in [LWC*02].

In real-time graphics, another level-of-detail hierarchy has also found very wide-spread application: in combination with trillinear interpolation, mip-maps of a texture, first introduced in [Wil83], are commonly applied to avoid aliasing artifacts arising from under-sampling textured area. In [TMJ98], Tanner et al. first make use of the mip-map hierarchy to allow for arbitrarily large virtual textures maintaining an active working set, similar to Clark. While Tanner et al. propose the use of a specialized graphics workstations, the concept of virtual texturing has in recent years regained popularity (e.g. [Bar08]) due to the increasing flexibility and general availability of GPUs.

BTF compression, streaming, and rendering: For the task of real-time rendering of BTF materials, a number of different solutions have been proposed. For a comprehensive overview we refer to [HF11]. At their core, almost all approaches have in common that they aim to reduce the huge amount of data in a BTF description to a more compact representation that will eventually fit on the GPU. One approach is to fit SVBRDFs to the BTF data. While this representation is well suited for evaluation on the GPU, the quality can suffer drastically by the reduction to an SVBRDF as the non-local effects of the light scattering in the material are lost. In a recent publication [WDR11], Wu et al. therefore combine a mixture of several fitted SVBRDF models with residual ABRDFs and propose to compress those via vector quantization. A second group of compression techniques is based on factorization. Here, the BTF is considered as a matrix or tensor of which a low-rank approximation is found. Recent comparisons [PSR13] indicate that on BTF data, *Full Matrix Factorization* (FMF) [KM03] often yields the best RMSE for a given compression ratio. The only mentioned exception is a BTF compression scheme based on *K-SVD* [RK09] that outperforms the FMF by a factor of 3 to 4 at comparable quality. However, an efficient real-time rendering technique for this compression has not yet been found. In [GMSK09], Guthe et al. employ a perceptually motivated BTF compression based on matrix factorization. Compression rates of about 500 : 1 are achieved with a high approximation quality. The authors observe that GPU memory can be saved by employing lower downsampled levels-of-details for some of the factorized data. In our paper, we will also save GPU memory by exploiting the fact that lower resolution versions of factorization components can be used. However, instead of reducing the level-of-detail once at compression-time, based on assumptions about viewing distance and angles, we store the factorized BTF data at multiple precomputed levels-of-detail. This allows us to dynamically decide at runtime which level-of-detail is necessary and can thus consider the actual view-point of the user.

Recently, data-driven compression methods for BTFs that are not based on factorization have been proposed as well. In [HF07], the authors follow a statistical modeling approach that achieves impressive compression ratios but in its nature is not capable of exactly reproducing the surface features of a given BTF. While this might be tolerable or even desired for the purpose of texture synthesis, it would for example not be applicable in the case of virtual surrogates for cultural heritage. In [HFM10], Havran et al. employ a compression based on multi-level vector quantization and in [TFLS11] Tsai et al. propose to use a decomposition in multivariate radial basis functions. Both methods provide high quality results for the reproduction of material appearance at real-time frame-rates. Unfortunately, no direct quality comparisons to FMF are given. However, the reported compression ratios are in the same region as achieved with FMF, so it is not to be expected that these techniques will reduce memory demand sufficiently to eliminate the memory issues of BTF rendering. In this paper, we use FMF compression, as it greatly facilitates the simplicity of the proposed progressive streaming and $L2$-norm based error-approximation for tile-prioritization. In future work one might consider the applicability of other compression methods for BSVTFs as well.

In [SRWK11], FMF compression is used for rendering BTFs in the Webbrowser via WebGL on commodity hard-

ware (NVIDIA GeForce 8800). The authors utilize the level-of-detail hierarchy implicated by the factorization to perform a progressive streaming of the BTF data over the internet. They employ an additional lossy image compression for the efficient transmission. However, the image compression does not allow fast random-access reconstruction of the compressed data any more, which is mandatory for the purpose of real-time rendering. Therefore, after transmission, the factorized data is unpacked into GPU memory again, occupying up to 2.5 GB for a single high quality BTF with 4 Megapixel. In contrast, with our proposed technique, scenes that contain several 4 Megapixel BTF materials with comparable compression ratios (660 : 1 in our case versus 428 : 1 in [SRWK11]), such as the one shown in Fig. 1 and 4, can be rendered in real-time with a much lower memory footprint (483 MB and 229 MB for the total scene).

Out-of-core rendering of reflectance data: For their editing system *BTFShop* [KBD07], Kautz et al. proposed an out-of-core rendering architecture for BTFs. For this, the uncompressed BTF data is split into tiles which are successively streamed to memory for editing and rendering. However, BTFShop is an editing application and not ment for real-time viewing purposes. The rendering relies on lazy updates and assumes that usually only a subset of pixels on the screen are changed and light and view directions remain constant. A slight rotation around the object would require to completely swap the cached tiles. This severely restricts the achievable frame-rates and prohibits streaming over a limited bandwidth network connection. In contrast, using the proposed BSVTFs allows changing the light and view directions even for scenes with many BTF materials in real-time with moderate bandwidth requirements.

On the related topic of surface light-field (SLF) rendering, Chen et al. presented the technique of *light field mapping* [CBCG02]. Here, the authors proposed to perform a spatial partition of the object's surface. In combination with factorization, vector quantization and image compression this allowed combining the SLFs for each such spatial part into textures that are suitable for rendering with the GPU. Images are then generated using multi-pass rendering, rasterizing the triangles of one spatial part at a time. While this algorithm allows for memory-friendly out-of-core rendering, it does not include level-of-detail and therefore requires the costly successive swapping of all textures for the visible parts of the object's surface in every frame.

3. Sparse Virtual Texturing

In this section, we briefly discuss the SVT algorithm [Bar08] and introduce our notation and implementation details.

We consider gray-scale images with $M \times N$ pixels depending on their context either as a matrix $\mathbf{I} = \mathbb{R}^{M \times N}$ or as a function $\mathcal{I} : (s,t) \in [0,M) \times [0,N) \subset \mathbb{R}^2 \mapsto i \in \mathbb{R}$, mapping from the continuous pixel-domain $[0,M) \times [0,N) \subset \mathbb{R}^2$ to intensity values in $\mathbb{R}$ by bi-linear interpolation of the matrix entries at the respective discrete rows and columns.

SVT considers the problem of representing a very large image $\mathbf{I}$ using a considerably smaller matrix $\mathbf{C} = \mathbb{R}^{O \times P}$, $O \ll M$ and $P \ll N$ as a cache. The technique exploits the fact that the display resolution itself is usually much smaller than the dimensions of $\mathbf{I}$. For rendering, it is therefore sufficient to hold only those parts of the image in memory, i.e. in the cache $\mathbf{C}$, that are visible on the screen at a given time. Additionally, they parts only have to be held in memory at the screen resolution, which has the additional benefit of avoiding aliasing artifacts due to under-sampling.

To this end, the original image $\mathbf{I}$ is decomposed into a set of disjoint tiles $\mathfrak{T} = \{\mathbf{T_i} \in \mathbb{R}^{T \times T} \subset \mathbf{I} | \forall_{\mathbf{i} \neq \mathbf{j}} \mathbf{T_i} \cap \mathbf{T_j} = \emptyset \wedge \mathbf{I} = \bigcup \mathbf{T_i}\}$ of size T. The tiles $\mathbf{T_i}$ are indexed by a two dimensional multi-index $\mathbf{i}$. More sets of tiles are generated for different resolutions $l = 0, \ldots, L$ by down-sampling the image $\mathbf{I}$, with L referring to the highest resolution, and are then denoted as $\mathfrak{T}_l$. In case portions of the image can sufficiently be represented in a lower resolution l, tiles from the set $\mathfrak{T}_l$ can be used. Note that tiles from this set will allow a larger coverage of the virtual image $\mathbf{I}$ at the same size T. We compute and decompose all down-sampled versions of the original image with resolutions of $\frac{M}{2^{L-l}} \times \frac{N}{2^{L-l}}$, $l = [0,1,...,L]$ until its content can eventually be expressed using the single tile in $\mathfrak{T}_0 = \{\mathbf{T}_{(0,0)}\}$, i.e. $\max(\frac{M}{2^L}, \frac{N}{2^L}) \leq T$. The content of the cache $\mathbf{C}$ is then compiled from that subset of tiles that form the visible part of the image $\mathbf{I}$ at a sufficient resolution. Hence, $\mathbf{C}$ is also referred to as the *tilecache*. If all space in the cache is already occupied on arrival of a new tile, free space will be made available by unloading existing tiles based on their priority (see Section 5.2). Tiles from multiple virtual textures are handled in a single tilecache. In our implementation, we take special care when manipulating the tilecache that at all times all parts of $\mathbf{I}$ are covered at least on a low-resolution level. This strategy prevents drawing errors due to cache misses in case of rapid user interaction.

To determine the information which tiles of which level have to be displayed, a *Feedback-image* $\mathcal{F} : (x,y) \in [0,X) \times [0,Y) \mapsto (\mathbf{i},l,\tau) \in \mathbb{R}^4$ is computed in regular intervals. Let $\Pi : \mathbb{R}^2 \to \mathbb{R}^2$ be a texture-mapping function that maps screen pixel coordinates (x,y) to texel coordinates $(s,t) \in [0,M) \times [0,N)$. For each pixel (x,y), the down-sampling level l and the index $\mathbf{i}$ of the tile $\mathbf{T_i}$ with the content for that pixel can be computed as

$$l = L - \log_2 \max\left(\left\|\frac{\partial \Pi(x,y)}{\partial x}\right\|, \left\|\frac{\partial \Pi(x,y)}{\partial y}\right\|\right) \quad (1)$$

$$\mathbf{i} = \left(\mathrm{mod}\left(\left\lfloor 2^l \frac{s}{M} \right\rfloor, 2^l\right), \mathrm{mod}\left(\left\lfloor 2^l \frac{t}{N} \right\rfloor, 2^l\right)\right)^T. \quad (2)$$

A pixel-shader is used to evaluate Equations 1 and 2. To support for multiple texture-images an additional texture-index τ is stored in the fourth channel.

In order to reassemble the original appearance of $\mathbf{I}$ from the possibly fragmented tiles that might also exhibit different resolutions, an indirection has to be performed for all texture-fetches during rendering. For each screen pixel (x,y), the texel coordinates $(s,t) = \Pi(x,y)$ are mapped to

coordinates in the tile-cache where the value for $\mathcal{I}(s,t)$ is stored. This requires to locate the appropriate tile in the tile-cache and find the correct offset within the tile itself. For this purpose, we maintain a *lookup-table* $\mathcal{L}:\mathbb{N}^2\rightarrow\mathbb{R}^3$ that holds the level l' in which a tile for (s,t) is available in the tile-cache (which might differ from the optimal level l) and the texel coordinates $\mathbf{i}'$ of its top-left corner in $\mathbf{C}$. Please note that $\mathcal{L}$ is considerably smaller than the original texture $\mathbf{I}$, as only one entry for every T^2-th texel is required. From this information, the coordinate $\mathbf{x}$ of the texel in the tilecache that needs to be fetched is computed as

$$\mathbf{x}=\mathbf{i}'+\left(2^{l'}s-M\left\lfloor 2^{l'}\frac{s}{M}\right\rfloor,2^{l'}t-N\left\lfloor 2^{l'}\frac{t}{N}\right\rfloor\right)^T. \quad (3)$$

We employ a separate lookup-table for each virtual texture.

4. BTF Real-time Rendering

We consider the discretized Bidirectional Texture Function $\rho(\mathbf{x},\omega_l,\omega_v)$ to be written as a matrix $\mathbf{B}_{i,j}$ with row indices i enumerating all combinations of view-/light-directions $\omega_\mathbf{l},\omega_\mathbf{v}$ and column indices j specifying the position $\mathbf{x}$ on the surface. For example, our highest resolution test datasets have 2048×2048 texels of spatial resolution, 151 view- and 151 light directions and three color channels. When arranging the colors as part of the ABRDFs, this results in a BTF matrix $\mathbf{B}\in\mathbb{R}^{68,403\times4,194,304}$ with about 287 billion entries. Compactly storing the matrix in half-precision floating point values, results in a datasize of 534.4 GB, which is certainly not applicable for real-time rendering.

In this paper, we build on the more compact FMF representation that can be obtained from $\mathbf{B}$ via singular value decomposition (SVD) [KM03]. Given the full SVD $\mathbf{B}=\mathbf{U}\Sigma\mathbf{V}^T$, a low-rank approximation is obtained by truncating the matrices $\mathbf{U}$ and $\mathbf{V}$ after C columns. Being a diagonal matrix, Σ can be multiplied with $\mathbf{V}$ prior to truncation, i.e. $\mathbf{V}:=\mathbf{V}\Sigma$. According to the Eckart-Young-Theorem [EY36] the SVD computes the best possible rank-C approximation of the original matrix under the L2-norm:

$$\arg\min_{\{\mathbf{U}_c,\mathbf{V}_c\}}\left\|\mathbf{B}-\sum_{c=1}^{C}\mathbf{U}_c\Sigma_{c,c}\mathbf{V}_c^T\right\|_F^2 \quad (4)$$

Here $\mathbf{U}_c$ and $\mathbf{V}_c$ denote the c-th column of the matrix $\mathbf{U}$ and $\mathbf{V}$ respectively, which in the context of BTFs are also referred to as Eigen-ABRDFs and Eigen-Textures, and $\Sigma_{c,c}$ denotes the c-th singular value. This way, only the more compact truncated matrices $\mathbf{U}'$ and $\mathbf{V}'$ have to be stored and an approximation of the BTF value can be obtained as $\mathbf{B}\approx\mathbf{B}'=\mathbf{U}'\mathbf{V}'^T$.

In the context of real-time rendering, the factorized representation has the important benefit of allowing random access to arbitrary values of the BTF without the necessity to reconstruct the full matrix $\mathbf{B}'$. Consider the c-th column of the matrices $\mathbf{U}'$ and $\mathbf{V}'$ as images $\mathcal{U}_c$ and $\mathcal{V}_c$. Then the BTF ρ can be approximated as

$$\rho'(\mathbf{x},\omega_l,\omega_v)=\sum_{c=1}^{C}\mathcal{U}_c(\omega_l,\omega_v)\cdot\mathcal{V}_c(\mathbf{x}). \quad (5)$$

If $\mathcal{U}$ and $\mathcal{V}$ are stored as textures on the GPU, Equation 5 can be efficiently evaluated in a shader-program. For directions and positions other than the discrete samples stored in $\mathbf{B}$, the values have to be interpolated. Instead of having to perform a costly $6D$ interpolation, in the factorized case a $2D$ interpolation for $\mathbf{x}$ in the spatial domain can be performed independently from a $4D$ interpolation in the angular domain.

We rely on the texture-units of the GPU to perform the spatial $2D$ interpolation for us when accessing the textures $\mathcal{V}$, by choosing a suitable layout of the Eigen-Textures. The four dimensional bidirectional interpolation in $\mathcal{U}$, however, has to be performed explicitly in the shader. For this, we follow an idea presented in [ND06] and pre-compute two separate two-dimensional Delaunay triangulations D_l and D_v for the sets of light and view direction samples of the BTF given in parabolic coordinates. We then raster each triangulation D into two RGB textures $\mathcal{D}$ and $\mathcal{B}$, containing the three direction indices of the enclosing Delaunay triangle and the three barycentric weights respectively. This way, during rendering the interpolated value for arbitrary view and light directions given in parabolic coordinates, can be evaluated in a GPU shader: For all 9 combinations of direction indices from $\mathcal{D}_l(\omega_\mathbf{l})$ and $\mathcal{D}_v(\omega_v)$, we perform a lookup into $\mathcal{U}$ and blend the values according to the barycentric weights. The small GPU memory overhead re-introduced by the index and weight textures can further be reduced in the case of rendering multiple BTF with the same angular sampling by sharing the textures between them.

5. Extension of SVT to BSVTFs

While for curved surfaces and perspective cameras almost all entries of the bi-directional reflectance properties in $\mathbf{U}'$ have to be accessed, the utilization of parts of the Eigen-Textures stored in $\mathbf{V}'$ follow the same consideration as conventional textures. Therefore, the idea of sparse virtual texturing could be directly applied in this case. The Eigen-Textures could be treated as an image with C channels and a spatial level-of-detail hierarchy can be constructed and decomposed into tiles.

However, this would not provide the best approximation as it does not take advantage of the property that the SVD compacts most of the information in the first few columns of $\mathbf{U}'$ and $\mathbf{V}'$, so that the contribution of later columns to the quality of the approximation decreases quickly. This observation has already found application in [SRWK11], where the columns of $\mathbf{U}'$ and $\mathbf{V}'$ were transmitted sequentially. In our case, the situation is far more general. Using SVT introduces a new degree of freedom, since every column of $\mathbf{V}'$ could be stored in a different spatial resolution.

We aim to combine both the spatial resolution and the approximation rank level-of-detail in a consistent manner. Instead of considering the matrix $\mathbf{V}'$ as one texture with multiple channels, we regard every column as an individual virtual $2D$ texture $\mathcal{V}'_c$. This way, the tiles of different columns are weighted against each other for utilization of the tilecache.

In principle, the goal of hierarchical level-of-detail rendering can be defined as the minimization of the rendering

error that can result from the restriction to a fixed tilecache size. In the case of BTFs, possible sources of error are an insufficient spatial resolution or insufficient number of factorization components for the low-rank approximation. Let image $\mathcal{S}$ denote the content of the screen when directly using the factorized BTF $\mathbf{B}'$ for rendering and $\tilde{\mathcal{S}}$ the content using SVT to access $\mathbf{V}'$. The rendering error under the L2 norm can be expressed as

$$\sum_{y=0}^{Y-1}\sum_{x=0}^{X-1}\left(\mathcal{S}(x,y)-\tilde{\mathcal{S}}(x,y)\right)^2, \tag{6}$$

i.e. the sum of squared differences over all screen pixels.

In our implementation, we do not directly minimize this term but instead propose a simplification. Let Π be the texture-mapping function used during rendering that maps from screen pixels (x,y) to the spatial position $\mathbf{x}$ in the BTF. Let furthermore $A_{\mathbf{x}}(\omega_i,\omega_o)=\rho'(\mathbf{x},\omega_i,\omega_o)$ denote the ABRDF encoded at that position in the factorized BTF. Then

$$\begin{aligned}&\sum_{y=0}^{Y-1}\sum_{x=0}^{X-1}\left\|A_{\Pi(x,y)}-\tilde{A}_{\Pi(x,y)}\right\|^2\\ =&\sum_{y=0}^{Y-1}\sum_{x=0}^{X-1}\left\|\sum_{c=0}^{C}\mathbf{U}'_c\mathcal{V}'_c(\Pi(x,y))-\sum_{c=0}^{C}\mathbf{U}'_c\tilde{\mathcal{V}}'_c(\Pi(x,y))\right\|^2\end{aligned}$$

denotes the L2 error of the ABRDF for every pixel reconstructed directly from the factorization (designated A) and reconstructed using SVT (designated $\tilde{A}$). Utilizing the SVD property of matrix $\mathbf{U}'$ being unitary, it is sufficient to consider the error for every individual virtual $2D$ texture $\mathcal{V}'_c$.

$$E=\left\|\left(\mathcal{V}'_c(\Pi(x,y))-\tilde{\mathcal{V}}'_c(\Pi(x,y))\right)\right\|^2. \tag{7}$$

Our proposed algorithm minimizes this error under the constraint of limited memory.

Please note, that even though the different columns in $\mathbf{U}'$ and $\mathbf{V}'$ have different importance for the quality of the BTF approximation, using the proposed minimization formulation we elegantly avoid the introduction of additional weighting-terms to balance the individual textures against each other. Furthermore, the proposed simplification of the rendering error minimization to an ABRDF error minimization has the additional advantage that the lighting in the virtual scene can be changed without the necessity to change anything in the tilecache utilization. Changes in view direction benefit from the availability of ABRDFs as well, as not all tiles in the tilecache have to be exchanged but only those which are affected by changes in visiblity or mip-level.

Without loss of generality, in the remainder of this paper we will assume that the individual Eigen-textures $\mathcal{V}'_c$ are laid out side-by-side in a large enough virtual texture $\mathcal{I}$ that will be used for SVT.

5.1. Level-of-detail strategy

While in the case of level-of-detail on geometry a variety of strategies for the artifact free refinement without inconsistencies exists, for SVT not too many details can be found in the literature. In this work, we essentially distinguish between two operations: `add` and `swap`.

The operation `add` will insert a tile $\mathbf{T}$ at free space in the tilecache. As a post-condition, we check whether any ancestor tile of $\mathbf{T}$ in the tile-hierarchy is now completely covered by its children. If so, the ancestor is removed from the tilecache, as it will not contribute to the pixels drawn on screen any more. `add` operations are only performed on tiles that have an ancestor in the tilecache. After the operation one or none (if an ancestor has been removed) of the free entries in the tilecache will be occupied.

The operation `swap` will remove two tiles $\mathbf{T}_{\mathbf{i}_1,l_1},\mathbf{T}_{\mathbf{i}_2,l_2}$ from the tilecache and instead insert a tile $T_{\mathbf{i}',l'}$ from a lower level $l'<\min(l_1,l_2)$ in the tile-hierarchy that covers those parts of $\mathbf{I}$ that were shown in $\mathbf{T}_{\mathbf{i}_1,l_1}$ and $\mathbf{T}_{\mathbf{i}_2,l_2}$, that is $\mathbf{i}'=\left\lfloor 2^{l'-l_1}\mathbf{i}_1\right\rfloor=\left\lfloor 2^{l'-l_2}\mathbf{i}_2\right\rfloor$. This operations will result in one free entry in the tilecache.

After all operations have been performed, the entries in the lookup-table are updated accordingly.

5.2. Tile prioritization

In order to minimize the ABRDF error from Equation 7, we weight the possible tiles that can be loaded into the tilecache against each other. For this, we roughly follow two measures:

1. The number of the pixels on the screen covered by the tile
2. The average reduction of the approximation error E for a pixel covered by the tile

As long as there is still free space in the tilecache we perform the `add` operations on tiles prioritized by these two criteria. For this we set the priority P of a tile $\mathbf{T}_{\mathbf{i}}$ at level l as $P=w(\mathbf{i},l,l-1)\cdot v(\mathbf{i},l)$, where

$$w(\mathbf{i},l,l')=\frac{1}{T^2}\sum_{x=0}^{T-1}\sum_{y=0}^{T-1}\left(\mathcal{T}_{\mathbf{i},l}(x,y)-\mathcal{T}_{2^{l-l'}\mathbf{i},l'}\left(x',y'\right)\right)^2 \tag{8}$$

denotes the maximum L2 difference of the tile to its lower resolution ancestor at level l' in the tile-hierarchy and

$$v(\mathbf{i},l)=\left|\left\{(\mathbf{i}',l')\in\mathcal{F}|l'\geq l\wedge\mathbf{i}=\left\lfloor 2^{l-l'}\mathbf{i}'\right\rfloor\right\}\right| \tag{9}$$

denotes the number of votes, i.e. pixels in the feedback image (see Section 3) that show the index values $\mathbf{i}$ and l of the tile or its descendants in the tile-hierarchy. (x',y') in Equation 8 denote the point in the lower resolution tile $\mathcal{T}_{2^{l-l'}\mathbf{i},l'}$ that maps to the same point in the virtual texture as (x,y) does in $\mathcal{T}_{\mathbf{i},l}$. The value P approximates the reduction in the error E in Equation 7 if $\mathbf{T}_{\mathbf{i},l}$ would be in the tilecache.

This definition for P is only valid for `add` operations on tiles of level l for which the parent at level $l-1$ is currently visible. Otherwise computing the votes v would be more complex, as several in-between steps would have to be considered. Since, `add` operations for a tile with a directly available parent are favorable for the application of streaming in Section 6, we restrict ourselves to the simple case.

BSVTF (244 MB) $C = 100$, 4032^2 pixel tilecache size | FMF BTF (316 MB) $C = 100$ | uncompressed (167 GB)

Figure 3: *BTF renderings using BSVTFs (left), FMF compression (center) and no compression (right). Upper and lower half of the images are lit from different light directions. All of the materials (front to back: leather, gravel, sponge, wood, velvet) exhibit complex view and light dependent material appearance. Whereas the uncompressed materials appear to be visibly sharper, there is hardly any noticeable difference between our technique and the FMF.*

While w is pre-computed for every tile of every Eigen-Texture, the pixel votes v, obtained at runtime from the feedback buffer, are simply repeated in the spatial layout of the components. This particular choice of v will also make sure that no space is wasted on tiles with unnecessarily high resolutions, i.e. levels that are higher than the ones in the feedback buffer, since those will have a priority of $P = 0$.

In case there is no free space left in the tilecache but further tiles could be added, we have to decide whether a `swap` operation should be performed to free space or not. Naturally, the `swap` operation will increase the error E, as it replaces higher resolution tiles $\mathbf{T}_{\mathbf{i},l}$ with a lower resolution substitute $\mathbf{T}_{\mathbf{i}',l'}$. We can approximate the rise in error by

$$\begin{aligned} c(l',\mathbf{i},l) &= \sum_{k=l'}^{l} w\left(\left\lfloor 2^{k-l}\mathbf{i}\right\rfloor, k, k-1\right) v(\mathbf{i},l) \\ &\approx w(\mathbf{i},l,l')v(\mathbf{i},l) \end{aligned}, \quad (10)$$

which is the accumulated approximated error of the portion of all in-between tiles with levels $k = l', \ldots, l$ that are currently covered by pixels from the high resolution tile and would therefore be revealed in case of a `swap`. Even though this particular approximation is not very accurate, it has the benefit that only one weight value $w(\mathbf{i},l,l-1)$, which is the same as the weight that we employ for computing P, has to be computed and stored per tile. Since we will replace exactly two tiles $\mathbf{T}_{\mathbf{i}_1,l_1}$ and $\mathbf{T}_{\mathbf{i}_2,l_2}$, the total increase in error or *cost* of this operation can be expressed by $c = c(l',\mathbf{i}_1,l_1) + c(l',\mathbf{i}_2,l_2)$.

In order to decide whether to perform a `swap` operation, we first find the three swap-candidates with the lowest cost $c^\star$ and compare this value with the highest priority $P^\star$ of the tiles that could be added. If $c^\star < P^\star$, this means that the approximated error of not having the tile with priority $P^\star$ in the tilecache is higher than the error induced by performing the `swap` operation with cost $c^\star$. Hence, we will reduce the total error E by first performing the least costly `swap` operation to obtain free space and then performing the highest priority `add` operation. Otherwise, we already found the best solution and will not perform any operation.

6. Streaming

Similar to other hierarchical level-of-detail techniques, the proposed BSVTFs are very well suited for streaming over a network. Tiles that have to be inserted into the tilecache by the `swap` or `add` operation are in this case requested from a streaming server.

To facilitate the transmission of tiles over a low-bandwidth network, we employ an additional compression to the tiles prior to submission that is inverted before the tile is inserted into the tilecache. As observed in [SRWK11], the Eigen-Textures obtained by the SVD show similar image statistics as natural images. Therefore, in principle every image-compression technique could be employed for this purpose. For example, in [KM03] Koudelka et al. utilize JPEG compression while in [SRWK11] Schwartz et al. employed a wavelet codec similar to JPEG2000.

For the compression we perform a discrete cosine transformation (DCT) on 8×8 pixel blocks of the tile-images and then apply a quantization with respect to a quality threshold similar to JPEG. The quantized data is then stored using deflate. The only mentionable difference to other off-the-shelve implementations is the fact that our compression operates on floating point values (half-precision).

To further improve the compression ratio and exploit the large redundancy present in the sets of tiles for different resolutions, instead of directly compressing the tiles, we compress the differences $\mathcal{T}'$ of a tile to its up-sampled parent in the tile-hierarchy $\mathcal{T}'_{\mathbf{i},l}(x,y) = \mathcal{T}_{\mathbf{i},l}(x,y) - \mathcal{T}_{\lfloor\frac{1}{2}\mathbf{i}\rfloor,l-1}(\frac{x}{2},\frac{y}{2})$. This procedure exploits the fact that due to our construction of the tile-hierarchy, most of the low-frequency components of the DCT are already covered by the parent tile. Thus, the amount of information that needs to be compressed is drastically reduced by using the difference image. The compressed size of a compressed tile depends on the choice of T and the user-determined quality threshold for the quantization. In our experiments we were able to obtain a compression ratio up to 6 : 1 with no perceivable artifacts.

In order to unpack the DCT compressed difference images after transmission, the respective parent tile is required. During an `add` operation this does not pose a problem, since we

decided in Section 5.2 that this operation should only be performed if a parent of the tile is still in the tilecache and hence available at the client side. In case of the `swap` operation, in the worst case all ancestor tiles will have to be requested as well in order to sequentially unpack all of them until the parent is available. However, in order for a `swap` operation to be possible, higher resolution tiles had to be added to the tilecache first. In turn this means that the full branch of the level-of-detail hierarchy up to this resolution and thus also all ancestors of the tile that has to be swapped in, had to be previously transmitted to the client. We therefore employ a *least recently used* caching strategy to keep as many tiles that have been received as possible in the client-side RAM.

Even before applying the transmission compression the size of the tiles is only in the order of a few Kilobytes. The Eigen-ABRDFs in $\mathbf{U}'$ on the other hand have a size of a few Megabytes per color channel (4.4 MB for the UBO2011 objects). Fortunately, in contrast to the tiles that have to be swapped in and out on demand, $\mathbf{U}'$ only has to be transmitted once and does not change during the rendering process. Still, loading this amount of data for multiple objects in advance over a low-bandwidth connection is not a good solution.

We therefore transmit the columns $\mathbf{U}'_c$ sequentially and interleave them in the tile-datastream. This way, only a few hundred Kilobytes have to be transmitted at once, allowing to start rendering considerably faster. In this case, the vectors $\mathbf{U}'_c$ have to be prioritized in a similar fashion as the tiles to decide whether to stream the next tiles or another column of $\mathbf{U}'$. From Equation 5, it becomes apparent that $\mathbf{U}'_c$ can only contribute to the BTF approximation if $\mathbf{V}'_c$ is available as well. We can therefore approximate the priority of the Eigen-ABRDFs by the sum of votes for all tiles $\mathbf{T}_{\mathbf{i},l}$ in the tilecache that are currently used to represent $\tilde{\mathcal{V}}'_c$, weighted by the average intensity of the tile, i.e. $\sum v(\mathbf{i},l)\|\mathbf{T}_{\mathbf{i},l}\|_F^2$. This weighting can be understood as the contribution the tile makes to not having a value for $\mathbf{V}'_c$ available at all, which would in turn render the request for $\mathbf{U}'_c$ pointless.

7. Evaluation

To assess the feasibility of our approach, we tested the level-of-detail rendering and streaming on several high-resolution BTFs from the OBJECTS2011 and OBJECTS2012 databases of the University of Bonn [SWRK11] as well as a collection of 100 measured BTFs obtained from material samples. Please see the additional multimedia material to get an impression of the complex material appearance effects captured in the BTF datasets.

All of our BTF materials have a high dynamic range and are represented in RGB color. In all cases, the angular sampling contained the same set of 151×151 directions ω_v, ω_v. Before uploading $\mathbf{U}$ to the GPU we furthermore compute an additional 152-th basis illumination in which we stored a pre-integrated value of all other lights for the efficient evaluation of a view-dependent ambient term in the fragment shader. The spatial resolution of the datasets varies from 512×512 to 2048×2048 texels (see Table 1 for details).

resolution	uncomp.	FMF	BSVTF	pre-proc.
Buddha, Donkey, Minotaur, Terracotta Soldier				
2048^2 px	534 GB	813 MB	460 MB	20 min
Almond Horn, Apple, Moulage 2, Pudding Pastry Strawberry				
1600^2 px	326 GB	501 MB	262 MB	12.5 min
Chess Piece, Ganesh, Moulage 1, Shoe				
1024^2 px	133 GB	213 MB	168 MB	5 min
Santa, 100 Materials				
512^2 px	33 GB	63 MB	53 MB	1 min

Table 1: *The employed datasets (Fig. 1 and 2).* Uncompressed, FMF *and* BSVTF *gives the filesizes for the different levels of compression. Here,* BSVTF *refers to the streaming ready file including headers, pre-computed weights, level-of-detail hierarchy and DCT compression. The* pre-processing *time refers to computing the BSVTF from the FMF BTF.*

As factorization compression we compute the SVD on the full BTF matrix $\mathbf{B}$ to obtain a rank $C = 100$ approximation according to Equation 4, which we will denote *FMF BTF*.

From the FMF BTFs, we generate the BSVTF by first creating a layout of the Eigen-Textures. To save texture-lookups in the shader, we store four values $\mathcal{V}'(\mathbf{x})$ per pixel as RGBA channels. Then we compute the sets of tiles for different resolutions $\mathfrak{T}_l$. In our experiments, we use a tile-size $T = 64$. We additionally extend the tile with 4 pixels of padding at each border to allow for trilinear filtering using the tilecache texture. This results in 40.5 KB per tile when employing 16 bit floating point numbers. Using the DCT compression this size is reduced to about 7 KB to 10 KB. We also pre-compute the weights between direct descendents $w(\mathbf{i},l,l-1)$ from Equation 8 and store them as 16 bit float as well. Finally, the Eigen-ABRDFs with 151×151 angular directions are stored for the three color channels. Here we employ the strategy of packing four components into the RGBA channels as well, resulting in packets of 534 KB for interleaving with the tile transmission. Details on the processing times and the resulting total file-sizes can be found in Table 1. The costs for generating the level-of-detail representation from factorized BTFs is negligible compared to the time requirements of the factorization. While computing the FMF compression for a 512×512 texel BTF took 20 minutes using a highly optimized GPU implementation, generating the DCT compressed tiles with a single-threaded CPU implementation took only one additional minute on a 2.4 GHz Intel Xeon.

We compiled six test-scenes from the available datasets:

1. all 100 materials, arranged on a grid of tori (Fig. 2),
2. all available objects from OBJECTS2011 and OBJECTS2012 on a BTF textured plane (Fig. 1),
3. the four 4 Megapixel objects (Fig. 4 and 5),
4. five selected materials, presented on cylinders (Fig. 3),
5. only the Buddha object,
6. only the Terracotta Soldier object.

The performance of the BSVTFs was measured using fly-through animations. Please see the videos in the additional multimedia material for an impression of the animations.

BSVTF (77 MB) $C = 100$, 1008^2 tilecache | BSVTF (229 MB) $C = 100$, 4032^2 tilecache | FMF (227 MB) $C = 7$ | FMF (3.17 GB) $C = 100$ | uncompressed (2.1 TB)

Figure 4: *Quality comparison on a scene with four* 4 Megapixel *BTFs that would exceed the memory of most GPUs. With a too small tilecache size (first image), our technique is not able to resolve the fine mesoscopic details. Using an appropriate tilecache size (second image), the BSVTFs still have a small GPU memory footprint and at the same time achieve a comparable quality to directly rendering the FMF BTF data (fourth image). The third image demonstrates the loss in quality when using FMF with a higher compression ratio to achieve the same small memory footprint. Due to the insufficient number of $C = 7$ columns, this rendering shows blurred meso-structure details, washed-out highlights and false-colors.*

Rendering with a screen resolution of 1280×720 and using a tilecache the size of 4032×4032 pixels, we achieve a comparable quality to FMF BTFs at real-time frame-rates. All tests were conducted on an Intel Core i7 950 with an NVIDIA GeForce GTX 680 GPU with 4 GB of GPU memory. Details on rendering performance and GPU memory consumption can be found in Table 2.

In all of our experiments with BSVTFs, the average total CPU utilization of the system was at 23% with 51% of the time spent on rendering and GUI, 46% on evaluating the feedback-image and deriving the list of operations, 2.4% in image decompression and the remaining 0.6% in network- or disk-IO. As expected, rendering with FMF BTFs resulted 12% CPU load, since here no other tasks than rendering and GUI had to be performed. The recorded frame-rates suggest that rendering with BSVTFs is about 28% less efficient than using FMF BTFs while requiring 23%-93% less GPU memory. In both cases the performance is mainly correlated with the triangle count of the scenes, not the number of BTFs.

Fig. 3 and 4, which depict scene 4 and 3 respectively, offer a qualitative comparison of BSVTFs with FMF BTFs and uncompressed BTFs. Whereas the uncompressed BTFs appear to be visibly sharper, there is hardly any noticeable difference between BSVTFs and FMF BTFs. Note that rendering the uncompressed BTFs has been performed using deferred shading from out-of-core data and is prohibitively costly. The hard-disk is a severe bottle-neck, resulting in several hours for one image with solely local illumination.

While we employed a tilecache with 4032×4032 pixels for our evaluation of the performance, the GPU memory footprint could be reduced even further by choosing a smaller cache size. Fig. 4 demonstrates the influence of a reduced tilecache size. Although the most obvious difference can be observed in the spatial resolution of surface details, the quality of the reflectance also suffers for too small tilecache sizes. For example, the copper parts of the Minotaur object show a shift in color and appear more dull. For more tilecache size comparison images, please refer to the additional multimedia material.

	#		FPS		GPU Memory	
	Δ	BTFs	BSVTF	FMF	BSVTF	FMF
1	180K	100	34 ±10	-	1.7 GB	6.2 GB
2	3.7M	15	10 ±2	-	483 MB	6.6 GB
3	2.3M	4	18 ±5	-	229 MB	3.2 GB
4	4.6K	5	72 ±16	106 ±23	244 MB	316 MB
5	50K	1	46 ±9	69 ±14	181 MB	0.8 GB
6	1.1M	1	24 ±3	29 ±6	181 MB	0.8 GB

Table 2: *Results of the performance evaluation on the test-scenes described in Section 7. The columns* #Δ *and* #BTF *denote the number of triangles and BTF materials in the scene.* FPS *are the average and standard deviation on the tested animation sequence. FPS for FMF BTFs are only available for scenes that fit into the GPU memory of our test system.*

In Fig. 5 we demonstrate the streaming over the network. After a transmission of 25 MB the scene already achieves a perceptual similarity, measured by the structural similarity index (SSIM) [WBS*04], of 95.4% to the converged BSVTF (i.e. no `add` or `swap` operation would further reduce the error). After transmitting about 100 MB the images become virtually indistinguishable.

Limitations: Although our evaluation shows that the proposed BSVTF is applicable in a number of scenarios and performs very well, the method also has a few limitations that need to be considered as well. First, the additional buffer updates, the regular texture-data uploads, and the additional texture fetches due to the indirection in the fragment shader show an unavoidable and significant impact on the frame-rate. Second, our current approach only uses a level-of-detail hierarchy on the Eigen-Textures. While this is very feasible for few but high resolution BTFs (e.g. scenes 2 and 3), it is less efficient in scenes with many but comparably low spatial resolution materials (e.g. scene 1). It will not help at all if instead of spatial resolution a high angular resolution of the BTF data would become the bottleneck. Finally, unless the

Figure 5: *Rendering quality after streaming different amounts of data over the network. The SSIM values predict the perceptual similarity between the images and are computed with respect to the converged version.*

movement of the user is somehow anticipated, a pre-fetching of data is hard to implement and resolution popping artifacts can not completely be eliminated, especially when streaming from a network connection with high latency.

8. Conclusion and Future Work

In this paper, we demonstrated that by adapting sparse virtual textures to factorized BTFs it becomes possible to render scenes with a large number of high resolution BTFs efficiently on the GPU. For this, we suggested a strategy to trade of spatial resolution and the accuracy of the reflectance representation. Furthermore, we demonstrated that this technique can be combined with an additional image compression codec and used for network transmission.

An important consideration of the proposed BSVTFs with regard to GPU memory is that only the spatial domain is covered by the level-of-detail hierarchy. We envision to overcome this limitation by extending the level-of-detail approach to the Eigen-ABRDFs as well, keeping only those that are most important to the current view-point in GPU memory. For this, a hierarchical factorization could be used, in which first the whole BTF is represented by a small number of columns C and then the residuum is subdivided into smaller subsets which are factorized individually.

Another direction of future research will be improving the network streaming by integrating a progressive refinement of the tiles, similar to [SRWK11]. This would allow smaller tile-sizes and thus faster responses to changes in view-point over low-bandwidth networks.

Acknowledgements: We would like to thank Nils Jenniche for his groundwork on the combination of SVT with BTFs. The research leading to these results was funded by the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 323567 (Harvest4D); 2013-2016.

References

[Bar08] BARRETT S.: Sparse virtual texture memory. In *Game Developer Conference* (2008). 2, 3, 4

[CBCG02] CHEN W.-C., BOUGUET J.-Y., CHU M. H., GRZESZCZUK R.: Light field mapping: efficient representation and hardware rendering of surface light fields. In *SIGGRAPH* (2002), pp. 447–456. 4

[Cla76] CLARK J. H.: Hierarchical geometric models for visible surface algorithms. *Commun. ACM 19*, 10 (1976), 547–554. 3

[DvGNK97] DANA K. J., VAN GINNEKEN B., NAYAR S. K., KOENDERINK J. J.: Reflectance and texture of real-world surfaces. In *IEEE Conference on Computer Vision and Pattern Recognition* (1997), pp. 151–157. 1

[EY36] ECKART C., YOUNG G.: The approximation of one matrix by another of lower rank. *Psychometrika 1* (1936), 211–218. 5

[GMSK09] GUTHE M., MÜLLER G., SCHNEIDER M., KLEIN R.: Btf-cielab: A perceptual difference measure for quality assessment and compression of btfs. *Computer Graphics Forum 28*, 1 (2009), 101–113. 3

[HF07] HAINDL M., FILIP J.: Extreme compression and modeling of bidirectional texture function. *PAMI 29*, 10 (2007), 1859–1865. 3

[HF11] HAINDL M., FILIP J.: Advanced textural representation of materials appearance. In *SIGGRAPH Asia Courses* (2011). 3

[HFM10] HAVRAN V., FILIP J., MYSZKOWSKI K.: Bidirectional texture function compression based on multi-level vector quantization. *CGF 29*, 1 (2010), 175–190. 3

[KBD07] KAUTZ J., BOULOS S., DURAND F.: Interactive editing and modeling of bidirectional texture functions. In *SIGGRAPH* (2007). 4

[KM03] KOUDELKA M. L., MAGDA S.: Acquisition, compression, and synthesis of bidirectional texture functions. In *Texture 2003 Workshop* (2003), pp. 59–64. 3, 5, 7

[LWC*02] LUEBKE D., WATSON B., COHEN J. D., REDDY M., VARSHNEY A.: *Level of Detail for 3D Graphics*. Elsevier Science Inc., New York, NY, USA, 2002. 3

[ND06] NGAN A., DURAND F.: Statistical acquisition of texture appearance. In *EGSR* (2006), pp. 31–40. 5

[PSR13] PAJAROLA R., SUTER S. K., RUITERS R.: Tensor approximation in visualization and computer graphics. In *EG Tutorials* (2013). 3

[RK09] RUITERS R., KLEIN R.: Btf compression via sparse tensor decomposition. *EGSR* (2009), 1181–1188. 3

[SRWK11] SCHWARTZ C., RUITERS R., WEINMANN M., KLEIN R.: Webgl-based streaming and presentation framework for bidirectional texture functions. In *VAST* (2011), pp. 113–120. 2, 3, 4, 5, 7, 10

[SWRK11] SCHWARTZ C., WEINMANN M., RUITERS R., KLEIN R.: Integrated high-quality acquisition of geometry and appearance for cultural heritage. In *VAST* (2011), pp. 25–32. 8

[TFLS11] TSAI Y.-T., FANG K.-L., LIN W.-C., SHIH Z.-C.: Modeling bidirectional texture functions with multivariate spherical radial basis functions. *PAMI 33*, 7 (2011), 1356–1369. 3

[TMJ98] TANNER C. C., MIGDAL C. J., JONES M. T.: The clipmap: a virtual mipmap. In *SIGGRAPH* (1998), pp. 151–158. 2, 3

[WBS*04] WANG Z., BOVIK A. C., SHEIKH H. R., MEMBER S., SIMONCELLI E. P., MEMBER S.: Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing 13* (2004), 600–612. 9

[WDR11] WU H., DORSEY J., RUSHMEIER H.: A sparse parametric mixture model for btf compression, editing and rendering. *Computer Graphics Forum 30*, 2 (2011), 465–473. 3

[Wil83] WILLIAMS L.: Pyramidal parametrics. In *SIGGRAPH* (1983), pp. 1–11. 3