Extreme simplification and rendering of point sets using algebraic multigrid

Extreme Simplification and Rendering of Point Sets using Algebraic Multigrid

Dennie Reniers Alexandru Telea

Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
d.reniers@tue.nl alext@win.tue.nl

Keywords: point set models, algebraic multigrid, extreme model simplification, real-time rendering

Abstract

We present a novel approach for extreme simplification of point set models in the context of real-time rendering. Point sets are often rendered using simple point primitives, such as oriented discs. However efficient, simple primitives are less effective in approximating large surface areas. A large number of primitives is needed to approximate even moderately simple shapes. However, often one needs to render a simplified version of the model using only a few primitives, thus to trade accuracy for simplicity. For this goal, we propose a more complex primitive, a sort of *splat*, that is able to approximate a larger and more complex surface area than the well-known oriented disc. To construct our primitive, we first decompose the target surface into quasi-flat regions, using an efficient algebraic multigrid algorithm. Next, we encode these regions into splats implemented using planar support polygons textured with color and transparency information and render the splats using a special blending algorithm. Our approach combines the advantages of mesh-less point-based techniques with traditional polygon-based techniques. We demonstrate our approach on various models.

1 Introduction

Interactive rendering of geometric models brings about the conflicting demands of high frame rates and good image quality. Changing the representation of the model to be rendered may help in achieving the right balance for particular applications. The recent direction of modeling and rendering using point primitives [6] instead of traditional triangle meshes is an example of this. Point-based rendering is more efficient for very complex models than traditional triangle rendering, because the scan-line coherence of triangles is lost when projected to a small screen space area. An additional advantage of point-based models is that the lack of connectivity information allows for efficient representation and easier editing of the model.

Many applications reduce the number of primitives that are needed to render a given model, trading off quality for model size and rendering speed. When the primitive count is reduced by more than two orders of magnitude, we speak of extreme model simplification [5]. In case of simplification of point set models, two options are available. First, one can use a lower amount of the same kind of simple point primitives as for the original finely sampled models [10], leading to a considerable decrease in simplification quality. Second, one can store more information per primitive [7, 9] so that less primitives are needed for the same simplification quality. Nevertheless, these techniques are still insufficient for extreme simplification of point sets, when our target primitive count is only a few hundred. The approximation power of the current point primitives is still too small for the large surface areas implied by the low primitive count of our extreme simplification goal. Indeed, the color and shape variation of larger surface areas can no longer be accurately captured by simple parametric models or radial basis functions.

The approach we present in this paper attempts to bridge the gap between point-based and polygon-based rendering. We aim to combine the point-based rendering model (many small, blended, mesh-less primitives) with the polygon-based model (a few large, textured, flat primitives). We want to make use of commodity graphics hardware for fast rendering, so we build our primitive model and rendering algorithm upon textured planar polygons. The trade-off for massively decreasing the primitive count and still working with a mesh-less representation is paid by a decrease in image quality. We control this trade-off by a multiscale approach, where every scale delivers a different number of primitives approximating the given model. Fine scales carry many primitives, and are close in rendering quality to the classic point-based rendering. Coarse scale levels address the issue of extreme model simplification.

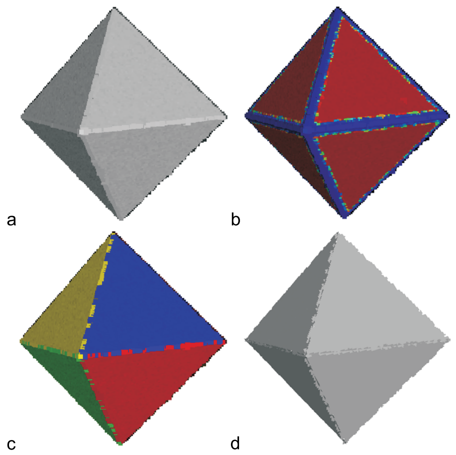

Figure 1: Global overview of our approach. An octahedron model of 16,000 points is rendered using point primitives (a). The color-coded surface classifier (b). The surface is decomposed into 8 domains, indicated by different colors (c). The model is rendered using 8 splats, or domain primitives (d).

Concisely put, we can summarize the quest of our method as “how to render point sets of hundreds of thousands of points with a few hundred mesh-less primitives”. Figure 1 illustrates this. An octahedron point set of 16,000 points is rendered using QSplat (Fig. 1a). The extreme simplification proposed by our method reduces it to eight splats, whose rendering is shown in Figure 1d. The advantage of our approach is clear; we render 8 hardware-accelerated splats, i.e. 8 textured and alpha-blended polygons, instead of 16,000 points.

Regarding extreme simplification of 3D models, our approach is related to the recently presented billboard clouds method [5]. Both methods yield a mesh-less polygon representation. However, the simplification heuristics, the implementation, obtained performance and trade-offs are significantly different for the billboard clouds technique and the one presented in this paper.

Our approach consists of three main stages: surface decomposition, primitive construction and rendering (see also Figure 2 for a detailed overview of our pipeline). Given our efficiency-motivated choice for textured polygons as rendering primitives, we decompose the point-set surface into quasi-flat regions. We compute surface flatness using a moment-based local surface classifier (Sec. 2.1) which is encoded into a finite element matrix (Sec. 2.2). The surface decomposition algorithm, based on an algebraic multigrid (AMG) method, is detailed in Section 2. The AMG method produces a multiscale representation of the input surface in terms of feature-aligned basis functions and corresponding support domains. Next, we construct one textured splat [12] for each domain and associated basis function, which we call a *domain primitive*. The domain primitives encode geometric and color information carried by each region’s point samples in the alpha, respectively color planes of a texture (Sec. 3). This effectively and efficiently replaces the original point model with a small set of textured primitives. Finally, we render the simplified model by blending together the splats (Sec. 4). The results of our method are discussed in Section 5. Section 6 concludes the paper and presents future work directions.

2 Surface Decomposition

The most complex hardware-supported primitive we avail of is a textured planar polygon. Consequently, we aim at decomposing the surface defined by the point set into a number of quasi-flat, or nearly flat, compact regions. These regions will be subsequently approximated by domain primitives, i.e. textured polygons. This decomposition is presented in the following. The method for constructing domain primitives from the quasi-flat regions is detailed further in Section 3. The complete pipeline is summarized in Figure 2.

Our decomposition method consists of three substeps. First, we use a local surface classifier to detect the point set regions corresponding to locally smooth, respectively non-smooth (curved) areas of the model (Sec. 2.1). Next, we encode this surface classifier in a finite element matrix (Sec. 2.2). Finally, we use AMG to produce a multiscale coarsening of this matrix (Sec. 2.3).

2.1 Local Surface Classification

Local surface classification attempts to assign a *smoothness* value to every point x in the point set, in order to distinguish between smooth, or quasi-flat, surface areas, and highly curved areas, such as the vicinities of edges, cusps, or tips. For this, we take into account the points in a small k-neighborhood N of x. We use a surface classifier based on the *zero and first moments* of N. We refer to [2] for a detailed description of this classifier. For this discussion it suffices to mention that computing the moments is similar to computing the surface variation [1, 8]. For this, principal component analysis on the k-neighborhood is used to obtain the eigenvectors e_0, e_1, e_2 with associated eigenvalues $\lambda_0 > \lambda_1 > \lambda_2$. The eigenvectors e_0 and e_1 form an approximate tangent plane at x, while e_2 is the normal of that plane. The surface variation can then be expressed as the variation λ_2 of the points along the normal e_2 normalized by the total variation along all three eigenvectors. The size k of the neighborhood clearly acts as discrete scale parameter or filter that removes small-scale surface noise. Figure 1b shows the surface classifier on a point set model. Smooth surface regions appear red, whereas highly non-smooth ones (e.g. creases) appear blue.

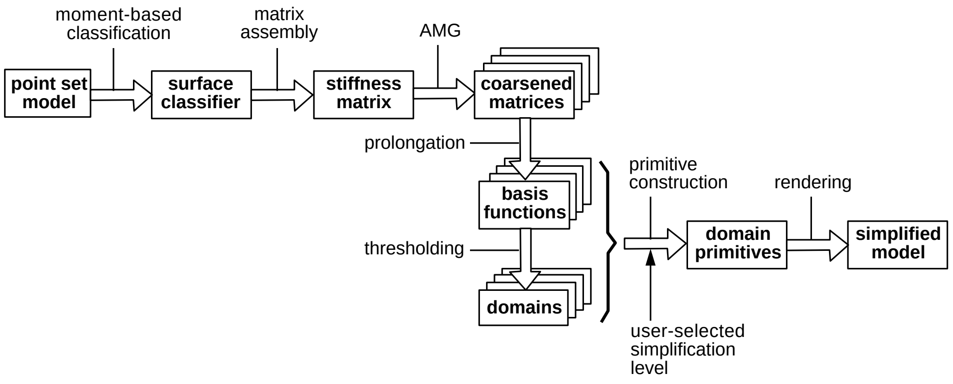

Figure 2: Overview of our approach

2.2 Matrix Encoding of Surface Classifier

Our goal is to use the local surface classifier introduced in the previous section to decompose the surface into quasi-flat components. For example, we would like to decompose the octahedron shown in Figure 1a into eight regions corresponding to its faces (Fig. 1c). We shall use for this decomposition an algebraic multigrid (AMG) approach (Sec. 2.3). As a prerequisite to using AMG, we need to 'convert' our classifier $\mathcal{C}$ into a mathematical operator $\mathcal{A}[\mathcal{C}]$ defined on the surface. We define this operator as:

$$\mathcal{A}[\mathcal{C}] := -\mathrm{div}_{\mathcal{M}}(\mathcal{C}\,\nabla_{\mathcal{M}}\;)\,,$$

where $\nabla_{\mathcal{M}}$ and $\mathrm{div}_{\mathcal{M}}$ are the gradient, respectively its dual divergence operator on a surface $\mathcal{M}$ embedded in $\mathbb{R}^3$. Essentially, $\mathcal{A}[\mathcal{C}]$ describes a non-uniform diffusion process on the surface $\mathcal{M}$, where the classifier $\mathcal{C}$ plays the role of diffusion coefficient. We are not interested in performing diffusion on the surface itself, just in a multiscale decomposition of the operator $\mathcal{A}[\mathcal{C}]$. To do this, we first need to discretize the operator, for which we use a finite element model. In case our surface discretization were a triangular mesh, we could directly compute A, the discrete matrix form of the operator $\mathcal{A}$. This is described in full detail in [4].

However, in our case we have a point set, not a global triangle mesh. Building the matrix A, as well as the underlying finite element model, is different in this case. We use the finite element model for point sets described in [2], which builds a *local* tangent finite element space at every point x_i in the point set. Basically, this method creates a local 2D triangulation of the neighborhood of x_i that is projected to the local tangent plane at x_i. The triangle fan around x_i defines the neighbor set.

$\tilde{A}_{ij}$ describes the coupling of point i with its neighbors j, from the point of view of i. Indeed, $\tilde{A}_{ij}$ may differ from $\tilde{A}_{ji}$, since the triangle fan computations of i and j are purely local. To yield a classical stiffness matrix A_{ij}, we symmetrize the computations by defining:

$$A_{ij} = \frac{1}{2}(\tilde{A}_{ij} + \tilde{A}_{ji}),$$

for $i \neq j$ and for the diagonal entries

$$A_{ii} = -\sum_{x_j \in \mathcal{N}(x_i)} A_{ij}.$$

The matrix A has now the same properties as a classical stiffness matrix defined e.g. on a triangulation mesh. Intuitively, A_{ij} is high if the neighbor points i and j are situated in a quasi-flat region. A_{ij} reaches its lowest values for neighbor points i and j that are separated by a crease. A is a sparse matrix, as the average size of N_i is under 10 neighbors per point. This matrix is the input of our surface decomposition, detailed in Section 2.3.

2.3 Algebraic Multigrid Decomposition

So far, we have encoded the surface's 'flatness', expressed by our moment-based classifier, into a stiffness matrix. We now proceed by performing a multiscale simplification, or coarsening, of this matrix. The aim of this phase is to deliver a multiscale of correspondingly simplified surfaces consisting of progressively larger quasi-flat regions.

For the matrix coarsening, we use an algebraic multigrid (AMG) algorithm. AMG was originally designed for solving large, sparse linear systems $Au = f$ coming from the discretization of scalar elliptic PDEs, such as diffusion problems. Briefly, given a fine-scale matrix $A^0 = A$, AMG attempts to compute a matrix sequence

$$A^l := R^l A^{l-1} P^l = (P^l)^T A^{l-1} P^l\,,$$

via a so-called Galerkin projection, or 'natural coarsening'. Here, the restriction R^l is the transpose of the prolongation, $R^l := (P^l)^T$.

The key element here is defining the prolongation matrices P^l that describe how coarse-scale (l) basis functions are computed from fine-scale ($l-1$) basis functions. AMG constructs prolongations such that the coarse-scale matrices A^l preserve the 'strong couplings' present in the fine-scale matrices A^{l-1}. For our application, it is important to mention that the construction of the prolongations $\{P^l\}_{l=0,\cdots,L}$ is equivalent to constructing a set of progressively coarser, problem-dependent bases $\{\Psi^{l,i}\}_{l=0,\cdots,L}$. On every level, these bases are aligned with the strong matrix couplings present on that level.

The number of basis functions between successive scales is reduced by a factor of approximately 2-3. This is inherent to our AMG implementation. For typical point set models, we thus obtain between 10 and 15 decomposition levels L. Moreover, the coarsest levels $L \cdots L-5$ usually contain up to a few hundred bases $\Psi^{l,i}$. Since our aim is to render every such base with a single graphic primitive (Sec. 3), instead of the initial tens up to hundreds of thousands of points, we can speak of an extreme simplification. The above bases $\Psi^{l,i}$ have relatively large supports. We define the *domain* $D^{l,i}$ of a basis function as the set of points where the basis function value exceeds a user-defined threshold τ:

$$D^{l,i} = \{x|\Psi^{l,i}(x) > \tau\}. \tag{1}$$

In practice, we set τ to approximately 0.05. This yields a multiscale surface decomposition into overlapping domains $D^{l,i}$. Since the coarsened matrix encodes surface flatness, the domains $D^{l,i}$ define regions of the input surface which are as quasi-flat as the surface's shape permits. For inherently curved surfaces, such as a ball, these domains will evidently become progressively less flat once one considers coarser levels. However, if permitted by the surface's shape, the decomposition correctly identifies flat surface components even at the coarsest level. In Figure 3b, we show a basis function located on a large flat surface region (the side face of the rocker-arm model). We can see that this basis function abruptly stops at the crease separating the model's side face from the upper face, as expected. In the flat area of the side face, the basis function decreases smoothly, since there is no curvature variation information. The overlapping of domains is clearly visible in Figure 3d, where each domain of the rocker-arm decomposition is colored with a distinct color (red, green, yellow, blue, or purple) different from its neighbor domains. Color mixing signals overlapping domains. In contrast, there is practically no such domain overlap for the octahedron (Fig. 3c). Here, every face corresponds to one single domain.

After decomposition, one level is chosen from the multiscale by the user. Finer levels deliver more domains of smaller size, which are thus implicitly closer to the quasi-flat requirement. Coarser levels may, especially for inherently curved objects, deliver domains which are far from the quasi-flat desiderate. Since

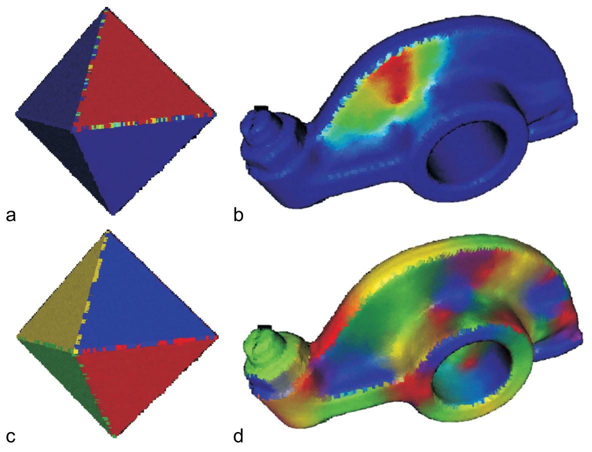

Figure 3: Basis function for the octahedron and rocker-arm model, using a blue-to-red colormap of the interval [0..1] (a,b). The domains are shown with distinct colors (c,d). Colors are mixed for overlapping domains.

we shall further use these domains to approximate our point set by a simplified rendering (Sec. 3), the level choice acts as a trade-off between performance and visual quality. Given our extreme simplification goal, we chose a level around $L-5$ in practice, where L is the coarsest decomposition level.

Summarizing, AMG can be considered to define a fuzzy clustering of the point set into quasi-flat domains: The bases $\Psi^{l,i}(x)$ define, on every level l, the degree of membership of every point x to every domain $D^{l,i}$. At points situated in clearly flat areas, such as the faces of the octahedron in Figure 3a, one basis function $\Psi^{l,i}$ will be close to unity, whereas all others $\Psi^{l,j}$, $j \neq i$, will be close to zero, as the sum of all bases at a point is always one (partition of unity). By thresholding (Eq. 1), we further decrease the number of bases acting upon a point to only those having non-negligible values. In this way, we further strengthen the partition of points into disjoint domains $D^{l,i}$. In areas of intermediate surface curvature (i.e. far from clear edges or flat zones), points will inherently be under the influence of several bases, i.e. the domains $D^{l,i}$ will overlap (Fig. 3d). Next, we map regions to graphical primitives (Sec. 3) and region overlap into a blending-based rendering algorithm (Sec. 4), in order to produce an image of our extremely simplified model.

3 Primitive Construction

The surface decomposition discussed in Section 2.3 delivers, on a given scale l, a set of basis functions $\Psi^{l,i}$ and associated quasi-flat domains $D^{l,i}$. We now construct a domain primitive for each domain D and basis function Ψ at the chosen scale. Since primitive construction is identical for every level l and domain and basis i, we now drop the indices l and i. When rendered together, domain primitives should convey an image close to the original point-based rendering. To maximize speed, we encode the information in D and Ψ in an efficient rendering combination: a support polygon P with a texture T. The complete process is illustrated in Figure 4. We first describe how the support polygon is constructed (Sec. 3.1). Next, the texture construction is detailed (Sec. 3.2).

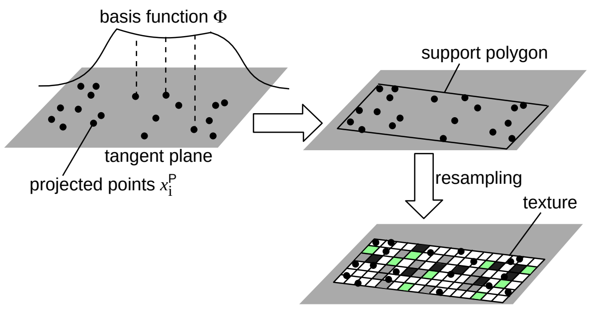

Figure 4: Primitive construction pipeline

3.1 Support Polygon

The support polygon P serves as planar approximation for the points $\{x_i\}$ contained in a (quasi-flat) domain D. The process of constructing P is depicted in Figure 5. To limit the geometric information loss produced by projecting the points of D onto a plane, we choose this plane to minimize the sum of squared distances to the points in D, using principal component analysis (PCA). This is similar to computing the local tangent planes (Sec. 2.1). However, here we use all points in a domain D, whereas local tangent planes used just small local neighborhoods. After projecting the points $\{x_i\}$ to $\{x_i^P\}$ on the tangent plane, a bounding polygon P is constructed. We compute P as a bounding rectangle, using the eigenvectors e_0 and e_1 of the PCA on D's points as the rectangle's main axes. While this does not deliver an optimal bounding rectangle, the result is only slightly sub-optimal in practice. We could construct a tighter fitting, more complex n-sided bounding polygon instead, e.g. using a convex hull algorithm. However, using rectangles for P is simple and efficient to implement, especially when performing texture mapping (Sec. 3.2).

3.2 Texture Construction

The support polygon P described in the previous section serves to carry a texture map T. This texture encodes two types of information extracted from the original point set: (unshaded) point colors, in the texture's color channels, and geometric (shape) information, in the texture's alpha channel, respectively. Point color information is simply transferred from the original 3D points $\{x_i\}$ to their 2D projections $\{x_i^P\}$. Next, the 2D projections get assigned transparency values equal to the basis function values $\Psi(x_i)$ at the original 3D locations x_i. Finally, from the set of 2D scattered points $\{x_i^P\}$, with color and transparency information, located inside the bounding rectangle P, we compute a texture T. This amounts to a resampling of the set $\{x_i^P\}$ on a regular grid of texels of user-specified resolution. For this, we have used two sets of basis functions: radial and linear, as follows.

Radial basis functions are 1 at the point sample and fall down to 0 radially. Different profiles are possible, such as constant, linear, or Gaussian. We set the fall-off radius proportional to the radius value available in every point $\{x_i\}$ of the point set [10]. By tuning this factor, as well as the profile, different degrees of color and transparency data smoothing can be achieved. Large fall-off radii generate smoother interpolations, but also overbright areas resulting from a violation of the partition of unity. In contrast, linear affine basis functions inherently enforce the partition of unity. We define such functions using a Delaunay triangulation [11] of the projected 2D point set $\{x_i^P\}$. For all texels inside the triangulation, we interpolate the color and transparency information using the linear basis functions. For texels outside the triangulation, but within a point's radius, i.e. texels close to the triangulation's boundary, we use radial basis functions. All other texels receive a default value of zero.

At this point, we have transferred the whole (simplified) point set information into a set of domain primitives consisting of textured polygons. Geometry is encoded both in the polygons' orientations as well as in the textures' alpha values – the latter encodes the object shape as captured by the basis functions. Color is naturally encoded in the texture color channels. Finally, if normal maps are supported by the graphics hardware at hand, point normal information can be stored in a similar texture, or normal map.

4 Primitive Rendering

In this section we will discuss how to render the domain primitives, thereby constructing a simplified view of the surface defined by the point set. The main idea here is to use the basis function information (Sec. 2.3), encoded as transparency (Sec. 3.2), to blend together the domain primitives into a smooth-looking surface. For shading we use either the domain primitive normals or normal maps when available. Blending the domain primitives requires special care. First, the interaction between blending and depth buffering must be taken care of. Second, the partition of unity property, i.e. the fact that basis functions sum to one at every point, holds only on the original 3D surface, but not on the projected 2D support polygons. These issues are explained next.

Depth buffering normally ensures that only the front-most fragment for each pixel is visible on the screen. However, blending requires that multiple overlapping surface fragments are combined per pixel. Simply disabling the depth test is erroneous in our case, as visible-surface determination is no longer performed then. Depending on the viewpoint, arbitrary basis func-

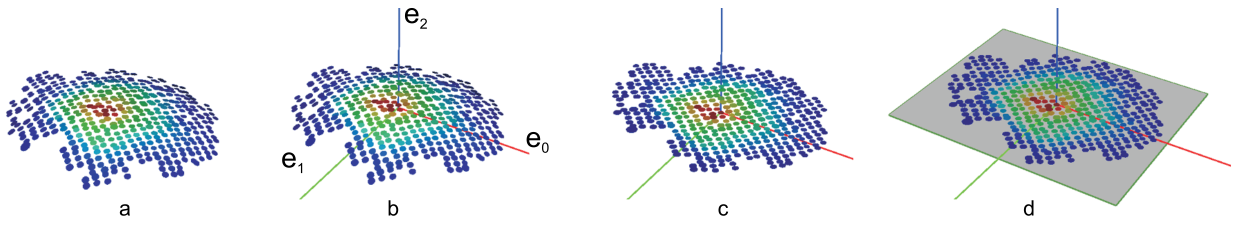

Figure 5: Domain points $\{x_i\}$ (a). PCA is performed to find the principal axes (b). Projected points $\{x_i^P\}$ on plane (e_0,e_1) (c). The range of projections of $\{x_i^P\}$ on e_0 and e_1 determines the bounding rectangle's size (d).

tion values from completely different parts of the surface may be projected to the same 2D screen area. Blending will sum up these values, whereas they would not be summed on the original 3D surface. The result is that surface parts that should normally be occluded are now blended with the occluding surface parts. We must thus solve the visibility problem differently. We make the observation that for each pixel only the *front-most* fragments should blend and be visible. By front-most fragments for a given pixel, we mean the fragments that overlap on the surface at its front-most intersection with a view ray cast from the viewpoint through the pixel. We call the correct set of fragments for a pixel the *prefix*, as it can be thought of as the prefix of a depth-sorted fragment list along the view ray (see Fig. 6).

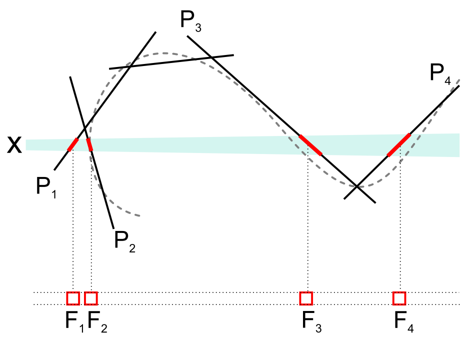

Figure 6: Simplified 2D view of a scene. The surface is indicated by the stippled curve, the domain primitives by straight lines. The fragment list for pixel x is indicated at the bottom. Its prefix consists of fragments F_1 and F_2 representing the surface's front-most intersection with the view ray through pixel x.

We stress that we want a simple rendering algorithm, without using programmable elements such as pixel shaders, so that simple graphics hardware suffices. Given this constraint, we must maintain our intermediate prefixes in the framebuffer during primitive rendering. Each incoming fragment must either additively blend with the current prefix or be discarded. After all domain primitives have been rendered, the prefix for each pixel must be complete, so that the framebuffer can be displayed. Hence, fragments must enter the graphics pipeline in a front-to-back order, as an arbitrary order would require sorting the fragments that make up the prefix. Since we cannot sort fragments explicitly, we sort the domain primitives by distance from the viewer to the primitive's center, so that their fragments enter the pipeline in a sorted manner. This holds, however, only when sorting is unambiguous, i.e. when domain primitives do not overlap in their Z extents. Ambiguous sorting causes an incorrect prefix, and thereby artifacts. Fortunately, in our case, this problem is diminished since an *exact* fragment ordering within the prefix is not important, as long as the prefix is correctly separated from the other fragments. In Figure 6 for example, the exact ordering of domain primitives P_1 and P_2 is not important, because the exact ordering of the fragments F_1 and F_2 does not matter.

We further note that basis functions are only allowed to be summed, and their domain primitives are only allowed to blend, when the domains overlap. When they do not overlap, summing them is not useful and may only lead to artifacts when they coincidentally project to the same screen area.

Combining all the above, we obtain the following algorithm (complete pseudocode is given in Figure 7). In each iteration we render the front-most domain primitive d that is not rendered yet, plus all the domain primitives whose domains overlap with d (called *domain neighbors*). We have now created a prefix for each pixel of d, because a) we render the primitives front-to-back and b) d was blended with all primitives it was allowed to, i.e. its neighbors. We now lock the pixels of d so that these prefixes cannot later be overwritten by other primitives. The prefixes of the neighbors' pixels (except d) are not locked, as they will be completed in later iterations. To lock the pixels, we use the depth buffer. Rendering a primitive d into the depth buffer effectively locks its pixels, as primitives of later iterations lie behind d, assuming an unambiguous primitive sort.

5 Results

We demonstrate the results of our approach for several models. We describe the cost of our domain primitive representation by the number of primitives needed and the number of texels in all their textures. The primitive count can be controlled indirectly by choosing the AMG level, as described in Section 2.3. The texel

```
$P \leftarrow$ list of sorted domain primitives
$R \leftarrow$ list of booleans initially false, of size of $P$
**for** $i$ from 1 to $length(P)$ **do**
  **if** $P_i$ is facing the camera **then**
    **if** not $R_i$ **then**
      render $P_i$ to color buffer with blending on
      $R_i \leftarrow \mathrm{true}$
    **end if**

    **for** each neighbor $P_j$ of $P_i$ **do**
      **if** not $R_j$ **then**
        render $P_j$ to color buffer with blending on
        $R_j \leftarrow \mathrm{true}$
      **end if**
    **end for**
    render $P_i$ to depth buffer
  **end if**
**end for**
```

Figure 7: The rendering algorithm

Model	rocker-arm	dinosaur	balljoint
#points	40k	56k	137k
AMG level	$L-5$	$L-6$	$L-5$
#polys	445	563	311
#texels	73k	74k	141k
sd time (s)	16	21	46
dpc time (s)	20	7	88

Table 1: Pre-processing times. L is the coarsest AMG scale for the particular model.

level	L-1	L-2	L-3	L-4	L-5	L-6
#polys	17	37	84	193	445	1012
128x128	2700	2010	1750	1300	590	220
256x256	940	730	670	600	420	220
512x512	270	218	204	190	155	130

Table 2: Framerates for the rocker-arm model. L is the coarsest AMG scale.

count can be controlled by choosing the texture resolution in the resampling step (Sec. 3.2).

Figure 8 shows images of three point-set models and their corresponding simplified rendering using domain primitives. Considering our extreme simplification goal, which often implies that these models are not meant for close-ups, we observe that important model features and its general structure are well captured even when the primitive count is considerably smaller than the original point count. Moreover, these models are not rendered with the extra surface detail normal maps provide, since our hardware did not support these.

The preprocessing timings for these models are shown in Table 1. All timings are measured on an Intel Pentium IV 2.4 GHz with 512 MB memory and a GeForce4 MX440. Preprocessing is divided into two steps. *sd* denotes the time needed to perform the surface decomposition (Sec. 2). This step has a computational complexity of $O(n)$, n being the number of point samples. *dpc* is the time needed to construct domain primitives (Sec. 3). This step takes $O(n \log n)$ for n domain points, given the Delaunay triangulation involved. Note, however, that the texture computation, involving resampling, is now entirely done in software. A simple and quick speed-up for the preprocessing can be easily gained if this step is directly performed in graphics hardware, for example by rendering the texture as a set of splats for the radial bases, or as a triangle mesh for the linear affine bases (Sec. 3).

Table 2 shows the rendering timings for the rocker-arm model, for seven different AMG levels and three screen sizes. Note that the amount of polygons is more or less doubled with each coarser scale (Sec. 2.3).

6 Conclusions

We have presented a new approach for creating extremely simplified representations of models, intended for rendering distant geometry. Our current implementation uses the more difficult case of a point set as an input. Point set surfaces pose extra challenges, as they do not allow a natural and direct definition of a finite element space upon them. Our method can further easily cope with triangular meshes, if this is desired (Sec. 2.3). Instead of using traditional point primitives, we introduce the domain primitive, a textured splat, which is better suited for representing the surface when using only a few primitives. By using color and transparency information stored as textures, domain primitives have more surface approximation power and are better able to capture shape and color variation than the same amount of other known primitives in point-based rendering. Domain primitives become most efficient and effective in terms of rendering performance and quality respectively when applied with the goal of displaying extremely simplified models.

Overall, the only user parameters of the complete pipeline are the classifier neighborhood size (Sec. 2.1), AMG scale (Sec. 2.3), and texel size (Sec. 3.2). We render our simplified model by a custom algorithm, as per-primitive blending requires a different visible-surface determination technique than standard depth buffering. The proposed algorithm uses only standard OpenGL 1.1 graphics hardware. Summarizing, we can consider the proposed domain primitive as a *multiscale generalization* of point primitives. Indeed, on the finest AMG scale, every point has exactly one domain primitive, which makes the two notions identical. On coarser scales, primitives adapt their shape to the surface shape. Moreover, primitives are rendered and blended using exactly the same mechanisms as in standard point-based rendering.

Some artifacts may be visible in the simplified renderings. The projection error is an inherent problem that is caused by approximation of the surface by

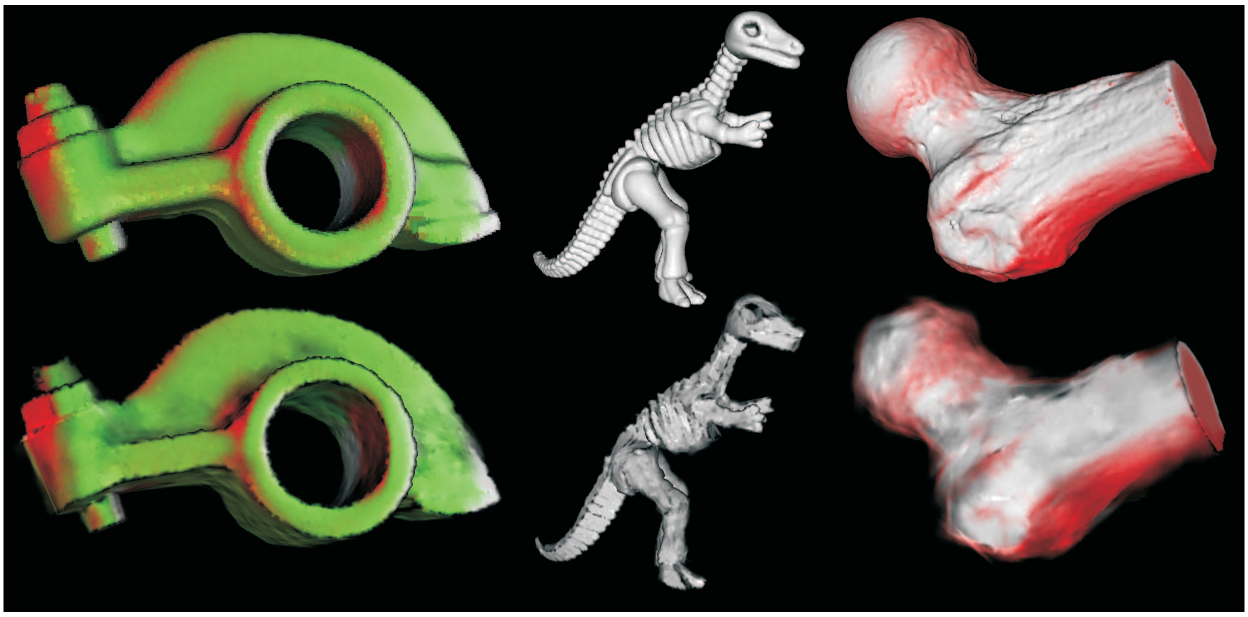

Figure 8: Screenshots of the rocker-arm, dinosaur, and balljoint models. For each model, the point-set rendering is shown right above the extreme simplification rendering.

blending flat primitives. Dark spots occur when the projection error is at its largest. Cracks may be visible at strong discontinuities, where flat domain primitives do not overlap. Interestingly, similar cracks are also visible in the extreme simplification method for triangular meshes proposed by Décoret et al. [5], as this method also approximates curved surfaces with flat textured primitives. A main difference between the above method and ours is that we blend primitives together using a continuous transparency signal determined by our basis function decomposition, whereas the above method uses transparency as a stencil mask, i.e. to turn on and off texture pixels. This often causes our cracks to be less visible.

Several directions of future research are envisaged. First, programmable graphics hardware can be used to enhance the flexibility, thus remove several artifacts, of the rendering algorithm used to combine the domain primitives. Second, one could try to combine several levels of the multiscale generated by the AMG to render primitives of different sizes and levels of detail together. Third, a challenging point, as with many visualizations, is to define a meaningful and computable error metric for measuring the visual quality of our simplifications. Finally, combining points and domain primitives in a hybrid rendering can open new ways to low primitive count, high quality rendering of 3D models.

References

[1] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, C.T. Silva, Point set surfaces, *Proc. of IEEE Visualization* 2001, pp 21-28

[2] U. Clarenz, M. Rumpf, A. Telea, Surface processing methods for point sets using finite elements, *Computers & Graphics* 28(6):851-868, 2004

[3] U. Clarenz, M. Rumpf, A. Telea, Robust feature detection and local classification for surfaces based on moment analysis, *IEEE TVCG*, 10(5):516-524, 2004

[4] U. Clarenz, M. Griebel, M. Rumpf, M.A. Schweitzer, A. Telea, Feature sensitive multiscale editing on surfaces, *The Visual Computer*, 20(5):329-343, Springer, 2004

[5] X. Décoret, F. Durand, F. Sillion, J. Dorsey, Billboard clouds for extreme model simplification, *Proc. of the ACM SIGGRAPH* 2003, pp 689-696

[6] M. Levoy, T. Whitted, The use of points as display primitives. Technical Report TR 85-022, Univ. of North Carolina at Chapel Hill, 1985

[7] A. Kalaiah, A. Varshney, Modeling and rendering points with local geometry, *IEEE TVCG*, 9(1):30-42, 2003

[8] M. Pauly, M. Gross, L. Kobbelt, Efficient simplification of point-sampled surfaces, *Proc. of IEEE Visualization* 2002, pp 163-170

[9] H. Pfister, M. Zwicker, J. van Baar, M. Gross, Surfels: surface elements as rendering primitives, *Proc. of ACM SIGGRAPH* 2000, pp 335-342

[10] S. Rusinkiewicz, M. Levoy, QSplat: a multiresolution point rendering system for large meshes, *Proc. of ACM SIGGRAPH* 2000, pp 343-352

[11] J.R. Shewchuk, Triangle: engineering a 2d quality mesh generator and delaunay triangulator, 1^{st} *Workshop of Applied Computational Geometry*, pp 124-133, ACM Press, 1996

[12] L. Westover, Footprint evaluation for volume rendering, *SIGGRAPH Computer Graphics*, 24(4):367-376, 1990