Dynamic Sampling and Rendering of Algebraic Point Set Surfaces

Gaël Guennebaud and Marcel Germann and Markus Gross

ETH Zurich

Abstract

Algebraic Point Set Surfaces (APSS) define a smooth surface from a set of points using local moving least-squares (MLS) fitting of algebraic spheres. In this paper we first revisit the spherical fitting problem and provide a new, more generic solution that includes intuitive parameters for curvature control of the fitted spheres. As a second contribution we present a novel real-time rendering system of such surfaces using a dynamic up-sampling strategy combined with a conventional splatting algorithm for high quality rendering. Our approach also includes a new view dependent geometric error tailored to efficient and adaptive up-sampling of the surface. One of the key features of our system is its high degree of flexibility that enables us to achieve high performance even for highly dynamic data or complex models by exploiting temporal coherence at the primitive level. We also address the issue of efficient spatial search data structures with respect to construction, access and GPU friendliness. Finally, we present an efficient parallel GPU implementation of the algorithms and search structures.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Curve and surface representations I.3.3 [Computer Graphics]: Viewing algorithms

1. Introduction

Point Set Surfaces (PSS) are a very attractive meshless surface representation defined by local moving least-squares (MLS) approximations of the data [Lev03,ABCO*03]. Since PSS embed a (locally controllable) low pass filter in the surface definition, they are de facto very well suited for point cloud denoising as well as for surface reconstruction. PSS have been successfully used in a wide range of applications [GP07], making them one of the most flexible surface representations for point sets.

Although the initial Levin's definition [Lev03] is still relatively expensive to compute, significant progress has been made to design simpler and more efficient definitions [AK04, AA04]. Such simple definitions can be interpreted as either a product space of two smooth vector fields or as purely planar approximations. More recently, Guennebaud and Gross [GG07] proposed an Algebraic Point Set Surface (APSS) framework to locally approximate the data using algebraic spheres. It turns out that this strategy exhibits high tolerance with respect to low sampling densities while retaining a tight approximation of the surface.

As a first major contribution of this paper we revisit the APSS problem and present a new, explicit and more generic solution that combines simplicity of formulation with efficiency of computation (section 3). A central feature of our new approach is its improved control of the curvature of the fitted sphere by means of one intuitive parameter. Not only does it allow the sphere fit to continuously degenerate to a plane fit, but it also permits users to invert or amplify surface microstructures.

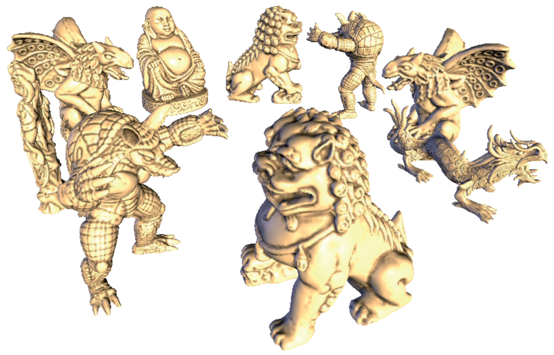
Figure 1: *APSS defined from 1.2M points rendered at 50/100 fps using our dynamic upsampling algorithm.*

Efficient high quality rendering of PSS is a non-trivial task. An obvious option is to use raycasting [AA03]. While interactive rates could be reported [WS05], such performance requires optimized point clouds and the construction of an expensive and spatially accurate data structure in a preprocess limiting the approach to static data.

As a second major contribution of this paper we make rendering likewise efficient for dynamically changing point sets. To this end, we designed a fast sampling and rendering algorithm based on forward warping and integrated APSS into this framework. Compared to the potentially superior image quality of raycasting, our algorithm avoids redundant evaluations and allows for quality/performance by adjusting the target density. The idea is to upsample the point cloud in a view dependent fashion and to render it using a standard splatting algorithm, e.g. [BHZK05]. A key feature of our algorithm is an adaptive upsampling scheme based on a new view dependent geometric error. In order to keep both the memory consumption and the number of generated splats at each frame as low as possible, we designed our algorithm to operate at the primitive level. Temporal coherence is accounted for by a low level cache mechanism and the inherent parallelism of our algorithm makes implementations on massive multi-core processors very efficient. Our rendering system is presented in section 4.

The most expensive parts of our rendering algorithm are the neighbor queries needed to evaluate the APSS during the projection of the splats onto the surface. In section 5 we present and compare several data structures with respect to our specific requirements including complexity of construction and update for dynamic point clouds as well as efficiency of access for static arrangements.

Our results in section 6 demonstrate that the algorithm is able to handle dynamic point clouds of up to 100k points in real time including the reconstruction of the data structure at each frame. In cases of local changes of the geometry our approach can even handle a few millions of samples without the need of caches or additional high level data structures.

2. Related Work

Point Set Surfaces

Point Set Surface (PSS) [ABCO*03] were initially defined as the set of stationary points of the Levin's moving least squares projection operator [Lev03]. This is an iterative procedure where at each step the point is projected onto a polynomial approximation of the data which is fitted from a local reference plane found using a non-linear optimization. By omitting the polynomial fitting step, Amenta and Kil [AK04] showed that the same surface can be defined and computed by weighted centroids and a smooth gradient field. This leads to a significantly simplified implicit surface definition and faster algorithms, especially in the presence of normals [AA04]. Such a simple approximation scheme can be extended to achieve convex interpolation using Hermite centroid evaluations [AA07].

Recently, Guennebaud and Gross [GG07] have shown that defining the surface by mean of sphere fitting significantly improves the robustness against low sampling density and reduces the approximation error of planar approaches while retaining high performance. Indeed, even though fitting polynomials allows to achieve tighter approximations, the approach breaks down as soon as the data cannot be locally represented as a height field. Another central limitation of the robustness of the previous definitions comes from the plane fit operation that becomes highly unstable when the sampling rate drops down. On the other hand, a higher order surface like a sphere better fit region of high curvature and performs better in the correct handling of sheet separation.

In the well know *multi-level partition of unity* implicit (MPU) technique [OBA*03], the authors also propose to directly fit quadrics to alleviate the limitations of polynomial fitting. In their work, the quadrics are fitted by adding a few point constraints away the surface neglecting the fact that the algebraic distance is not linear. Another related approach is *sparse low-degree implicit* (SLIM) [OBA05] where the geometry consists of points equipped with bivariate polynomials. Efficient rendering is accomplished by blending the primitives in screen space. However this approach still suffers from the polynomial fitting limitations and does not properly define a smooth surface since it depends on the view direction.

Dynamic Point Cloud Refinement

A common approach to control the sampling density of a point cloud spread over an implicit surface is to use a particle simulation procedure [Tur92, PGK02]. Targeting real-time performance, Guennebaud et al. present both a dyadic [GBP04] and $\sqrt{3}$ [GBP05] iterative refinement schemes based on the construction of accurate one ring neighborhoods. Again, in practice it turns out that such iterative refinement schemes are too costly to be able to handle dynamic data. For instance the authors report a point generation rate around 300k/s while a highly dynamic point cloud require a similar rate but per frame instead of per second.

In their initial PSS work [ABCO*03], Alexa et al. present an efficient upsampling algorithm where the tangent plane of each input sample is uniformly upsampled and projected onto a precomputed polynomial approximating the underlying surface. Since the so-generated "patches" have to overlap each other, this strategy does not provide properly upsampled point clouds like the previous methods do. Conversely, the simplicity of this scheme makes it amenable to efficient GPU implementation. For this reason we adopted their basic upsampling procedure, but in our case the generated splats are indeed projected onto the true PSS hence yielding a smooth surface and avoiding both preprocessing and extra storage requirements.

Geometric Error Metric

Related to our work are so called geometric error metrics that allow to locally adapt the sampling density according to a given threshold. Several meshless methods have been proposed for offline processing [PGK02, WK04]. The sequential point trees rendering system [DVS03] includes a view dependent geometric error for multi-resolution rendering of point clouds. However their approach cannot be extended to refinement and is quite expensive to compute. In this paper we therefore present a new view dependent geometric error metric tailored for upsampling combined with splatting.

3. Surface Definition

In this section we present our surface definition as a generalized version of the Guennebaud et al.'s APSS approach [GG07]. For the reader's convenience we will briefly summarize the fundamentals of APSS and we refer to the previous paper for the details.

3.1. General Settings

Throughout the paper we will consider as input a set of points $P = \{\mathbf{p}_i \in \mathbb{R}^3\}$ equipped with normals $\mathbf{n}_i$ and radii r_i representing the local point spacing. If the normals are not available, then they can be computed in a preprocess using, for instance, an algebraic sphere fitting without normals [GG07]. Similarly, the radii r_i can be computed using a local estimation of the density. These radii are used to define the following adaptive weighting scheme

$$w_i(\mathbf{x}) = \phi\left(\frac{\|\mathbf{p}_i - \mathbf{x}\|}{r_i \cdot h}\right) \tag{1}$$

describing the weight of the point $\mathbf{p}_i$ for any point $\mathbf{x} \in \mathbb{R}^3$. Here, h is a global scale factor allowing to adjust the influence radius of every point, and ϕ is a smooth, decreasing weight function for which we use the following compactly supported polynomial

$$\phi(x) = \begin{cases} (1-x^2)^4 & \text{if } x < 1 \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

3.2. The APSS approach

The key idea of APSS is to locally approximate the point cloud by a fitted algebraic sphere that moves continuously in space. An algebraic sphere is defined as the 0-isosurface of the scalar field $s_{\mathbf{u}}(\mathbf{x}) = [1,\ \mathbf{x}^T,\ \mathbf{x}^T\mathbf{x}]\mathbf{u}$, where $\mathbf{u} = [u_0, \ldots, u_4]^T \in \mathbb{R}^5$ is the vector of scalar coefficients describing the sphere. Then, the APSS S_P approximating the point cloud P yields as the zero set of an implicit scalar field $f(\mathbf{x})$ representing the distance between the evaluation point $\mathbf{x}$ and a locally fitted algebraic sphere $\mathbf{u}(\mathbf{x})$:

$$f(\mathbf{x}) = s_{\mathbf{u}(\mathbf{x})}(\mathbf{x}) = \left[1,\ \mathbf{x}^T,\ \mathbf{x}^T\mathbf{x}\right]\mathbf{u}(\mathbf{x}) = 0\,. \tag{3}$$

The sphere $\mathbf{u}(\mathbf{x})$ is obtained by minimizing given distances between itself and the neighbors of $\mathbf{x}$ in a weighted least square sense. The original algorithm minimizes the *positional constraints* $s_{\mathbf{u}}(\mathbf{p}_i) = 0$ and the *derivative constraints* $\nabla s_{\mathbf{u}}(\mathbf{p}_i) = \mathbf{n}_i$ simultaneously such that:

$$\mathbf{u}(\mathbf{x}) = \arg\min_{\mathbf{u}} \sum_i w_i(\mathbf{x})\left(s_{\mathbf{u}}(\mathbf{p}_i)^2 + \|\nabla s_{\mathbf{u}}(\mathbf{p}_i) - \mathbf{n}_i\|^2\right)\,. \tag{4}$$

This minimization yields a standard system of linear equations. In order to reduce the dependence on the scale of the point cloud, the authors proposed to scale the derivative constraints by an arbitrarily large value. They observed that this choice makes the fitting more robust to low sampling density, less prone to oscillations and less sensitive to outliers.

3.3. Direct APSS

The novel and generalized fitting procedure we propose minimizes the two key constraints (positional, derivative) separately and starts with the derivative constraints. We observe that, unlike the positional constraints, the derivative constraints provide sufficient information to determine the four coefficients u_1 to u_4 that define the gradient of the sphere. Intuitively, the positional constraints are subsequently only used to specify the isovalue to approximate the data. In practice, minimizing the set of derivative constraints yields the normal equation below which can be solved explicitly.

$$\begin{bmatrix} \sum w_i(\mathbf{x})\mathbf{I}_3 & 2\sum w_i(\mathbf{x})\mathbf{p}_i \\ (2\sum w_i(\mathbf{x})\mathbf{p}_i)^T & 4\sum w_i(\mathbf{x})\mathbf{p}_i^T\mathbf{p}_i \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} \sum w_i(\mathbf{x})\mathbf{n}_i \\ 2\sum w_i(\mathbf{x})\mathbf{p}_i^T\mathbf{n}_i \end{bmatrix} \tag{5}$$

Next, using the algebraic distance constraints to determine the value of u_0, we obtain the following explicit solution for the coefficients of $\mathbf{u}(\mathbf{x})$:

$$\begin{aligned} u_4 &= \beta\frac{1}{2}\frac{\sum w_i\mathbf{p}_i^T\mathbf{n}_i - \sum\tilde{w}_i\mathbf{p}_i^T\sum w_i\mathbf{n}_i}{\sum w_i\mathbf{p}_i^T\mathbf{p}_i - \sum\tilde{w}_i\mathbf{p}_i^T\sum w_i\mathbf{p}_i} \\ \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} &= \sum\tilde{w}_i\mathbf{n}_i - 2u_4\sum\tilde{w}_i\mathbf{p}_i \\ u_0 &= -[u_1 u_2 u_3]\sum\tilde{w}_i\mathbf{p}_i - u_4\sum\tilde{w}_i\mathbf{p}_i^T\mathbf{p}_i \end{aligned} \tag{6}$$

where $w_i = w_i(\mathbf{x})$ and $\tilde{w}_i$ is the normalized weight of the sample $\mathbf{p}_i$: $\tilde{w}_i = w_i / \sum_j w_j$. β is an additional scalar parameter that, for the time being, is equal to one. The equations for the gradients are omitted, but can be derived easily.

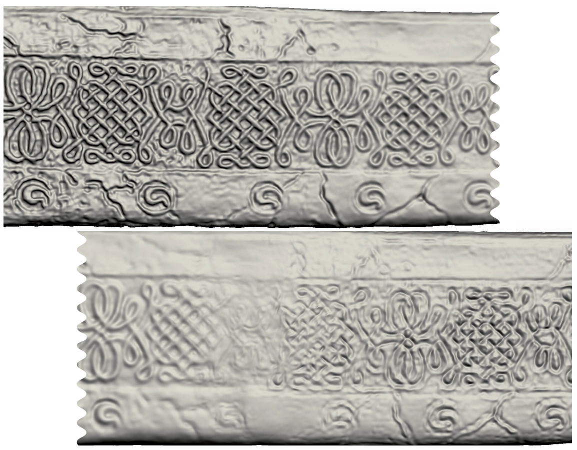

Figure 2: *Smooth variation of the curvature parameter β from 8 to 1 (top) and 1 to −4.5 (bottom). Note the inversion effect on the surface microstructure.*

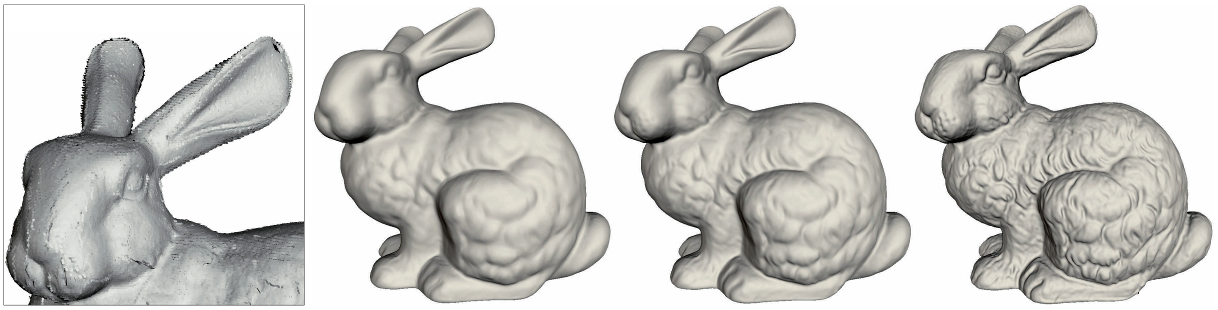

Figure 3: *APSS reconstruction of a noisy data (left) using different values of the curvature control parameter β: -1, 0.5, 5.*

3.4. Plane Fit and Curvature Control

If we utilize our new spherical fitting procedure to fit a plane by setting $u_4 = 0$ (i.e., $\beta = 0$) the remaining formulas correspond *exactly* to the ones of a plane fit with normal averaging as used in the PSS definition of [AA04]. By introducing the parameter β, we can therefore continuously tweak our algebraic spherical fit from a pure planar fit and a pure spherical fit for $\beta \in [0,1]$. This is illustrated in figures 3 and 2.

More generally, this parameter allows us to control the curvature of the fitted sphere. For instance, a negative value of β inverts the curvature, thus increasing the APSS smoothing effect or even allowing to invert the surface features as depicted in figure 2-bottom. Note however, that extreme settings of β reduce the stability of the representation which means that this feature should only be used for densely sampled models. Conversely, a value greater than one tends to exaggerate the surface features (figure 2-top).

This new feature enhancement is particularly useful in the case of noisy data as it enables to preserve the surface structure while removing the noise (figure 3-right). Note that it is also possible to locally adapt this parameter over the input point cloud and still retain a continuous surface as long as β varies smoothly. Figure 2 depicts an example.

4. Realtime Upsampling

In this section, our main goal is to design an efficient and as flexible as possible rendering procedure of APSS. As explained in the introduction, we opted for a forward warping approach. The general principle of our algorithm is relatively simple. After building the spatial data structure required for a fast evaluation of the surface (see section 5), the rendering of a frame is basically a four steps procedure which is entirely performed on the GPU:

1. **Selection** of the visible samples. This step includes common view frustum and back face culling.
2. **Adaptive up-sampling** of the neighborhood of each selected sample yielding a dense set of splats.
3. **Projection** of the splats onto the APSS.
4. **Splatting** of the generated splats.

The central step is the second one which is described in the next section 4.1. Some additional issues regarding the projection steps are discussed in section 4.2. A parallel implementation is presented in section 4.3 and extended to support temporal coherence in section 4.4. For the remainder of this section, we use the following naming convention: a *sample* represents an input point $\mathbf{p}_i$, while a *splat* defines a newly created point on the surface for the purpose of rendering.

4.1. View Dependent Upsampling

Remark that each input sample $\mathbf{p}_i$ of normal $\mathbf{n}_i$ and radius r_i can be interpreted as an oriented disk. The task is to generate sufficiently many splats onto these disks such that the resulting set of points is dense enough to be rendered using a splatting algorithm. For the upsampling we opted for a simple but very efficient approach that generates a regular pattern of $m \times m$ splats within the quad bounding the disk of each input sample (figure 4a).

The main issue is now to determine m for each input sample $\mathbf{p}_i$. A first criterion is obviously to ensure that the screen space sizes of the splats are below a given pixel threshold t_s. To this end we use the following approximation

$$m = \eta \frac{r_i}{t_s} \tag{7}$$

where η is the scale factor between the viewport and the input point $\mathbf{p}_i$. A simple choice for t_s is one or two pixels. However, when the underlying surface is flat and parallel to the screen, for instance, this choice produces prohibitively many splats without contributing to visual quality.

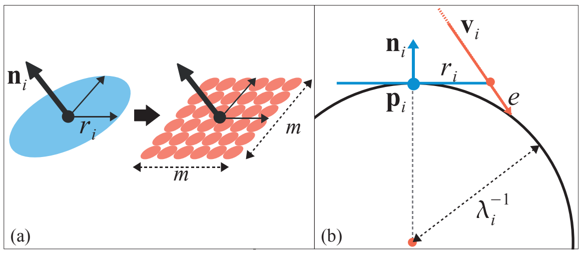

Figure 4: *(a) Illustration of the upsampling algorithm. (b) 2D view to illustrate our view dependent geometric error. The tangent disk representing $\mathbf{p}_i$ is displayed in blue and the viewing ray in red.*

A more sophisticated approach makes our refinement procedure both feature and view sensitive. For this purpose we propose a new view dependent geometric error metric. The key idea is to define the error as the maximal distance between the sample and the underlying surface along the view direction (figure 4b). Since the accurate computation of this error is too expensive for realtime, we approximate the underlying surface by a tangent sphere passing through $\mathbf{p}_i$. The radius of this sphere is conservatively estimated from the principal curvature λ_i:

$$\lambda_i = \max_j \frac{\|\mathbf{n}_i - \mathbf{n}_j\|}{\|\mathbf{p}_i - \mathbf{p}_j\|} . \tag{8}$$

A second approximation is to assume that the view direction $\mathbf{v}_i = \mathbf{p}_i - \mathbf{c}$ remains constant over the input sample, where $\mathbf{c}$ is the position of the camera. Using these approximations, the computation of our view dependent error e boils down to a 2D problem considering the plane spanned by the vectors $\mathbf{n}_i$ and $\mathbf{v}_i$ and passing through $\mathbf{p}_i$. Correspondingly, the input sample can now be seen as a segment, and the sphere as a circle. e is eventually obtained by computing the intersection between the circle and the ray with direction $\mathbf{v}_i$ and origin in the extremity of the splat in the direction of $\mathbf{v}_i$. This is illustrated in figure 4b. Finally the number of generated splats is set such that the minima of e and r_i in screen space is below the threshold t_s:

$$m = \sqrt{\eta \frac{\min(e, r_i)}{t_s}} . \tag{9}$$

The square root is needed because e varies quadratically if we consider the underlying surface to be a sphere. The effect of this strategy is shown in figure 5 where we see that both the high curvature regions and the silhouettes are properly taken into account by our geometric error.

4.2. MLS Projection

After upsampling, each generated splat has to be projected onto the underlying APSS. There are three different projection procedures [AA04]. The *simple* one iteratively projects the sample onto the best locally fitted sphere: $\mathbf{q}_{n+1} = project(\mathbf{q}_n, \mathbf{u}(\mathbf{q}_n))$. The *almost orthogonal* always projects the initial point $\mathbf{x}$: $\mathbf{q}_{n+1} = project(\mathbf{x}, \mathbf{u}(\mathbf{q}_n))$, and the orthogonal projection uses the real gradient of the scalar field.

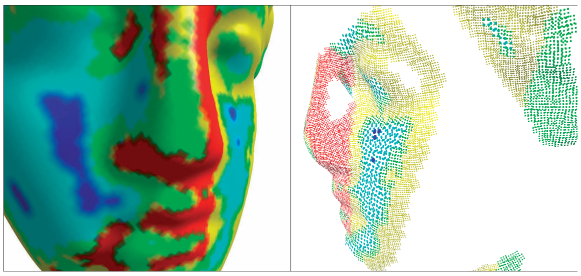

Figure 5: *Illustration of our view dependent geometric error metric with color coded density. The right picture shows the generated splats utilized for rendering the left images. Note the accurate culling (view and backface) and the higher density near the silhouettes.*

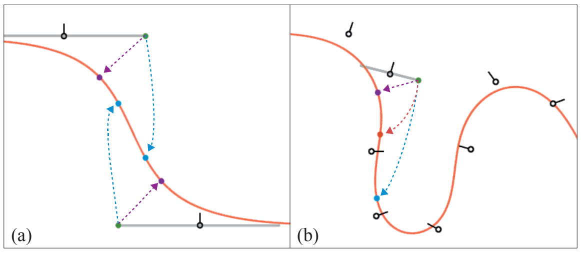

Figure 6: *Comparisons of the three projection procedures for some splat extremities. Blue: simple. Red: almost orthogonal. Magenta: orthogonal. Note that in (a) the simple and almost orthogonal operators overlap.*

As illustrated in figure 6a, the orthogonal projection appears to not be suited for our purpose since, in addition to be prohibitively expensive, it can lead to uneven sample distributions. The *simple* projection leads to the best coverage but may sometimes project points very far away (figure 6b). Finally, the *almost orthogonal* projection appears to be the best choice since it is simple and efficient, has reasonable convergence, and still provides a good coverage when the tangential components of the input samples do not overlap.

Note that after the projection, the distances between the generated splats vary and hence their radii might have to be updated. In particular, if their distances increase too much, then some holes could appear. However, since an APSS approximates the input point cloud, the point spacing has a tendency to decrease instead of increasing. Indeed, we observed an average shrinking of 98% with a negative and positive standard deviations of 18% and 3.7% respectively. Moreover, since the radii of the generated splats are set to the pattern resolution, a hole free rendering is achieved while the point spacing does not increase by a factor greater that $\sqrt{2}$. In practice, we therefore never experimented such holes.

4.3. Parallel Implementation

The presented algorithms can be implemented on massively parallel multicore architectures. As a main implementation target we employ a GeForce 8800 using CUDA, but our concepts can be applied to most other multi-core processor units as well. We assume the reader to be familiar with the parallel programming model offered by CUDA and refer to [NVI07] for an introduction. Since current GPUs process multiple data in parallel in an SIMD fashion, a first challenge is to avoid sparse computations. A second challenge is to avoid thread communication which constitutes a major source of bottleneck.

The different stages of our algorithm are illustrated in figure 7. We assume the input samples to be sequentially stored in memory. The selection step consists of building a vector $\mathbb{V}$

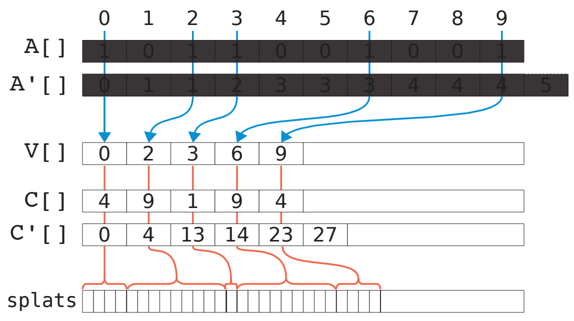

Figure 7: *Parallel implementation of the selection (blue part) and upsampling (red part) algorithms.*

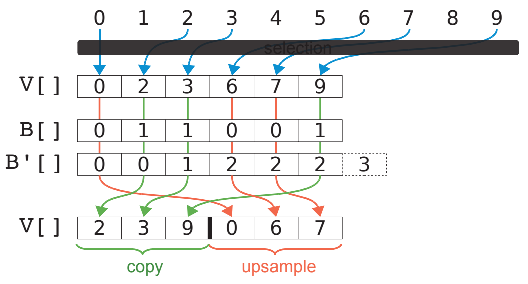

Figure 8: *Illustration of the prefix sum based classification used in our low level temporal coherence algorithm.*

which contains the list of the selected sample indices. In order to avoid the need of a global and shared counter to build this index vector, we use the following three step algorithm:

```
1. foreach $\mathbf{p_i}$
     A[i] = is_visible($\mathbf{p_i}$) ? 1 : 0
2. A′ = prefix_sum(A[i])
3. foreach $\mathbf{p_i}$
     if A[i] then V[A′[i]] = i
```

The two `foreach` loops are parallelized with potentially one thread per sample. The clue of this procedure is the use of an efficient parallel *prefix sum* algorithm [Ble90, HSO07] to compute the vector A′ such that $\mathtt{A'[i]} = \sum_{j=0}^{i-1} \mathtt{A[j]}$. Indeed, after this step, the vector A′ contains for each sample the number of selected samples having a lower index. For a selected sample that number exactly corresponds to its position in the target index vector V. Note that if nb is the number of input samples then A′[nb] holds the number of selected samples nb_v.

For the view dependent upsampling algorithm we have to generate a variable number of splats per selected sample. In addition, all the new splats have to be sequentially stored in memory. To this end we use the same technique as before. For each selected sample, we write to the *counter vector* C the number of splats that have to be generated. Next, we use the prefix sum algorithm to build a new vector C′ from C, and in a third step we fill a vector of splats as follows:

```
foreach $j_i \in$V
   for k=0 to C[i]-1
     splats[C′[i]+k] = gen_splat(k,$\mathbf{p}_{j_i}$)
```

Only the first loop is parallelized and the projection step becomes trivially parallelizable.

In fact, the latest GPUs support the required features (i.e., the *geometry shader* and *transform feedback buffer*) to implement the previous algorithms without the need of *prefix sum* passes. While the presented approach is somewhat more complex to implement, it provides several advantages: it is surprisingly much faster, in particular for the upsampling step where we observed a speedup of an order of magnitude, and it is less dependent on the actual architecture and thus more generic. Moreover, it gives us more flexibility to design an efficient low level temporal coherence mechanism described in the next section.

4.4. Low Level Temporal Coherence

In the previous algorithm, all the splats required for rendering are generated from scratch for each frame. If some parts of the model are static, we can additionally take advantage of temporal coherence and only perform local updates in the active splat buffer (list of the generated splats). To achieve this goal, we modify the previous parallel algorithm as follows. First, we add two additional attributes nb_i and δ_i to each input sample representing the number of generated splats and the position of the first splat in the actual splat buffer respectively. Secondly, the selection step is slightly modified such that nb_i is set to zero if the i^{th} sample is culled which has for effect to invalidate its corresponding set of rendering splats.

Next, after computing the vector V, we compute and update the number of generated splats nb_{j_i} for each $j_i \in$V. Then we sort the vector V into two distinct segments such that the indices of all samples with valid refined splats are sequentially stored at the first segment of V. To this end, we use a similar principle as for the selection step (see figure 8). During the update of the values nb_{j_i}, we build a vector B such that B[i] equals one if nb_{j_i} changed and zero otherwise. Next we compute the vector B′ from B using a *prefix sum*, we then read the number of samples which are still valid nb_c =B′[nb_v], and do the sort as follows:

```
foreach $j_i \in$V
   k = B[i]==1 ? B′[i] : $nb_c$+B′[i]+i
   V[k] = $j_i$
   C[k] = $nb_{j_i}$
```

At the same time we *pack* the value nb_{j_i} into the vector C such that the *prefix sum* algorithm can efficiently compute C′ without the need of additional, performance-killing indirections.

Finally, the new splat buffer is obtained by simple copies from the old one for the first part of V and by using the up-sampling procedure followed by the MLS projection for the second part. Note that during the copy and upsampling passes, we have to update each δ_{j_i} from C′[i] such that we are able to perform the copies at the next frame. In the case of a dynamic point cloud, we have to invalidate the set of rendering splats of each sample $\mathbf{p}_i$ for which at least one of its neighbors, or itself, has changed. This is simply achieved by setting $nb_i = 0$ for such samples.

4.5. High Level Culling

For very large models, i.e. beyond a few millions of splats, it becomes expensive to check at each frame the whole set of input splats. For such complex scenes it is common to add a high level data structure (DS) enabling per block culling (including occlusion culling) and low resolution LOD selection. Adding such mechanisms to our refinement procedure is a relatively simple task. Let us assume that each cell of the high level DS stores its set of sample indices sequentially in the device memory. Moreover, each cell stores the smallest radius value of its samples such that we can rapidly and conservatively determine if the point cloud is already dense enough for this cell using equation 7. At each frame, we traverse the DS and build two sequential index buffers using efficient device to device memory copies representing respectively the list of the samples which are dense enough and the list of the samples which have to be processed by our low level up-sampling algorithm. This classification allows us to apply on the first buffer a simplified version of our algorithm where the geometric error evaluation and the upsampling are disabled.

5. Data Structures for Efficient GPU Implementations

In this section we present and discuss several octree variants for efficient MLS spherical fitting on a GPU. Given a query point $\mathbf{x}$, the purpose of such a data structure (DS) is to find every sample $\mathbf{p}_i$ such that the weight $w_i(\mathbf{x})$ is strictly positive. According to the weighting scheme presented in section 3, the influence of a sample $\mathbf{p}_i$ is bounded by a ball of center $\mathbf{p}_i$ and radius $r_i h$. Our problem is hence equivalent to finding all the balls containing $\mathbf{x}$.

5.1. Redundant Octree

One common option is to use a space partitioning where each cell stores the list of all the samples (balls) that intersect the cell, e.g. like in [WS05]. In the following we call such a DS *redundant*. Here, smaller cells lead to more efficient queries but the construction/updates and memory cost are more expensive.

Usually such DS store the references to the samples at the leaves only. In this work we propose a different and, to our knowledge, new strategy where the basic principle is to store a sample at a given level that only depends on its influence radius. In particular we propose an octree using the following iterative insertion procedure. Starting from an empty octree, we insert a point $\mathbf{p}_i$ into the finest level l having a cell size greater than $t \cdot 2r_i h$, where t controls the tradeoff. Thanks to the regularity of an octree, this insertion procedure can be accomplished very efficiently. For each cell of the virtual grid of level l covered by the ball of center $\mathbf{p}_i$ and radius $r_i h$, we compute the location code [FP02] of the cell in the tree and insert i into it using a fast top down procedure which, at the same time, builds the potentially missing nodes.

Compared to a conventional *split* and *sort* construction strategy, we observed a speedup of an order of magnitude. Moreover, with our strategy dynamic updates are as simple and efficient as with a grid for instance. Note that performing a neighbor query with such an octree does not require any recursive process and is thus very well suited for GPUs. Indeed, all we have to do is to go down the tree visiting all the nodes containing $\mathbf{x}$ until we reach a leaf. This traversal can be implemented very efficiently using bitwise operations [FP02]. A typical choice for the tradeoff value is to take $t = 1$ which guarantees that the bounding ball of a sample does not overlap more than 2^3 cells.

5.2. Single Reference Octree

While the previous approach delivers optimal query performance, such a DS is not optimal with respect to memory consumption and is still relatively expensive to build and update. When construction time and/or memory consumption matter, we show that the previous octree can easily be adapted so that each sample is referenced only once.

Indeed, let each sample be inserted into the unique cell containing it at the finest level and having a cell size greater than $t \cdot 2r_i h$. Then all the neighbors of a query point $\mathbf{x}$ are guaranteed to be found by visiting at each level l of the hierarchy, all the cells intersecting the ball of center $\mathbf{x}$ and radius $\frac{w_l}{2t}$, where w_l is the cell size of the level l. Moreover, since the samples are referenced only once, such a DS can also be used to perform other kind of neighbor queries or to use another weighting scheme. For instance, while increasing the global scale factor h with a redundant DS requires to rebuild it, here we only have to scale the radius of the query balls. On the other hand, the queries are obviously much more expensive since we have to loop over several cells at each level of the hierarchy.

5.3. GPU Construction

In some applications the construction of the DS is a very critical issue. In such cases we propose to build the DS directly on the GPU, thus reducing the CPU to GPU memory transfers at the same time. However, our GPU based construction algorithm currently support only a particular case of octrees where all the cells of every level are explicitly stored. Such octrees can be seen as a dyadic pyramid of grids. In the reminder of the paper we will refer to redundant pyramid and single pyramid for such versions of our previous octrees.

Purcell et al. [PDC*03] proposed to build a spatial grid on a GPU using a bitonic sort. While inherently parallel, this approach appears to be quite expensive and is not easily applicable to our redundant DS.

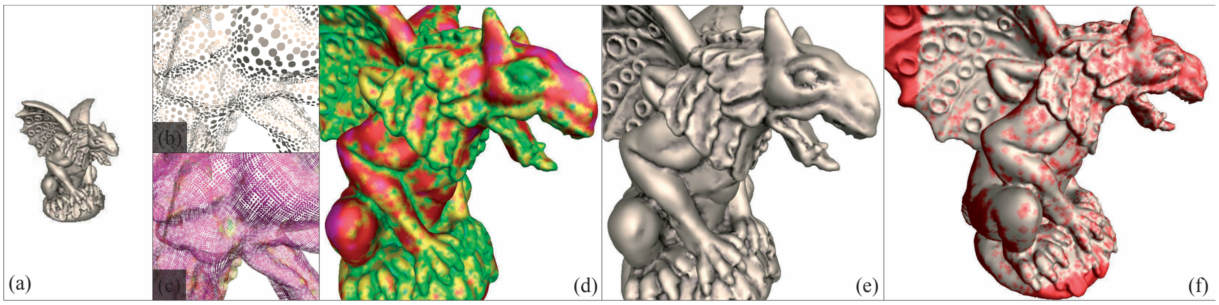

Figure 9: *Pictures taken from the walkthrough of figure 13. (a) frame #1. (b) and (c) frame #370 showing respectively the input point cloud and the refined one with color coded density. (d) and (e) frame #411 rendered with color coded density and accessibility shading respectively. (f) frame #412 with temporal coherence activated where the red parts correspond the newly generated splats.*

We therefore opted for a more basic algorithm: for each sample $\mathbf{p}_i$, we insert its index i into the lists of all the cells covered by it. First, let us remark that with our DS the number of covered cells by a single sample is bounded by a given value b, e.g., with a single pyramid $b = 1$. We therefore avoid the need of a dynamic stack by storing the indices of the cells as linked lists stored in a buffer where the b elements from $b \cdot i$ to $b \cdot (i+1)$ are allocated to the sample $\mathbf{p}_i$. Another issue is that two different threads cannot update the list of the same cell at the same time. Here, because the risk of conflict is relatively low, we opted for a mutex strategy. Since it is not possible to allocate one mutex per cell, and real mutex are not implementable on current GPU, we use the following algorithm. We allocate a vector M of m bytes in shared memory. When a thread has to edit the cell of index i_c, the thread writes its index i_t at the position $M[i_c\%m]$ and repeats this step until $M[i_c\%m]$ is indeed equal to i_t. Then the cell can be safely updated. Obviously this algorithm only works if all the threads are executed in parallel and in a SIMD fashion. We therefore have to run a single warp per SIMD core. Furthermore, this technique does not allow to synchronize the threads which are not in the same SIMD core since there is no shared memory between them. Our solution is therefore to build one DS per SIMD core and to merge them in an additional pass.

Finally, in order to optimize the memory access during the neighbor queries, we observed that it often pays off to sort the list buffer such that the list elements of a cell are sequentially stored in memory. This operation is efficiently implemented using a *prefix sum* algorithm similar to our upsampling algorithm (see section 4.3).

6. Results

In this section we present some results obtained with a Core2 duo 2.4GHz processor and an NVIDIA GeForce 8800 GTX. Our dynamic upsampling algorithm is entirely implemented in CUDA, except the final rendering of the generated splats which is performed using a standard multi-pass splatting algorithm with OpenGL [BHZK05]. All our results were obtained without the high level culling system.

Comparison of Data Structures

In section 5 we presented four different DS where the tradeoff accuracy versus build time is controlled by the parameter t. The influence of this parameter is shown by the graphs in figure 11. For comparison, these results were obtained from a roughly uniform point cloud allowing us to include an optimally sized *redundant grid*. Note that the step effects shown by these graphs are due to both the uniformity of the input and the dyadic refinement of the cells.

As expected, the redundant DS are much more sensitive to this parameter than the other ones. It appears that all the redundant DS exhibit very similar query and CPU based construction performances. However, the redundant pyramid remains interesting since its simplicity allows to enable a faster GPU based construction. Even though the *single* DS exhibit relatively poor query performance, they remain, by far, the fastest to build and the most compact ones. Note that the construction performances were obtained using a single thread for the CPU versions, and 32 threads times 8 cores for the CUDA versions. Even though such settings are far to allow to use the GPU at its best efficiency, our CUDA implementations are still an order of magnitude faster.

Upsampling Performance

Figure 13 shows the cost of each different part of our algorithm for a walkthrough around a static and non uniform model of 85k points stored in a redundant octree. The sequence starts with a zoom shown in figures 9abc and includes a large jump between two consecutive frames, hence explaining the peak in the temporal coherence graph. The figures 9ef depict the behavior of our temporal coherence algorithm in this case of temporal incoherence. As we can see, with the temporal coherence enabled, the rendering cost of a frame is mainly dominated by the splatting algorithm, thus proving the efficiency of our approach.

Figure 12 shows that the performance of the MLS projections does not only depend on the accuracy of the DS, but also on the coherence of the input data. This behavior is due to both the SIMD architecture of current GPUs and the coherence of the memory accesses. This figure was obtained

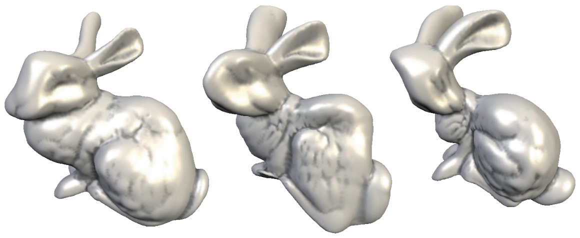

Figure 10: *Global deformation involving realtime construction of the data structure.*

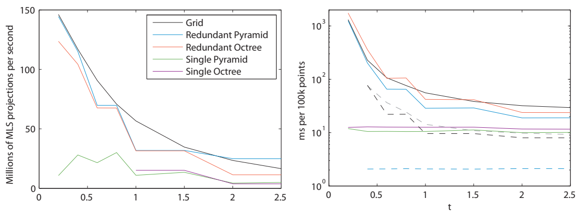

Figure 11: *Performance of the MLS projections (left) and construction time (right) for varying values tradeoff t and for different data structures. The solid and dashed lines stand for the CPU and GPU DS constructions respectively.*

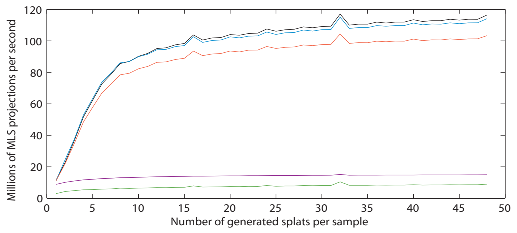

Figure 12: *Performance of the MLS projection (in million of projection per second) according to the number of generated splats per sample. We use the same color scheme than in figure 11.*

for an input point cloud randomly sorted in space. As we can see, the projection rate significantly varies from a few millions to more than 100 millions of projections per second depending on the number of generated splats per input sample. Hence, generating more but coherent splats, as in the closeup views, might indeed increase the performance.

Figure 10 depicts a global and fast deformable object rendered by our system at about 45 fps. In this sequence the DS (*redundant pyramid*) is reconstructed from scratch at each frame and represents about 25% of the rendering time while the deformation itself represents 50%. Note that, as suggested in [GG07], this sequence was rendered with *curvature shading* using the radius of the fitted spheres as an approximation of the surface curvature. Our system can also handle large models as illustrated in figure 1.

7. Discussion and Conclusion

In this paper we first presented a new general solution to perform the spherical fits used to define APSS. In particular, our new approach offers a curvature control parameter allowing to inverse or enhance the surface microstructures. Since this feature is embedded into the surface definition, high smoothing can be accomplished while preserving the surface structures. However, it would be interesting to better understand the properties of this feature and compare it to other nonlinear smoothing techniques. While very intuitive to use, let us remind that this parameter has to be used with some care since inverting the sphere curvature significantly reduces the stability of the representation to low sampling density.

We also presented a realtime parallel upsampling framework of APSS targeting efficient and flexible rendering. To achieve our goals we had to do some compromises. For instance, the simplicity of the upsampling scheme makes it not really well suited for applications other than rendering. Moreover, a hole-free rendering can only be guaranteed if the set of the input tangent quads used to generate the rendering splats cover the entire surface. To overcome this limitation we could use a higher order approximation for generating the splats, as in the SLIM method [OBA05] for instance. This would, however, requires additional preprocessing and storage requirements. In practice, the use of a non orthogonal projection operator handles such cases well.

We emphasise that our approach can easily be used to render most of point based representations, like implicit MLS [SOS04, Kol05] or MPU implicits [OBA*03] for instance. Our approach can also easily be extended to support both an anisotropic weighting scheme [AA06] and anisotropic upsampling pattern that is particularly useful during anisotropic deformations for instance.

Compared to latest adaptive mesh-based refinement techniques [BS07], our approach is entirely performed on the GPU in a parallel fashion that allows us to handle very large input models. Moreover, since we do not have any connectivity to manage, we successfully designed a very efficient low level temporal coherence mechanism. It would therefore be interesting to adapt our techniques to the refinement of meshes by replacing the input tangent quads by the polygons of the mesh. As future work, we would also investigate faster parallel algorithms for the construction of the spatial data structures, and in particular for octrees for which we currently do not have efficient solution.

Finally, we believe that the high degree of flexibility offered by our approach, combined with its high performance allow to investigate new applications for Point Set Surfaces, like interactive shape modeling for instance.

Acknowledgments

This work was supported by the ERCIM “Alain Bensoussan” Fellowship Programme. The models of figures 1 and 9 are provided courtesy of the AIM@SHAPE Shape Repository (Chinese Lion, Gargoyle, Buddha) and the Stanford 3D Scanning Repository (Armadillo, Asian Dragon, Bunny).

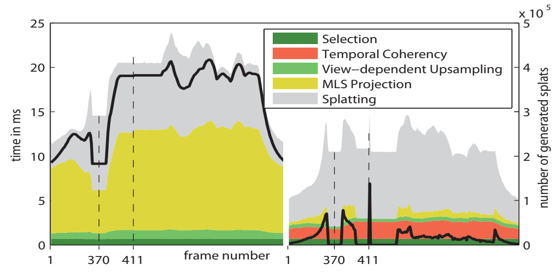

Figure 13: *Rendering cost in ms of each parts of our rendering algorithm for a walkthrough, with (left) and without (right) temporal coherence. The bold curve corresponds to the number of MLS projections.*

References

[AA03] ADAMSON A., ALEXA M.: Approximating and intersecting surfaces from points. In *Proceedings of the Eurographics Symposium on Geometry Processing 2003* (2003), pp. 230–239.

[AA04] ALEXA M., ADAMSON A.: On normals and projection operators for surfaces defined by point sets. In *Proceedings of the Eurographics Symposium on Point-Based Graphics* (2004).

[AA06] ADAMSON A., ALEXA M.: Anisotropic point set surfaces. In *Afrigraph '06: Proceedings of the 4th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa* (2006), ACM Press, pp. 7–13.

[AA07] ALEXA M., ADAMSON A.: Interpolatory point set surfaces - convexity and hermite data. *ACM Transactions on Graphics, to appear* (2007).

[ABCO*03] ALEXA M., BEHR J., COHEN-OR D., FLEISHMAN S., LEVIN D., SILVA C. T.: Computing and rendering point set surfaces. *IEEE Transactions on Computer Graphics and Visualization 9*, 1 (2003), 3–15.

[AK04] AMENTA N., KIL Y.: Defining point-set surfaces. *ACM Transactions on Graphics (SIGGRAPH 2004 Proceedings) 23*, 3 (2004), 264–270.

[BHZK05] BOTSCH M., HORNUNG A., ZWICKER M., KOBBELT L.: High-quality surface splatting on today's GPUs. In *Proceedings of Symposium on Point-Based Graphics 2005* (2005), pp. 17–24.

[Ble90] BLELLOCH G. E.: Prefix sums and their applications. In *Synthesis of Parallel Algorithms*, John H. Reif (Ed.), 1990.

[BS07] BOUBEKEUR T., SCHLICK C.: A flexible kernel for adaptive mesh refinement on gpu. *Computer Graphics Forum to appear* (2007).

[DVS03] DACHSBACHER C., VOGELGSANG C., STAMMINGER M.: Sequential point trees. In *ACM Transactions on Graphics (SIGGRAPH 2003 Proceedings)* (2003), ACM Press, pp. 657–662.

[FP02] FRISKEN S., PERRY R.: Simple and efficient traversal methods for quadtrees and octrees. *Journal of Graphics Tools 7*, 3 (2002).

[GBP04] GUENNEBAUD G., BARTHE L., PAULIN M.: Dynamic surfel set refinement for high-quality rendering. *Computers & Graphics 28* (2004), 827–838.

[GBP05] GUENNEBAUD G., BARTHE L., PAULIN M.: Interpolatory refinement for real-time processing of point-based geometry. *Computer Graphics Forum (Proceedings of Eurographics 2005) 24*, 3 (2005), 657–666.

[GG07] GUENNEBAUD G., GROSS M.: Algebraic point set surfaces. *ACM Transactions on Graphics (SIGGRAPH 2007 Proceedings) 26*, 3 (2007), 23.1–23.9.

[GP07] GROSS M., PFISTER H. (Eds.): *Point Based Graphics*. Morgan Kaufmann, 2007.

[HSO07] HARRIS M., SENGUPTA S., OWENS D. J. D.: Parallel prefix sum (scan) with cuda. In *GPU Gems 3*, Hubert Nguyen (Ed.), 851-875, 2007.

[Kol05] KOLLURI R.: Provably good moving least squares. In *ACM-SIAM Symposium on Discrete Algorithms* (San Francisco, CA, Jan. 2005), pp. 1008–1018.

[Lev03] LEVIN D.: Mesh-independent surface interpolation. *Geometric Modeling for Scientific Visualization* (2003), 181–187.

[NVI07] NVIDIA: Nvidia cuda: Programming guide. url: http://developer.nvidia.com/object/cuda.html, 2007.

[OBA*03] OHTAKE Y., BELYAEV A., ALEXA M., TURK G., SEIDEL H.-P.: Multi-level partition of unity implicits. *ACM Transactions on Graphics (SIGGRAPH 2003 Proceedings) 22*, 3 (2003), 463–470.

[OBA05] OHTAKE Y., BELYAEV A., ALEXA M.: Sparse low-degree implicit surfaces with applications to high quality rendering, feature extraction, and smoothing. In *Proceedings of the Eurographics Symposium on Geometry Processing 2005* (2005), pp. 149–158.

[PDC*03] PURCELL T. J., DONNER C., CAMMARANO M., JENSEN H. W., HANRAHAN P.: Photon mapping on programmable graphics hardware. In *Proceedings of Graphics Hardware* (2003), Eurographics Association, pp. 41–50.

[PGK02] PAULY M., GROSS M., KOBBELT L. P.: Efficient simplification of point-sampled surfaces. In *Proceedings of the conference on Visualization '02* (2002), pp. 163–170.

[SOS04] SHEN C., O'BRIEN J. F., SHEWCHUK J. R.: Interpolating and approximating implicit surfaces from polygon soup. *ACM Transactions on Graphics (SIGGRAPH 2004)* (2004), 896–904.

[Tur92] TURK G.: Re-tiling polygonal surfaces. In *Computer Graphics* (1992), ACM Press, pp. 55–64.

[WK04] WU J., KOBBELT L.: Optimized sub-sampling of point sets for surface splatting. *Computer Graphics Forum (Eurographics 2004 Proceedings) 23*, 3 (2004), 643–652.

[WS05] WALD I., SEIDEL H.-P.: Interactive ray tracing of point based models. In *Proceedings of the Eurographics Symposium on Point Based Graphics 2005* (2005), pp. 9–16.