First publ. in: Proceedings - IEEE International Conference on Shape Modeling and Applications 2006, SMI 2006 : 14 - 16 June 2006, Matsushima, Japan / sponsored by IEEE Computer Society Visualization and Graphics Technical Committee. - Los Alamitos, Calif. [u.a.] : IEEE Computer Society, 2006. - S. 45-56. - ISBN 978-0-7695-2591-4

Hardware Rendering of 3D Geometry with Elevation Maps

Tilo Ochotta Stefan Hiller

Department of Computer and Information Science,
University of Konstanz, Germany

Abstract

We present a generic framework for realtime rendering of 3D surfaces. We use the common elevation map primitive, by which a given surface is decomposed into a set of patches. Each patch is parameterized as an elevation map over a planar domain and resampled on a regular grid. While current hardware accelerated rendering approaches require conversion of this representation back into a triangle mesh or point set, we propose to render the elevation maps directly in a hardware accelerated environment. We use one base data set to render each patch in the common vertex and fragment shader pipeline. We implement mesh- or point-based rendering by using a base mesh or a base point set respectively. This provides the basis for the underlying primitive for the final rendering. We show the benefits of this method for splat rendering by replacing attribute blending through a simplified and fast attribute interpolation. This results in rendering acceleration as well as an improvement in visual quality when compared to previous approaches.

1 Introduction

Today's 3D acquisition devices, in particular laser scanning systems, are capable of producing complex high quality representations of real world objects in a relatively short time [18]. The resulting rapid increase in accessible 3D content requires displaying the data that consist of a large number of primitives at interactive rates. The most common approach, which is efficiently implemented in graphics hardware and therefore used in many applications, is the rendering with polygons, e.g., triangle meshes. However, mesh rendering suffers from fixed topological relationships between vertices. The construction of this connectivity and its efficient representation is a non-trivial problem, such as triangulation [10] and stripification [31, 21] of 3D data sets.

Point-based rendering was invented by Levoy and Witted [19] and extended by Grossman and Dally [12]. The point primitive does not rely on connectivity information and is therefore suitable for applications, such as dynamic shape modelling [25]. Also with regard to efficient rendering, points have shown to be a suitable primitive in hardware accelerated environments [27, 5]. However, current splatting implementations on graphics hardware require several rendering passes for blending attributes of overlapping splats. The blending itself tends to produce blurring artifacts that are visible at a closeup range.

We consider hardware-based rendering of 3D geometry using the height-field representation [23]. As denoted in figure 1, a given surface is decomposed into a number of patches, each of which is resampled as an elevation map over a compactly supported planar domain, and hence it can be held as a 2D texture. In [23] it has been shown that this representation can be used for high-performance com-

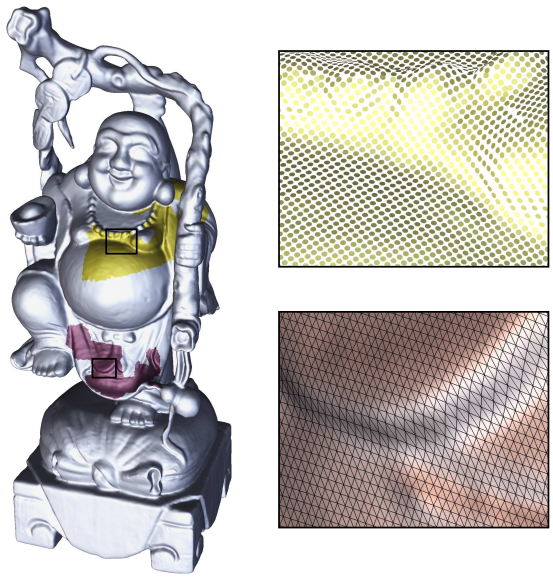


Figure 1. Patches on the Shakyamuni statue with 1.6 million points (left) can be rendered with splats (top) and triangles (bottom). Our high-quality splat renderer achieves 12 fps.

pression of point-sampled geometry. Thus, it enables us to archive a large number of 3D models at low memory usage.

In this paper we focus on benefits of the elevation map representation for hardware-accelerated rendering. We consider the scenario that 3D models are available in compressed form, e.g., in a digital library by using the approach in [23]. In their model representation, each patch consists of an elevation map and parameterization parameters. The parameters consist of a reference point in 3D space and a normal vector that define a base plane, yielding a generic surface parameterization.

To render this data, the straight forward approach is to decode the model into main memory and to render the data as a simple list of primitives with common methods [5]. However, drawbacks of this method are that neighbor information is lost. Moreover, the decoded model has a higher memory usage than is required for our elevation maps, since point positions are in the (x, y, z) representation.

We propose to render the elevation maps directly. More specifically, the graphics card holds a base data set, e.g., a regularly structured triangle mesh or point set. In order to render a patch, its elevation map is applied to the base data set, which is then transformed according to plane parameters. Our approach has the following advantages:

- **Efficient geometry representation**: Images resulting from resampling can efficiently be held in graphics hardware, e.g., using textures. The rendering is implemented by using a base data set that is similar for all patches in the model. This results in less memory usage than is required for holding the model in its original form.

- **Rendering of triangles and splats**: Rendering models with our representation allows us to easily switch between triangle- and splat-based rendering. From the regular structure of the elevation maps we directly derive a triangulation as well as a stripified representation of a patch in linear time. We also realize splat rendering by displaying the base data set as points with associated normals and radii.

- **Efficient attribute interpolation**: We propose an interpolation scheme that removes blurring artifacts of state-of-the-art splat renderers that results in an improvement of the overall visual quality. We achieve this by exploiting neighbor information in the regularly resampled model. This interpolation is efficiently implemented in the fragment shader using fast texture lookups. This procedure allow us to replace time-consuming multi-pass rendering by fast one-pass rendering.

The remaining part of this paper is organized as follows. In the next section we review related work in the field of patch-based model representation and hardware-accelerated rendering of complex 3D geometry, focusing on splatting techniques for point-rendering. In section 3 we introduce our rendering pipeline including construction of elevation map data for given point- and mesh-based 3D models. After discussing experimental results in section 4, we conclude our work and outline future work.

2 Related Work

2.1 Hardware Rendering

Since rendering of complex 3D meshes is already efficiently implemented on today's GPUs, we focus on recent advances in splat-based hardware rendering of point-sampled geometry.

Point-rendering was introduced by Levoy and Whitted [19] and improved by Grossman and Dally [12]. Pfister et al. [26, 32] propose to use points with normals and radii that define discs (surfels) that locally approximate a given surface. Alexa et al. [3] render a model by constructing a moving least squares surface for a given set of points. The surface is represented by a set of projections onto the surface, and hence, an arbitrary number of points can be used to render the model. This approach has been extended for ray tracing in [1].

QSplat was invented [28] to cope with dense data sets, e.g., with several million points. Here the point set is hierarchically partitioned into a set of bounding spheres that are used for dynamic level of detail rendering. The problem of rendering such large models is also discussed in [8] where an octree is constructed for a given densely sampled data set. This hierarchical data structure is used for efficient storage and fast rendering. However, current graphics hardware is not designed to implement these advanced rendering pipelines.

Krüger et al. [16] propose to represent and render densely sampled point models through a set of runs, in which positions of neighboring points are encoded using a chain coding like method. They use space filling twelve sided polyhedrons which results in advantageous relationships between neighboring cells.

Hardware acceleration capabilities for rendering of points with current graphics cards has been studied in [27]. During the past three years, an improvement to methodical and performance aspects was introduced by Botsch et al. [6, 7, 33, 5]. A main contribution of their work was to use hardware accelerated and screen aligned point sprites. The point sprites are projected onto the surface and cut by discarding fragments in order to simulate circular or elliptical splats. The pipeline is implemented in at least two consecutive rendering passes. In the first pass, the model is rendered into the depth buffer in order to avoid incorrect over-

lapping splats. The second pass is required to accumulate attributes, such as color or normal vector, for smooth blending between overlapping splats. The attribute values of each fragment are weighted according to the distance of the fragment to the splat center using a Gaussian function. In the final fast normalization pass, the attribute values for each fragment are divided by the sum of weights of contributions from overlapping splats. This framework can be extended with the approach in [30], in which distances of fragments to their splat centers are mapped to the depth buffer. The resulting splats appear as screen aligned Voronoi regions.

Although splatting can be implemented on graphics hardware, it has some drawbacks, such as blurring artifacts at a closeup view. This is caused by blending through Gaussian filtering in regions where splats overlap. Overlapping splats are also a problem with respect to rendering speed. Intense overlapping results in undesirable overdraw, meaning that significantly more fragments are produced than are finally displayed. Another limitation is that splat blending in current GPU-based implementations relies on multiple rendering passes for blending of overlapping splats. This leads to loss in rendering speed, and moreover, it is more difficult to integrate multipass approaches into hybrid rendering systems, e.g., models are rendered using different primitives, such as splats together with triangles.

2.2 Patch-based Representation

Representing 3D geometry through a set of parameterized patches has been successfully developed during recent years in the computer graphics. Given a mesh, Lee et al. [17] consider a displaced subdivision surface by constructing a smoothed control mesh, which is displaced by scalar valued maps. Gu et al. [13] proposed geometry images: complete surface is parameterized on a planar domain and resampled on a regular grid, providing one image that represents the surface. They extend their method to multichart geometry images [29] in order to achieve a better approximation of the original model. However, this method uses a more complex parameterization than just displacing geometry orthogonal to a base domain, and thus, is less suitable for a real time rendering application.

Another approach for representing geometry with height fields is given by spatial patches [14] . However, their method tends to end up with a number of patches about two to three magnitudes higher than the one we propose to use. For example, the bunny model is represented by 1526 patches, while we only need 30 patches applying our construction method. It is essential to represent the model with a small number of patches, since the rendering of each patch requires a number of API calls to the graphics driver.

Height fields have also been used for processing point-sampled geometry using spectral methods. Pauly and Gross [24] decompose a point model into a set of patches that are resampled to rectangular images. These images are analyzed and modified in the frequency domain using discrete Fourier transform for geometry filtering. In [23], this approach has been used for the application of compression. The difference to [24] is that patches are not resampled to rectangular images, but rather have irregular shapes. The resampling technique is also optimized in the sense that points are resampled on the MLS surface of the original model.

Except for the work of Ivanov and Kuzmin [14], all these approaches only discuss applications of a patch representation, such as modelling, compression and shape analysis, rather than addressing rendering. In our work, we focus on the problem of fast rendering of these data on the GPU under the constraint to keep the data in a compact representation. We furthermore discuss the generality of the elevation maps rendering approach with regards to using different primitives such as triangles and points. Furthermore, we focus on the advantages of exploiting the regular structure of the elevation maps for attribute interpolation for splat rendering.

3 Pipeline

Our proposed renderer is capable of displaying mesh- as well as splat-based surfaces. Our input data consists of a set of planes in 3D space that are expressed by a reference point r in $\mathbb{R}^3$ and a normal vector n. For each plane we have a height map that gives the signed distance of each point to its plane in 3D space. Moreover, the elevation maps have irregular shapes, thus, the planes have compact supports.

For completeness, we will revisit the scheme of patch construction as described in [23]. Please note that this method relies on point-based input data. In the next section we show that this approach can be extended to mesh-based surfaces in a straight forward fashion.

3.1 Patch Construction and Resampling

We consider a given 3D model as set of points $\mathcal{M} \subset \mathbb{R}^3$, e.g., points that have been acquired by a laser range scanner and approximate the real surface. For this set we associate a continuous surface $\mathcal{S} = \mathcal{S}(\mathcal{M}) \subset \mathbb{R}^3$, which we need to evaluate geometry normal vectors for the input model. For point-based models we define $\mathcal{S}$ by the Moving-Least-Squares surface (MLS) [3, 2] for which the surface normal can be evaluated for any point on $\mathcal{S}$. For mesh models we define $\mathcal{S}$ to be the set of piecewise linear interpolation surfaces that are defined by the triangles. Please note that the resulting surface may not be differentiable on the positions of the input points (vertices). For these points we therefore define a surface normal by averaging the normals from incident triangles.

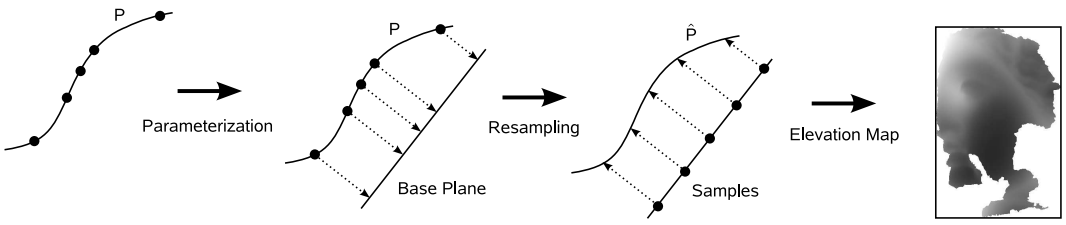


Figure 2. Construction of elevation maps for a given surface patch $\mathcal{P}$ (left); original samples are projected onto the base plane, defining the support of the elevation maps (left middle); the support is resampled on a regular grid, providing new samples which are elevated in order to produce points on the original surface $\mathcal{P}$. The resulting surface $\hat{\mathcal{P}}$ is an approximation of the original one (middle right); the resulting elevation map is a shaped 2D image (right).

We now define a surface patch $\mathcal{P}$ as subset of $\mathcal{S}$. Given $\mathcal{P}$, we consider

$$\hat{n}(\mathcal{P}) = \arg\min_{n\in\mathbb{R}^3} \max_{x\in\mathcal{P}} \|n_{\mathcal{P}}(x) - n\|,$$

whereas $n_{\mathcal{P}}(x)$ is the surface normal of $\mathcal{P}$ in x, $\|n_{\mathcal{P}}(x)\| = 1$. The vector $n(\mathcal{P}) = \hat{n}(\mathcal{P})/\|\hat{n}(\mathcal{P})\|$ gives the axis of the normal cone, when considering the normals in $\mathcal{P}$. For computation of $n(\mathcal{P})$ we use miniballs [11], while we evaluate the surface normals at a number of points in the patch.

Given $n(\mathcal{P})$ we compute the cone aperture $A(\mathcal{P}) := \min_{x\in\mathcal{P}} \left(1 - \frac{1}{2}\|n_{\mathcal{P}}(x) - n(\mathcal{P})\|^2\right)$. A condition that $\mathcal{P}$ can be parameterized as height field is $A(\mathcal{P}) \geq \cos\phi$, $\phi = 90°$. In practice, we choose an angle of $\phi < 90°$ in order to bound the variance of surface normal vectors in the patch.

As in [23, 24], our patch layout is constructed in a split-merge fashion. We start by letting $\mathcal{U} := \{\mathcal{S}\}$. For any $\mathcal{P} \in \mathcal{U}$ we evaluate $A(\mathcal{P})$ and replace $\mathcal{P}$ in $\mathcal{U}$ by two new patches $\mathcal{P}_0$ and $\mathcal{P}_1$ with $\mathcal{P}_0 \cup \mathcal{P}_1 = \mathcal{P}$, and $\mathcal{P}_0 \cap \mathcal{P}_1 = \emptyset$. To obtain $\mathcal{P}_0$ and $\mathcal{P}_1$, we split up $\mathcal{P}$, if $A(\mathcal{P}) < \cos\phi$. The splitting is performed using principal component analysis (PCA), whereby $\mathcal{P}$ is split through its center and across its major principal axis, which is computed by solving the Eigensystem of the covariance matrix of $\mathcal{P}$ and selecting the Eigenvector with the largest absolute Eigenvalue.

After splitting, we have a set of patches $\{\mathcal{P}_i\}$, each of which fulfills the normal cone condition. In a second phase we reduce the number of patches by iteratively merging pairs, which lead to the least increase in normal approximation error $E(\mathcal{P})$:

$$E(\mathcal{P}) = \int_{x\in\mathcal{P}} \|n_{\mathcal{P}}(x) - n(\mathcal{P})\|^2 dx.$$

For fast patch merging, we use a priority queue that holds merge pair candidates of adjacent patches with respect to their error increases. We merge the candidate $(\mathcal{P}_i, \mathcal{P}_j)$ with the least increase in error as long as $A(\mathcal{P}_i \cup \mathcal{P}_j) < \cos\phi$.

For each patch $\mathcal{P}_i$, we have a reference plane that is given by the axis vector n_i of the normal cone and a reference point r_i. We choose r_i such that the signed distance between the original point $p \in \mathcal{P}_i$ to its projection onto the plane is positive: $\langle r_i - p, n_i\rangle < 0$. Having the reference plane, we construct a regularly sampled version of $\mathcal{P}_i$, namely $\hat{\mathcal{P}}_i$, by placing sample positions on the plane on a regular grid with a defined resolution. The resolution is chosen such that the number of samples in $\hat{\mathcal{P}}_i$ is approximately the same as the number of points in $\mathcal{P}_i$. If the original surface $\mathcal{S}$ is defined by a mesh, we linearly interpolate the elevation values of points in $\hat{\mathcal{P}}_i$ from elevation values in $\mathcal{P}_i$. In the case that $\mathcal{S}$ represents a MLS surface, we use the iterative method in [23] in order to find the elevation values for the samples in $\hat{\mathcal{P}}_i$. The process of parameterization and resampling is summarized in figure 2.

Using the resampling procedure, we also implement an elevation map test, since the normal cone condition is a necessary, however, not a sufficient condition for the underlying surface to be an elevation map. During resampling, we compare the elevation values of each sample to elevation values in the 8-neighborhood. When the resulting variation is larger than a given threshold, we declare the patch not to be an elevation map over a planar domain. Using PCA, we split up all patches that do not fulfill the resampling elevation variation test.

After partitioning and resampling we have a set of patches $\hat{\mathcal{P}}_i$ that give an approximation $\hat{\mathcal{S}}$ of $\mathcal{S}$. Each patch $\hat{\mathcal{P}}_i$ consists of the following components:

- **Plane parameters**: These are three values (r_i, n_i, s_i) that denote the position and the orientation of the reference plane in 3D space and the sampling step length respectively.
- **Elevation map**: These are the elevation values for each sample on the reference plane with compact support. For points that do not belong the map we set the elevation value to be zero.

Besides the elevation maps, we have additional attribute maps, e.g., normal maps which store the surface normal for each sample.

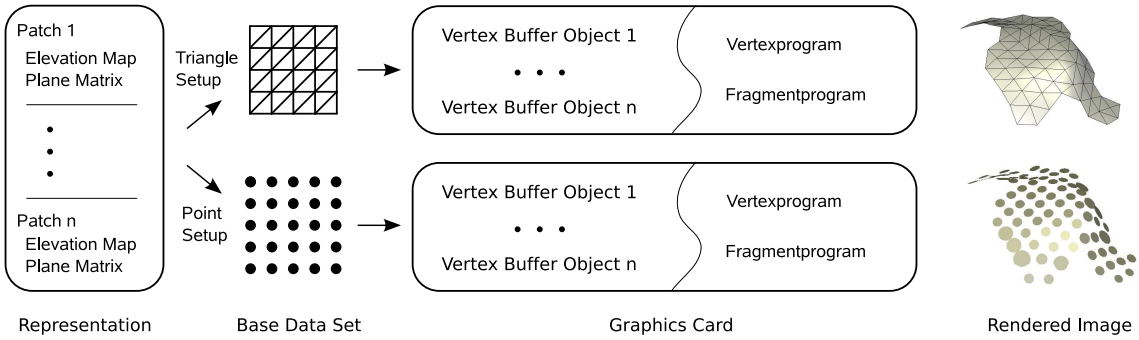


Figure 3. Overview of our rendering pipeline; Elevation maps are encoded using vertex buffer objects and are passed to the vertex and fragment shader unit of the GPU; we implement either mesh-based or point-based rendering by using a mesh or points as base data set respectively.

3.2 Representation on the GPU

Both, elevation maps and side information, can efficiently be stored in the GPU memory. Figure 3 shows an overview of our rendering pipeline. Due to regular sampling, point positions on the plane can directly be represented by an integral number, namely the row and column of the respective point on the grid. For hardware-accelerated rendering of the resampled model $\hat{\mathcal{S}}$ we apply the standard vertex and fragment shader pipeline to a base data set, which consists of a regularly distribution of rendering primitives, e.g., triangles or points. To render a patch, the vertex shader identifies and elevates the individual samples in the base data set according to the elevation map. In order to move the samples to their position in 3D space, a single matrix multiplication is applied. Considering the fact that our maps have arbitrary shapes, we need to remove vertices in the base data set when rendering a patch. Since current vertex shaders do not allow removing vertices, this is performed in the fragment shader, where fragments can be discarded. However, this straight forward approach has the disadvantage that the vertex unit produces an overhead of geometry that leads to performance loss. With our maps this waste may grow up to about 80% of the entire model.

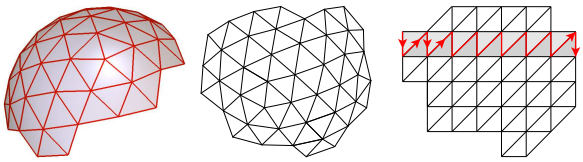
We therefore propose to store a list of addresses of map supported samples in the elevation map of each patch by using vertex buffer objects. For each patch we have two components. Firstly, the elevation map that holds the geometric information, and secondly, the index map that defines the support of the elevation map as well as topological information for mesh rendering. We postpone the description of the index map to the next two subsections in which we discuss mesh and point rendering in more detail.

For both rendering primitives, we use the same type of maps that hold elevation and attributes, such as the normal vector for each sample in $\hat{\mathcal{S}}$. For efficient storage we quantize elevation values to 16 bits, which is sufficient even for models with complex geometry [9]. The normal vectors are expressed in polar coordinates, which are also quantized with 16 bits each. In experiments we found that quantization to this level leads to rendering without noticeable artifacts. For each sample we have a total bit budget of 48 bits that are stored using three short integral numbers. In comparison to rendering the original model $\mathcal{S}$, we have a slight overall memory savings compared to the common approach, although if similar quantization of coordinates and normals is used.

3.3 Mesh Rendering

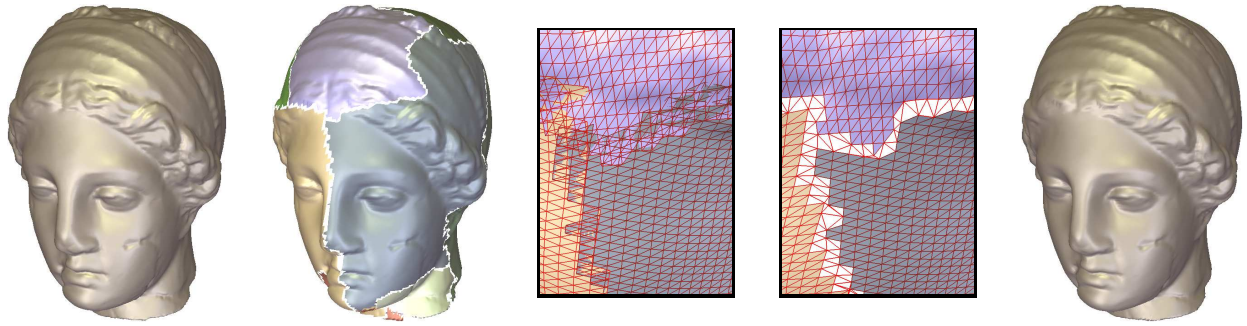
To render the complete model, we implement mesh rendering by using a single mesh as the base data set. This means that only the base set needs to be triangulated, see figure 3, which can be performed in linear time.

However, rendering the base mesh for each patch leads to problems on the patch boundaries. Since our patches have irregular shapes, the removal of vertices would be necessary. We emphasize that this leads to problems, since vertices on the boundary are connected to vertices outside of the patch. To overcome this problem we store a triangulation for each patch individually using triangle strips that can



sphere patch | parameterized mesh | triangle strip on resampled mesh

Figure 4. A given patch in mesh representation (left) is parameterized on a plane (middle) and resampled on a grid (right). The resulting triangulation is represented through triangle strips.



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

Figure 5. Patch gaps occur when approximating the original model (a) by a resampled mesh with patches (b); To close the gaps we propose two methods, firstly, we let patches slightly overlap (c); secondly, we develop a method that fills the gaps with triangles (d); the resulting rendering provides similarly good visual results for both methods (e).

directly be derived from the base triangulation and the support of the elevation map. These strips are constructed in linear time by traversing the triangles with map supported vertices row by row (see figure 4). The stripification reduces the total memory usage, since vertex indices are redundantly stored when holding the data as triangle soup. The advantage of this stripification method is its simplicity of computation. As will be shown in the results section, the combination of vertex position encoding (see subsection 3.2) and stripification reduces to total memory usage of the original model down to about 70%.

For each patch we set up two vertex buffer objects. The first one holds the addresses and quantized normal vectors of the supported vertices as described in the last subsection. The second VBO is the index buffer that holds the stripified triangle mesh of the patch.

A basic problem of rendering the patches with a base mesh is the occurrence of patch gaps due to lack of connectivity information between patches (figure 5b). In the past, several methods have been proposed that deal with closing such gaps, e.g., the zippering method in [29], and the gap filling approach in [4]. The main idea of the first method is to partition the gaps into cut paths that are used to zipper opposing boundary parts in consecutive unification steps. The second approach [4] considers orthogonal projections of border vertices onto opposing edges. For time-efficient processing a priority-queue is maintained that holds candidates for consecutive merging steps. Both methods produce high-quality gap closings, but they rely on changing positions of vertices on the border of the patches. In our setting we want to keep the vertices fixed and find a pure retriangulation of the gaps. To achieve this we propose a new gap filling method that is inspired by [4] and works as follows.

For each patch $\hat{P}_i$ in the resampled surface $\hat{\mathcal{S}}$ we consider the boundary $\partial\hat{P}_i$ which is the set of points on edges that share only one triangle in the triangulation of $\hat{P}_i$ and also the boundary vertices that are incident to these edges. Our goal is to connect vertices on the border of each patch with edges on another patch. This provides new triangles that connect the patches. For this purpose we consider projections of boundary vertices $v \in \partial\hat{P}_i$ onto $\partial\hat{P}_j$, $i \neq j$. More precisely, for each $\hat{P}_i$ and border vertex $v \in \partial\hat{P}_i$ we compute

$$p_j(v) := \arg \min_{p \in \partial\hat{P}_j} \|p - v\|. \tag{1}$$

Since $p_j(v)$ is a point on the boundary $\partial\hat{P}_j$, this point will either be on a boundary edge or a boundary vertex. In the first case we consider the pair v_1, $v_2 \in \partial\hat{P}_j$ of incident vertices to this edge. In the second case we choose one incident edge to the vertex $p_j(v)$ and also consider the pair of incident vertices v_1 and v_2. We now form and insert a new triangle between the three vertices v, v_1 and v_2, see figure 6. In a consecutive step we update the border information for edges and vertices that have been affected by this insertion step as follows:

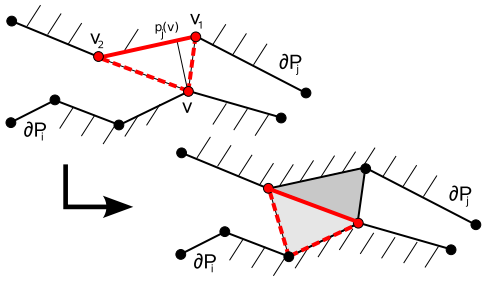


Figure 6. Triangle insertion in our gap filling method; upper left: projection of border vertex v onto an opposing edge (v_1, v_2); lower right: triangles are inserted as long as they do not intersect with an existing part of the surface.

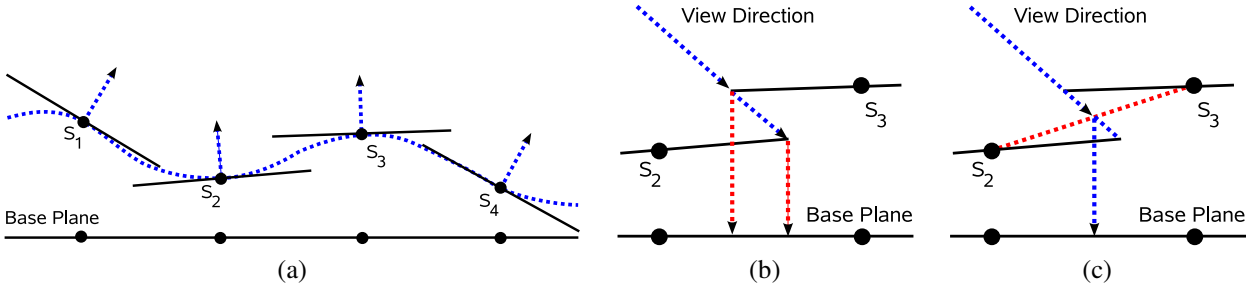


Figure 7. Our attribute interpolation method; (a) Splats on the resampled surface $\hat{S}$ with associated normal vectors. The dotted line indicates smoothly interpolated normal vectors that we aim to achieve; (b) Projection of fragment positions onto the base plane for computation of interpolation weights leads to discontinuities; (c) We intersect the viewing ray with a plane through the samples in the neighborhood to achieve a continuous attribute interpolation.

- The edge (v_1, v_2) becomes an inner edge, since it shares two triangles.
- We have two new border edges, namely (v, v_1) and (v, v_2), if there is no previously inserted triangle with an edge (v, v_1) or (v, v_2) respectively.
- The vertex v is no longer a border vertex, but an inner vertex, once all edges incident to v have become inner edges.

When inserting a new triangle (v, v_1, v_2), we check for intersections with triangles incident to v, v_1 and v_2 to circumvent the occurrence of overlapping triangles. This is mandatory, since in (1) there is no topological condition that prevents the triangle (v, v_1, v_2) to intersect with an existing part of the surface.

Additionally to the gap filling method that produces watertight models (figure 5d), we propose a second approach in which we do not connect the patches. For high-quality rendering only, we assert that it is rather sufficient to allow slightly overlapping patches so that the gaps disappear (figure 5c). We implement this overlapping by growing the patches beyond their borders with triangles that are not farther apart than a certain distance to the boundary of the original patch. Although, this is a rather simple approach, it results in high visual quality (figure 5e).

Results for our two gap closing methods are shown in figure 5. Figure 5a shows the original Venus model and a rendering of the resampled model with patch gaps. Figure 5b shows overlapping patches and the final rendering of the reconstructed model. Figure 5c demonstrates the construction of new triangles in our gap filling method. Triangles are iteratively inserted by projecting border vertices onto opposing edges and by forming triangles that do not overlap with an existing part of the surface. The resulting closeup view of a region with gaps and the resulting rendering is shown in 5d.

3.4 Splat Rendering with Attribute Interpolation

Splat rendering as a variant of point rendering is gaining more and more interest, since it can be implemented in a hardware accelerated environment and points do not rely on topological constraints. In our scenario, we implement a splat renderer by using a base point set. The vertex buffers hold an elevation value, a normal vector, and a splat radius for each point. The index buffer holds a list of addresses for points in the support of the elevation map.

The resulting splat model can be rendered in a vertex and fragment shader pipeline as proposed in [6]. This method has proven to provide a good tradeoff between splatting quality and rendering speed using a two pass rendering approach. In the first pass, splats are rendered into the depth buffer in order to enable blending only for splats that slightly differ in depth value. In the second pass, weighted contributions of overlapping fragments are accumulated by alpha blending. The weighted sums are normalized in a final normalization rendering pass.

In our framework, the points are regularly sampled on the base plane of each patch. We propose to exploit this information to implement a splat renderer that provides smooth interpolation of attribute values without time consuming blending. Figure 7(top) shows an example of four splats with associated normal vectors. Each fragment corresponds to a specific position on a splat. For this position we derive a normal vector that is an interpolation of the three splat normal vectors in the neighborhood. In the following, we denote points in $\hat{S}$ by p, and their projection onto the respective base plane by $\bar{p}$. Given a fragment f with position p_f in world space, we project back p_f onto the base plane, yielding a point $\bar{p}_f$ on the plane. We identify the three samples $(\bar{p}_1, \bar{p}_2, \bar{p}_3)$ in the patch that form a triangle $\bar{t}$ that contains $\bar{p}_f$.

The straight forward interpolation approach is established by interpolating attributes for fragment f using the barycentric coordinates of point $\bar{p}_f$ in triangle $(\bar{p}_1, \bar{p}_2, \bar{p}_3)$. We emphasize that this leads to discontinuities in the resulting rendering, since splats do not produce continuous surfaces. In the example in figure 7b, we show the scenario of a fragment being produced by two possible splats. Using this direct interpolation method leads to different interpolation weights due to spatial gaps between splats (indicated by the two red-dotted lines).

To achieve smooth interpolation, we propose to use both, attribute information and elevation values from neighboring points. This enables us to construct a plane h through the three points p_1, p_2 and p_3. We intersect the viewing ray with the plane h, yielding a point q. We now use the projection of q onto the base plane to interpolate attribute values for fragment f. Figure 7c shows that we achieve identical interpolation weights for the fragments intersecting the viewing ray on both splats.

Projecting fragment positions onto the base plane may cause a situation in which attributes cannot be interpolated due to missing neighbor information, e.g., at the patch boundaries. In these cases we discard the respective fragments in the fragment stage which prevents visual artifacts. To ensure hole free rendering, we let patches slightly overlap during construction, as required for rendering with a base mesh, see figure 5c.

We implement our interpolation scheme in the fragment shader, where attribute values for fragments are needed for lighting. For each fragment, we evaluate the attribute and elevation values of points p_1, p_2 and p_3 by using fast texture lookups. The corresponding plane is evaluated by a small number of cross product computations.

Since our attribute interpolation method replaces commonly used splat blending, it is no longer necessary to accumulate contributions of overlapping splats. This means that we reduce the rendering complexity by one rendering pass which results in a significant acceleration of the entire splat rendering pipeline. The interpolation also removes blurring artifacts that are visible in the closeup view when using standard splatting techniques. This provides the same superior visual quality as achieved with high resolution meshes.

The drawback of our method is a slightly increased memory usage in comparison to traditional splat rendering. Due to hardware limitations, per point elevation and normal data has to be stored twice on the GPU: Firstly, as VBO for the vertex shader, and secondly, as texture for the interpolation in the fragment shader. We stress that this problem can be fixed by using the textures also in the vertex shader. However, vertex texture lookups on current GeForce architectures still limit the vertex rate to a peak throughput of 33 millions vertices per second. A corresponding implementation would lead to a significant loss in rendering speed.

4 Results

For our experiments, we use a 2.8GHz Intel Pentium4 CPU, 2 GB DDR-RAM and a GeForce 6800GT/AGP/SSE2 graphics card running Linux, using OpenGL2.0 API with Nvidia driver version 76.76. Our shaders are implemented using the Nvidia Cg shading language [20].

To compare our renderer to state-of-the-art methods, we render the original models as triangle meshes and circular splats. The original meshes are rendered in two ways, firstly, as a simple triangle list, and secondly, using triangle strips that are computed in a preprocessing step by using the NvTriStrip library [22]. For splat rendering of the original models, we reimplemented the renderer in [6]. We consider both variants, fast, but low-quality splat rendering without blending, and slower, but high-quality rendering with blending.

Figure 9 shows renderings of the Dragon model using the different rendering approaches. The three closeup views show renderings of our representation in figure 9a. The closeup view in figure 9b illustrates the results using a mesh setup. Figures 9c and 9d use a point setup with attribute blending and attribute interpolation respectively. Our splat attribute interpolation method (figure 9d) provides visual results that are similar to the quality achieved through rendering our representation with the base mesh, figure 9b. In the splat rendered image (figure 9c), there are visible blurring artifacts. This is due to the fact that the Gaussian pixel color blending does not lead to reconstruction of accurate lighting. With our interpolation method, we achieve visual results similar to the quality of mesh rendering.

Figures 10 and 11 compare the rendering performance for models of various complexity for the different rendering approaches discussed in this paper. Comparing the results with respect to memory efficiency and rendering speed, we note three major results:

1. Rendering the models with our representation in a mesh setup yields better frame rates, and at the same time, requires significantly less memory in comparison to rendering the original model as a pure triangle list. The NvTriStrip method [22] produces slightly better triangle strips than our method and therefore achieves a slight improvement in rendering speed.

2. With our interpolation method, we achieve higher frame rates than by using the standard blending technique. Since our method is implemented using only one rendering pass, we achieve a significant acceleration in rendering speed compared to the commonly used blending technique. For large models, e.g., the Model of Imperia [15], our interpolation scheme provides frame rates that are slightly below the rates that are achieved by plain splat rendering without blending.

Figure 8. Model of Imperia rendered using 3.6M splats without interpolation at a screen resolution of 1024×1024 with 9.6 fps. When splat sizes drop down to few pixels and neither blending nor interpolation improves the visual quality.

3. At a comparable model complexity, mesh rendering is still outperforming splat rendering with respect to speed, since current graphics API and hardware has been aggressively optimized for fast triangle rendering. In contrast, the splat rendering pipeline has to be simulated through shader programs, which makes this approach more time-consuming. However, rendering very dense and complex data sets, e.g., the Model of Imperia with about 3.6 million points (figure 8), reduces the advantages of mesh rendering. The cause is that the size of the triangles drops down to a few pixels, and the graphics card cannot profit from the hardware encoded triangle rasterization. Generally speaking, the rendering speed mainly depends on the throughput rate of the fragment shader, and less on the vertex shader.

Figure 12 shows renderings of the David head. In the closeup view we observe a better visual quality for our interpolation method (right figure). The middle and the left figure show splat rendering without blending, using our elevation maps and the original data set respectively.

Figure 13 shows that our interpolation method still achieves satisfying visual quality, when reducing the density of the resampled model. We reduced the number of splats from 1.6 million points in the full data set, figure 13b, by factor ten.

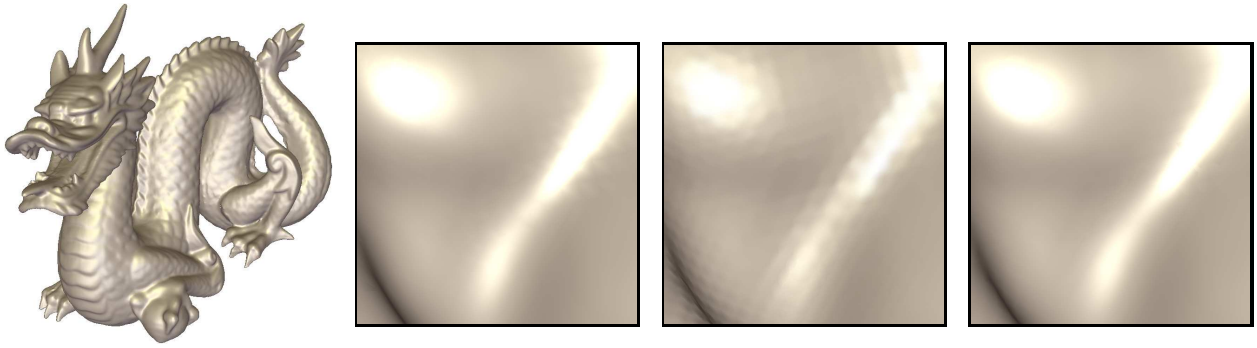
5 Discussion and Future Work

We presented a framework for rendering of 3D surfaces with elevation maps. We have shown that we can approximate a complex 3D model consisting of a large number of primitives with a relatively small number of patches. We profit from exploiting the regular structure of the underlying base domains in the context of a fast and high-quality attribute interpolation for splat rendering. This leads to an improvement in both, visual quality and rendering speed.

A positive side effect of our elevation maps is that we obtain fast mesh rendering using triangle strips that can directly be derived from the regular structure. Simultaneously, the memory usage is lower in comparison to standard stripification.

Another important advantage of our framework is that we bring splat rendering in line with mesh rendering in two ways. Firstly, since we do not need to blend overlapping splats, we reduce the rendering complexity to only one rendering pass. This makes our method suitable for hybrid applications in which mesh and splat rendering primitives are combined. Secondly, due to correct attribute interpolation, we obtain consistent lighting for neighboring fragments. With our rendering scheme, we are also able to integrate common GPU shader algorithms that have originally been designed for triangle rendering in a splat-based environment, e.g., Phong lighting, texture mapping, and environment mapping. Using a similar rendering pipeline for triangles and points, allows for choosing the application specific best rendering primitive. For example, mesh rendering can provide best results at a closeup view, while points have shown to be superior in level-of-detail applications [28].

We can extend our approach of elevation maps to higher order base domains. By fitting polynomials to the original patches, we can replace the patch planes by patch polynomials, which may be more suitable for approximating the original surface.



(a) (b) 70.7 fps (c) 15.8 fps (d) 21.5 fps

Figure 9. Renderings of the Dragon with elevation maps; complete model with splats and interpolation (a); closeup view for rendering with a base mesh (b); rendering with splats with blending (c); rendering with splats and attribute interpolation (d).

model	Balljoint	David Head	Dragon	Sphere	Shakyamuni	Model of Imperia
our representation						
# samples	138k	403k	442k	673k	1.6M	3.6M
# patches	19	383	303	39	208	769
our representation (mesh)						
# triangles	267k	760k	833k	1.32M	3.24M	7.0M
memory usage	1.57M	4.58M	5.04M	7.73M	18.7M	41.6M
frames/s	173	75.6	71.4	53.2	19.9	9.8
our representation (splats)						
memory usage (blended)	1.31M	3.84M	4.21M	6.42	15.6	34.8M
frames/s (blended)	31.9	15.6	15.1	16.4	8.0	4.7
memory usage (interpolated)	2.09M	6.14M	6.75M	10.3M	25.0M	55.7M
frames/s (interpolated)	36.2	19.2	20.3	23.9	11.4	8.9

Figure 10. The rendering performance of our elevation maps for various models at a screen resolution of 1024×1024.

model	Balljoint	David Head	Dragon	Sphere	Shakyamuni	Model of Imperia
original mesh						
# vertices	137k	398k	437k	655k	1.6M	3.6M
# triangles	274k	794k	871k	1.3M	3.2M	7.2M
memory usage	2.90M	8.39M	9.23M	13.8M	35.7M	76.0M
frames/s	102	71.8	67.2	28.9	21.9	10.4
memory usage (stripified)	2.15M	6.25M	7.0M	9.6M	25.5M	56.1M
frames/s (stripified)	194	80.2	75.3	49.8	21.9	10.3
original splats						
# splats	137k	398k	437k	655k	1.6M	3.6M
memory usage	1.83M	5.31M	5.8M	8.6M	22.7M	48.1M
frames/s (not blended)	74.7	30.6	32.1	31.3	15.3	8.4
frames/s (blended)	32.4	15.8	14.7	16.2	7.5	4.3

Figure 11. Rendering performance of the original models using the triangle and splat primitive. The positions of the vertices/points, as well as normal vectors have been quantized 16 bits per value.

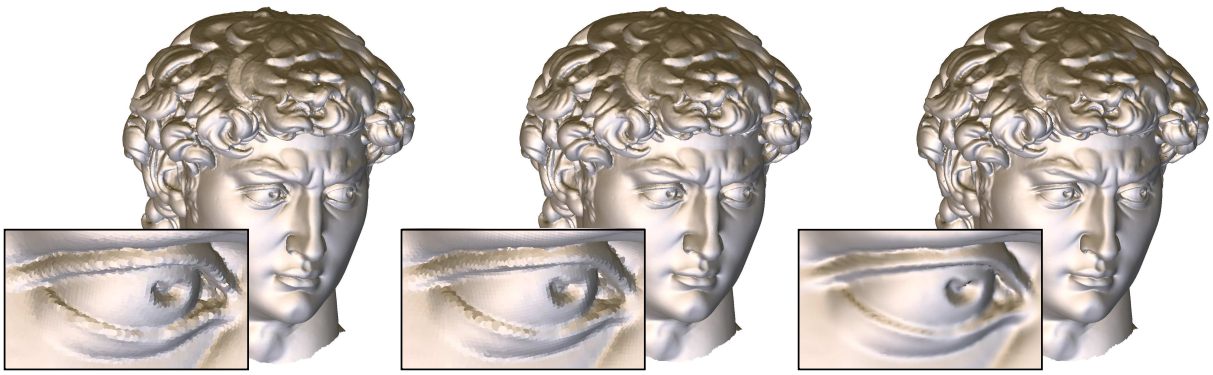


Figure 12. Rendering of David Head with the original splat model without blending (left), with our representation using splats also without blending (middle), and with our representation using interpolation (right).

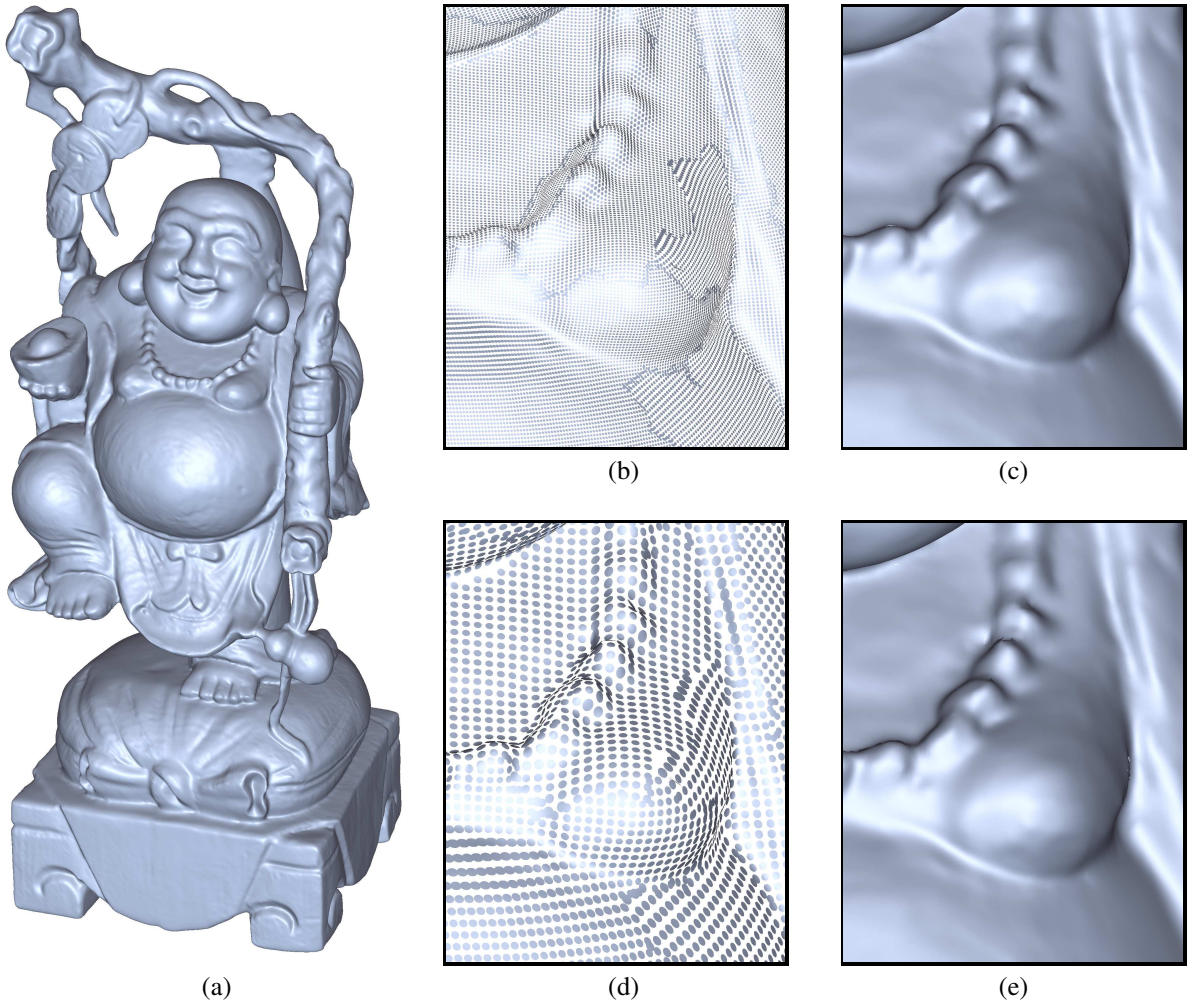


Figure 13. Rendering of the Shakyamuni statue with our interpolation method (a); closeup views with different resolutions of the resampled model (b) and (d), and the final renderings with attribute interpolation in (c) and (e), respectively.

Acknowledgments

We thank Dietmar Saupe and Marc Alexa for helpful discussions and Jörg Strecker, Felicia Burkhard and Marcel Nause for scanning the Shakyamuni-Statue and the Model of Imperia. We thank Anna Dowden-Williams for proofreading the paper. This work was supported by the DFG Graduiertenkolleg 1042 'Explorative Analysis and Visualization of Large Information Spaces'.

References

[1] A. Adamson and M. Alexa. Ray tracing point set surfaces. In *Proc. Shape Modeling International*, pages 272–279, 2003.

[2] M. Alexa and A. Adamson. On normals and projection operators for surfaces defined by point sets. In *Proc. Symposium on Point-based Graphics*, pages 149–155, 2004.

[3] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. Silva. Computing and rendering point set surfaces. *IEEE Transactions on Computer Graphics and Visualization*, 9(1):3–15, 2003.

[4] P. Borodin, M. Novotni, and R. Klein. Progressive gap closing for mesh repairing. In *Proc. Computer Graphics International*, pages 201–213, 2002.

[5] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. High-quality surface splatting on today's GPUs. In *Proc. Symposium on Point-based Graphics*, pages 17–24, 2005.

[6] M. Botsch and L. Kobbelt. High-quality point-based rendering on modern GPUs. In *Proc. Pacific Graphics*, pages 335–343, 2003.

[7] M. Botsch, M. Spernat, and L. Kobbelt. Phong splatting. In *Proc. Symposium on Point-based Graphics*, pages 25–32, 2004.

[8] M. Botsch, A. Wiratanaya, and L. Kobbelt. Efficient high quality rendering of point sampled geometry. In *Proc. Workshop on Rendering*, 2002.

[9] M. Deering. Geometry compression. In *Proc. ACM SIGGRAPH*, pages 13–20, 1995.

[10] T. K. Dey, J. Giesen, and J. Hudson. Delaunay based shape reconstruction from large data. In *Proc. IEEE Symposium in Parallel and Large Data Visualization and Graphics*, pages 19–27, 2001.

[11] B. Gärnter. Fast and robust smallest enclosing balls. In *Proc. 7th Annual European Symposium on Algorithms*, pages 325–338, 1999.

[12] J. Grossman and W. Dally. Point sample rendering. In *Proc. Workshop on Rendering*, pages 181–192, 1998.

[13] X. Gu, S. Gortler, and H. Hoppe. Geometry images. *ACM Transactions on Graphics*, 21(3):355–361, 2002.

[14] D. V. Ivanov and Y. Kuzmin. Spatial patches - a primitive for 3D model representation. *Computer Graphics Forum*, 20(3):511–521, 2001.

[15] Konstanz 3D Model Repository, Apr. 2006, http://www.inf.uni-konstanz.de/cgip/projects/surfac/.

[16] J. Krüger, J. Schneider, and R. Westermann. DUODECIM - a structure for point scan compression and rendering. In *Proc. Symposium on Point-based Graphics*, pages 99–107, 2005.

[17] A. Lee, H. Moreton, and H. Hoppe. Displaced subdivision surfaces. In *Proc. ACM SIGGRAPH*, pages 85–94, 2000.

[18] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The Digital Michelangelo Project: 3D scanning of large statues. In *Proc. ACM SIGGRAPH*, pages 131–144, 2000.

[19] M. Levoy and T. Whitted. The use of points as a display primitive. Technical Report TR 85-022, Computer Science Department, University of North Carolina, January 1985.

[20] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics*, 22(3):896–907, 2003.

[21] G. Meenakshisundaram and D. Eppstein. Single-strip triangulation of manifolds with arbitrary topology. *Computer Graphics Forum*, 23(3):371–379, 2004.

[22] NVIDIA Corporation, NvTriStrip library, Feb. 2004, http://developer.nvidia.com.

[23] T. Ochotta and D. Saupe. Compression of point-based 3D models by shape-adaptive wavelet coding of multi-height fields. In *Proc. Symposium on Point-based Graphics*, pages 103–112, 2004.

[24] M. Pauly and M. Gross. Spectral processing of point-sampled geometry. In *Proc. ACM SIGGRAPH*, pages 379–386, 2001.

[25] M. Pauly, R. Keiser, B. Adams, P. Dutre, M. Gross, and L. J. Guibas. Meshless animation of fracturing solids. *ACM Transactions on Graphics*, 24(3):957–964, 2005.

[26] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface elements as rendering primitives. In *Proc. ACM SIGGRAPH*, pages 335–342, 2000.

[27] L. Ren, H. Pfister, and M. Zwicker. Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. *Computer Graphics Forum*, 21(3):461–470, 2002.

[28] S. Rusinkiewicz and M. Levoy. QSplat: a multiresolution point rendering system for large meshes. In *Proc. ACM SIGGRAPH*, pages 343–352, 2000.

[29] P. Sander, Z. Wood, S. Gortler, J. Snyder, and H. Hoppe. Multi-chart geometry images. In *Proc. Symposium on Geometry Processing*, pages 146–154, 2003.

[30] J. O. Talton, N. A. Carr, and J. C. Hart. Voronoi rasterization of sparse point sets. In *Proc. Symposium on Point-based Graphics*, pages 33–37, 2005.

[31] X. Xiang, M. Held, and J. S. B. Mitchell. Fast and effective stripification of polygonal surface models. In *Proc. Symposium on Interactive 3D Graphics*, pages 71–78, 1999.

[32] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Surface splatting. In *Proc. ACM SIGGRAPH*, pages 371–378, 2001.

[33] M. Zwicker, J. Räsänen, M. Botsch, C. Dachsbacher, and M. Pauly. Perspective accurate splatting. In *Proc. Graphics Interface*, pages 247–254, 2004.