Vis Comput
DOI 10.1007/s00371-014-0924-7

ORIGINAL ARTICLE

GPU-based polygonization and optimization for implicit surfaces

Junjie Chen · Xiaogang Jin · Zhigang Deng

Abstract Despite the popularity of polygonization of implicit surfaces in graphics applications, an efficient solution to both polygonize and optimize meshes from implicit surfaces on modern GPUs has not been developed to date. In this paper, we introduce a practical GPU-based approach to efficiently polygonize and optimize iso-surface meshes for implicit surfaces. Specifically, we design new schemes to maximally exploit the parallel features of the GPU hardware, by optimizing both the geometry (vertex position, vertex distribution, triangle shape, and triangle normal) and the topology (connectivity) aspects of a mesh. Our experimental results show that, besides significant improvement on the resultant mesh quality, our GPU-based approach is approximately an order of magnitude faster than its CPU counterpart and faster than or comparable to other GPU iso-surface extraction methods. Furthermore, the achieved speedup becomes even higher if the resolution of the iso-surface is increased.

Keywords Implicit surface · Polygonization · Mesh optimization · GPU parallelization

J. Chen
State Key Laboratory of CAD&CG, Zhejiang University, Hangzhou, People's Republic of China
e-mail: chenjunjie@cad.zju.edu.cn

X. Jin (✉)
State Key Laboratory of CAD&CG, Zhejiang University, Hangzhou 310058, People's Republic of China
e-mail: jin@cad.zju.edu.cn

Z. Deng
Computer Science Department, University of Houston, Houston, USA
e-mail: zdeng4@uh.edu

1 Introduction

Implicit surface, mathematically defined as the zero-level iso-surface of a scalar-valued function, $F : R^3 \rightarrow R$, is a widely used fundamental surface representation in computer graphics applications. It can be visualized in two different ways: ray-tracing or polygonization. While ray-tracing algorithms can produce realistic and accurate rendering results, they are notoriously time-consuming. Therefore, in recent years polygonization of implicit surfaces has been popularized in numerous graphics applications due to its efficiency and flexibility.

Over the years, many polygonization algorithms have been developed to convert implicit surfaces into corresponding 3D meshes. Among them, the marching cubes (MC) algorithm is probably the most used (or de-facto) algorithm for such a task due to its simplicity and efficiency. To accelerate the MC algorithm, researchers have designed various strategies including octree data structure [3,4] and GPU-specific acceleration schemes [7]. However, raw 3D meshes extracted by the MC algorithm often suffer from artifacts. A variety of algorithms have been proposed to further optimize the polygon meshes [1,6,8,12], but these mesh optimization approaches typically run on the CPU, which is inefficient especially for large-scale meshes. Furthermore, if we have extra geometry information about a mesh, which is the case in polygonized implicit surfaces, the optimization would be much more accurate. However, the main technical challenges are, on the one hand, the extracted polygons are unorganized and thus not convenient for further optimization, and on the other hand, many existing optimization methods are not suitable for parallelization. Therefore, an important yet under-explored research question is: can we design an efficient and practical approach to both polygonize and optimize iso-surface meshes for implicit surfaces directly on GPU?

Published online: 05 March 2014

Inspired by the above observations, in this paper, we introduce a practical GPU-based approach to efficiently polygonize and optimize iso-surface meshes for implicit surfaces. Specifically, at the polygonization stage, compared with existing GPU-based MC algorithms [7], we design special data structures for GPU acceleration and construct connectivity information by computing neighbors and detecting boundaries. At the surface optimization stage, we design new schemes to maximally exploit the parallel features of the GPU hardware, by optimizing both the geometry (vertex position, vertex distribution, triangle shape, and triangle normal) and the topology (connectivity) aspects of a mesh. Through our comparison experiments, we demonstrate that, besides significant improvement on the resultant mesh quality, our approach can achieve approximately an order of magnitude faster speed than CPU-based algorithms for the same task, and is faster than or comparable to other GPU-based methods, in particular for large-scale datasets.

The main contribution of our work is a new parallel approach to efficiently construct high-quality polygon meshes from implicit surface representations. Different from other polygonization algorithms, which generate a polygon soup or a group of indexed triangles, our approach adds a stage of connectivity reconstruction that enables optimization of not only vertex positions but also normals, regularity and even connectivity itself. Our approach can be used in a broad variety of graphics applications including, but not limited to, surface reconstruction, implicit surface modeling, and volume data visualization.

The remainder of the paper is organized as follows. Section 2 briefly reviews related research efforts. Section 3 describes how to perform iso-surface polygonization on GPUs. Section 4 describes the parallel mesh optimization algorithms on GPUs. Finally, experimental performances and concluding remarks are presented in Sects. 5 and 6, respectively.

2 Related work

2.1 Iso-surface extraction

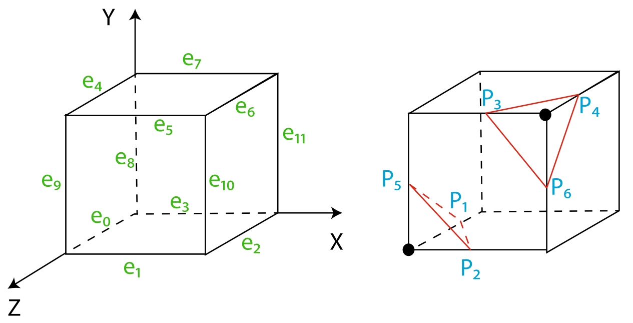
The marching cubes (MC) algorithm introduced by Lorensen and Cline [19] is one of the widely used algorithms in scientific visualization and 3D surface reconstruction applications [9]. In this seminal approach, 3D space is partitioned into cubes; the cubes are polygonized. Unlike the uniform space partition scheme in the standard MC algorithm, Bloomenthal proposes an adaptive octree space partition scheme to improve the algorithm efficiency [3,4]. While the octree structure gives a higher performance of the polygonization process, constructing the adaptive octree on GPUs is significantly less efficient than the uniform space partition scheme. Lempitsky extracts iso-surfaces from binary volumes by imposing higher order smoothness [14]. In addition, Hartmann uses a continuation method to triangulate implicit surfaces [11], but this method is not suitable for parallelization.

Many research efforts have been focused on the development of parallel approaches to improve the performance of implicit surface polygonization. Shirazian et al. [29] take advantage of multicore processors and SIMD optimization to render complex implicit models. Dias et al. [7] propose a parallel marching cubes algorithm to polygonize convolution molecular surfaces. Tatarchuk et al. [31] use programmable shaders to extract and render iso-surfaces. Löffler et al. [18] parallelize the dual marching cubes method [22] to extract manifold surfaces from terrain data set. Schmitz et al. [28] implement a modified dual contouring on the GPU. Different from the original dual contouring algorithm, they replace the QR factorization with a particle-based approach, which iteratively moves the vertices of the quad towards the iso-surface. There are also several CUDA-based versions of the marching cubes algorithm. Nvdia CUDA SDK [39] provides a simple parallel implementation that generates unorganized triangles which are enough for visualization but difficult to perform further optimization. The most recent CUDA-based MC is probably the Griffin et al.'s version [10]. To get indexed triangles, they sort and hash the vertex buffer and index triangles by binary searching the vertices in a hash buffer. Connectivity information is implicitly used when performing summation in an one-ring neighbor. Different from their algorithm, our connectivity information is explicitly calculated, making it more efficient for neighbor access.

2.2 Mesh optimization

Mesh optimization is the process of improving the quality of a mesh. Specifically, mesh quality is often referred to the sampling, grading, regularity, size or shape of its elements [1]. As one of the early efforts, Taubin [32] introduces Laplacian operations for mesh optimization by iteratively moving each vertex towards the center of its neighbors. Laplacian mesh optimization is originally designed to smooth a noisy mesh, but it can be used to improve triangle shape as well, because the Laplacian vector contains both normal and tangential components. Researchers have used mean curvature flows [6] or bilateral filters [8,12] to smooth the mesh while preserving its features as much as possible. These methods move the vertices only in normal direction, without taking triangle shape into account. Ohtake and Belyaev [24] simultaneously optimize the smoothness and regularity of a mesh by combining the mean curvature flow and the tangential component of the Laplacian vector. These methods are implemented on CPU. But since they adopt a local optimization strategy, these methods are inherently suitable for GPU implementation.

A frequently used method to make vertices evenly sampled is centroidal Voronoi tessellation (CVT). Liu et al. [15] prove that the CVT energy is almost C^2 smoothness and therefore can be optimized by a Newton-like method which is much faster. CVT later is extended to hyperbolic space [25] and supports acceleration on the GPU [26]. A similar idea is to perform optimization with particles by adding a repelling force among them. Meyer et al. [20] use an energy function which is compact, scale-invariant curvature-dependent, and efficient for computation to sample implicit surfaces. Meyer et al. [21] extract high-quality iso-surface meshes from dynamic particles. They first compute a median axis to characterize the feature size, and then adaptively sample particles and generate triangle meshes using a Delaunay triangulation triangulations. Zhong et al. [36] map the anisotropic space into a higher dimension isotropic one, sample and optimize vertices of the surface and then map them back to the original space to get anisotropicity. Liu et al. [16] extract and optimize iso-surfaces simultaneously from a point cloud, taking advantage of the relation between original data and the reconstructed mesh, which is similar to our idea. The main difference between these optimization methods and ours is that we optimize the mesh directly while they construct a mesh as the final step.



Grid Edge	e_0	e_1	e_5	e_6	e_9	e_{10}
Vertex Buffer	P_1	P_2	P_3	P_4	P_5	P_6
Vertex Counter	1	1	1	1	1	1
Vertex Index	1	2	3	4	5	6

Face Index	1			2		
Grid Edge	e_0	e_1	e_9	e_5	e_{10}	e_6
Face Buffer	1	2	5	3	6	4

Fig. 1 *Top* face extraction in a cube. Here grid edges are marked in *green*. *Bottom* buffers used in the face extraction process. The "Vertex Index" is prefix summed from the "Vertex Counter". We have omitted those grid edges that do not contain a vertex

3 Iso-surface polygonization on GPU

3.1 Approach overview

The standard MC algorithm [19] typically results in a group of unorganized triangles (polygon soup). However, for many applications (e.g., mesh editing and deformation), organized triangles with connectivity information are needed. In this work, we polygonize iso-surfaces from implicit surfaces on GPU in the following way: first, we generate mesh vertices and faces from implicit surfaces or volumetric data. Then, we construct connectivity information by computing neighbors and detecting boundaries. The finial output data from our approach include the following data structures:

- *Vertex buffer*: the coordinates of each vertex,
- *Face buffer*: the indices of three vertices of each face,
- *Neighbor vertex buffer*: the indices of all the vertices in one-ring neighbor of each vertex,
- *Neighbor face buffer*: the indices of all the faces in one-ring neighbor of each vertex,
- *Boundary buffer*: the true/false mark of each vertex.

3.2 Mesh extraction

The main difference between our MC algorithm and the standard MC algorithm [19], besides the parallelization aspect, is that the extracted faces by our MC algorithm are stored in the form of vertex indices rather than vertex coordinates in the standard MC algorithm. The essential problem here is how to automatically generate vertex indices, which is non-trivial in parallel situation because they are listed in a serial manner. We use a parallel prefix-summation (scan) method [27] to determine the index of a vertex, utilizing the CUDA Data Parallel Primitives (CUDPP) library [38].

Figure 1 shows an example of extracting two faces in a cube. Given a volumetric data in a regular grid form (raw data or sampled from an implicit function), first, we process each grid edge to extract all the vertices. At the same time, a vertex counter buffer is generated to indicate the number (0 or 1) of vertices on each grid edge. Then, we prefix-sum the buffer so that each vertex is assigned with an index, which is actually the number of visited vertices before this vertex is assigned. The total number of vertices is computed alongside. Similarly, we compute the total number of faces. Finally, we deal with each cube to extract faces, and the indices of their vertices are referred from the summed vertex counter buffer.

Algorithm 1 describes our parallization of the MC algorithm. Note that we do not address the ambiguity problem of MC, which can be corrected by algorithms such as [23] and [2]. Furthermore, this ambiguity problem will not essentially affect our parallelization of mesh extraction.

Algorithm 1 Parallel Iso-surface Extraction

```
1: for each grid edge $i$ parallely do
2:   if it intersects with the iso-surface then
3:     Set its vertex counter $vc_i = 1$
4:     Compute vertex position $P_i$
5:   else
6:     Set its vertex counter $vc_i = 0$
7:   end if
8: end for
9: Prefix-sum $vc$ to generate vertex indices
10: for each grid cube $i$ parallely do
11:   if it intersects with the iso-surface then
12:     Compute the number of triangles $k$ in the cube
13:     Set its face counter $fc_i = k$
14:   else
15:     Set its face counter $fc_i = 0$
16:   end if
17: end for
18: Prefix-sum $fc$ to generate face indices
19: for each grid cube $i$ parallely do
20:   for each triangle $f$ in the cube do
21:     Find the grid edges where $f$'s vertices locate
22:     Find $f$'s vertex indices according to the grid edges
23:   end for
24: end for
```

Algorithm 2 Parallel Construction of Connectivity Information

```
1: for each face $f : v_1v_2v_3$ parallely do
2:   Add $v_1$ to $v_2$'s neighbor
3:   Add $v_2$ to $v_3$'s neighbor
4:   Add $v_3$ to $v_1$'s neighbor
5:   Add $f$ to $v_1$'s face neighbor
6:   Add $f$ to $v_2$'s face neighbor
7:   Add $f$ to $v_3$'s face neighbor
8: end for
```

3.3 Connectivity construction

Our algorithm frequently needs to sum across a vertex's one-ring neighbors. With indexed vertices, this can be accomplished via CUDA atomic functions [10]. Different from [10], we explicitly store the one-ring neighbor information of a vertex. We do use atomic functions, only once, to construct one-ring connectivity. After that, summation across neighbors can be processed inside the GPU kernel. We present our connectivity construction approach in Algorithm 2. By just adding a vertex or a face into another's neighbor, Algorithm 2 seems trivial at first glance. However, our experiments show that it is actually very time-consuming, because we have to use atomic functions when parallel writing data into other threads' resources. This is also the reason we try to avoid such function calls as much as possible.

3.4 Detection of boundary vertices

Detecting boundary vertices is an important part in our approach, because they behave differently from interior vertices in the follow-up optimization process. Besides, Algorithm 2 is incomplete on boundaries. An interior edge is shared and thus processed by two faces, so the end points mutually include each other into their neighbors. A boundary edge, however, is only processed once, making one end point lose its counterpart as a neighbor. To this end, we introduce Algorithm 3 to detect boundary vertices and recapture the lost information. In brief, for any vertex, the algorithm traverses its neighbors to find whether all of them consider this vertex as their neighbor. If not, this vertex will be detected as a boundary vertex.

Algorithm 3 Parallel Detection of Boundary Vertices

```
1: for each vertex i parallely do
2:   for each $v_j$ in $v_i$'s neighbor do
3:     if cannot find $v_i$ in $v_j$'s neighbor then
4:       Mark $v_j$ as a boundary vertex
5:       Add $v_i$ to $v_j$'s neighbor
6:     end if
7:   end for
8: end for
```

4 Mesh optimization on GPU

4.1 Approach overview

To parallelly optimize a polygonized iso-surface on GPU, we adapt the surface flow method [24]. This method moves mesh vertices on a surface without changing its topological connectivity. We modify the three flows in [24] to make it more efficient. These flows optimize vertex positions, normals, and triangle shape (regularity), respectively. Unlike CVT or particle-based methods [15,20], connectivity of the mesh is fixed during our optimization stage with surface flows. This might not be optimal, but keeping connectivity unchanged and explicitly stored is more efficient to access neighbors of a vertex, which is required in the normal adjustment and regularity adjustment. We employ an parallel edge flipping operation to optimize the connectivity and further improve the quality of the mesh. Figure 2 illustrates the pipeline of our optimization stage.

4.2 Vertex position adjustment

The vertices of the initial mesh are not exactly on the iso-surface because the extraction process is approximative. We project the vertices onto the zero-level set as close as possible. Since the vertices are already close to the implicit surface, the Newton method can be used to adjust their positions. For an implicit surface $f(x, y, z) = 0$ and a nearby vertex $\mathbf{P}$, we move it along its gradient direction $\nabla f(\mathbf{P})$ to find a position $\mathbf{P}'$ on the surface. This is equivalent to find t which satisfies

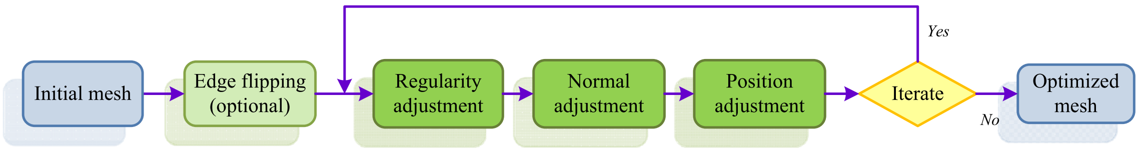


Fig. 2 Overview of our optimization scheme

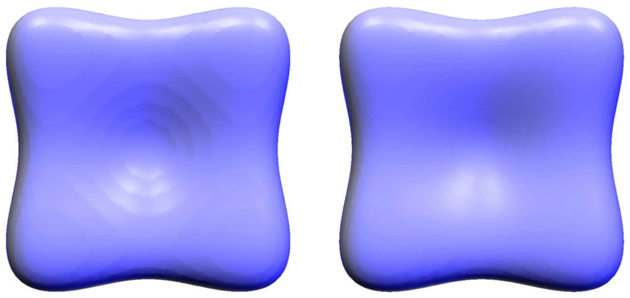


Fig. 3 Vertex position adjustment. The used tooth surface is: $x^4+y^4+z^4-x^2-y^2-z^2=0$. An initial mesh with 41,728 triangles, extracted by MC (*left*). The updated result using ten iterations of Eq. (4), which took 0.570 ms on GPU (*right*)

the following function:

$$f(\mathbf{P}') = f(t\nabla f(\mathbf{P}) + \mathbf{P}) = 0. \tag{1}$$

If $\mathbf{P}$ is close to the solution, i.e., $t\nabla f(v)$ is small, we use a first-order Taylor expansion for approximation as follows:

$$f(\mathbf{P} + t\nabla f(\mathbf{P})) \approx f(\mathbf{P}) + (t\nabla f(\mathbf{P}))\nabla f(\mathbf{P}) = 0. \tag{2}$$

Thus, we get

$$t = -\frac{f(\mathbf{P})}{||\nabla f(\mathbf{P})||^2}, \tag{3}$$

and consequently

$$\mathbf{P}' = \mathbf{P} + t\nabla f(\mathbf{P}) = \mathbf{P} - \frac{f(\mathbf{P})\nabla f(\mathbf{P})}{||\nabla f(\mathbf{P})||^2}. \tag{4}$$

The difference between our vertex position adjustment method and the "Z-flow" method used in [24] is that our approach does not need to compute the step size for gradient decent. Theoretically, iteratively using Eq. (4) can attract vertices to an iso-surface arbitrarily close. In practice, the achieved precision is limited to GPUs. Figure 3 shows the result of optimizing a tooth surface using the vertex position adjustment.

4.3 Regularity adjustment

The above procedure smoothes regions with low curvatures. However, as demonstrated in the top row of Fig. 4, there are

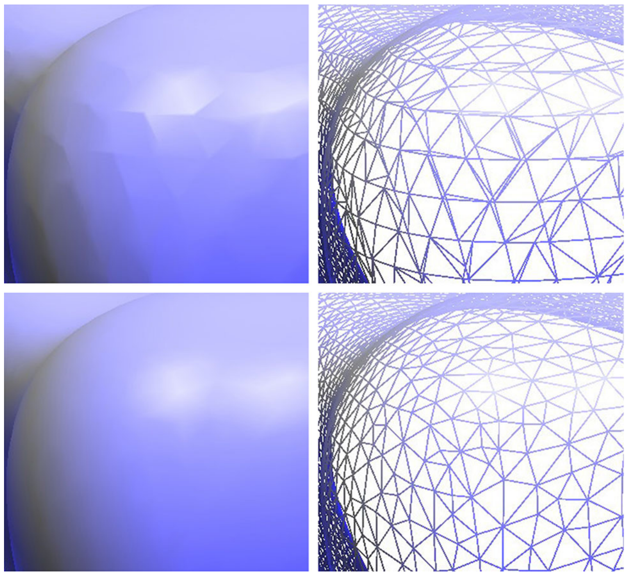


Fig. 4 Regularity adjustment: corner of the tooth surface

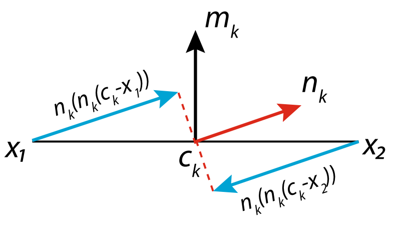
still artifacts in high curvature regions. The naive MC algorithm may produce elongated triangles because the initial mesh could be unevenly sampled. In low curvature regions, the sample rate does not obviously affect the geometry and visualization. On the contrary, it is more non-linear in high curvature regions, which leads to uneven sampling. Since the sampling rate depends only on the 3D regular grid, the unevenly sampled regions will appear to be under-sampled and thus the mesh will not be smooth in such regions. The vertex updating procedure, although giving a better geometrical approximation, does not improve the distribution of mesh vertices, because it operates only on the normal direction. As a result, it is the tangential-direction adjustment that we need for further optimization.

The simplest way is to use the tangential Laplacian operator [5] as follows:

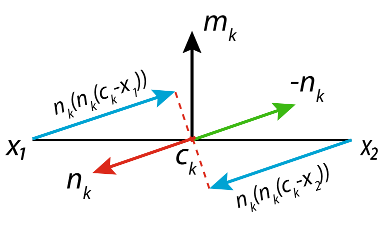
$$\mathbf{P}' = \mathbf{P} + \mathbf{L} - (\mathbf{L}\cdot\mathbf{n})\mathbf{n}, \tag{5}$$

where $\mathbf{L}$ is the discrete Laplacian operator, and $\mathbf{n}$ is the normal. The uniform Laplacian $\mathbf{L} = \frac{1}{n}\sum_{j\in N(i)}(\mathbf{v}_j - \mathbf{v}_i)$ is used here because it is simple and the cotangent Laplacian is parallel to the normal [6].

Ohtake et al. [24] use a similar "R-flow" equation: $\mathbf{P}' = \mathbf{P} + C(\mathbf{L} - (\mathbf{L}\cdot\mathbf{n})\mathbf{n})$ with a step-size coefficient $C = 0.1$.



(a)



(b)

Fig. 5 **a** If the angle between the mesh normal m_k and the target normal n_k is large, the *triangle* (*red dotted line*) will shrink severely. **b** If the two normals form an *obtuse angle*, the *triangle* will result in an opposite normal. In this figure, the *black line* segment and the *black arrow* represent a *triangle* and its normal, respectively. The *red arrow* is the target normal, and the *blue arrows* are the updating vectors that move the vertices to the new positions as the *red dotted line* stands for

A small step size ensures small displacements in the normal direction. On the contrary, we use a large step size (equivalent to $C = 1$) to accelerate the convergence. Enlarged position errors can be reduced by the vertex updating procedure mentioned above. Compared to the method in [24] which needs 20 iterations, an average of 3∼5 iterations are enough in our approach. Another difference between our approach and the work of [24] is that they use the mesh normal while we use the real normal in Eq. (5). Since calculating a discrete normal requires one-ring information, which leads to considerable amount of data transfer, it is preferred to postpone the estimation of discrete normals to the rendering stage. The bottom row of Fig. 4 is the result after one iteration of regularity adjustment with vertex adjustment.

4.4 Normal adjustment

Taubin [33] smoothes a mesh by first smoothing its normal field and then repositioning the mesh vertices according to the new normals. By exploiting the orthogonality between the normal and the three edges of a face, his method uses the gradient descent algorithm to update vertex positions as follows:

$$\mathbf{x}_i' = \mathbf{x}_i + \lambda \sum_{k \in F_V(i)} \sum_{j \in \partial f_k} \mathbf{n}_k(\mathbf{n}_k(\mathbf{x}_j - \mathbf{x}_i)). \tag{6}$$

Sun et al. [30] proved that if $\lambda \le 1/3|F_V(i)|_{\max}$ ($F_V(i)$ is the set of faces that share a common vertex V_i), then the vertex updating Eq. (6) is convergent.

Our approach uses a simplified version suggested in the work of [30], as described below:

$$\mathbf{x}_i' = \mathbf{x}_i + \frac{1}{6} \sum_{k \in F_V(i)} \mathbf{n}_k(\mathbf{n}_k(\mathbf{c}_k - \mathbf{x}_i)), \tag{7}$$

where $\mathbf{c}_k$ is the centroid of triangle k.

We found, however, such iterative normal updating process may generate artifacts if the source normal (the mesh normal) and the target normal diverge greatly. Figure 5 is a 2D illustration of the situation mentioned above. From the figure, we can find that when the angle between the two normals are large, the triangle has a visually noticeable shrinkage. It is even worse when the angle is obtuse as shown in Fig. 5b. Since $(-\mathbf{n}_f)((-\mathbf{n}_f)(\mathbf{c}_f - \mathbf{x}_i))$ is equivalent to $\mathbf{n}_f(\mathbf{n}_f(\mathbf{c}_f - \mathbf{x}_i))$, the resultant normal (the green arrow) will be opposite to the target normal, which may lead to folded triangles. Figure 6 is the result of updating a tooth surface by a sphere normal field, using Eq. (7). Although the shape is still globally correct, some triangles on the sphere have inward normals and thus are displayed in the front-culling mode.

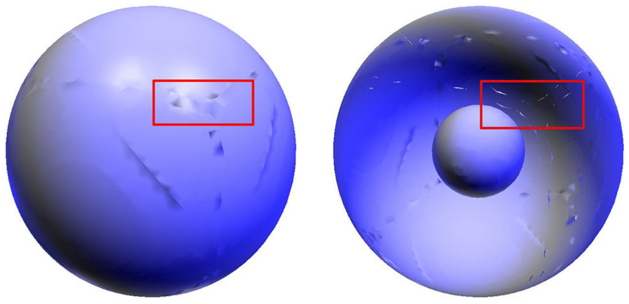


Fig. 6 Updating a tooth surface using a sphere normal field. The input tooth surface (*left*); some *triangles* with negative normals lie on the front surface (*right*), i.e., they are incorrectly flipped

It is noteworthy that applying the regularity adjustment before the normal adjustment can nicely solve the above issue. In other words, this not only evens mesh edges but also expands folded triangles. Figure 7 shows the result of optimizing iso-surfaces extracted from implicit functions.

4.5 Discrete data

In some cases, we do not have an analytic function, or computing function values and/or gradients is computationally expensive. This requires us to use discrete data in the form of, for example, a uniform grid. The above optimization process needs to calculate the function value for an arbitrary position,

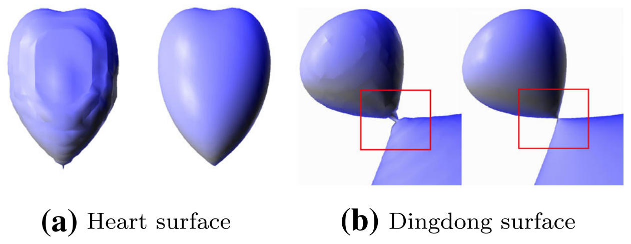


Fig. 7 Iso-surface optimization. Note that fine details are recovered by our approach

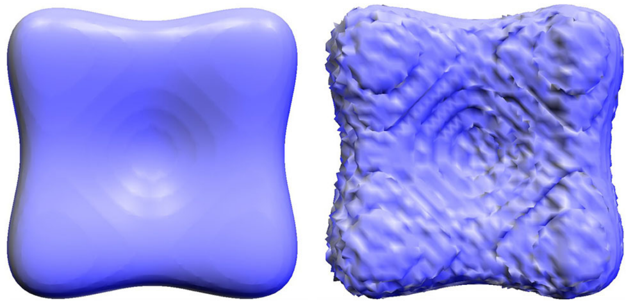


Fig. 8 Artifacts on discrete data. *Left* the initial mesh; *right* the optimized result using a trilinear interpolation instead of the implicit function

which can be achieved by a linear interpolation in discrete situations. However, a linear interpolation approach may result in rough surfaces as shown in Fig. 8. Although a higher order data smoothing approach can be employed, it is costly for some practical applications. Note that optimization for discrete data does not necessarily mean to move a vertex to its corresponding exact position. Therefore, we can optimize the mesh using the regularity adjustment and the normal adjustment only. Figure 9 is the result of extracting and optimizing iso-surface from volumetric data.

4.6 Edge flipping

An initial mesh can be significantly optimized using above-mentioned three surface flows. One advantage of our algorithm is that we construct mesh connectivity after extraction, enabling us to perform the normal optimization and regularity optimization which need one-ring neighbor information. The connectivity information, however, can be more widely used. In this section, we introduce a parallel algorithm that will modify the connectivity. Such a stage is not mandatory, but it further optimizes triangles' shape and distribution of vertices.

A vertex of a triangle mesh is called regular if the number of its neighboring vertices is 6 for interior vertices or 4 for boundary vertices, referred to as the target valence in this writing. According to Euler's formula, fully regular meshes can be generated only for a very limited number of input models, namely those that topologically are (part of) a torus [5]. Our algorithm uses edge flipping, which is more efficient on the GPU than changing the number of vertices and faces.

Computing flipping neighbors We use Eq. (8) to compute the profit value of flipping an edge. If its profit value is larger than a threshold th, this edge needs to be flipped. But this does not mean it will be flipped since we need to ensure that every vertex flips only one linking edge each time. This constraint is exactly enforced to guarantee the consistency among their neighbors for parallelization.

$$\begin{aligned}\text{Profit}(e_{ij}) = {} & \delta(\text{valence}(j) > \text{targetValence}(j)) \\ & + \delta(\text{valence}(h) < \text{targetValence}(h)) \\ & + \delta(\text{valence}(k) < \text{targetValence}(k)), \end{aligned} \tag{8}$$

where $\delta(x)$ is a Boolean function. Note that we do not count in $\delta(\text{valence}(i) > \text{targetValence}(i))$ because we perform profit test only for those vertices whose valences are larger than their target valences. Next, for an edge that passes the profit test, if all of its relevant four vertices are available (i.e., they have no linking edge waiting to be flipped), as illustrated in Fig. 10, then it will be flipped at the next stage. We store vertex j as vertex i's flipping neighbor and vice versa.

Flipping edges For a vertex that has an edge to be flipped, we adjust its neighbor information as well as its right-side

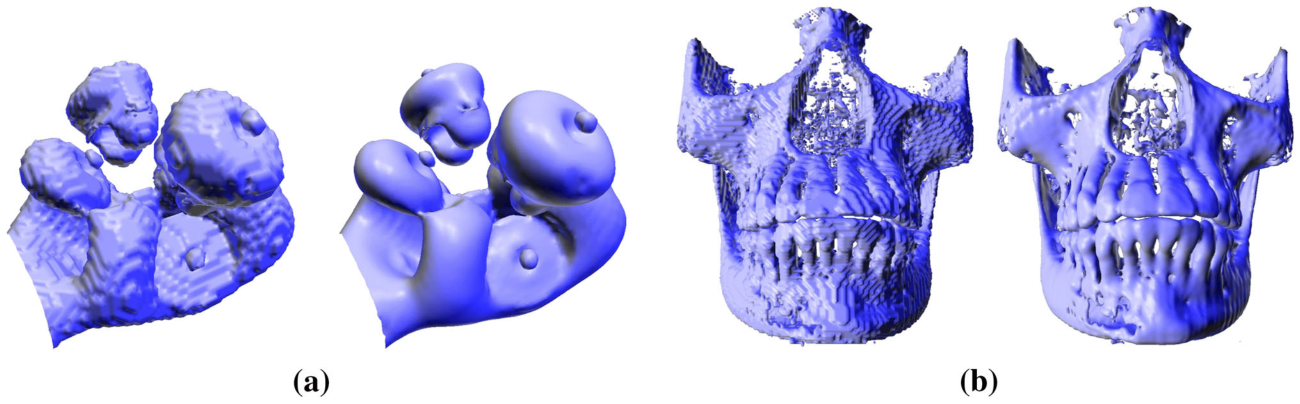
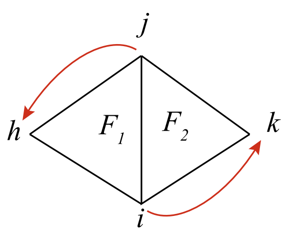


Fig. 9 Iso-surface optimization of volumetric data. **a** A neghip data in a 64^3 volume; **b** a skull data in a 128^3 volume



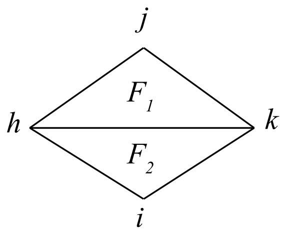


Fig. 10 Illustration of the edge flipping process

vertex's neighbor information, and change its right-side face. The left side is processed by its flipping neighbor (i.e., the other vertex on the flipped edge). Figure 10 illustrates the edge flipping process.

Multi-threshold converging The only parameter in the above algorithm is the threshold th. A simple strategy is to set th as a constant. For example, th=0 means that we perform edge flipping if and only if at least two vertices benefit from it. There is a trade-off when choosing a proper th value. Specifically, if th is too large, the resulting mesh may contain many non-regular vertices; and if it is too small, the algorithm converges slowly and may lead to uneven sampling. In our approach, we use a multi-threshold strategy, which progressively reduces the th value from 2 to 0 (see Fig. 11). Our experimental results show that our strategy requires fewer iterations and can robustly eliminate most non-regular vertices.

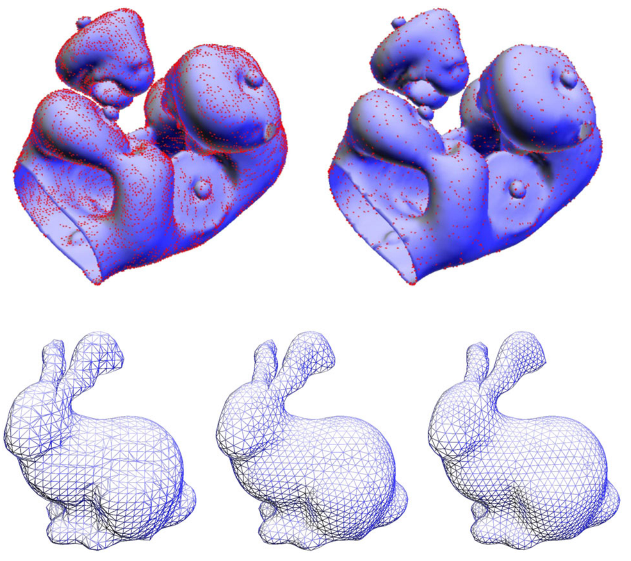


Fig. 11 Optimization of mesh regularity. *Top* a neghip data where non-regular vertices are highlighted in *red*. *Left* the initial mesh; *right* the optimized mesh. *Bottom* a RBF bunny data. *Left* the initial mesh; middle: after regularity adjustment; *right* after edge flipping

5 Results and analysis

We implemented our approach on an off-the-shelf computer with Intel Q9400 2.66GHz CPU. An NVIDIA GeForce GTX 260 GPU with CUDA 3.2 architecture was used for parallelization.

The performance statistics of our algorithm is listed in Table 1. For the extraction stage, the computational complexity of our algorithm is $O(n^3)$, where n is the dimension of the regular grid. For the connectivity reconstruction stage and the optimization stage, the complexity is $O(n_F)$, where n_F is the number of mesh faces. Note that the time spent on the connectivity construction increases dramatically with the mesh size. As mentioned in Sect. 3, this can be explained by atomic operations at this stage, which we do not want to postpone to the next stage. The surface flow time largely depends on the complexity of the implicit function. For example, the diamond surface is a trigonometric surface, while the dingdong surface is a polynomial surface. As a result, their vertex adjustment time is far away from being proportional to their vertex numbers at the optimization stage.

Table 2 reports the achieved CPU/GPU speedup of our approach. Here, we use the diamond surface with different

Table 1 Performance statistics of our approach

Model	# of vertices	# of faces	Extraction	Connectivity construction	Position adjust	Normal adjust	Regularity adjust	Edge flipping	Total
Heart	2,004	4,004	0.893	0.427	0.074	0.085	0.059	1.676	2.41
Tooth	5,232	10,460	0.552	0.819	0.095	0.117	0.082	1.663	2.841
Bucky	7,817	15,088	0.587	1.32	N/A	0.195	0.104	2.47	3.402
Dingdong	10,676	21,078	2.453	2.009	0.145	0.334	0.131	2.414	7.512
Neghip	24,747	49,176	3.339	4.481	N/A	1.114	0.578	3.621	16.28
Engine	76,864	153,812	9.294	12.96	N/A	1.768	0.994	22.51	36.06
Skull	219,280	430,286	12.08	41.54	N/A	4.163	2.316	59.69	86.02
Diamond	545,274	1,078,862	16.81	65.44	2.56	10.48	7.015	73.56	182.5

Position, normal and regularity adjustment are timed for one iteration. For the total time, we use five iterations of the three surface flow, and the time of edge flipping is not included. All timings are given in milliseconds

Table 2 The achieved CPU/GPU speedup for the diamond surfaces with different resolutions

Model	# of vertices	Polygonization			Optimization			Total		
		CPU	GPU	Speedup	CPU	GPU	Speedup	CPU	GPU	Speedup
Diamond_16	8,580	6.36	2.3	2.77	296.62	6.571	45.1	302.98	8.871	34.2
Diamond_32	34,068	44.5	8.47	5.25	938.21	16.565	56.6	982.71	25.035	39.3
Diamond_64	136,236	262.8	30.95	8.49	4644.1	79.65	58.3	4906.9	110.6	44.4
Diamond_128	545,274	1627	111.5	14.5	17859	257.6	69.3	19486	369.1	52.8

All timings are given in milliseconds

Table 3 Running time of our MC and other GPU MC methods

Model	# of faces	Ours	HPDC_10	TVCG_12
Chmutov_64	64 k	3.27	2.13	7.97
Chmutov_128	190 k	9.29	5.46	22.6
Diamond_128	107 k	16.8	8.60	71.7
Diamond_172	1,623 k	29.8	16.7	137.1

All timings are given in milliseconds

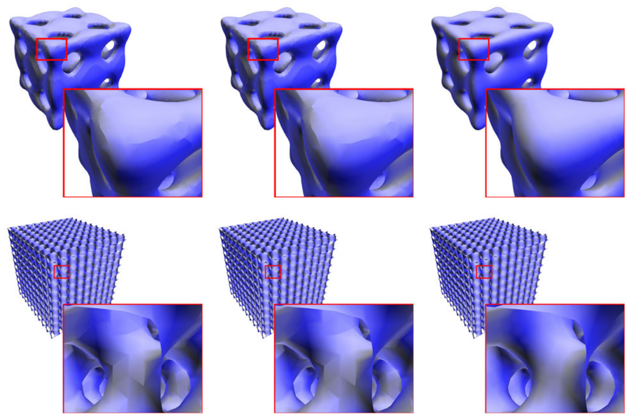


Fig. 12 Extraction and optimization with Chmutov surface (*top row*) and diamond surface (*bottom row*). HPDC_10 (*left*); TVCG_12 (*middle*); ours (*right*)

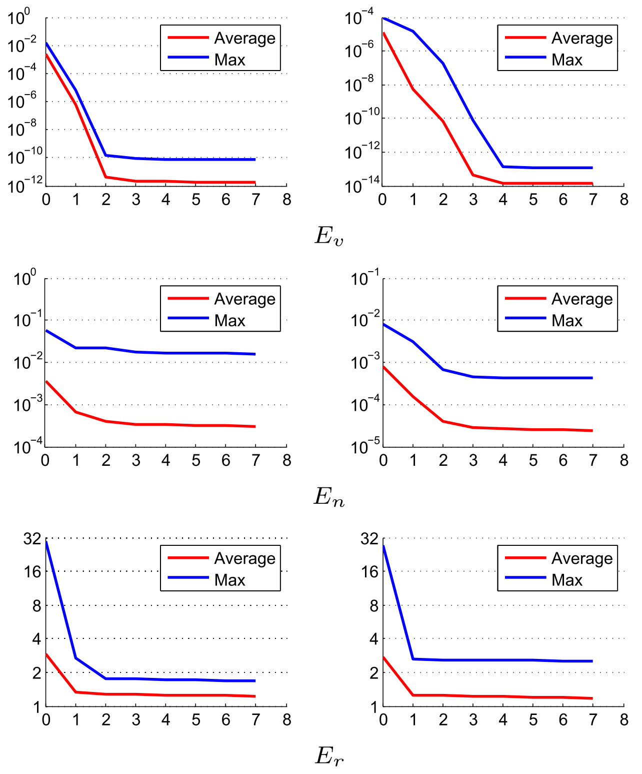


Fig. 13 Average and maximum errors decrease with the iterations. *X*-axis is the number of iterations and *Y*-axis is the error metric. *Left* Chmutov surface; *right column* dingdong surface

resolutions. Our results show that the speedup (i.e., acceleration ratio) achieved by our approach is higher if the resolution of iso-surface is increased.

We compared our algorithm with several existing GPU-based MC algorithms in Fig. 12 and Table 3. HPDC_10 [7] is faster than our implementation because their method only generates a triangle soup and requires fewer GPU kernels. A triangle soup is good enough for visualizing molecular surfaces but too simple for further optimization. Similar to our method, TVCG_12 [10] generates indexed triangles with vertex hashing and binary search. Their method requires fewer memory but is about 2–4 times slower than ours.

For an implicit surface $f(\mathbf{x}) = T$, we introduce three quantitative error metrics to analyze the optimization process:

- *Position error*: $E_v = |f(v) - T|^2$;
- *Normal error*: $E_n = |\nabla f(v) - n(v)|^2$, where $n(v)$ is the normal of vertex v;
- *Regularity*: $E_r = \frac{|t|_\infty}{2\sqrt{3}r}$, where $|t|_\infty$ and r denote the greatest edge length and the inradius of t.

Figure 13 and Table 4 show the effectiveness of our optimization process. The error metrics mentioned above are greatly reduced after optimization. As shown in Fig. 13, errors converge quickly as the iteration goes. Our experiments show that 3∼5 iterations are typically sufficient to get a high-quality mesh for most cases.

Dual Contouring (DC) [13] is one of the most widely used polygonization methods, which optimizes vertex positions to

Table 4 Maximum and average errors of position, normal and regularity of polygonized implicit surfaces before and after optimization

	Position error		Normal error		Regularity error	
	Average	Max	Average	Max	Average	Max
Dindgong_64	8.89e−07/1.40e−14	5.92e−6/2.87e−13	2.08e−04/3.98e−06	3.14e−03/1.38e−04	3.28/0.12	212/3.71
Chmutov_64	2.37e−03/1.94e−12	0.014/1.53e−10	3.53e−03/2.85e−04	0.056/0.018	2.54/0.09	107/0.59

Table 5 Quality comparison with GPU-based dual contouring

	Position error		Normal error		Regularity	
	Dual contouring	Our method	Dual contouring	Our method	Dual contouring	Our method
Chmutov	1.72^{-12} (7.06^{-11})	2.06^{-12} (7.87^{-11})	9.32^{-4} (1.86^{-2})	2.98^{-4} (1.58^{-2})	1.78 (8.91)	1.23 (1.67)
Dingdong	1.31^{-14} (1.74^{-13})	1.27^{-13} (1.33^{-11})	3.46^{-4} (6.08^{-3})	2.43^{-12} (4.33^{-4})	1.82 (28)	1.25 (3.26)

Average errors are listed, with maximum errors given in parentheses

Table 6 Time comparison (in millisecond) with a GPU-based Dual Contouring

Model	# of vertices	Extraction	Connectivity construction	Optimization	Total	GPU dual contouring
Chmutov	23134	3.09	3.12	5.99	12.2	6.80
Dingdong	10559	2.75	1.68	3.05	7.48	6.05

Here we use five iterations in optimization

get a high-quality iso-surface. We compared our optimization with the GPU Dual Contouring algorithm (DC) [28]. Table 5 shows that the position errors are similar for the two algorithms. However, our approach produces smaller normal and regularity errors. Table 6 shows the time comparison of the two algorithms. The main difference between our algorithm and DC is that ours has a connectivity reconstruction stage while DC cannot deal with one-ring information. Our algorithm is relatively slower than the GPU-based DC due to the connectivity reconstruction. However, this stage is the reason that we can optimize not only the vertex positions but also normals, triangle aspect ratio and other potential surface flows that require neighbor information.

6 Conclusion and future work

In this paper, we present an efficient and practical GPU-based approach to polygonize and optimize iso-surfaces for implicit surfaces. Our experimental results show that our approach can robustly generate high-quality triangle meshes efficiently, and our GPU-based approach is approximately an order of magnitude faster than its CPU counterpart. Furthermore, the achieved speedup becomes even higher if the resolution of the iso-surface is increased. Our method can be used for implicit function and volume data, and is possible to be extended to other data models such as offset surface [34,35]. Our approach can also be applied in sketch-based modeling using implicit surfaces to get high-quality tesselated meshes [37].

Although our algorithm is effective and efficient, it has some limitations. First, if we perform a connectivity surgery, such as the edge flipping algorithm in Sect. 4.6, although the regularity of a mesh can be further improved, the increased time is not negligible. Second, due to our explicit storage of neighbor information, memory consumption will be a problem for large meshes.

Our approach can be further improved in several directions. First, to efficiently handle some cases where high-resolution initial meshes are input or we need to deal with very large volume datasets, incorporating a divide-and-conquer strategy and GPU memory management becomes necessary, which is absent in our current approach. A possible way to deal with large data can be found in [17]. Second, how to reduce the GPU bandwidth of data transmission when iteratively optimizing the mesh remains to be further explored, and we would like to leave it for future work. Third, at the mesh regularity adjustment step, our current data structure can be further improved so that no atomic operation is needed. In the future, we will explore the possibility of mesh optimization by locally fitting an implicit surface and evolving the noisy mesh towards it.

Acknowledgments Xiaogang Jin was supported by the National Natural Science Foundation of China (Grant No. 61272298) and Zhejiang Provincial Natural Science Foundation of China (Grant No. Z1110154). Zhigang Deng was supported by the Joint Research Fund for Overseas

Chinese, Hong Kong and Macao Young Scientists of the National Natural Science Foundation of China (Grant No. 61328204).

References

1. Alliez, P., Ucelli, G., Gotsman, C., Attene, M.: Recent Advances in Remeshing of Surfaces. Shape Analysis and Structuring, pp. 53–82. Springer, Berlin Heidelberg (2008)
2. Bischoff, S., Kobbelt, L.: Isosurface reconstruction with topology control. In: Proceedings of the 10th Pacific Conference on Computer Graphics and Applications, pp. 246–255 (2002)
3. Bloomenthal, J.: An Implicit Surface Polygonizer. Graphics gems IV, pp. 324–349. Academic Press Professional Inc, San Diego (1994)
4. Bloomenthal, J.: Polygonization of implicit surfaces. Comput. Aided Geom. Des. **5**(4), 341–355 (1988)
5. Botsch, M., Kobbelt, L., Pauly, M., Alliez, P., Lévy, B.: Polygon Mesh Processing. A K Peters Ltd., USA (2008)
6. Desbrun, M., Meyer, M., Schröder, P., Barr, A.: Implicit fairing of irregular meshes using diffusion and curvature flow. In: Proceedings of the 26th annual Conference on Computer Graphics and Interactive Techniques, pp. 317–324 (1999)
7. Dias, S., Bora, K., Gomes, A.: CUDA-based triangulations of convolution molecular surfaces. In: Proceedings of the 19th ACM International Symposium on High Performance, Distributed Computing, pp. 531–540 (2010)
8. Fleishman, S., Drori, I., Cohen-Or, D.: Bilateral mesh denoising. ACM Trans. Graph. **22**(3), 950–953 (2003)
9. Gomes, A., Voiculescu, I., Jorge, J., Wyvill, B., Galbraith, C.: Implicit Curves and Surfaces: Mathematics, Data Structures and Algorithms. Springer, Berlin (2009)
10. Griffin, W., Wang, Y., Berrios, D., Olano, M.: Real-time GPU surface curvature estimation on deforming meshes and volumetric data sets. IEEE Trans. Vis. Comput. Graph. **18**(10), 1603–1613 (2012)
11. Hatmann, E.: A marching method for the triangulation of surfaces. Vis. Comput. **14**(3), 95–108 (1998)
12. Jones, T., Durand, F., Desbrun, M.: Non-iterative, feature-preserving mesh smoothing. ACM Trans. Graph. **22**(3), 943–949 (2003)
13. Ju, T., Losasso, F., Schaefer, S., et al.: Dual contouring of hermite data. ACM Trans. Graph. **21**(3), 339–346 (2002)
14. Lempitsky, V.: Surface extraction from binary volumes with higher-order smoothness. In: IEEE Conference on Computer Vision and Pattern Recognition, pp. 1197–1204 (2010)
15. Liu, Y., Wang, W., Lévy, B., et al.: On centroidal Voronoi tessellation-energy smoothness and fast computation. ACM Trans. Graph. **28**(4), 101 (2009)
16. Liu, Y., Yuen, M.: Optimized triangle mesh reconstruction from unstructured points. Vis. Comput. **19**(1), 23–37 (2003)
17. Liu, Y., Yuen, M., Tang, K.: Manifold-guaranteed out-of-core simplification of large meshes with controlled topological type. Vis. Comput. **19**(7–8), 565–580 (2003)
18. Löffler, F., Schumann, H.: Generating smooth high-quality isosurfaces for interactive modeling and visualization of complex terrains. In: Proceedings of the Vision, Modeling, and Visualization Workshop (2012)
19. Lorensen, W., Cline, H.: Marching cubes: a high resolution 3D surface construction algorithm. ACM Comput. Graph. **21**(4), 163–169 (1987)
20. Meyer, M., Georgel, P., Whitaker, R.: Robust particle systems for curvature dependent sampling of implicit surfaces. In: Shape Modeling and Applications, pp. 124–133 (2005)
21. Meyer, M., Kirby, R., Whitaker, R.: Topology, accuracy, and quality of isosurface meshes using dynamic particles. IEEE Trans. Vis. Comput. Graph. **13**(6), 1704–1711 (2007)
22. Nielson, M.: Dual marching cubes. In: IEEE Visualization, pp. 489–496 (2004)
23. Nielson, G., Hamann, B.: The asymptotic decider: resolving the ambiguity in marching cubes. In: Proceedings of the 2nd Conference on Visualization, pp. 83–91 (1991)
24. Ohtake, Y., Belyaev, A.: Mesh optimization for polygonized isosurfaces. Comput. Graph. Forum **20**(3), 368–376 (2001)
25. Rong, G., Jin, M., Guo, X.: Hyperbolic centroidal voronoi tessellation. In: Proceedings of the 14th ACM Symposium on Solid and Physical Modeling, pp. 117–126 (2010)
26. Rong, G., Liu, Y., Wang, W., et al.: GPU-assisted computation of centroidal Voronoi tessellation. IEEE Trans. Vis. Comput. Graph. **17**(3), 345–356 (2011)
27. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore GPUs. In: Processing of IEEE International Symposium on Parallel and Distributed, pp. 1–10 (2009)
28. Schmitz, A., Dietrich, A., Comba, D.: Efficient and high quality contouring of isosurfaces on uniform grids. In: IEEE XXII Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI), pp. 64–71 (2009)
29. Shirazian, P., Wyvill, B., Duprat, J.: Polygonization of implicit surfaces on multi-core architectures with SIMD instructions. In: Eurographics Symposium on Parallel Graphics and Visualization, pp. 89–98 (2012)
30. Sun, X., Rosin, P., Martin, R., Langbein, F.: Fast and effective feature-preserving mesh denoising. IEEE Trans. Vis. Comput. Graph. **13**(5), 925–938 (2007)
31. Tatarchuk, N., Shopf, J., DeCoro, C.: Real-time isosurface extraction using the GPU programmable geometry pipeline. In: ACM SIGGRAPH 2007 courses, pp. 122–137 (2007)
32. Taubin, G.: A signal processing approach to fair surface design. In: Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques, pp. 351–358 (1995)
33. Taubin, G.: Linear anisotropic mesh filtering. IBM Research Report RC22213 (W0110–051). IBM T. J. Watson Research Center (2001)
34. Wang, C., Leung, Y., Chen, Y.: Solid modeling of polyhedral objects by layered depth-normal images on the GPU. Comput. Aided Des. **42**(6), 535–544 (2010)
35. Wang, C., Manocha, D.: GPU-based offset surface computation using point samples. Comput. Aided Des. **45**(2), 321–330 (2012)
36. Zhong, Z., Guo, X., Wang, W., et al.: Particle-based anisotropic surface meshing. ACM Trans. Graph. **32**(4), 99 (2013)
37. Zhu, X., Jin, X., Liu, S., et al.: Analytical solutions for sketch-based convolution surface modeling on the GPU. Vis. Comput. **28**(11), 1115–1125 (2013)
38. http://code.google.com/p/cudpp/
39. https://developer.nvidia.com/cuda-downloads

Junjie Chen is a PhD candidate of the State Key Lab of CAD&CG, Zhejiang University. He received his BSc degree from Dalian University of Technology in 2008. His research interests include implicit surface modeling and geometric processing.

Xiaogang Jin is a professor of the State Key Lab of CAD&CG, Zhejiang University. He received his BSc degree in computer science in 1989, MSc and PhD degrees in applied mathematics in 1992 and 1995, all from Zhejiang University. His current research interests include implicit surface computing, special effects simulation, mesh fusion, texture synthesis, crowd animation, cloth animation and facial animation.

Zhigang Deng is currently an Associate Professor of Computer Science at the University of Houston (UH) and the Founding Director of the UH Computer Graphics and Interactive Media (CGIM) Lab. His research interests include computer graphics, computer animation, virtual human modeling and animation, and human computer interaction. He earned his Ph.D. in Computer Science at the Department of Computer Science at the University of Southern California in 2006.