Comput Visual Sci
DOI 10.1007/s00791-006-0043-8

REGULAR ARTICLE

Extreme simplification and rendering of point sets using algebraic multigrid

Dennie Reniers · Alexandru Telea

Received: 6 October 2005 / Accepted: 13 June 2006

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, this requires using many primitives to render even moderately simple shapes. Often, one wishes to render a simplified model using only a few primitives, thus trading accuracy for simplicity. For this goal, we propose a more complex primitive, called a *splat*, that is able to approximate larger and more complex surface areas than oriented discs. We construct our primitive by decomposing the model into quasi-flat regions, using an efficient algebraic multigrid algorithm. Next, we encode these regions into splats implemented as 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 method on various models.

Keywords Point set models · Algebraic multigrid · Extreme model simplification · Real-time rendering

Communicated by G.Wittum.

D. Reniers (✉) · A. Telea
Department of Mathematics and Computer Science,
Eindhoven University of Technology, Den Dolech 2,
5600 MB, Eindhoven, The Netherlands,
e-mail: D.Reniers@tue.nl

A.Telea
e-mail: A.C.Telea@tue.nl

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 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.

The representation accuracy, and ultimately the rendering quality of point models is usually achieved by using simple point primitives that approximate small surface areas. Simple point primitives, such as flat discs, have just a few parameters: position, radius, color, and normal. This makes them fast to render and efficient to store. The drawback of this simplicity is that a point can accurately describe only a small surface area. Very dense samplings are needed to obtain a good rendering quality, even for relatively simple models, resulting in slower renderings than necessary.

Many applications, such as interactive and level-of-detail rendering, data transmission, and object simplification and matching, need to 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 with more than two orders of magnitude, we speak of extreme model simplification [10]. In case of simplification of point set

models, two options are available, as follows. First, one can use a lower amount of the same kind of simple point primitives as for the original finely sampled models. The multiresolution point rendering system QSplat [18] is a good example of such a system. The simplification quality may considerably decrease, given the limited approximation power of simple primitives such as oriented discs. Nonuniform sampling techniques [15] may alleviate this problem by distributing more points of smaller size in areas of high variations of the model's shape and/or color and less larger points in flat, single-color areas. However, the extreme simplification requirement of reducing the primitive count from hundreds of thousands to several hundred may lead to poor quality when using such simple primitives.

A second option in the simplification is to store more information per primitive. This yields primitives with a higher approximation power for the model's geometry and/or color information, thus less primitives needed for the same simplification quality. For example, differential points [12] store local differential geometric information. The accompanying simplification algorithm delivers a sparser point set whose local sampling density reflects the local surface variation. Surfels [16] store pre-filtered texture, bump, and displacement maps in a multiresolution hierarchy and reduces the model sampling rate to match the output screen resolution.

Nevertheless, the above point primitives and multiresolution 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 variability of larger surface areas can no longer be accurately captured by simple parametric models or radial basis functions. For example, the six sides of a cube model could be optimally captured by six square-shaped primitives, one for each face. However, even if we availed of square-shaped primitives, six primitives would not suffice in case of fine-scale color detail on the faces. In this case, many point primitives are needed, as points usually carry a single color per primitive. In contrast, the classical polygon-based approach would effectively represent the cube using six textured polygon primitives.

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. On the one hand, our approach inherits the well-known mesh-less advantages inherent to point sets. On the other hand, we use textures to increase the primitive's geometric and color approximation power, hence delivering considerably less primitives than in standard point-based rendering. 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 classical point-based rendering. Coarse scale levels address the issue of extreme model simplification.

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 hundreds of mesh-less primitives". Figure 1 illustrates the above. 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 Fig. 1d. The advantage of our approach is clear; we render eight hardware-accelerated splats, i.e. eight 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 [10]. Both methods yield a mesh-less polygon representation. However, the simplification heuristics, the implementation, obtained performance

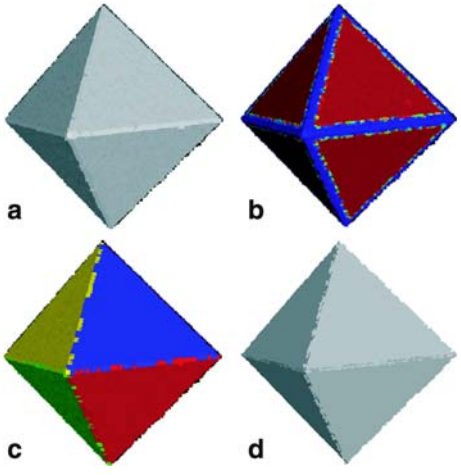

Fig. 1 Global overview of our approach. An octahedron model of 16,000 points is rendered using point primitives (**a**). The color-coded surface classifier (**b**). The surface is decomposed into eight domains, indicated by different colors (**c**). The model is rendered using eight splats, or domain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

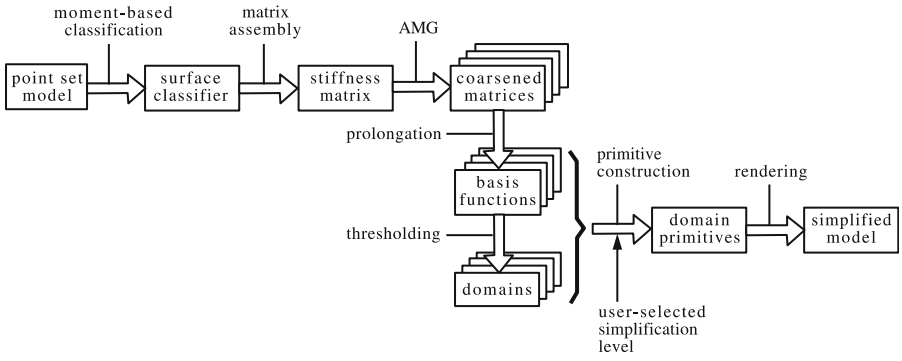

Fig. 2 Steps of the extreme simplification and rendering of point sets

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 Fig. 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 (Sect. 2.1) which is encoded into a finite element matrix (Sect. 2.2). The surface decomposition algorithm, based on an algebraic multigrid (AMG) method, is detailed in Sect. 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 splat, or *domain primitive*, for each domain and associated basis function. The domain primitives encode geometric and color information carried by each region's point samples in the alpha, respectively color planes of a texture (Sect. 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 (Sect. 4). The results of our method are discussed in Sect. 5. Section 6 concludes the paper and presents future work directions.

2 Surface decomposition

Recalling our goal, we want to quickly render a 3D surface using just a few mesh-less primitives. As sketched in Sect. 1, in a primitive-based (in contrast with image-based) approach, 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 Sect. 3. The complete pipeline is summarized in Fig. 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 (Sect. 2.1). Next, we encode this surface classifier in a finite element matrix (Sect. 2.2). Finally, we use AMG to produce a multiscale coarsening of this matrix (Sect. 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 aim, we take into account the points n_i in a small 3D spatial neighborhood N of x. N is chosen such that (a) it contains a minimal number of points for stable classifier computation and (b) the radius of N is not larger than the size of the features we want to be visible in the approximation. For the first requirement, we efficiently compute N as the k nearest neighbors of x, for given k. For the second requirement, we define N as the ball $B_\epsilon(x)$ of given radius ϵ centered at x. In practice, we combine the above, i.e. prescribe a minimum number of neighbors $k_{\min}$, to enforce the first requirement. If the $k^{th}_{\min}$ closest neighbor of x is closer to x than the prescribed minimal feature size ϵ, we consider all additional nearest neighbors in $B_\epsilon(x)$.

Given the above, we use a surface classifier based on the *zero and first moments of N*. This classifier is described in great detail in [3], so here we limit ourselves to a concise presentation thereof. Moments allow us both to distinguish between smooth and non-smooth surface parts (classification) and also to stably compute local tangent planes to the surface. We use this latter feature when assembling our surface classification matrix (Sect. 2.2).

For a continuous surface $\mathcal{M}$, the zero moment is given by the local barycenter of $\mathcal{M}$ with respect to an Euclidean ball $B_\epsilon(x)$ centered at x:

$$M^0_\epsilon(x) = M^0_\epsilon := \fint_{B_\epsilon \cap \mathcal{M}} x \, \mathrm{d}x. \tag{1}$$

The first order moment is then defined as:

$$\begin{aligned} M^1_\epsilon(x) &:= \fint_{B_\epsilon \cap \mathcal{M}} (x - M^0_\epsilon) \otimes (x - M^0_\epsilon) \, \mathrm{d}x \\ &= \fint_{B_\epsilon \cap \mathcal{M}} (x \otimes x \; - M^0_\epsilon \otimes M^0_\epsilon) \; \mathrm{d}x \,, \end{aligned} \tag{2}$$

where $y \otimes z := (y_i z_j)_{i,j=1,\ldots,3}$. The first moment approximates the matrix $\Pi_{\mathcal{T}_x \mathcal{M}}$ that describes the projection onto the surface local tangent space $\mathcal{T}_x \mathcal{M}$ [4]. If $\lambda_0 > \lambda_1 > \lambda_2$ are the eigenvalues of the 3×3 symmetric matrix M^1_ϵ, then the corresponding eigenvector e_2 is the normal of the approximate tangent plane, whereas e_1 and e_0 form a 2D coordinate system in the plane itself. Figure 3a illustrates the above in two dimensions. The zero and first moments (Eqs. 1 and 2) are computed numerically as sums over the sample points. This is very similar to the principal component analysis, or so-called 'surface variation', used in [1,14,21]. Essentially, the radius ϵ in both our moment-based and the surface variation approaches has the role of a filter size: The tangent plane ignores, or filters out, features significantly smaller than ϵ.

Next, we define our surface classifier $\mathcal{C}_\epsilon$ as:

$$\mathcal{C}_\epsilon = G\left(\frac{||M^0_\epsilon(x) - x|| \lambda_2(M^1_\epsilon(x))}{\epsilon \, \lambda_0(M^1_\epsilon(x))} \right), \tag{3}$$

with $G(s) = (\alpha + \beta \, s^2)^{-1}$ with suitably chosen $\alpha, \; \beta > 0$. In all our applications, we have fixed $\alpha = 0.01$ and $\beta = 100$. The function G causes $\mathcal{C}_\epsilon$ to be close to 1 in relatively smooth surface areas and one up to several orders of magnitude smaller close to edges or cusps, thus making surface classification easier. We used a similar classifier for processing of point surfaces [3,6]. Moreover, we used an analogous classifier, though defined on triangular mesh surfaces instead of point sets, to decompose these surfaces into quasi-flat areas. [5].

Figure 1b shows the surface classifier on a point set model, for different values of k closest points. Smooth surface regions appear red, whereas highly non-smooth ones (e.g. creases) appear blue. The value of k, essentially correlated with ϵ, clearly acts as a filter that removes small-scale surface noise.

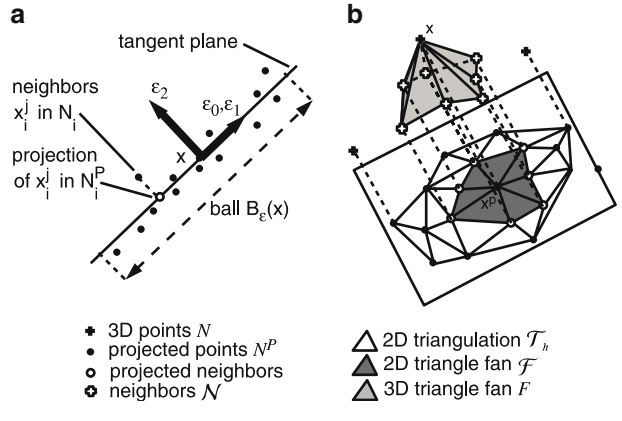

Fig. 3 Tangent plane (**a**) and local triangulation (**b**)

2.2 Matrix encoding of surface classifier

Our goal is to use the local surface classifier introduced in Sect. 2.1 to decompose the surface into quasi-flat components. For example, we would like to decompose the octahedron shown in Fig. 1a into eight regions corresponding to its faces (Fig. 1c). We shall use for this decomposition an algebraic multigrid (AMG) approach (Sect. 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 this, we use a finite element model, as follows. In case our surface discretization were a triangular mesh, we could directly compute A, the discrete matrix form of the operator $\mathcal{A}$:

$$A_{ij} = \sum_l \mathcal{C}(T_l) \, \nabla_{T_l} \Phi_i \cdot \nabla_{T_l} \Phi_j \, |T_l|, \tag{4}$$

where Φ_i are the usual linear affine basis functions defined on the mesh triangles, $\Phi_k(x^j_i) = \delta_{kj}$, for all mesh nodes k and j. Here $\{T_l\}_l$ denotes the triangle fan around every mesh node, ∇_{T_l} the gradient of the affine triangle T_l, $\mathcal{C}(T_l)$ the classifier value for triangle T_l computed by averaging the nodal classifier values (Eq. 3), and $|T_l|$ the area of this triangle respectively. Assembling A is done in the usual manner, i.e. by iterating over all mesh triangles, next over all pairings of local nodal basis functions, followed by updating the entries A_{ij} corresponding to these pairs. The above procedure is described in full detail for triangle meshes in [5].

However, in our case we avail of 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. Essentially, we use the finite element model for point sets described in [3], which we briefly outline in the following for the sake of clarity and completeness. As described in Sect. 2.1, we compute a tangent plane $\{x \in \mathbb{R}^3 \mid e_2 \cdot (x - x_i) = 0\}$ to every point x_i of our point set using the eigenvector e_2 of the second moment's vanishing eigenvalue. Next, we project all neighbors x_i^j in the neighbor set N_i of x_i onto it. To simplify notation, we add x_i as x_i^0 in N_i. We obtain the projected neighbor set $N_i^p = \{X_i^j\}_{j=0,\cdots k}$ in the tangent plane (e_0,e_1) (Fig. 3a). Next, we compute the Delaunay triangulation $\mathcal{T}_i$ of the projected points N_i^p, using an efficient and robust software package [20]. Note that this is a 2D triangulation taking place in the tangent plane. From $\mathcal{T}_i$, we select the *triangle fan* $\mathcal{F}_i^p = \{T_i^p\}_i$ of projected triangles T_i^p around the projected seed point X_i^0 (Fig. 3b). This triangle fan allows us to define the neighbor set $\mathcal{N}_i^p$ of x_i as being the points x whose projections X_i^j are used in the triangle fan $\mathcal{F}_i^p$. Finally, we denote by $\mathcal{N}_i$ the 3D points that correspond, via projection, to $\mathcal{N}_i^p$.

The above scheme allows us to define a *local* tangent finite element space at every point x_i in the point set. We proceed now to assemble the preliminary matrix entry:

$$\tilde{A}_{ij} = \sum_l \mathcal{C}(T_l)\, \nabla_{T_l}\Phi_i \cdot \nabla_{T_l}\Phi_j \, |T_l|, \tag{5}$$

similarly to Eq. 4, but selecting the triangle T_l from the local triangle fans $\mathcal{F}_i = \{T_l\}_l$. Integration takes place on the 3D triangle fan $\mathcal{F}_i$, and not on its 2D projection $\mathcal{F}_i^p$, since we want to encode a strong coupling between points in quasi-flat regions and weak coupling between points separated e.g. by a crease.

$\tilde{A}_{ij}$ 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:

$$A_{ij} = \frac{1}{2}(\tilde{A}_{ij} + \tilde{A}_{ji}), \tag{6}$$

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

$$A_{ii} = -\sum_{x_j \in \mathcal{N}(x_i)} A_{ij}. \tag{7}$$

The matrix A has now the same properties as a classical stiffness matrix defined e.g. on a triangulation mesh (Eq. 4). 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. Among others, A is a sparse matrix, as the average size of N_i is under ten neighbors per point. This matrix is the input of our surface decomposition, detailed in Sect. 2.3. We next comment a few issues on the proposed matrix construction scheme, for the full details referring to [3].

The local triangle fan scheme is robust, for two main reasons. First, we use a relatively large k-closest point set of 50..100 points. This makes the moment-based evaluation of the tangent plane stable [4]. Moreover, the tangent plane need not be very accurate, since we use it only as a support for performing the Delaunay triangulation. Next, we select just a small subset $\mathcal{F}_i^p$ of 5..10 triangles from this triangulation, which makes the neighbor $\mathcal{N}_i$ computation very robust. Practically, the above mean that small variations of the tangent plane orientation, caused by the moment evaluation, will not change a point's neighbor set [3]. Finally, the symmetrization phase (Eqs. 6 and 7), needed in those relatively unfrequent cases when the neighbor relation is not symmetric, can be seen as a filter acting on the encoded surface classifier. Even though this may smooth out individual classifier values, we shall further use just a coarsened (simplified) version of the matrix A (Sect. 2.3), so the symmetrization phase has negligible side-effects.

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, $R^l := (P^l)^T$. Figure 4 shows the general working of the AMG algorithm.

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 $\{P^l\}_{l=0,\ldots,L}$ is equivalent to con-

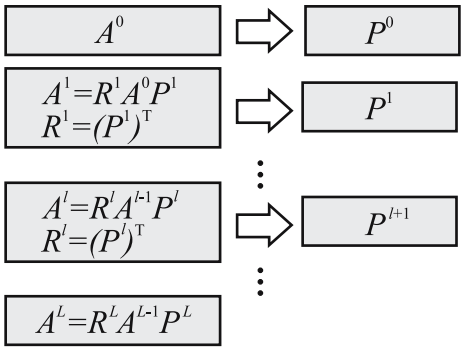

Fig. 4 Overview of the AMG algorithm

structing a set of progressively coarser, problem-dependent bases $\{\Psi^{l,i}\}_{l=0,\ldots,L}$. On every level, these bases are aligned with the strong matrix couplings present on that level. This is done using the notion of *algebraic smoothness* [2,17], based on the general observation that a simple Gauss–Seidel relaxation scheme damps components in the direction of matrix eigenvectors associated with large eigenvalues. Consequently, the coarse bases $\Psi^{l,i}$ are chosen such that they deal with the remaining components of an eigenvector decomposition. Since AMG uses just the previous level $l-1$ when constructing the prolongation P^l, it works efficiently even for very large matrices A^0: Its complexity is $O(n)$, where n is the number of rows of the initial matrix A^0.

Although the theory and design of AMG are rather involved, it can be used as a 'black box' for many applications requiring coarsening of a matrix that encodes some fine-scale, problem-dependent coupling information. Examples of previous applications are multiscale image segmentation [19], large graph layouts [13], vector field clustering [9], and multiscale surface decomposition [5]. In particular, we use here the same surface classifier, AMG implementation, and parameter settings as for the surface decomposition method presented in [5]. The main difference is that here we build our stiffness matrix using finite elements defined on point-based surfaces, instead of classical triangle meshes. Finally, we mention that any AMG tool (see e.g. *http://www.mgnet.org*) can be used for our matrix coarsening. Specifically, we use exactly the same AMG implementation which was used in [5] and [9], which is described in further detail in [7,8]. Our AMG implementation, in contrast e.g. to [17], treats robustly only symmetric, diagonally dominant matrices, which is the main technical reason for which we applied the symmetrization phase described earlier in Sect. 2.1.

The number of basis functions between successive scales is reduced by a factor between 2 and 3 approximately. This is inherent to our AMG implementation. For typical point set models, we obtain thus between 10 and 15 decomposition levels L. Moreover, the coarsest levels $L \cdots L-5$ usually contain between under 10 and up to a few hundred bases $\Psi^{l,i}$. Since our aim is to render every such base with a single graphic primitive (Sect. 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:

$$D^{l,i} = \{x | \Psi^{l,i}(x) > \tau\}. \tag{8}$$

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 on the coarsest level. In Fig. 5b, 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 well visible in Fig. 5d, 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. 5c). Here, every face corresponds to one single domain.

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

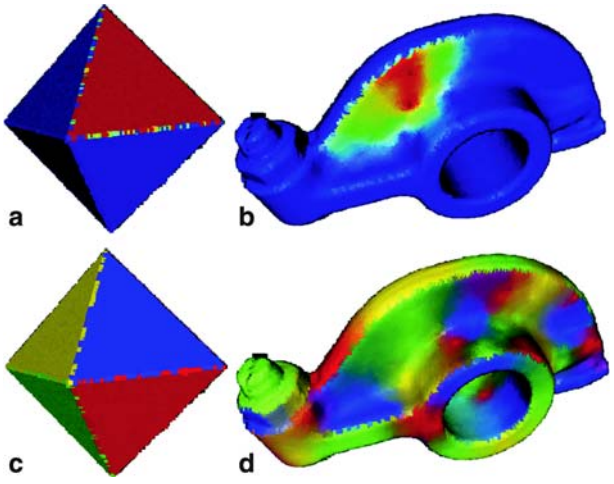

Fig. 5 Basis function for the octahedron and rocker-arm model, using a blue-to-red color map of the interval [0..1] (**a**, **b**). The domains are shown with distinct colors (**c**, **d**). Colors are mixed for overlapping domains

points situated in clearly flat areas, such as the faces of the octahedron in Fig. 5a, 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. 8), 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 be inherently under the influence of several bases, i.e. the domains $D^{l,i}$ will overlap (Fig. 5d). Next, we map regions to graphical primitives (Sect. 3) and region overlap into a blending-based rendering algorithm (Sect. 4), in order to produce an image of our extremely simplified model.

3 Primitive construction

The surface decomposition discussed in Sect. 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

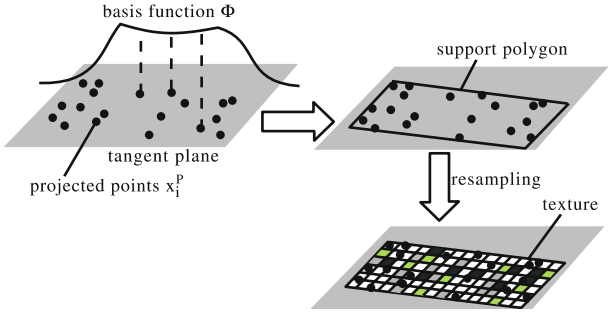

Fig. 6 Primitive construction pipeline

identical for every level l and domain and basis i, we drop now 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 Fig. 6. We first describe how the support polygon is constructed (Sect. 3.1). Next, the texture construction is detailed (Sect. 3.2).

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 Fig. 7. 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 (Sect. 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

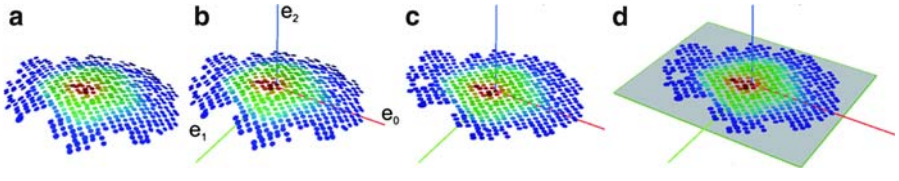

Fig. 7 Domain points $\{x_i\}$ (**a**). PCA is performed to find the principal axes (**b**). Projected points $\{x_i^P\}$ on plane (e_0,e_1) (**c**). The range of projections of $\{x_i^P\}$ on e_0 and e_1 determines the bounding rectangle's size (**d**)

and efficient to implement, especially when performing texture mapping (Sect. 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: 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 [18]. 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 over-bright 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 of the projected 2D point set $\{x_i^P\}$. Triangles created for domain concavities must be removed, because they do not belong to the domain surface (Fig. 8). These triangles can be identified by having at least one edge which length is more than the sum of the two incident vertices' radii. 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.

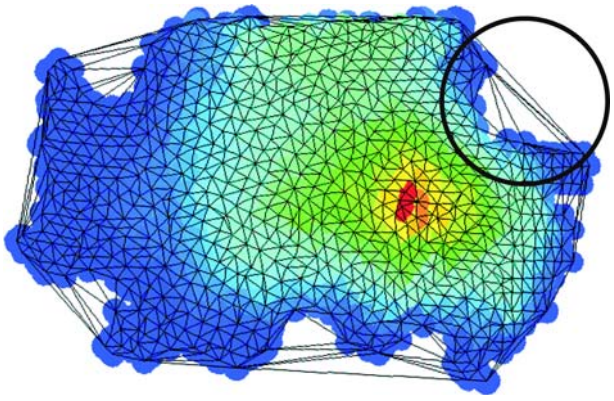

Fig. 8 Domain triangulation. A domain concavity is *encircled*. These triangles are removed from the final result

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. However, using normal maps during rendering would require programmable graphics hardware, whereas one of our requirements was to have a minimal method which requires only the standard, fixed graphics pipeline of OpenGL 1.1 to be used. This makes our method applicable on low-end graphics hardware, which makes sense if we think this is precisely the type of hardware on which one would want to render highly simplified 3D models.

4 Primitive rendering

In this section we will discuss how to render the domain primitives, thereby reconstructing a simplified view of the surface defined by the point set. The main idea here is to use the basis function information (Sect. 2.3), encoded as transparency (Sect. 3.2), to blend together the domain primitives into a smooth-looking surface. However, 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 overall to one, 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 function values from completely different parts of the surface may be projected to the same 2D screen area. Blending

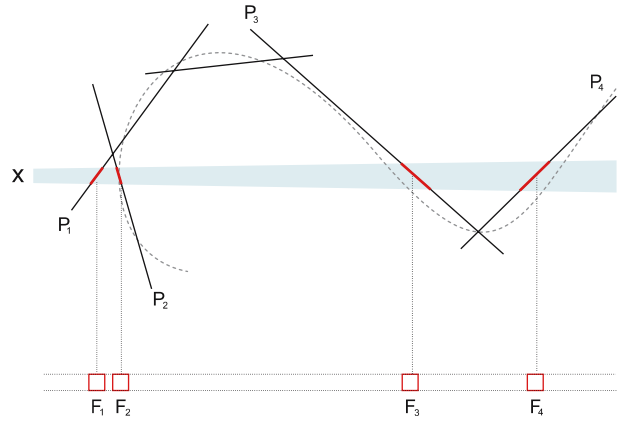

Fig. 9 Simplified 2D view of a scene. The surface is indicated by the *stippled curve*, the domain primitives by *straight lines*. The fragment list for pixel x is indicated at the *bottom*. Its prefix consists of fragments F_1 and F_2 representing the surface's front-most intersection with the view ray through pixel x

```
L ← list of sorted domain primitives
R ← list of booleans initially false, of size of L
for i from 1 to length(L) do
  if L_i is facing the camera then
    if not R_i then
      render L_i to color buffer with blending on
      R_i ← true
    end if

    for each neighbor L_j of L_i do
      if not R_j then
        render L_j to color buffer with blending on
        R_j ← true
      end if
    end for
    render L_i to depth buffer
  end if
end for
```

Fig. 10 The rendering algorithm

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. 9).

Ideally, we could proceed as follows for finding these prefixes: For each pixel x, cast a ray from x and call p its intersection with the original surface. Next, additively blend the primitives P_i whose domains D_i contain p and only allow pixel x to be altered. However, as explained previously, our rendering model operates primitive-based and not image-based. Furthermore, we do not wish to use the original 3D surface in our rendering. Given this, we now present an object-based approximation to the above rendering algorithm.

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, as e.g. for the painter's algorithm [11]. 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 Fig. 9 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 Fig. 10). 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 be overwritten later by other primitives. The prefixes of the neighbors' pixels (except

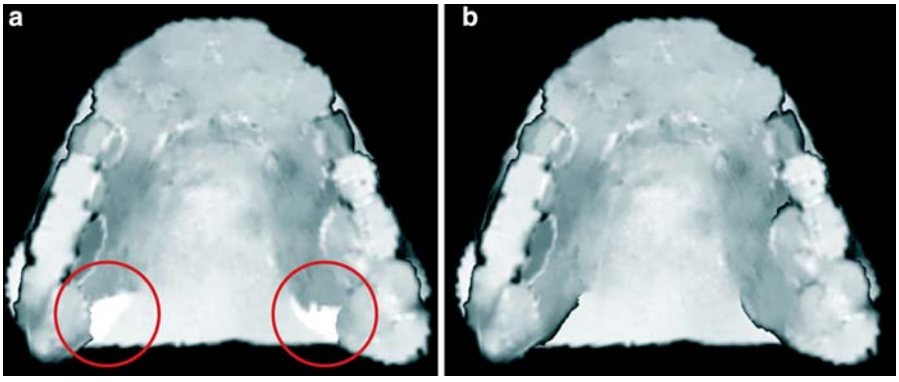

Fig. 11 If two domains are considered neighbors even when they only share one point, artifacts may occur

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.

Two (overlapping) domains are considered neighbors if the fraction $\frac{\#common\ points}{\#points}$ for at least one of the two domains exceeds a given threshold. In practice, the threshold 0.05 performs well for all models we tested. The above threshold prevents domains that overlap too slightly (e.g. just over a few points), to be considered neighbors and thus blended. In Fig. 11a two artifacts, visible as bright spots, are encircled that result from incorrect neighbor computation. These bright parts are caused by the incorrect blending of a 'molar' domain primitive and a 'palate' domain primitive. When the required number of common domain points is raised, the artifacts disappear as can be seen in Fig. 11b.

The above domain primitive neighbor information may be regarded as a coarse form of topological mesh structure, so one may argue that our extreme simplification model, although based on overlapping domain primitives, is not entirely mesh-less. However, looking at a rendering of the support polygons themselves, we can see these are far from forming a consistent mesh, so we cannot consider our primitive set as being just a mesh-based model simplification (Fig. 12).

We have also experimented with two other rendering algorithms that do not need neighbor information. The first one blends all fragments within a uniform distance from the front-most fragment, using a two-pass method similar to the one described in [18]. The second one uses back-facing polygons to divide the fragment list into groups. Fragments are allowed to blend only within a group. Both these algorithms deliver poorer image-quality than the algorithm using neighbor information. Concluding, while our domain primitive model does not necessarily need topological information, the results are better if we use it for rendering.

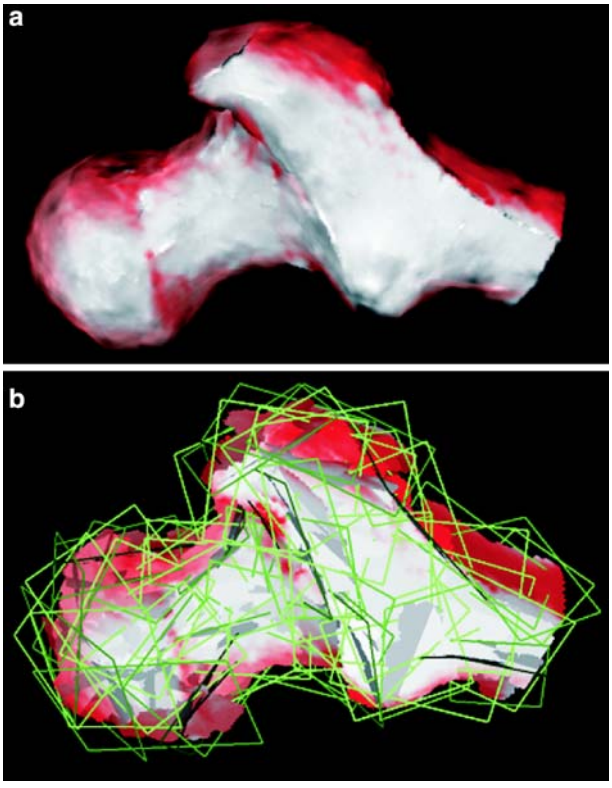

Fig. 12 Image **b**, in which the domain primitives are rendered opaque and with a *green outline*, shows that our simplification (**a**) is mesh-less

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 Sect. 2.3. The texel count can be controlled by choosing the texture resolution in the resampling step (Sect. 3.2).

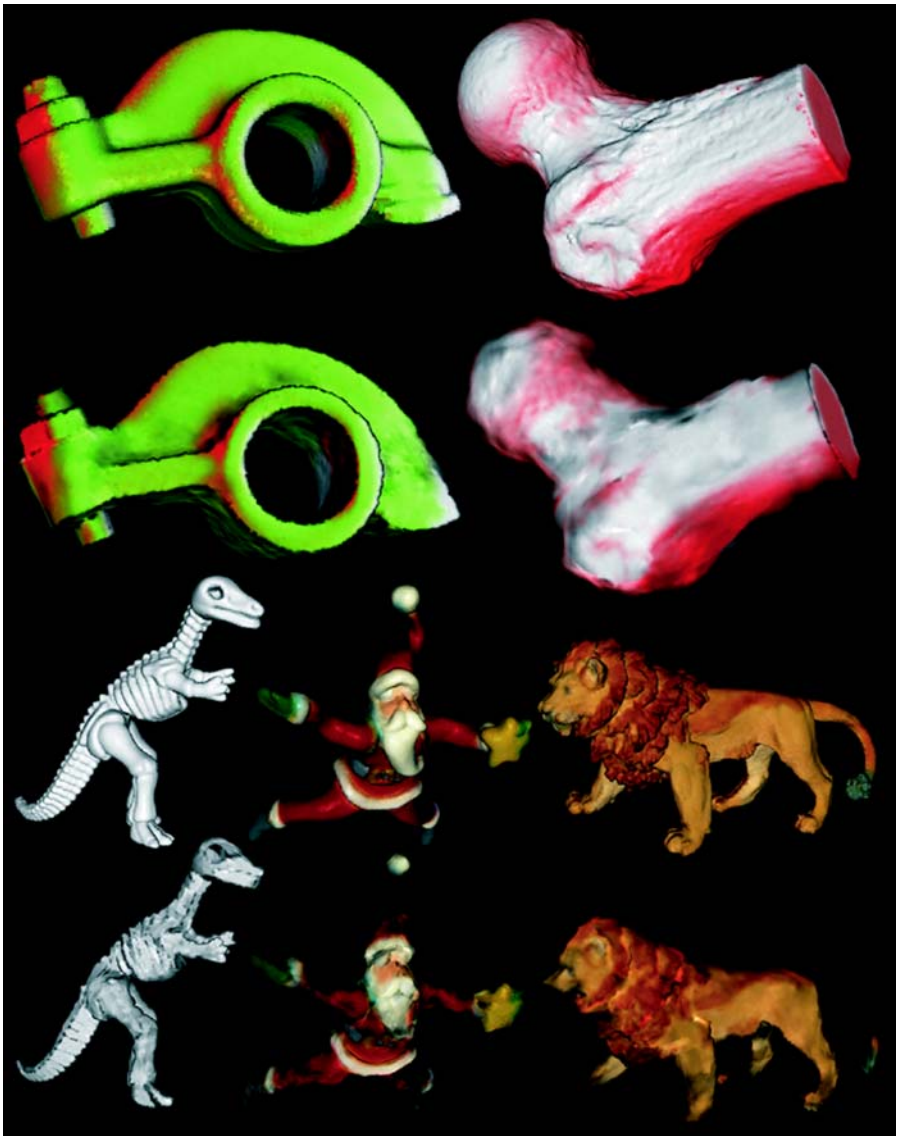

Fig. 13 Screenshots of the rocker-arm, balljoint, dinosaur, santa, and lion models. For each model, the point-set rendering is shown right above the extreme simplification rendering

Figure 13 shows images of several 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 [10], 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 normals 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 in two steps. *sd* denotes the time needed to perform the surface decomposition (Sect. 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 (Sect. 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

Table 1 Pre-processing times

Model	No. of points	AMG level	No. of polys	No. of texels	sd time (s)	dpc time (s)
Rockerarm	40k	*L*-5	445	73k	16	20
Dinosaur	56k	*L*-6	563	74k	21	7
Santa	50k	*L*-5	364	60k	24	16
Balljoint	137k	*L*-5	311	141k	46	88
Lion	180k	*L*-4	160	111k	54	134

L is the coarsest AMG scale for the particular model

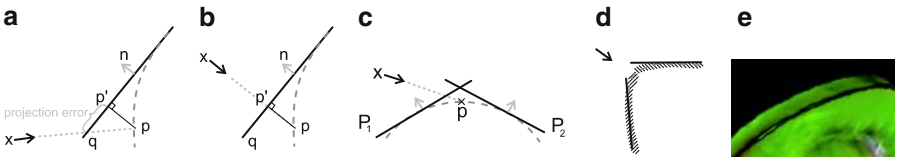

Fig. 14 Rendering artifacts. The projection error (**a**, **b**), dark spots (**c**), and cracks (**d**, **e**)

Table 2 Framerates for the rocker-arm model

Level	*L*-0	*L*-1	*L*-2	*L*-3	*L*-4	*L*-5	*L*-6
No. of polys	5	17	37	84	193	445	1,012
128 × 128	3,200	2,700	2,010	1,750	1,300	590	220
256 × 256	1,100	940	730	670	600	420	220
512 × 512	305	270	218	204	190	155	130

L is the coarsest AMG scale

for the preprocessing can be easily gained if this step is directly performed in graphics hardware by rendering the texture as a set of splats, for the radial bases, and triangle mesh, for the linear affine bases, respectively (Sect. 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 (Sect. 2.3).

Several rendering artifacts may occur, depending on the model, as follows. First, a projection error appears since we approximate curved domains by flat domain primitives. Consider Fig. 14a. Let p be the intersection for pixel x with the surface represented by the dotted curve. The color and basis function values of point p are captured in the color and alpha channels at the perpendicular projection of p onto the domain primitive, namely at p'. However, when the domain primitive is rendered, x will be painted with the fragment at q, not p'. The error increases when the angle between the viewing direction (indicated by the arrow) and the domain primitive's normal n increases. There is no error when the viewing is perpendicular to the domain primitive (Fig. 14b). As a result of the projection error, we might sum two fragments that do not represent the same part of the surface. This may be visible as a slightly too dark, too bright, or incorrect hue, spot on the surface. When the projection error is at its maximum, i.e. when the viewing direction is perpendicular to the surface normal, dark spots may occur. Consider Fig. 14c. Domain primitive P_2 that represents p is not rendered because it is backfacing the viewing direction. The pixel x is thus darker than it should be. Because this effect is greatest when the viewing direction is perpendicular to the surface normal, it is most noticeable at the contours of a model (see e.g. Fig. 12a).

A second problem is that of cracks, which occur at strong discontinuities, or ridges, of the surface. Such ridges represent areas of low coupling in the stiffness matrix (Sect. 2.2). Consequently, the AMG decomposition creates two disjunct domains for both discontinuity sides. The domain primitives separated by the ridge may not always intersect (Fig. 14d). Depending on the viewpoint, a crack is visible in the rendering, as shown in Fig. 14e for the rocker-arm model. We investigate solving this problem by adding simple point primitives in the rendering for the discontinuities, yielding a hybrid point and domain primitive approach.

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 too, if this is desired (Sect. 2.3). Instead of using traditional point primitives, we introduce the domain primitive, a sort of 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 more able to capture shape and color variability 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.

Domain primitives are created by a largely automated and robust pipeline consisting of surface classification, surface decomposition, and primitive construction. The methods used for surface classification and decomposition, i.e. moment-based classifiers and algebraic multigrid respectively, have been already successfully used on a variety of complex datasets [3–6]. Implementing the AMG tool is difficult, but fortunately several packages are already available that can be used, virtually

as black-boxes, as no parameters need to be specifically set. AMG's multiscale decomposition provides discrete simplification levels, enabling an image quality and framerate tradeoff. Finally, the primitive construction stage treats every domain primitive independently, which adds to the elegance of our approach. Overall, the only user parameters of the complete pipeline are the classifier neighborhood size (Sect. 2.1), AMG scale (Sect. 2.3), and texel size (Sect. 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.

Overall, some artifacts are sometimes visible in the simplified renderings. The projection error is an inherent problem that is caused by approximation of the surface by 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. [10], 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 method of Décoret et al. uses transparency as a stencil mask, i.e. to turn on and off texture pixels. This causes our cracks to be often less visible. Also, in contrast to Décoret et al., we do not use normal maps to encode detail surface variation, but just draw flat textured polygons. Hence, the quality of our rendering should be compared actually with a *flat shading* of an unstructured polygon soup containing the same small number of polygons, such as illustrated in Fig. 12b. From this perspective, we argue that the relative quality loss of our method is more than reasonable.

Another aspect concerns the usage of the transparency, or alpha, channel. As explained so far, we reserve this channel for combining the multiscale basis functions by means of additive blending. This leaves the question whether our method can handle half-transparent rendering of simplified objects. To render a simplified object with transparency $\alpha \in [0, 1]$, we use the rendering method presented so far in Sect. 4, but scale the values of the texture-encoded basis functions with α. This scaling can be achieved easily and efficiently by using the imaging operations of the OpenGL 1.1 fixed pipeline.

Several directions of future research are envisaged. First and foremost, programmable graphics hardware can be used to enhance the flexibility, thus remove several artifacts, of the rendering algorithm used to combine the domain primitives. The by far most effective addition to our method would be the addition of normal maps, stored in similar textures as our current basis functions. Normal maps would massively improve the rendering quality and also allow a very simple and effective real-time relighting strategy. However, as explained earlier, this would require programmable graphics hardware, which is typically not present on the low-end machines which need extreme model simplification, such as web servers, for example. 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 efficiently computable error metric for measuring the visual quality of our simplifications. As a starting point, this could be done by rendering the original and simplified objects with the same viewing parameters and measuring the (normalized) difference between the resulting images. 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. Alexa, M., Behr, J., Cohen-Or, D., Fleishman, S., Levin, D., Silva, C.: Point set surfaces. In: Proceedings of IEEE visualization, pp. 21–28 (2001)
2. Brezina, M., Cleary, A.J., Falgout, R.D., Henson, V.E., Jones, J.E., Manteuffel, T.A., McCormick, S.F., Ruge, J.W.: Algebraic multigrid based on element interpolation (AMGe). SIAM J. Sci. Comp. **22**(5), 1570–1592 (2000)
3. Clarenz, U., Rumpf, M., Telea, A.: Surface processing methods for point sets using finite elements. Comput. Graph. **28**(6), 851–868 (2004)
4. Clarenz, U., Rumpf, M., Telea, A.: Robust feature detection and local classification for surfaces based on moment analysis. IEEE TVCG **10**(5), 516–524 (2004)
5. Clarenz, U., Griebel, M., Rumpf, M., Schweitzer, M.A., Telea, A.: Feature sensitive multiscale editing on surfaces. Visual Comput. **20**(5), 329–343, Springer (2004)
6. Clarenz, U., Rumpf, M., Telea, A.: Fairing of point based surfaces. In: Proceedings of computer graphics international (CGI), pp. 600–603 (2004)
7. Grauschopf, T., Griebel, M., Regler, H.: Additive multilevel preconditioners based on bilinear interpolation, matrix-dependent geometric coarsening and algebraic multigrid coarsening for second order elliptic PDEs. Appl. Num. Math. **23**(1), 63–96 (1997)

8. Griebel., M., Oeltz, D., Schweitzer, M.A.: An algebraic multigrid method for linear elasticity. SIAM J. Sci. Comp. **25**(2), 385–407 (2003)
9. Griebel, M., Rumpf, M., Preusser, T., Schweitzer, M.A., Telea, A.: Flow field clustering via algebraic multigrid. In: Proceedings of IEEE visualization, pp. 35–42 (2004)
10. Décoret, X., Durand, F., Sillion, F., Dorsey, J.: Billboard clouds for extreme model simplification. In: Proceedings of the ACM SIGGRAPH, pp. 689–696 (2003)
11. Foley, J., Van Dam, A., Feiner, S., Hughes, J.: Computer graphics: principles and practice. 2nd edn, Addison-Wesley, New York, pp. 649–720 (1993)
12. Kalaiah, A., Varshney, A.: Modeling and rendering points with local geometry. IEEE TVCG **9**(1), 30–42 (2003)
13. Koren, Y., Carmel, L., Harel, D.: ACE: a fast multiscale eigenvector computation for drawing huge graphs. In: Proceedings of IEEE information visualization, pp. 137–144 (2002)
14. Pauly, M., Gross, M., Kobbelt, L.: Efficient simplification of point-sampled surfaces. In: Proceedings of IEEE visualization, pp. 163–170 (2002)
15. Pauly, M.: Point primitives for interactive modeling and processing of 3D geometry. PhD Thesis, ETH Zürich (2003)
16. Pfister, H., Zwicker, M., Van Baar, J., Gross, M.: Surfels: surface elements as rendering primitives. In: Proceedings of ACM SIGGRAPH, pp. 335–342 (2000)
17. Ruge, J.W., Stüben, K.: Efficient solution of finite difference and finite element equations by algebraic multigrid. In: Paddon, D.J., Holstein, H. (eds) Multigrid methods for integral and differential equations. Oxford University Press, Oxford (1985)
18. Rusinkiewicz, S., Levoy, M.: QSplat: a multiresolution point rendering system for large meshes. In: Proceedings of ACM SIGGRAPH, pp. 343–352 (2000)
19. Sharon, E., Brandt, A., Basri, R.: Fast multiscale image segmentation. In: Proceedings of IEEE computer vision and pattern recognition, pp. 70–77 (2000)
20. Shewchuk, J.R.: Triangle: engineering a 2d quality mesh generator and Delaunay triangulator. In: 1st workshop of applied computational geometry, pp. 124–133 (1996)
21. Xie, H., Wang, J., Hua, J., Qinh, H., Kaufman, A.: Piecewise C^1 continuous surface reconstruction of noisy point clouds via local implicit quadric regression. In: Proceedings of IEEE visualization, pp. 198–206 (2003)