Efficient Surface Reconstruction From Noisy Data Using Regularized Membrane Potentials

Andrei C. Jalba and Jos B. T. M. Roerdink, Senior Member, IEEE

Abstract—A physically motivated method for surface reconstruction is proposed that can recover smooth surfaces from noisy and sparse data sets. No orientation information is required. By a new technique based on regularized-membrane potentials the input sample points are aggregated, leading to improved noise tolerability and outlier removal, without sacrificing much with respect to detail (feature) recovery. After aggregating the sample points on a volumetric grid, a novel, iterative algorithm is used to classify grid points as exterior or interior to the surface. This algorithm relies on intrinsic properties of the smooth scalar field on the grid which emerges after the aggregation step. Second, a mesh-smoothing paradigm based on a mass-spring system is introduced. By enhancing this system with a bending-energy minimizing term we ensure that the final triangulated surface is smoother than piecewise linear. In terms of speed and flexibility, the method compares favorably with respect to previous approaches. Most parts of the method are implemented on modern graphics processing units (GPUs). Results in a wide variety of settings are presented, ranging from surface reconstruction on noise-free point clouds to grayscale image segmentation.

Index Terms—Graphics processing units (GPU), mass-spring system, membrane potential, point cloud, regularization, surface reconstruction, volumetric segmentation.

I. INTRODUCTION

In the area of surface reconstruction the purpose is to obtain a digital representation of a real, physical object or phenomenon described by a cloud of points, which are sampled on or near the object’s surface. The growing interest in this field is due to the increasing availability of point-cloud data, such as may be obtained from medical scanners, laser scanners, vision techniques (e.g., range images), and other modalities.

In computer vision, shape recovery is a classical problem, whose goal is to derive a 3-D scene description (e.g., surface normal and surface depth) from one or more 2-D images. All techniques that recover shape are commonly called “shape-from-X,” where X can be shading, stereo, texture, or silhouettes, etc. (see [1]–[6] and the references therein). For example, in the stereo problem, one first extracts features (e.g., corners, lines, etc.) from a collection of input images and then solves the so-called correspondence problem, i.e., matching features across images. After obtaining depth information at the locations of the extracted features, one needs to reconstruct the surfaces of the objects present in the scene. One way of achieving this is by using techniques that reconstruct surfaces from point clouds.

Reconstructing a surface from unorganized point clouds is a challenging problem because the topology of the real surface can be very complex, the acquired data may be nonuniformly sampled and the data may be contaminated by noise. In addition, the quality and accuracy of the data sets strongly depend upon the acquisition methodology. Furthermore, the computational cost of reconstructing surfaces from large datasets can be prohibitive. Most of the existing reconstruction methods were developed postulating that precise and noise-free data is available. Therefore, they cannot meet the demands posed by noisy and/or sparse data.

In this paper, which is a greatly extended version of [7], we propose a novel, physically based technique for surface reconstruction, which employs regularized membrane potentials evaluated on a volumetric grid. Although the input data are noisy and sparse, the output surfaces are smooth. The purpose of the membrane potentials is twofold: first, to aggregate data points; second, to remove outliers due to noise. The process in which gaps between the data points are bridged by a slowly varying scalar field will be referred to as aggregation.

The contributions of this paper are as follows.

• A new method for aggregating input data points, based on regularized-membrane potentials, as an alternative approach to the widely employed distance transform (Section III-A). The main advantage of our surface-reconstruction approach is a greatly improved robustness with respect to noise.

• A fast, iterative algorithm for classifying grid points into exterior and interior to the surface (Section III-B).

• A new method for surface smoothing (Section III-C) based on a mass-spring system enhanced with a bending-energy minimizing term, ensuring that the final triangulated surface is smooth (i.e., smoother than piecewise linear).

Our formulation handles noisy as well as nonuniform data sets, it works in any dimension, and is very competitive with previous approaches as regards computing costs. We also take advantage of the increased computational power of modern graphics hardware, and present implementations of most parts of the method on graphics processing units (GPUs). The method can also be used to perform surface reconstruction starting from grayscale volumetric data, leading to image segmentation.

We demonstrate the flexibility and power of the proposed method in a wide variety of settings (Section V). In particular, we show that the proposed method: (i) quickly reconstructs surfaces of very large models, (ii) is reliable even under heavy (shot

IEEE TRANSACTIONS ON IMAGE PROCESSING, VOL. 18, NO. 5, MAY 2009 1119

Manuscript received June 10, 2008; revised January 21, 2009. Current version published April 10, 2009. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Xuelong Li.

The authors are with the Institute for Mathematics and Computing Science, University of Groningen, 9700 AK Groningen, The Netherlands (e-mail: andrei@cs.rug.nl; j.b.t.m.roerdink@rug.nl).

Digital Object Identifier 10.1109/TIP.2009.2016141
and Gaussian) noise conditions, and (iii) copes well with diverse inputs obtained from particle systems, contours, and grayscale volumetric data—all these in the absence of any orientation information.

II. PREVIOUS AND RELATED WORK

There are two main categories of surface representation, called explicit and implicit. We shall classify various existing techniques according to the kind of surface representation employed. As our method is based on an implicit representation, we shall focus our overview on this type. Two important representatives of the first class are parametric and triangulated surfaces. Parametric surfaces are represented by parametric patches (e.g., NURBS, B-spline and Bézier patches), which can represent smooth surfaces and can cope with nonuniform data. Major drawbacks are that patches should be combined to form closed surfaces (which can be very difficult for arbitrary data sets), and noise in the data set is difficult to handle. Triangulated surfaces are usually obtained using tools from computational geometry. Methods that rely on this representation extract subsets of faces of triangulations to yield the reconstructed surfaces [8]–[11]. These methods exactly interpolate the data, and, therefore, are rather sensitive to noise. Moreover, inserting hundreds of thousands of points into a triangulation is computationally expensive. Examples are Alpha Shapes [11], the (Power) Crust algorithm [8], [9], and the Ball-Pivoting algorithm [12].

When using implicit surface representations, one traditionally computes a signed distance function so that the reconstructed implicit surface can be represented by an iso-contour (usually at iso-value zero) of this function [13]–[17]. These methods exactly interpolate the data, and, therefore, are rather sensitive to noise. Moreover, inserting hundreds of thousands of points into a triangulation is computationally expensive. Examples are Alpha Shapes [11], the (Power) Crust algorithm [8], [9], and the Ball-Pivoting algorithm [12].

When using implicit surface representations, one traditionally computes a signed distance function so that the reconstructed implicit surface can be represented by an iso-contour (usually at iso-value zero) of this function [13]–[17]. These methods exactly interpolate the data, and, therefore, are rather sensitive to noise. Moreover, inserting hundreds of thousands of points into a triangulation is computationally expensive. Examples are Alpha Shapes [11], the (Power) Crust algorithm [8], [9], and the Ball-Pivoting algorithm [12].

A. Aggregation of the Input Data Points

This step assigns the input data points to cells of a 3-D grid, using the CIC interpolation scheme. Accordingly, a constant numerical value (we fix this value to one), representing the contribution of each data point to the initial (heat) distribution, is spread to the eight nearest cell centers. The weights are given by the overlap volumes of a box, centered around the data point under consideration, with the neighboring voxels. If several points contribute to the same cell, the values are accumulated. As we will see below, the nonempty grid cells will serve as sources generating potentials on the grid.

The nonempty grid cells, called source points, are regarded as sources for the physical simulation of heat flow, as defined by the linear diffusion equation

$$\frac{\partial u}{\partial t} = \nabla^2 u \tag{1}$$
where \( u \) is the concentration of diffusing material, with the original volume data \( f \) as initial condition, i.e., \( u(t = 0) = f \). Aggregation using (1) has the disadvantage that it converges to a constant steady state. That is, the sizes of the support regions around the cells corresponding to the input points increase with the number of iterations, so that the diffusion would eventually converge to a constant solution covering the whole volume. An illustration of this effect, in the 1-D case, is shown in the left graph of Fig. 2. Two heat sources are placed at positions \( x = 10 \) and \( x = 40 \). From the temporal evaluation of the pure diffusion process, one can easily notice that after \( t > 500 \) the positions of the two maxima (corresponding to the sources) can barely be distinguished.

Since we are not interested in the steady state of linear diffusion, a criterion is required for choosing a stopping time. This can be done with the help of an additional reaction term, which keeps the steady state close to the initial value, leading to the regularized membrane equation (see also [36]–[38])

\[
\frac{\partial u}{\partial t} = \nabla^2 u + \beta(f - u).
\] (2)

The physical analogy of (2), without the diffusive term, is a patch of passive neural membrane, which can be modeled with a serial RC (resistor, capacitor) circuit. The capacitor represents the fact that cellular membranes are good electrical insulators, whereas the resistor depicts the leakage of current through the membrane. The “\( \beta \)” term in (2) demarcates the amount by which the current in the RC circuit has diverged from its original value. This term ensures that the reaction-diffusion equation reaches a steady state not far from the original values of \( f \). However, the problem of choosing a proper stopping time for the linear diffusion is shifted to finding a suitable value for the parameter \( \beta \). To alleviate this problem we have chosen the value of \( \beta \) equal to the absolute value of the original signal \( f \) at each voxel location \( x \), yielding

\[
\frac{\partial u}{\partial t} = \mu \nabla^2 u + |f|(f - u)
\] (3)

where \( \mu \) is a small regularization constant which controls the amount of smoothing. Note that we have used \( \beta = |f| \) in (3) so that we can also handle negative values of \( f \); this will be necessary when performing interpolation in Section III-C, see (4). The choice of \( \mu \) is not critical, as we show in Section V.

Although a closed-form solution of (3) is not at all trivial to obtain, one can use the method of eigenfunction expansion to find an approximate solution. The temporal behavior of the solution using the first 20 terms in the eigenfunction expansion is illustrated in the right graph of Fig. 2. Note that the positions of the maxima can easily be demarcated even for large values of the parameter \( t \), i.e., only after \( t > 4000 \) a plateau of constant value appears between the two maxima. In fact, the steady-state solution of this equation linearly interpolates the data points, whereas the transient solutions are equivalent to Gaussian interpolants in space which decay exponentially in time. Therefore, one can conclude that the formulation of the diffusion process given in (3) is suitable for the purpose of aggregating the input points, provided that the process is stopped before reaching some huge value of the time parameter \( t \). However, the diffusion process should not be stopped too early, because then the “gaps” between the projected input points will not be bridged.

B. Classification of Grid Cells

After aggregation, a method is needed which separates the exterior grid points from the interior ones, thus defining the primary implicit surface. This method should start from the bounding box of the computational grid, follow increasing paths of the scalar field on the grid towards the source points, and label grid cells as exterior, as it proceeds. After the propagation has stopped at regional maxima and ridges, the boundary separating the remaining (interior) points from the exterior points can be traced to yield the reconstructed surface. For this purpose, we will first examine the tagging algorithm of Zhao et al. [19] and then propose a fast, iterative algorithm that fulfills our requirements.

The tagging algorithm of Zhao et al. [19] starts by labeling points on the bounding box of the computational domain as exterior and all other points as interior. This method assumes that the (un-signed) distance transform of the grid cells corresponding to the input points has been computed on the grid. Then, those interior points that have at least one exterior neighbor are labeled as temporary (unknown) boundary points and are inserted into a sorted heap. Next, the subset defined by
the remaining interior points is swept to march the temporary boundary points inwards, towards the input points, as follows. The temporary boundary point with the largest distance (on top of the heap) is checked whether it has at least one interior neighbor with a larger or equal distance value. If not, the point is taken out from the heap, turned into an exterior point, and all its interior neighbors are inserted into the heap. If it does have such a neighbor, the point is removed and turned into a final boundary point (none of its neighbors is added to the heap). This process is repeated until the maximum distance of the temporary boundary points is smaller than some preset distance threshold.

Algorithm 1  Labeling of Grid Points as Exterior, Boundary, and Interior

1: Label all points p as INTERIOR (assign label[p] := INTERIOR)
2: Enqueue points p of bounding box in queue Q and assign label[p] := TRIAL
3: repeat
4:   while Not empty queue Q do
5:     p := Dequeue Q
6:     if ∃n neighbor of p with label[n] = INTERIOR and u[n] << u[p] then
7:       label[p] := EXTERIOR
8:     for all neighbors n of p with label[n] = INTERIOR do
9:       Enqueue n
10:    label[n] := TRIAL
11:    repeat_flag := false
12:   for all points p with label[p] = TRIAL do
13:      if ∃n neighbor of p with label[n] = INTERIOR then
14:        Enqueue p
15:        label[p] := EXTERIOR;
16:        repeat_flag := true
17:       until repeat_flag = false
18:   for all points p with label[p] = EXTERIOR do
19:      if ∃n neighbor of p with label[n] = INTERIOR then
20:        label[p] := BOUNDARY.

Since each grid point is visited at most once, the tagging algorithm described above has a time-complexity of $O(M \log M)$, where $M$ is the number of grid points, and the factor $\log M$ comes from sorting. This algorithm is computationally expensive for large values of $M$. Therefore, instead of adapting this algorithm to meet our requirements, we have developed a fast iterative sweeping algorithm, the pseudo-code of which is given in Algorithm 1 (see also Fig. 3 for its flowchart diagram).

Our algorithm starts also by labeling all points as interior (line 1). Then, the points situated on the bounding box of the grid are inserted in a queue (enqueued) and assigned some temporary value, TRIAL. Then, the subspace is swept as follows. Each trial point is removed from the queue (dequeued) and checked to see if it has at least an interior neighbor that has a smaller value of the potential scalar field (line 6). Only if it does not have such a neighbor, the point is turned into an exterior point and all its interior neighbors are inserted into the queue, as trial points. Otherwise, none of its neighbors is enqueued and its label remains untouched. This case, in which the marching front reaches an interior point with a smaller value, may occur in two situations: (i) either the front has just arrived at the true location of the boundary separating surface interior from exterior, or (ii) the point has a neighbor which has been labeled beforehand. In the first case, the algorithm should stop turning inte-
rior neighboring points into exterior points, since these points
are situated on the other side of the advancing front, and they
are truly interior points. The second case is usually encoun-
tered when the marching front approaches concave regions,
and it happens because the algorithm uses a queue as opposed to a
sorted heap. However, we can solve this ambiguity in the fol-
lowing iterative manner. All trial points which have at least
an interior neighbor are deemed (temporarily) exterior points
and enqued. Also, the existence of such points is signaled
by turning the boolean variable repeat_flag to true (lines
12–16). Then, the whole process is repeated within the repeat-
unitl loop, until no more such points remain. At the end of
the algorithm (lines 18–20), those exterior points which have at least
one interior neighbor are labeled as boundary points. Since each
point is visited at most twice, one iteration of the algorithm will
be completed in no more than O(M) operations. However, a
concern may arise regarding the number of iterations (in the
repeat-until loop) required by the algorithm to converge. Al-
though this number is data-dependent, we did not encounter in
our experiments any data set for which the algorithm would not
terminate in less than five iterations. In practice, our algorithm
is faster than the tagging algorithm of Zhao et al., while CPU
timings suggest a linear-time algorithm. The classification of
grid cells can also be formulated as a linear convection problem,
whose velocity is the gradient of the membrane potential, unlike
Zhao et al. who use the gradient of the distance function. Fur-
ther, one can simplify the convection problem and turn it into
an eikonal one, which can be solved in linear time using the
fast sweeping method of [39]. Although we did not explore this
possibility, we believe that our fast iterative algorithm is better
suited for grid labeling as it does not need to evaluate the gra-
dient of the membrane potential, which is singular at the posi-
tions of the nonempty grid cells.

C. Surface Smoothing and Polygonization

After classification, one can use Bloomenthal’s method [34]
to polygonize the implicit surface given by the zero level set of
the scalar field $g$ defined by

\[
g(x, y, z) = \begin{cases} 
-1, & \text{if } (x, y, z) \text{ is labeled as INTERIOR} \\
0, & \text{if } (x, y, z) \text{ is labeled as BOUNDARY} \\
1, & \text{if } (x, y, z) \text{ is labeled as EXTERIOR}.
\end{cases}
\]  

(4)

1) Interpolation Using Membrane Potentials: Direct polygo-
nization will cause “staircase” artefacts in the resulting mesh
(see Fig. 4, left column). A better approach is to interpolate
the implicit surface using the reaction-diffusion process (3) a
second time, with the labeled grid points as sources. Sources
are instantiated only at the locations of the interior and exterior
grid points [see (4)], since the membership of boundary points is
uncertain. By tracing the zero iso-contour a smooth scalar field
emerges and the implicit surface is turned into a triangulated
one. Since boundary voxels form thin bands along surface
borders, a small number of iterations is required, resulting in fast
computation. The triangulated surface, which is a better approx-
imation to the real surface than the initial one, is used as initial-
ization for the more computationally demanding mass-spring
system, described next.

2) Mesh Smoothing With a Mass-Spring System: Assuming
that the correct topology has been inferred and the triangulated
surface possesses consistent orientation (see Section V-D for a
justification), we propose a mass-spring system for obtaining
a larger degree of smoothing. That is, each edge of each
triangular patch comprising the mesh is modeled by a spring
and each vertex is regarded as a particle with a small mass.
Since we utilize triangular elements, we do not need to include
extra cross springs to afford resistance against shearing (see
[40]). In addition, we integrate an extra energy term such
that the bending energy of the system is minimized. This
has the beneficial effect, analogous to curvature flow, that
the triangulated surface is smoothed by moving its vertices
along their normals with a speed proportional to the (normal)
curvature.

We start by defining nodes $p_i, i = 1, 2, \ldots, N$, of the mass-
spring network, where node $p_i$ has mass $m_i$ and position vector
$x_i(t) = [x_i(t), y_i(t), z_i(t)]$. We denote by $N_i$ the set of neigh-
bors of $p_i$, i.e., all particles $p_j$ with an edge $c_{ij}$ between $p_i$ and
$p_j$. Let spring $s_{ij}$ connect nodes $p_i$ and $p_j$, have rest length $l_{ij}$
and stiffness $c_{ij} = c$, where $c$ is a constant; also, let $r_{ij} =$

![Fig. 4. Smoothing examples (grid resolutions: $33 \times 33 \times 80, 79 \times 80 \times 43,$
$21 \times 54 \times 80$). Left: no smoothing; center: smoothing by reaction-diffusion po-
tentials; right: smoothing by the reaction-diffusion process and the enhanced
mass-spring system.](image-url)
\( \mathbf{x}_j - \mathbf{x}_i \) be the vector separating the two nodes. Then the energy of spring \( s_{ij} \) can be expressed as
\[
E_{s_{ij}} = \frac{c}{2} \left( \mathbf{r}_{ij} - l_{ij} \right)^2.
\] (5)
The potential energy \( E_{s_{ij}} \) of the spring \( s_{ij} \) gives rise to a force \( f_{s_{ij}} \) acting on particle \( p_i \) due to particle \( p_j \)
\[
f_{s_{ij}} = -\nabla_{\mathbf{x}_i} E_{s_{ij}} = c \left( \mathbf{r}_{ij} - l_{ij} \right) \frac{\mathbf{r}_{ij}}{\left| \mathbf{r}_{ij} \right|}.
\] (6)

Smoothing a mesh by minimizing a membrane energy functional [37] can be regarded as the physical simulation of a mass-spring network with zero-rest length springs that will shrink to a single point. On the one hand, because such behavior of the mass-spring system is undesirable for our purposes, the rest lengths of the springs should be chosen such that they reflect the lengths of the edges of the initial (un-deformed) mesh. On the other hand, in order to facilitate the relaxation of the mesh structure into a smooth configuration, the rest lengths of the springs should be smaller than the initial lengths of the edges of the mesh, i.e., we use a percentage of the initial edge lengths.

The bending energy of an ideal, thin flexible flat plate of elastic material is defined as the sum of squared curvatures of spring \( s_{ij} \), the conservative force \( f_{s_{ij}} \) acting on particle \( p_i \) due to particle \( p_j \), minimizing the bending energy, is
\[
f_{b_{ij}} = -\nabla_{\mathbf{x}_i} E_{b_{ij}} = \frac{4}{9} \mathbf{n}_i \cdot \mathbf{r}_{ij} \mathbf{n}_i \left( \frac{\mathbf{r}_{ij}}{\left| \mathbf{r}_{ij} \right|^3} - \frac{8}{3} \frac{\mathbf{r}_{ij} \cdot \mathbf{r}_{ij} \mathbf{n}_i}{\left| \mathbf{r}_{ij} \right|^5} \right). \] (7)

Thus, according to (8) the movement of particle \( p_i \) due to its interaction with particle \( p_j \) will be primarily in the direction of \( \mathbf{n}_i \), whereas the second term in (8) forces the particle to slightly move in the direction of \( -\mathbf{r}_{ij} \) to compensate for the former movement. The normals \( \mathbf{n}_i \) are computed as the average of the normals of the triangular faces incident to each point \( p_i \).

The potential energy of a particle \( p_i \) of the mass-spring system due to its interactions with a neighboring particle \( p_j \), \( j \in \mathcal{N}_i \) is given by
\[
E_i = \alpha E_{s_{ij}}(\mathbf{r}_{ij}) + (1 - \alpha) E_{b_{ij}}(\mathbf{r}_{ij}; \mathbf{n}_i)
\] (9)
where the first term represents the energy (5) of the spring connecting the particles, the second term is the bending energy (7), and \( \alpha \) is a scalar weight.

The variations of particle potentials with respect to positions yield forces acting on particles. The corresponding system of differential equations is
\[
f_i = \sum_{j \in \mathcal{N}_i} \left( \alpha f_{s_{ij}} + (1 - \alpha) f_{b_{ij}} \right)
\] (10)
IV. IMPLEMENTATION ON GPU HARDWARE

In this section, we describe GPU implementations of most constituent parts of the proposed surface-reconstruction method: (i) an iterative method for the computation of the regularized-membrane equation, (3), and (ii) an implementation of the mass-spring system from Section III-C2. Referring to Fig. 1, this means that the aggregation stage [step (2)] and both smoothing methods, based on nonlinear diffusion and on the mass-spring system [paralyly step (4) and step (5)] are implemented on GPU hardware.

A. Regularized-Membrane Potentials

The regularized-membrane equation, (3), is discretized using finite differences (forward differences in time and central differences in space), such that the discrete update rule is

\[
u_{i,j,k}^{n+1} = u_{i,j,k}^n + \Delta t \left[ \mu \sum_{(l,m,n) \in \mathcal{N}_0} u_{i+l,j+m,k+n}^n - 6u_{i,j,k}^n + f_{i,j,k} (f_{i,j,k} - u_{i,j,k}^n) \right]
\]

where \( \Delta t \) is the time step (\( \Delta t \leq (\Delta x \Delta y \Delta z)/(6\mu) \) for stability reasons), \( n \) is the current iteration, and \( \mathcal{N}_0 \) denotes the neighbors of location \((i,j,k)\) using 6-connectivity. This update rule can be straightforwardly encoded in a fragment program running on GPU hardware, as follows. Volumes \( f \) and \( u \) (initially \( u(l = 0) = f \)) are encoded as luminance (or red color) components into two collections \( T_f \) and \( T_u \) of 2-D texture maps. At each iteration, each texture \( t_u \in T_u \) representing a slice of the 3-D volume \( u \) is updated by (i) enabling the fragment program, (ii) passing it textures \( t_f, t_u \), left \((t_u^-)\) and right \((t_u^+)\) neighboring textures of \( t_u \), and parameters \( \Delta t, \mu \), and (iii) rendering a quad which triggers the computation. At the end of the computation, temporary texture \( t_{t2} \) contains the updated values. Note that to update the next slice \( t_{t2}^+ \), the previous values of \( t_u \) are needed, and not the updated values stored in \( t_{t1} \). After slice \( t_{t2}^+ \) has been updated as well, we store its values into a second temporary texture \( t_{t2} \). Then, texture \( t_{t2} \) is copied at its proper location (i.e., \( t_u \)), and the temporary textures are swapped. The computations for the next slices proceed in a similar manner. Special care should be taken when updating the first and last slices, such that the desired (Dirichlet) boundary conditions are implemented.

A problem with this approach lies in the encoding of volumes \( u \) and \( f \) as scalar-only components into textures. Texture storage allows to represent vectors of up to four components, corresponding to the RGBA color components. This approach will not only be more efficient in storage but also in computations, since four scalar operations can be performed in parallel by GPUs, at no additional computational cost. However, if we simply encode 4-component vectors of data into RGBA textures along rows, the neighbors which would be retrieved by performing texture lookups at the left and right of the current position would not be the correct ones. An example is shown in the first row of Fig. 5. Assume that the neighbors at the left of the location in the middle (starting with index 4) are needed. By performing the lookup, the values with indexes \((1,2,3)\) are retrieved. However, these are not the correct ones, e.g., the value with index 0 is not a left neighbor (in the original data) of the value with index 4, etc. To address this, we pack the data along rows using the permutation \((4j+i) \rightarrow (i(W)/(4)+j)\), where \( W \) is the width of the texture, \( i = 0, 1, 2, 3 \) and \( j = 0, 1, \ldots, (W)/(4) \). For the case in Fig. 5, the result after packing is shown in the second row. Now the desired operation can be performed in parallel for four values. However, a new issue appears: the values in the first and last locations of any row (marked in Fig. 5) do not have left and right
neighboring locations, respectively, although in the original data they do. Still, the corresponding values can be retrieved by performing lookups at the very last position for the left neighbors of the first locations, and conversely, at the first position for the right neighbors of the last locations. Of course, there exists no left neighbor of location 0 nor a right one for the last location (11 in our example). The unpacking scheme needed to retrieve the values when the whole computation has been finished can easily be deduced.

In our implementation, first the “central region” of each slice is updated by rendering a quad with position \((1, 1, W − 1, H − 1)\). Then, two horizontal lines with \(y\) coordinates 0 and \(H\), and two vertical lines with \(x\) coordinates 0 and \(W\) are rendered, which are required anyway to implement boundary conditions. The fragment programs for rendering the vertical lines also take care of the lookups specific to our packing scheme. The advantage is that no extra texture lookups into dependent textures are needed, to implement the indirect texture addressing otherwise required, see [44].

### B. Enhanced Mass-Spring System

The main computational steps required to update the position \(x_t\) of a particle \(p_t\) are (i) estimation of surface normal \(n_t\); (ii) computation of the resulting force \(f_t\) on the particle (see (11)), and (iii) position update using the update rule in (13).

Normals \(n_t\) are computed as the average of the normals of the triangular faces incident to each vertex \(p_t\). Thus, for each vertex \(p_t\) one needs to maintain a list of triangular faces incident to \(p_t\). Likewise, a list of vertices adjacent to vertex \(p_t\) is needed when computing the spring force acting on \(p_t\). Both lists are computed and stored into textures using a layout similar to that in [45]. The details are as follows.

Particle positions and vertex indexes of each face are encoded in two RGB textures \(t_x\) and \(t_f\), respectively. For storing surface normals we use two additional textures \(t_{N,face}\) and \(t_{N,face}\), initialized as follows. For each vertex \(p_t\), the indexes of the incident faces in \(p_t\) are stored consecutively into the RG components of texture \(t_{N,face}\) as segments (i.e., along rows). Each segment’s starting address and length are stored in the indirection texture \(t_{N,face}\). The computations are triggered by drawing a quad with size equal to that of texture \(t_x\), and they are performed by a fragment program which is passed textures \(t_x\), \(t_f\), \(t_{N,face}\), \(t_{N,face}\) as input parameters. First, the fragment program performs a lookup into texture \(t_{N,face}\) to obtain the address of its corresponding segment from texture \(t_{N,face}\). Then, each incident face is identified by performing a lookup into texture \(t_{N,face}\) along the current segment. Next, the indexes of the vertices of the current incident face are found from texture \(t_f\). Finally, the positions of the vertices of the current incident face are retrieved from texture \(t_x\), and the contribution of this face to the normal in the current vertex \(p_t\) (corresponding to the current fragment) can be computed. At the end of the computation, the normals are stored in an OpenGL vertex array \(V_g\).

An approach similar to the one just described, which uses two additional textures \(t_{X,face}\) (for indirection encoding) and \(t_{X,face}\) (for adjacency), is used for the computation of the resulting forces as given in (11). However, it is computationally cheaper to combine this step with the last step, i.e., position update. According to (13), at the end of each integration step the current position \(x_t\) of particle \(p_t\) is stored in variable \(x_t\). Thus, a useful optimization is to simply swap array pointers (in

---

<table>
<thead>
<tr>
<th>Model</th>
<th>No. Points</th>
<th>Grid</th>
<th>No. Vertices</th>
<th>No. Triangles</th>
<th>Error (%)</th>
<th>Aggregation Time (s)</th>
<th>Marching Time (s)</th>
<th>Interpolation Time (s)</th>
<th>Smoothing Time (s)</th>
<th>Total Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buddha</td>
<td>543,625</td>
<td>170x400x170</td>
<td>297,062</td>
<td>594,182</td>
<td>0.08</td>
<td>1.3</td>
<td>6.9</td>
<td>0.9</td>
<td>4.9</td>
<td>15.2</td>
</tr>
<tr>
<td>Armadillo</td>
<td>172,974</td>
<td>337x400x307</td>
<td>370,840</td>
<td>741,572</td>
<td>0.05</td>
<td>4.9</td>
<td>29.3</td>
<td>3.8</td>
<td>5.4</td>
<td>41.8</td>
</tr>
<tr>
<td>Dragon</td>
<td>433,375</td>
<td>400x284x184</td>
<td>383,680</td>
<td>767,390</td>
<td>0.08</td>
<td>2.1</td>
<td>13.4</td>
<td>1.5</td>
<td>6.9</td>
<td>24.6</td>
</tr>
<tr>
<td>Hand</td>
<td>327,322</td>
<td>400x283x143</td>
<td>190,664</td>
<td>381,372</td>
<td>0.06</td>
<td>1.6</td>
<td>11.7</td>
<td>1.2</td>
<td>3.1</td>
<td>18.2</td>
</tr>
<tr>
<td>Asian Dragon</td>
<td>3,609,600</td>
<td>400x226x269</td>
<td>206,142</td>
<td>412,290</td>
<td>0.06</td>
<td>2.5</td>
<td>18.7</td>
<td>1.8</td>
<td>4.3</td>
<td>28.2</td>
</tr>
</tbody>
</table>
terms of GPU hardware this means exchanging textures). We use an extra texture $t_{\text{new}}$ which stores particle positions from the previous time step. The fragment program computes the force acting on the current particle (associated with the present fragment), then uses (13) to update its position. At the end of the computations, the new positions are made available into texture $t_{\text{p}}$ and vertex array $V_g$. (The vertex array can be used, if desired, to render the mesh during its evolution.) When advancing to the next step, texture $t_{\text{p}}$ is used for computing the normals, and then the two textures $t_{\text{p}}$ and $t_{\text{new}}$ are swapped.

The approach outlined above is implemented in two rendering passes: one for the computation of normals and the other for updating particle positions. However, the size of textures $t_{\text{N,faces}}$ or $t_{\text{X,weight}}$ may become larger than the maximum texture size allowed by OpenGL. In this case, the surpluses are encoded into additional textures and extra rendering passes are needed to finalize the whole computation. Actually, very large data sets of hundreds of thousands of vertices (see Section V) require four passes, two for each computational step (computation of normals and update of particle positions).

The speedups obtained on a GeForce FX 7900 GTX graphics processor compared to optimized CPU algorithms (run on a machine with an AMD Opteron processor at 2.4 GHz) vary from 7 to 10 for the computation of the regularized-membrane potentials. Also, the mass-spring system performs at interactive rates of about 10 to 25 fps, for fairly large meshes, see Section V. A possible optimization is to use an algorithm for mesh decimation prior to mesh smoothing, so that the number of vertices and faces becomes smaller.

Currently, the grid point classification and polygonization steps are performed on the CPU, see Fig. 1. The reason is that current GPU hardware does not permit an efficient implementation of a flooding process, analogous to a standard CPU counterpart using a queue. Of course, it is possible to achieve the same result using an iterative brute-force method. Yet, the performance will not be any better than that of a sparse, queue-based computation run on the CPU. Although first attempts have been made to surface polygonization on GPUs [46], the performance was worse than that of standard CPU implementations. Moreover, in our method this step is performed only once, instead of many times as required for other applications (e.g., iso-surface browsing).

V. RESULTS

All computations were performed on a system equipped with a Pentium IV processor at 3.0 GHz and a GeForce FX 7900 GTX GPU.

A. Large Data Sets

The parameters were set as follows. Aggregation was done by computing the membrane potential, as in Section IV-A; we used $N_m = 20$ iterations and the value of parameter $\mu$ was set to 0.1. The parameters of the Verlet integrator [see (14)] for the mass-spring system were $\Delta t = 0.1$ and $t = 10$. The force weight in (10) was set to $\alpha = 0.1$, to emphasize the bending-energy minimizing term. The rest lengths of the springs were set to 90% of the initial edge lengths. Finally, the largest dimension of the grid was set to 400 and the remaining two dimensions were obtained by uniform scaling of the bounding box of the
sample points. Below we use the same values of the parameters (unless stated otherwise).

The results are shown in Fig. 6. Timings (in seconds) of each step of the method, for the models shown in Fig. 6, are given in Table I. The time taken is well under 1 min. The most expensive parts are the labeling and the second stage of smoothing by the enhanced mass-spring system. The computations have the same order of magnitude, and the computational cost generally depends exclusively on the grid size (e.g., compare the Asian Dragon model with the Armadillo model).

Table I provides some statistics. The sixth column shows the approximation error—an indication of the quality of reconstruction. This error is an upper bound for the average distance from the data points to the surface, and it is computed as the average distance from the data points to the centers of mass of the mesh triangles. The error is given as a percentage of the diagonal of the bounding box of the data points.

### B. Coping With Noise

Next we consider the method under noise and random subsampling conditions.

1) **Shot Noise:** A certain number of empty voxels was changed by assigning them the value one, i.e., the same numeric value used to assign the input points. The number of corrupted voxels is expressed as a percentage of the number of source points. We used nearest-neighbor interpolation for grid assignment, as this results in a binary volume and represents a fair setting, without a priori information. The results of this experiment are shown in Fig. 7; the initial number of source points was 9,830.

For the computation of the membrane potential, the number of iterations $N_m$ was increased from 20 to 100. The reason is that a large number of iterations results in a large aggregation support covering most of the exterior volume around the object, which will be correctly labeled as exterior. Note that the method is able to reconstruct the surface of the cactus shown throughout Fig. 7 even when as much as 80% of the source points were corrupted by noise.

2) **Gaussian Noise:** The input points were corrupted with zero mean Gaussian noise with standard deviations $\sigma = 0.5, 1.0, 1.5(\%)$, expressed as percentages of the length of the diagonal of the bounding box of the grid. The results are shown in Fig. 8. The grid size was $210 \times 200 \times 114$. The parameters were set as in the previous section, with one exception—the stopping time $t$ of the mass-spring system was increased from 10 to 25.

Unlike methods which rely on distance transforms, our method can cope with large amounts of Gaussian noise. In fact, in the third case ($\sigma = 1.5(\%)$) of Fig. 8, one percent of the diagonal of the bounding box means that $\sigma = 3.1$, which implies that the coordinates of most points were randomly translated in the interval $[-9.3; 9.3]$. Yet, even then the method is able to output smooth surfaces, with errors bounded by $\varepsilon_f$.  

![Fig. 10. Mesh smoothing comparison. Left-to-right, top-to-bottom: original mesh, method of Jones et al. [49], curvature flow [47], [48], our method.](image)
3) Random Sampling: We used a relatively large grid (288 x 408 x 410), such that the number of source points (100, 523) is comparable to the number of input points (100, 759). Then, keeping the grid resolution constant, we randomly sub-sampled the set of source points and performed reconstruction using only the sampled points; we used nearest-neighbor interpolation for grid assignment, and the parameters of the method were set as in the previous section. Fig. 9 shows the results. Note that the method yields very good results, even with as few as 10% of the source points.

C. Comparison to Other Methods

1) Mesh Smoothing: We compared our mesh-smoothing by a mass-spring system with curvature flow [47], [48], and with the noniterative, feature-preserving method of Jones et al. [49]. The results are shown in Fig. 10. The stopping time for the iterative methods (i.e., curvature flow and mass-spring system methods) was set to \( t = 300 \), whereas the parameters of the noniterative method were set to \( \sigma_f = 2, \sigma_g = 10 \) (to smooth large features), which yielded the best result. Note that the noniterative method preserves too many mesh details, whereas, at the other extreme, curvature flow smears out even large mesh features. Our proposed method seems to offer the best tradeoff between mesh smoothness and feature preservation. In addition, it can be efficiently implemented on GPU hardware, unlike the noniterative method.

2) Surface Reconstruction: We compared our surface reconstruction method to that of Hoppe et al. [14], to the Power Crust algorithm by Amenta et al. [8], [9] and to the level set method of Zhao et al. [19], see Fig. 11. The time taken by our method for the model in Fig. 11 was 56 s on a grid with dimensions 450 x 320 x 206; the reconstruction error was 0.06. It took 4 min by the method of Hoppe et al. to reconstruct the same model. Note that some holes are visible in the triangulated surface, since we increased the parameter controlling the sampling of the unknown surface as much as possible, in an attempt to reconstruct fine surface details. Although the reconstructed surface is smooth, fine surface details are lost. This method can tolerate Gaussian noise provided that each sample point has on average the same distance to its neighbors. However, the method does not tolerate shot noise, see Fig. 12.

The highest resolution of the reconstructed surfaces is obtained by methods which interpolate the data points, similar to the Power Crust algorithm, see Fig. 11. However, the time taken to reconstruct surfaces of large models (see Table I) within floating-point precision is two orders of magnitude larger than...
that of the proposed method. Since this method interpolates the data points, it cannot cope with either types of noise which we considered.

One method which lends itself to an efficient GPU implementation is the level set method of Zhao et al. [19]. We performed a direct comparison of our method and this level set method, both implemented similarly on GPU hardware, as in Section IV. Statistics, reconstruction quality and timings for the latter method are shown in Table II. Instead of using the full minimal surface model, this method can be accelerated by first convecting the surface towards the data points and then using a small number of iterations (say 10) of curvature-based smoothing, to obtain the final (smooth) surface. The penultimate column of Table II shows the total CPU time for the minimal surface model, whereas the last one shows the timings for the convection model with curvature-based smoothing. Note that even in the latter case, our method is about an order of magnitude faster than the level set method. Also, as shown in Fig. 13, this method does not tolerate shot noise, due to its reliance on the distance transform for aggregation.

One of the fastest techniques for surface reconstruction is the MPU method of Ohtake et al. [30]. Comparing the result from Table I on the Dragon data set with that from Table II in [30] one observes that our method is 2.3 times faster than the MPU method, at the same accuracy (8.0 \times 10^{-4}). However, if larger accuracy is needed, their method may be more efficient. Nevertheless, the method assumes that accurate normal estimates are available.

D. Error Analysis of the Framework

Assume that the grid resolution agrees with the (appropriate) sampling rate of the unknown surface to be reconstructed, i.e., each point of the data set is assigned to a distinct grid cell.
the small-cell-size limit, the CIC interpolation scheme becomes nearest neighbor interpolation. Also, we assume a clean input data set. After aggregation, a smooth scalar field emerges with regional maxima at the locations of the data points and ridges along straight segments connecting adjacent data points. After labeling, the exterior and interior layers bound the surface of the object closely, such that (i) grid points corresponding to input points in the boundary layer are enclosed between the interior and exterior layers, and (ii) the surface of the exterior/interior layer represents a manifold. Therefore, if \( h_x = h_x^0 g_x, h_y = h_y^0 g_y, h_z = h_z^0 g_z \) are the grid-cell sizes, \( h_x, h_y, h_z \) are the dimensions of the bounding box of the data points, and \( g_x, g_y, g_z \) are the number of cells in the \( x, y, z \) directions, then a bound on the reconstruction error is given by the length of the diagonal of a grid cell, i.e., \( \varepsilon = \sqrt{h_x^2 + h_y^2 + h_z^2} \). Implicit surface interpolation by (3) with initial condition (4) cannot increase the reconstruction error since grid cells labeled as interior/interior maintain their labels due to the similarity term. Moreover, interpolation using (3) yields a smooth field at boundary locations, which can only decrease the reconstruction error, though the error bound remains the same. After interpolation, the gradient field has correct orientation, without singular points, and, therefore, the reconstructed surface is consistently oriented. Also, when the grid resolution is large enough, any of the surfaces of the interior/exterior layers has the same topology as the unknown surface, and, therefore, the reconstructed surface will have the same topology.

In the presence of noise, surface features smaller than the noise amplitude in the data set can obviously not be recovered. However, as we showed in Section V-B, the method is noise tolerant, albeit the error bound increases up to \( \varepsilon_n = \eta + \sqrt{h_x^2 + h_y^2 + h_z^2} \), where \( \eta \) denotes the standard deviation of the noise. These error bounds remain the same even if the mapping of data points to nonempty grid cells is not one-to-one. The mass-spring system potentially increases the overall reconstruction error. Therefore, an essential requirement is that our mass-spring system preserves the features of the triangulated surface. We showed in Section V-C that this is indeed the case.
E. Reconstruction Error

To verify our claims made in Section V-D, we studied the behavior of the reconstruction error (cf. Table I) when grid resolution increases, see Table III. As can be seen, the reconstruction error is always bounded by \( \varepsilon_r \), even when the mass-spring system is used for mesh smoothing. Only at small grid resolutions the reconstruction error increases when mesh smoothing is applied, because our mesh smoother preserves small features of the triangulated surface.

The results shown in Table III were obtained using a CPU implementation, unlike those of Table I. The reason is the small amount of memory available on our current GPU hardware, which would not allow using grids larger than those of Table I.

F. Other Results

1) Particle Systems: To evaluate whether the proposed method can be used to triangulate surfaces sampled with particle systems, we used the particle system from [50] to provide the input data points; we have used two grayscale volumetric data sets, obtained by Magnetic Resonance Angiography (MRA). The results are shown in Fig. 14. Note that the method copes rather well with these noisy inputs, and yields in both cases smooth, approximating surfaces.

2) Contours: A Magnetic Resonance Imaging (MRI) scanner outputs parallel cross sections of the area under observation. Thus, one can extract polygonal contours in every cross section, either manually or using tools from image analysis [51]. Example results obtained by the proposed method using this kind of input data sets are shown in Fig. 15. Note that although these data sets are highly nonuniform, the method is able to output rather smooth surfaces.

3) Grayscale Volumetric Data: Another application is (semi-automatic) volumetric segmentation, based on the observation that instead of assigning the input points to the volumetric grid, one can use directly a grayscale volume as input. That is, we start by computing some contrast measure (e.g., gradient magnitude) within the input volume and by assigning heat sources proportional to this measure. Then, the method proceeds as usual with computing membrane potentials on this grid.

For the purpose of automatic segmentation one may use the bounding box as the initial surface, after the potentials have been computed. The resulting implicit surface defines the boundaries of the objects present in the input volume. This is in accordance with the very definition of image segmentation, i.e., the process by which an image is divided in its constituent parts. This implies that segmentation should produce a complete partitioning of the image such that object contours are closed and precisely localized. Note that since our method relies on an implicit-surface representation, the latter requirement is fulfilled.

The results of an experiment involving two sets of 3-D MRA volume data are shown in Fig. 16. The experimental setup was as follows. Since these data sets are likely to be noisy, we performed regularization of the input volumes (i.e., convolution...
with a Gaussian kernel of width $\sigma = 2.5\sigma$, prior to estimating the gradient magnitude. As can be seen from Fig. 16, most important structures were correctly recovered.

VI. LIMITATIONS

In closing, we list a number of (current) limitations of our method.

Surface features smaller than the grid size are not appropriately reconstructed, see Fig. 17. A possible solution would be to increase the grid resolution at the expense of larger computational time and memory requirements. Also, the method is not geometrically adaptive, but we are currently investigating an adaptive, multiresolution approach based on data-structures similar to octrees which can also be efficiently implemented on GPUs.

As is usual for methods that employ implicit surface representations, we assume that the surfaces to be reconstructed are closed, though the method does perform intrinsic hole filling by minimal surfaces. Fig. 18 shows that by increasing $N_m$ (number of iterations used for computing membrane potentials), increasingly larger holes can be filled, at the expense of higher computational requirements. Note that there are no samples at the bottom of the vase object, as can be seen from the first image of Fig. 18.

A related problem is that if surface sheets come close to each other, e.g., see the knot surface in Fig. 4, and a large $N_m$ is used, concavities and holes of the object may be filled.

VII. CONCLUSION

We have introduced a novel framework for surface reconstruction starting from unorganized point clouds without orientation information, and demonstrated its effectiveness in various experimental settings. The method can be used to efficiently reconstruct surfaces from clean as well as noisy data sets, and in our opinion, this represents an advantage over existing methods. The method can deliver multiresolution representations of the reconstructed surface, and can be used to perform reconstruction starting from particle systems, contours or even grayscale volumetric data leading to image segmentation.

Most constituent parts of the method have been implemented on GPU hardware. A problem which we encountered is the relatively small amount of video memory available on current graphics cards, which limits the sizes of the data sets which can be loaded in texture memory. Nevertheless, it is our belief that the method is flexible enough to be used and adapted to diverse tasks, ranging from accurate surface reconstruction on noise-free data sets to grayscale image segmentation.

REFERENCES


Andrei C. Jalba received the B.Sc. (1998) and M.Sc. (1999) degrees in applied electronics and information engineering from the “Politehnica” University of Bucharest, Romania, and the Ph.D. degree from the Institute for Mathematics and Computer Science, University of Groningen, Germany, in 2004.

His research interests include computer vision, pattern recognition, image processing, and parallel computing.

Jos B. T. M. Roerdink (SM’95) received the M.Sc. degree (1979) in theoretical physics from the University of Nijmegen, The Netherlands, and the Ph.D. degree (1983) from the University of Utrecht.

Following his Ph.D. and a two-year position (1983–1985) as a Postdoctoral Fellow at the University of California, San Diego, both in the area of stochastic processes, he joined the Centre for Mathematics and Computer Science, Amsterdam, The Netherlands. He worked there from 1986–1992 on image processing and tomographic reconstruction.

He was appointed Associate Professor (1992) and Full Professor (2003), respectively, at the Institute for Mathematics and Computing Science, University of Groningen, where he currently holds a Chair in Scientific Visualization and Computer Graphics. His research interests include biomedical visualization, neuroimaging, and bioinformatics.