GPU-Assisted Computation of Centroidal Voronoi Tessellation

Guodong Rong, Yang Liu, Wenping Wang, Xiaotian Yin, Xianfeng David Gu, and Xiaohu Guo

Abstract—Centroidal Voronoi tessellations (CVT) are widely used in computational science and engineering. The most commonly used method is Lloyd's method, and recently the L-BFGS method is shown to be faster than Lloyd's method for computing the CVT. However, these methods run on the CPU and are still too slow for many practical applications. We present techniques to implement these methods on the GPU for computing the CVT on 2D planes and on surfaces, and demonstrate significant speedup of these GPU-based methods over their CPU counterparts. For CVT computation on a surface, we use a geometry image stored in the GPU to represent the surface for computing the Voronoi diagram on it. In our implementation a new technique is proposed for parallel regional reduction on the GPU for evaluating integrals over Voronoi cells.

Index Terms—Centroidal Voronoi Tessellation, Graphics Hardware, Lloyd's Algorithm, L-BFGS Algorithm, Remeshing.

1 Introduction

Voronoi diagrams are well studied in computational geometry and have many applications in areas like computer graphics, visualization, pattern recognition, etc. [1], [2]. An evenly-spaced tessellation of a given domain Ω is produced by a special type of Voronoi diagram, called *Centroidal Voronoi Tessellation* (CVT); see for example Fig. 1. The uniformity of the cells of an optimal CVT has been conjectured by Gersho [3] and proved in 2D [4], while confirmed empirically in 3D [5]. This property makes the CVT useful in many applications, including graph drawing [6], decorative arts simulation [7], [8], [9], grid generation and optimization [10], vector field visualization [11], [12], surface remeshing [13], [14], [15], [16] and medial axis approximation [17]. In this paper we study how to speed up the computation of the CVT using the GPU.

1.1 Preliminaries

We first present the definition and properties of the CVT and some typical algorithms for computing the CVT. More details about the CVT can be found in the survey [18].

Given n sites $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ in a domain $\Omega \subset \mathbb{R}^N$, the Voronoi diagram is defined as the collection of the *Voronoi cells* Ω_i, $i = 1, 2, \ldots, n$, defined as

$$\Omega_i = \{\mathbf{x} \in \Omega : \|\mathbf{x} - \mathbf{x}_i\| < \|\mathbf{x} - \mathbf{x}_j\|, i \neq j\}.$$

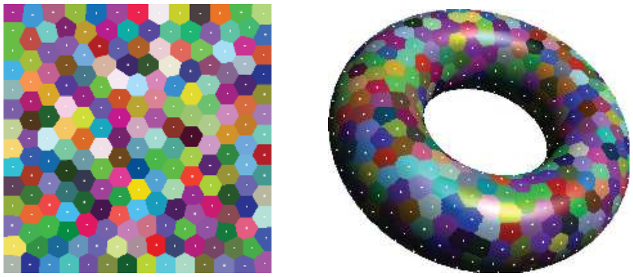


Fig. 1. (a) The CVT of a square 2D domain with 200 sites; (b) The constrained CVT on a torus with 600 sites.

The *centroidal Voronoi tessellation* is a special Voronoi diagram in which each site $\mathbf{x}_i$ coincides with the centroid $\mathbf{c}_i$ of its Voronoi cell Ω_i:

$$\mathbf{c}_i = \frac{\int_{\Omega_i} \rho(\mathbf{x})\mathbf{x}\,\mathrm{d}\sigma}{\int_{\Omega_i} \rho(\mathbf{x})\,\mathrm{d}\sigma}, \tag{1}$$

where $\rho(\mathbf{x}) > 0$ is a density function. An example of a CVT of 200 sites in a square is shown in Fig. 1(a).

The CVT energy function F is defined for the *ordered set* of samples (i.e. sites) $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$ in Ω as

$$F(\mathbf{X}) = \sum_{i=1}^{n} f_i(\mathbf{X}) = \sum_{i=1}^{n} \int_{\Omega_i} \rho(\mathbf{x})\|\mathbf{x} - \mathbf{x}_i\|^2\,\mathrm{d}\sigma, \tag{2}$$

where the Ω_i are the Voronoi cells of the sites $\mathbf{x}_i$. It can be shown [18] that the regions Ω_i of the sites $\mathbf{x}_i$ form a CVT in the domain Ω if and only if the gradient of $F(\mathbf{X})$ vanishes, that is, a critical point of $F(\mathbf{X})$. Therefore, a necessary condition for F to be locally minimized is that the regions Ω_i form a CVT. A CVT that locally minimizes F will be called a *stable CVT*. In practice one often seeks a stable CVT since it usually produces a more regular and compact tessellation than a CVT that is not a local minimizer of F.

- *G. Rong and X. Guo are with the Department of Computer Science, University of Texas at Dallas, Richardson, TX, USA.*
  E-mail: {guodongrong, xguo}@utdallas.edu.
- *Y. Liu is with Project ALICE, INRIA, Villers les Nancy, France.*
  E-mail: liuyang@loria.fr.
- *W. Wang is with the Department of Computer Science, University of Hong Kong, Hong Kong, China.*
  E-mail: wenping@cs.hku.hk.
- *X. Yin and X. Gu are with the Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY, USA.*
  E-mail: {xyin, gu}@cs.sunysb.edu.

Furthermore, a CVT that globally minimizes F is called an *optimal CVT*. Because there are a large number of stable CVTs when the number of sites is large, in general, it is difficult to compute an optimal CVT.

For a non-convex domain, the centroid of a cell Ω_i computed by (1) may lie outside Ω_i. In this case the centroid is replaced by the *constrained centroid* inside Ω_i, defined as

$$\mathbf{c}_i^* = \arg\min_{\mathbf{p}\in\Omega_i} \int_{\Omega_i} \rho(\mathbf{x})||\mathbf{x}-\mathbf{p}||^2 \,\mathrm{d}\sigma. \tag{3}$$

When all the sites coincide with the constrained centroids within the domain Ω, the CVT is called a *constrained centroidal Voronoi tessellation* (CCVT) [19]. The CCVT can also similarly be defined on a surface $S \subset \mathbb{R}^3$, i.e. by constraining all the sites to lie on the surface. Fig. 1(b) shows an example of a CCVT of 600 sites on a torus.

Two commonly used algorithms for computing the CVT are Lloyd's algorithm [20] and the L-BFGS algorithm [21]. Although converging faster than Lloyd's algorithm, the L-BFGS algorithm still needs a long time to compute a CVT. In this paper we describe how to leverage the parallel computational power of the programmable graphics processing unit (GPU) to speed up these two methods for computing the CVT in 2D and on surfaces.

Using the GPU for computing the CVT in 2D is a straightforward idea, since a 2D domain can be represented naturally by a 2D texture in the GPU. The idea of computing the CCVT on a surface is similar, but we need to first construct a parametric representation of the surface using the geometry image. With the geometry image represented as a texture in the GPU, we run the jump flooding algorithm [22] to compute the Voronoi diagrams in each iteration of CVT computation. Here the pixels of the geometry image store the 3D coordinates of sampled points on the surface and we use the Euclidean distance in 3D for computing the Voronoi diagram. We note that this is different from the method in [15], which computes Voronoi diagrams using distances in a 2D parametric domain.

1.2 Contributions

Our contributions are efficient implementations of Lloyd's algorithm and the L-BFGS algorithm on the GPU for the computation of the CVT in 2D and on a surface. We propose a new technique for computing Voronoi diagrams on surfaces and a novel way of using vertex programs to perform the regional reduction over Voronoi cells. Significant speedup is achieved by our GPU programs in various cases of CVT computation.

All our GPU programs are implemented with the shader language Cg. Although general purpose languages on the GPU (e.g. CUDA) are more popular now, our tests show that Cg is better suited for implementing the algorithms for CVT computation. We will explain the reasons behind this in more details in Section 5.4.

The remainder of the paper is organized as follows: Section 2 briefly reviews related work. Section 3 explains how to compute the CVT on a 2D plane with the GPU. The idea is then extended in Section 4 to computing the CCVT on surfaces. The experimental results and comparisons are given in Section 5. Section 6 concludes the paper with discussions of future research.

2 Previous Work

We will briefly review existing algorithms on the CPU for computing the CVT. We also give a brief survey of previous work on using the GPU to compute the Voronoi diagram.

2.1 CVT Algorithms

MacQueen's probabilistic method is one of the earliest methods for computing the CVT [23]. Ju et al. integrated MacQueen's method and Lloyd's algorithm on a parallel platform [24]. The most commonly used algorithm for computing the CVT in 2D/3D is Lloyd's algorithm [20] for its simplicity and robustness. However this method has linear convergence and is very slow in practice. The multi-grid method has been proposed to accelerate Lloyd's algorithm [25].

It has recently been shown that the CVT energy function is C^2 continuous [21]. Justified by the C^2 smoothness of the CVT energy function, Liu et al. applied a quasi-Newton method – the L-BFGS algorithm – to computing the CVT in 2D, 3D and on surfaces, and showed that the algorithm is faster than Lloyd's algorithm [21].

2.2 GPU Algorithms

With the rapid advance of the GPU, the general-purpose computation on the GPU (GPGPU) has become an active topic [26]. In the following we will review previous work on using the GPU to compute Voronoi diagrams.

Hoff et al. [27] built a right-angle cone for every site and rendered them from bottom to get a Voronoi diagram of these sites. Denny's method [28] is similar to Hoff et al.'s but changes the cones to depth textures to get better quality and speed. Fischer and Gotsman [29] lifted the sites to a paraboloid and rendered planes tangent to the paraboloid to obtain the Voronoi diagram, thus avoiding tessellating the cones. Note that the Voronoi diagrams computed by GPU algorithms in 2D are defined by color-coded pixel maps, hence they are only discrete approximations to the true Voronoi diagrams defined by polygons. All these algorithms are designed for computation in 2D, and it is not clear how to extend them to compute the Voronoi diagram on a surface.

The jump flooding algorithm (JFA) [22] is another method for computing the Voronoi diagram in a 2D discrete domain represented in pixels. The JFA propagates information (e.g. coordinates of the sites in the Voronoi diagram problem) from the sites to all other pixels in parallel, similar to the flood-filling algorithm, but with faster speed due to its use of varying step lengths. Cao et al. [30] proposed the parallel banding algorithm (PBA) which is faster than the JFA. But it is not clear how to extend the PBA to compute the Voronoi diagram on a surface. We will use the JFA to compute the Voronoi diagram in 2D as well as on a surface. Like [19], [21] we use the Euclidean distance to approximate the geodesic distance on a surface.

To compute the Voronoi diagram on a surface, one may compute a 3D Voronoi diagram and find its intersection with the surface. The GPU has also been used to compute 3D Voronoi diagrams [27], [31], [32], [33], [34]. Weber et al. [35] adopted the raster scan method in the geometry image to compute the Voronoi diagram using the geodesic distance on a surface. This method handles disk-like open surfaces only. The geodesic distance, though more accurate than the Euclidean distance, is much less efficient to compute on a mesh surface, even with GPU acceleration.

Vasconcelos et al.'s work [36] is the only known successful attempt so far using the GPU to compute the CVT in a plane. They implemented the Lloyd's method on the GPU to compute the CVT on a 2D plane, focusing on the computation of the centroids. A predefined mask is used to conservatively estimate the Voronoi cell for every site. Since the diameter of a Voronoi cell may be very big, to ensure an accurate result, the mask must be as big as the whole texture, which makes the method inefficient. Bollig [37] also proposed a similar algorithm computing the CVT on the GPU, but his method is prone to errors for regions with high curvature.

In contrast, we use the vertex program to perform scattering and use the framebuffer blending function to accumulate the coordinates. Our approach is simpler and works well for computing both the CVT on a 2D plane and the CCVT on a surface.

3 CVT on 2D Plane

There are two main steps in both Lloyd's algorithm and the L-BFGS algorithm: 1) computing the Voronoi diagram of the current sites, and 2) finding new positions of the sites for the next iteration. For step 1, we compute a discrete approximation of the Voronoi diagram using the jump flooding algorithm (JFA) [22] on the GPU. We propose a new regional reduction method for efficiently computing various integrals needed in step 2. In the following we will briefly review the JFA and present the regional reduction technique.

3.1 Jump Flooding Algorithm

Suppose that there is a site in an $n \times n$ texture in the GPU and we want to propagate some information (e.g. the coordinates of the site) from the site to all the other pixels. The JFA performs this in several passes. For an initial step length k that is a power of 2 (e.g. $2^{\lceil \log n \rceil}$), a pixel at (x, y) passes its information to its neighbors (eight at most) at $(x+i, y+j)$, where $i, j \in \{-k, 0, k\}$ (Fig. 2). Then in the subsequent passes, the same propagation is performed for a pixel using the step length that is half of the previous step length k. The iteration is stopped when the step length reaches 1. Fig. 2 illustrates how the JFA fills up an 8×8 texture using three passes with the single initial site at the bottom left corner.

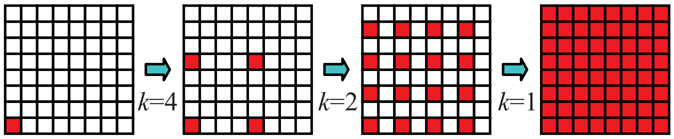


Fig. 2. The iterations of the JFA for an 8×8 texture with an initial site at the bottom left corner.

When using the JFA to compute the Voronoi diagram in a 2D texture, there are multiple sites and the information to be propagated from each site is its coordinates. Upon receiving the coordinates of different sites, each pixel compares its distances to these sites to find the nearest site whose Voronoi cell it belongs to. Thus all the pixels are classified to form a Voronoi diagram.

Despite its fast speed, the JFA may misclassify a small number of pixels [22]. We use 1+JFA [34], an improved version of JFA, to compute the Voronoi diagram in our GPU implementations. On average, the error rate of 1+JFA is less than 0.25 pixels in a texture with the resolution of 512×512 for less than 10,000 sites, which is accurate enough for most practical graphics applications utilizing the CVT. To be brief, we will refer 1+JFA as JFA as well.

A sufficiently large initial step length is needed to ensure that each pixel is reached by its nearest site. On the other hand, one should try to use a small but "safe" initial step length to reduce the computation time incurred by unnecessary JFA passes. A safe choice is $2^{\lceil \log n \rceil}$, which is necessary for computing a Voronoi diagram when the sites are distributed in such a way that each Voronoi cell is narrow and long, as the cells generated by a sequence of collinear sites. However, this initial step length is, overall, very conservative because after a few iterations of CVT computation (with either Lloyd's algorithm or the L-BFGS algorithm) the sites are normally already distributed rather evenly, therefore a much smaller initial step length would be sufficient for each pixel to be reached by its nearest site.

This consideration leads us to use the following scheme for selecting the initial step length. In the VD computation of the first CVT iteration, the initial step length of the JFA is set to be $2^{\lceil \log n \rceil}$. For the VD computation in the next t CVT iterations, we compute the average distance from each site to all the pixels in its Voronoi cell using the regional reduction (to be introduced in next subsection), and set the initial step length of the JFA to be the double of the maximum of all these average distances. Our experiments show that $t = 5$ gives satisfactory performance. Then in the

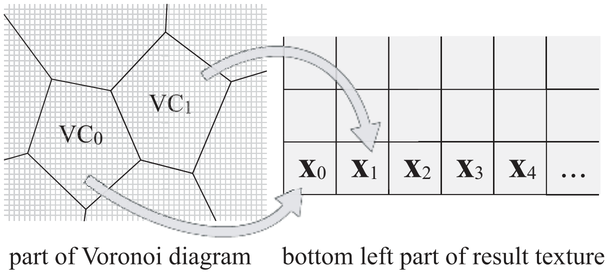


Fig. 3. Illustration of regional reduction. All the points in the Voronoi cell i (VC$_i$) are translated to the pixel corresponding to the site $\mathbf{x}_i$.

remaining CVT iterations (i.e. after 5 iterations) the initial step length of the JFA is set to be that used in the 5th iteration.

3.2 Regional Reduction

Several different integrals over the Voronoi cells of all the sites need to be evaluated in CVT computation, that is, for computing the value of the CVT energy function or computing the centroid as needed by the Lloyd algorithm. To approximate these integrals, we need to perform summations over the pixels of all the Voronoi cells in parallel, which calls for solving the so-called *regional reduction* problem.

In the reduction problem, one needs to reduce a number of values to a single one, such as sum, maximum, minimum, etc. More precisely, the reduction problem takes as input a set of values $v_1, \ldots, v_n$ and outputs a single value $v = v_1 \oplus \ldots \oplus v_n$, where $\oplus$ is an associative and commutative operator. Thus, summation is a special case of the reduction problem.

The reduction problem can be solved in $O(\log n)$ passes on the GPU using a fragment program [38]. Existing algorithms can only perform global reductions, that is, reducing values of *all* the pixels in a texture into one single value. However, in CVT computation the domain Ω is decomposed into multiple regions (that is, Voronoi cells) Ω_i and we need to compute the sums of values of the pixels of different regions in parallel. Therefore, we face a *regional reduction* problem rather than a global one.

We propose to use the method of rendering points for regional reduction. A single point is rendered for every pixel, and its position is changed in the vertex program. All the points in the Voronoi cell of site $\mathbf{x}_i$ are translated to the same position decided by its ID i. For example, the site $\mathbf{x}_i$ corresponds to the position $(i/w, i\%w)$, where w is the width of the texture used and "/" and "%" are division and modulus operators for integers, respectively. The result is a texture containing the reduction values for all the Voronoi cells, one pixel for each site, which are packed in the order of the site's ID. Fig. 3 shows an illustration of this operation.

First, every point is rasterized into one fragment. The fragment program processing these fragments then writes the values to a texture recording the results. The depth test or framebuffer blending operations can be used to reduce all the values corresponding to the same pixel to a single value which is stored in a result texture. For example, if we want to compute the maximum of the values, we can write the values into the depth channel as well as the color channels for every fragment, and set the depth test function so that only the fragment with the maximum value is stored into the result texture (e.g. using `glDepthFunc(GL_GREATER)`). If we want to compute the sum of the values, we can write the values into color channels for every fragment, and set the blending function so that the values of all the fragments are added and the result is stored into the result texture (e.g. using `glBlendFunc(GL_ONE, GL_ONE)`).

Our regional reduction method works for a connected region as well as a set consisting of disconnected regions. This is important because the Voronoi diagram of a surface may contain disconnected Voronoi cells near the boundaries of the geometry image (see Section 4). Furthermore, a Voronoi cell in a pixel plane may contain disconnected parts [39]; such a case can be handled properly by our regional reduction method.

3.3 Lloyd's Algorithm in 2D

Every iteration in Lloyd's algorithm contains two steps: 1) computing the Voronoi diagram of the current sites; and 2) computing the centroids for Voronoi cells as the new sites in the next iteration. These two steps are iterated until certain termination condition is met.

On a 2D plane the centroids are computed using (1). Using regional reduction the integrations in (1) are approximated by summations, so we have

$$\mathbf{c}_i = \frac{\sum_{\mathbf{x}\in\Omega_i} \rho(\mathbf{x})\mathbf{x}\,\Delta\sigma}{\sum_{\mathbf{x}\in\Omega_i} \rho(\mathbf{x})\,\Delta\sigma} = \frac{\sum_{\mathbf{x}\in\Omega_i} \rho(\mathbf{x})\mathbf{x}}{\sum_{\mathbf{x}\in\Omega_i} \rho(\mathbf{x})}, \tag{4}$$

where the $\mathbf{x}$ is the pixel position and $\Delta\sigma$ the constant area occupied by each pixel.

The numerator and denominator in (4) are computed using the regional reduction in the same pass since they share the same domain. The numerator is a 2D vector stored in red and green channels of the texture, and the denominator is a scalar value stored in the blue channel.

We stress that the coordinates of each site of the Voronoi diagram are floating point numbers, albeit they are stored in the discrete pixel closest to it. Since the Voronoi cells are composed of pixels, the Voronoi boundaries have only pixel precision. The algorithm will stop when the Voronoi diagram does not change. Alternatively, a different termination condition can be used by checking if the value of CVT energy function F or its gradient has reached a threshold.

3.4 L-BFGS Algorithm in 2D

The L-BFGS algorithm is a quasi-Newton algorithm that is more efficient than Lloyd's method for CVT computation [40], [21]. In every iteration of the L-BFGS algorithm,

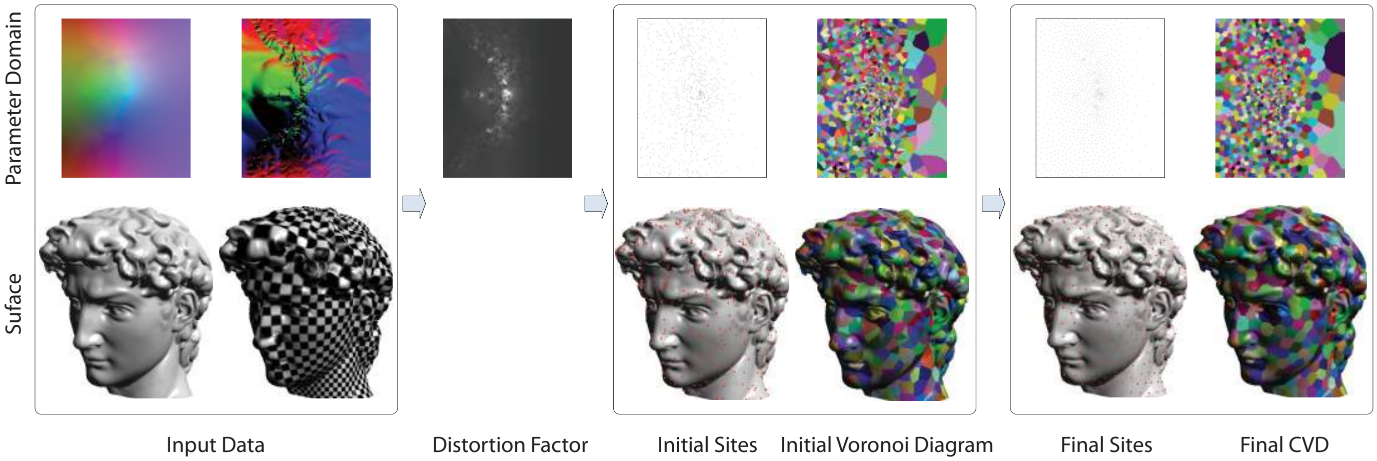


Fig. 4. The pipeline for computing a CCVT on a surface. From left to right: the input David Head model together with its geometry image ((r,g,b)=(x,y,z)) and normal vector image ((r,g,b)=(n_x, n_y, n_z)), distortion factors (modulated for visual clarity), the initial sites and the corresponding Voronoi diagram, and the final sites with the CCVT. The top row shows the parameter domain, and bottom row shows the surface.

we need to compute the CVT energy F and its partial derivatives with respect to all the sites. These values are accumulated with those values of the m preceding iterations to approximate the inverse Hessian matrix.

More specifically, define $\mathbf{s}_k = \mathbf{X}_k - \mathbf{X}_{k-1}$ and $\mathbf{y}_k = \nabla F_k - \nabla F_{k-1}$, where $\mathbf{X}_k$ and ∇F_k are the ordered set of sites and the gradient of the CVT energy function F at the k-th iteration, respectively. Then the approximated inverse Hessian matrix $\mathbf{H}_k$ is updated by

$$\mathbf{H}_k = \mathbf{V}_k^T \mathbf{H}_{k-1} \mathbf{V}_k + \rho_k \mathbf{s}_k \mathbf{s}_k^T \tag{5}$$

where $\rho_k = 1/(\mathbf{y}_k^T \mathbf{s}_k)$, and $\mathbf{V}_k = \mathbf{I} - \rho_k \mathbf{y}_k \mathbf{s}_k^T$. The new sites $\mathbf{X}_{k+1}$ for the next iteration is given by $\mathbf{X}_{k+1} = \mathbf{X}_k - \mathbf{H}_k \nabla F_k$. More details of the L-BFGS algorithm are given in [21].

The first-order partial derivative of F with respect to the site $\mathbf{x}_i$ is [2]:

$$\frac{\partial F}{\partial \mathbf{x}_i} = 2 \int_{\Omega_i} \rho(\mathbf{x})(\mathbf{x}_i - \mathbf{x}) \, \mathrm{d}\sigma. \tag{6}$$

This integral is again approximated by summation as:

$$\frac{\partial F}{\partial \mathbf{x}_i} = 2 \sum_{\mathbf{x} \in \Omega_i} \rho(\mathbf{x})(\mathbf{x}_i - \mathbf{x}) \, \Delta\sigma. \tag{7}$$

The summation is computed using the regional reduction in the same pass as computing the energy function F. These values are written into a texture and read back to the CPU, where they are used to compute the approximated inverse Hessian matrix $\mathbf{H}_k$ using (5) and then the new sites $\mathbf{X}_{k+1}$. Hence, our implementation of the L-BFGS method is not entirely done on GPU.

4 CCVT on Surfaces

In this section we will explain the pipeline of computing the CCVT on a surface, following the flow in Fig. 4.

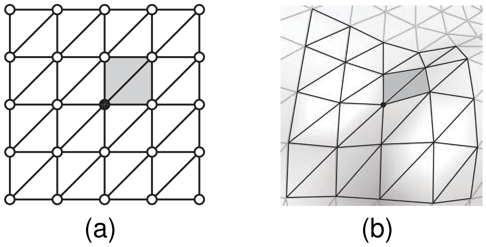


Fig. 5. (a) A regular triangulation in a geometry image. (b) The corresponding triangulation on the surface.

4.1 Geometry Image for Surface Representation

For a given surface we first compute a *conformal parameterization* [41] of the surface over a 2D rectangular domain and use the parameterization to construct a regular quad mesh to represent the surface. Then the surface can be represented by a 2D texture called the *geometry image* [42] whose pixels store the 3D coordinates of the corresponding mesh vertices. The red, green and blue channels of a geometry image store the 3D coordinates of the mesh vertices, respectively. The surface normal vectors at the vertices are also stored in a 2D texture of the same size, called *normal vector image*, which will be used in the L-BFGS algorithm for computing the CCVT on a surface.

Due to parameterization distortion, each pixel is assigned a weight equal to the area it covers on the surface. This weight is computed as follows. Suppose that the grid is subdivided to give a regular triangulation as shown in Fig. 5. Every pixel is supposed to cover one-third of each triangle incident to the corresponding vertex of the pixel. Thus the weight of the pixel is set to 1/3 of the areas of all the triangles incident to the corresponding vertex. The weight will be called the *distortion factor* and is used as $\Delta\sigma$ in (4) and (7).

The pipeline of computing the CCVT is shown in

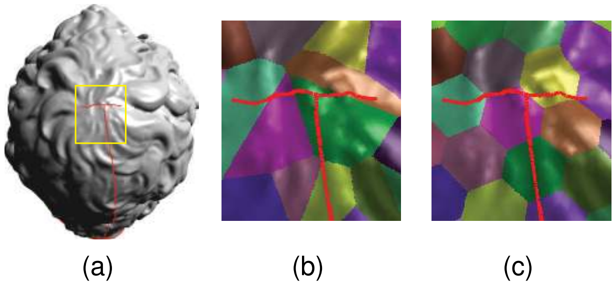


Fig. 6. (a) Cutting edges (red) on David Head. The enlarged region in the yellow box is shown with (b) the initial Voronoi diagram, and (c) the final CVT.

Fig. 4. The left most part of Fig. 4 shows the geometry image and the normal vector image of the surface of David Head in the top row, and the surface itself and a checkerboard texture showing the parameterization in the bottom row. The computed distortion factors are shown in the left middle part of Fig. 4.

Like the 2D case, computing the CCVT on a surface using either Lloyd's algorithm or the L-BFGS algorithm takes two main steps in each iteration: 1) computing the Voronoi diagram of the current sites on the surface, and 2) computing the new sites for the next iteration. We will explain these two steps in the following subsections.

4.2 Computing Voronoi Diagram on Surfaces

Suppose that a given surface is represented as a geometry image. We first store the 3D coordinates of the initial sites in the nearest corresponding pixels of the geometry image, and then perform the JFA to compute a Voronoi diagram on the surface. Note that the Euclidean distance between points in 3D space is used as approximation to the geodesic distance when computing the Voronoi diagram on the surface.

We use a single geometry image for parameterizing a surface of arbitrary topology, with necessary topological cutting to facilitate parameterization. Due to the topological cut, a Voronoi region on a surface may be split into disconnected regions in the geometry image. This is usually not a problem for the JFA, because the information of a site can reach every pixel during the JFA procedure as long as it is not killed en route [34]. Figure 6(b) demonstrates the correct Voronoi cells of initial sites near the cutting edges generated by the JFA. According to our experiments, even in some rare cases where errors occur near the cutting edges in certain iteration, its affection will get eliminated during later iterations. As the result, the final CVT result is always very good (see Figure 6(c) for example).

Equipped with the routine for computing the Voronoi diagram on a surface, we now explain how to implement Lloyd's algorithm and the L-BFGS algorithm on the GPU to compute the CCVT on a surface.

4.3 Lloyd's Algorithm on Surfaces

Using the geometry image to compute the CCVT on a surface with Lloyd's algorithm follows the same procedure as for computing the CVT in 2D, except that we need to compute the constrained centroids using (3). The following property about the constrained centroid will be useful [19]: if $\mathbf{c}_i$ and $\mathbf{c}_i^*$ are the centroid and the constrained centroid of the Voronoi cell Ω_i respectively, then $\mathbf{c}_i\mathbf{c}_i^*$ is parallel to the surface normal vector at $\mathbf{c}_i^*$. On the other hand, we observe that if $\mathbf{c}_i'$ is the nearest point in Ω_i to $\mathbf{c}_i$ and it is not on the boundary of Ω_i, then $\mathbf{c}_i\mathbf{c}_i'$ is parallel to the surface normal vector at $\mathbf{c}_i'$. Based on these observations we find it an effective heuristic to use the nearest point $\mathbf{c}_i'$ as the constrained centroid $\mathbf{c}_i^*$ in Lloyd's algorithm, although they are not always identical, since $\mathbf{c}_i\mathbf{c}_i'$ being parallel to the surface normal vector at $\mathbf{c}_i'$ is not a sufficient condition for $\mathbf{c}_i'$ to be the constrained centroid.

The nearest point $\mathbf{c}_i'$ is computed as follows. For the corresponding mesh vertex of every pixel in Ω_i, we compute the nearest point within its six incident triangles to the centroid $\mathbf{c}_i$. To avoid redundant computations, we only need to check two incident triangles for every vertex (e.g. the two shaded triangles in Fig. 5 for the center vertex), and the other four incident triangles will be dealt by other neighboring vertices. In this way we find a nearest point to the centroid $\mathbf{c}_i$ for every pixel in Ω_i. Then by computing the minimum of the distances from these points to $\mathbf{c}_i$ using the regional reduction, we find the nearest point $\mathbf{c}_i'$ within Ω_i to $\mathbf{c}_i$.

The number of iterations needed by both Lloyd's and the L-BFGS algorithms to obtain the CVT can be reduced significantly if the initial sites are roughly evenly distributed on the surface. To obtain such initial sites, we use the distortion factors as a probability density function to sample the initial sites in the geometry image. Therefore, a pixel with a larger distortion factor is more likely to be selected as the initial site. The right middle part of Fig. 4 shows 1,000 initial sites sampled according to the distortion factors and the Voronoi diagram generated in the parameter domain (top row), as well as on the surface (bottom row). The final sites and the CCVT are shown in the right most part of Fig. 4.

4.4 L-BFGS Algorithm on Surfaces

The initial sites for the L-BFGS algorithm are also sampled according to distortion factors. In every iteration, to update the sites we need to evaluate the CVT energy function F and its partial derivatives. Because the sites are constrained to be on the surface, we only use the tangential components of the partial derivatives as

$$\left.\frac{\partial F}{\partial \mathbf{x}_i}\right|_{\Omega} = \frac{\partial F}{\partial \mathbf{x}_i} - \left(\frac{\partial F}{\partial \mathbf{x}_i}\cdot \mathbf{N}(\mathbf{x}_i)\right)\mathbf{N}(\mathbf{x}_i), \tag{8}$$

where $\mathbf{N}(\mathbf{x}_i)$ is the surface normal vector at $\mathbf{x}_i$ stored in the normal vector image. Therefore we can use shaders

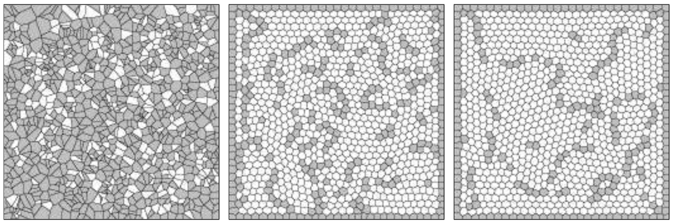

(a) (b) (c)

Fig. 7. The Voronoi diagram of 1,000 initial sites in the 2D domain of $[-1,1]\times[-1,1]$ (a), and CVT results generated by the L-BFGS algorithm using (b) GPU and (c) CPU. (Unshaded cells are hexagons.)

to evaluate F and its partial derivatives on the GPU efficiently, in the same way as in the 2D case.

The updated sites computed in the L-BFGS algorithm may not lie exactly on the surface. We compute their nearest points in their respective Voronoi cells on the surface as the new sites on the surface.

5 Experimental Results and Discussion

We implement our programs using Microsoft Visual C++ 2005 and NVIDIA Cg 2.0. The hardware platform is Intel Core 2 Duo 2.93GHz with 2GB DDR2 RAM, and NVIDIA GeForce GTX 280 with 1GB DDR3 VRAM. For the L-BFGS algorithm, we use our HLBFGS library [43]. We have compared our results with the CPU version of Lloyd's algorithm and the CPU version of the L-BFGS algorithm in [21]. We use $m = 7$ for all L-BFGS algorithms in our experiments; that is, we use the gradients of the 7 previous iterations to estimate the inverse Hessian in our implementation.

All the programs in this paper use IEEE standard float point numbers (32-bit).

5.1 Results of CVT in 2D

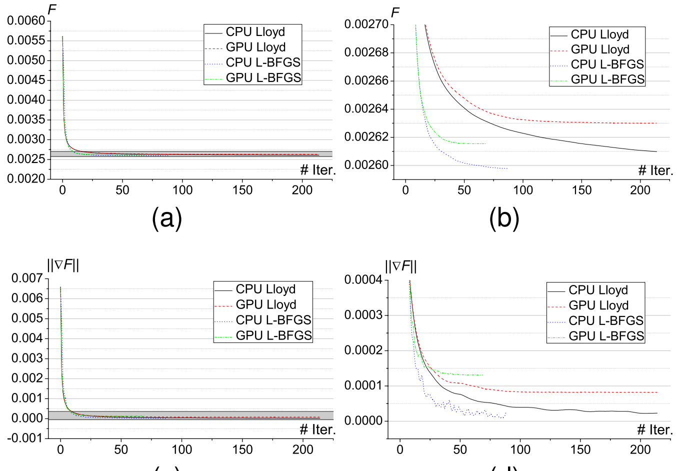
The first test example is the computation of the CVT in the 2D domain of $[-1,1]\times[-1,1]$ mapped to a 512×512 texture, with 1,000 random initial sites. The CVTs computed by the GPU program and CPU program are shown in Fig 7. It is clear that the uniformity of the sites is greatly improved in both results. We plot the CVT energy values and their gradients versus the number of iterations for Lloyd's algorithm and the L-BFGS algorithm in Fig. 8. The red and green curves are for our GPU programs and the black and blue curves for CPU ones. For clarity, we show zoom-in views of the shaded regions in Fig. 8(b) and Fig. 8(d).

The experiments show that the GPU result is very close to the CPU one, although there are fewer hexagon cells in the GPU result and the the final CVT energy value generated by the GPU is slightly higher. We may evaluate the quality of an approximate CVT by considering its *relative difference* from CVT computed with the CPU, defined as

$$\frac{CVT\_energy - final\_energy\_CPU}{final\_energy\_CPU}\times 100\%,$$

where CVT_energy is the CVT energy of the approximate CVT. For the tests shown in Fig. 8 the relative difference is 0.69% for the result of Lloyd's algorithm on GPU and 0.58% for that of the L-BFGS algorithm on GPU. These differences can be attributed to two factors. First, a different local minimum with higher energy is produced by the GPU program. Second, the quantization errors in the GPU implementation are responsible.

Table 1 compares the total running times of all iterations of different programs for this example. The GPU program and CPU program of the same algorithm use the same number of iterations. Because of the line search used in the L-BFGS minimizer, the function evaluating gradients and energy value may be called more than once in every iteration. Within every function call, we need to rebuild the Voronoi diagram and compute the gradients of the CVT energy, and this is the most time-consuming part of the L-BFGS algorithm and dominates the running time. For this reason, $\#Iter.$ for the L-BFGS method in Table 1 is the number of the function calls.



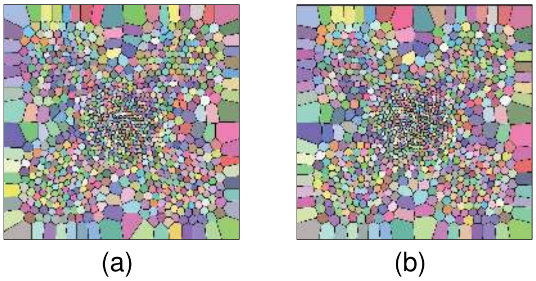
(a) (b)

(c) (d)

Fig. 8. Energy values (a) and their gradients (c) of CPU and GPU programs for Lloyd's algorithm and the L-BFGS algorithm using 1,000 sites in the 2D domain of $[-1,1]\times[-1,1]$. (b) and (d) are zoom-in views of shaded regions in (a) and (c).

TABLE 1
Comparison of energy F and running time (in seconds) of different programs using 1,000 initial sites in the 2D domain of $[-1,1]\times[-1,1]$.

Program	#Iter.	GPU		CPU	
		F	Time	F	Time
Lloyd	216	2.628×10^3	0.550	2.610×10^3	1.535
L-BFGS	92	2.612×10^3	0.512	2.597×10^3	0.703


(a)

(b)

Fig. 9. CVT results of 1,000 initial sites generated by the L-BFGS algorithm in the 2D domain of $[-1,1] \times [-1,1]$ with the density function $\rho(\mathbf{x}) = e^{-20x^2-20y^2} + 0.05\sin^2(\pi x)\sin^2(\pi y)$ using (a) GPU and (b) CPU.

TABLE 2
Comparison of energy F and running time (in seconds) of different programs using 1,000 initial sites in the 2D domain of $[-1,1] \times [-1,1]$ with non-constant densities.

Program	#Iter.	GPU		CPU	
		F	Time	F	Time
Lloyd	627	6.524×10^{-5}	2.529	6.163×10^{-5}	28.530
L-BFGS	376	6.185×10^{-5}	2.301	6.116×10^{-5}	16.889

We see that the GPU program of Lloyd's method is several time faster than the CPU program, because all the computations for Lloyd's algorithm are executed on the GPU. However, the GPU program of the L-BFGS method has only about 30% speedup over its CPU counterpart. That is because, although the most time-consuming parts for the L-BFGS algorithm are executed on the GPU, some of its computations are done on the CPU which takes about 23% of the total time and is not accelerated by the GPU; and the communication between the CPU and the GPU also takes about 5% of the total time.

We have also compared our GPU program of Lloyd's method to the algorithm proposed by Vasconcelos et al. [36]. For 1,000 sites, even with a very small mask (32×32), their program needs 1.416 seconds for the same number of iterations as in Table 1. If the size of the mask is set to be same as the screen resolution (512×512), the running time becomes 44.521 seconds.

Our GPU programs can also compute the CVT with a non-constant density function, as the density can be sampled and stored as floating point numbers in a texture. An example is shown in Fig 9, comparing the CVTs computed by the GPU program and the CPU program of the L-BFGS method. The final CVT energy values and the total running time of all iterations are listed in Table 2.

5.2 Results of CCVT on Surfaces

We will compare our GPU programs and CPU programs of Lloyd's algorithm and the L-BFGS algorithm using five surface models: Torus (Fig. 1), Lion (Fig. 10), Sculpture (Fig. 11), Body (Fig. 13), and David Head (Fig. 4). Table 3 lists information about these five models. All of our experiments use 1,000 initial sites on the surface sampled according to distortion factors. For each model, the same set of initial sites are used as input for all programs. The CPU programs utilize the fast algorithm introduced in [44] which can greatly accelerate the computation of the intersection between the surface and the 3D Voronoi diagram. For every model the geometry image is pre-computed with user interaction in less than 10 seconds. This time is not included in the total running time reported below.

The final energy values F and the total running time of all iterations are listed in Table 4 and Table 5. Like the 2D cases, for the L-BFGS algorithm, the number of the function calls for VD computation is listed, rather than the number of iterations. The GPU program and the CPU program have the same number of iterations for Lloyd's algorithm or the same number of function calls for the L-BFGS algorithm.

TABLE 3
Information of the five models used in this paper. The last two columns are for the number of boundaries and the resolution of geometry images.

Model	#Vertices	#Faces	Genus	#B	GI
Torus	4,096	7,938	1	0	512×512
Lion	5,321	10,200	0	0	512×190
Sculpture	5,471	10,364	3	0	512×372
Body	13,978	27,295	0	2	512×456
David Head	51,038	101,144	0	1	512×416

TABLE 4
Comparison of energy F and running time (in seconds) of GPU and CPU programs for Lloyd's algorithm.

Model	#Iter.	GPU		CPU	
		F	Time	F	Time
Torus	500	6.381×10^{-4}	8.235	6.304×10^{-4}	81.594
Lion	135	1.458×10^{-2}	0.918	1.388×10^{-2}	20.25
Sculpture	183	4.928×10^{-3}	2.229	4.568×10^{-3}	27.86
Body	262	1.801×10^{-4}	3.760	1.776×10^{-4}	52.985
David Head	215	9.998×10^{-4}	2.937	9.839×10^{-4}	144.688

TABLE 5
Comparison of energy F and running time (in seconds) of GPU and CPU programs for the L-BFGS algorithm.

Model	#Iter.	GPU		CPU	
		F	Time	F	Time
Torus	47	6.362×10^{-4}	0.808	6.307×10^{-4}	7.531
Lion	28	1.472×10^{-2}	0.207	1.386×10^{-2}	3.500
Sculpture	26	4.965×10^{-3}	0.331	4.582×10^{-3}	3.329
Body	67	1.790×10^{-4}	1.004	1.774×10^{-4}	13.703
David Head	33	1.010×10^{-3}	0.456	9.854×10^{-4}	15.609

TABLE 6
Comparison of the standard deviations of different uniformity measures for 1,000 initial sites, and the sites in CCVT results on the surface of Lion.

Program	STDEV(r_i)	STDEV(d_i)	STDEV(a_i)
Initial Sites	1.450×10^{-2}	1.809×10^{-2}	3.717×10^{-3}
Lloyd - GPU	7.422×10^{-3}	9.246×10^{-3}	1.902×10^{-3}
Lloyd - CPU	3.228×10^{-3}	6.090×10^{-3}	1.002×10^{-3}
L-BFGS - GPU	6.992×10^{-3}	9.657×10^{-3}	1.805×10^{-3}
L-BFGS - CPU	2.868×10^{-3}	5.824×10^{-3}	9.046×10^{-4}

It is observed that the GPU programs perform about one order of magnitude faster than their CPU counterparts. This speedup is more than that of the 2D case because the CPU programs for computing the CCVT of a surface need to compute a 3D Voronoi diagram and find its intersection with the surface, which is a very time consuming task compared with computing a Voronoi diagram in a 2D domain. Although the GPU programs also compute distances in 3D, the whole algorithms are still efficiently performed in a 2D domain.

The relative differences between the GPU and CPU results range from 0.88% to 8.36% due to different distortions of surface parameterizations. Fig. 10 and Fig. 11 show two sets of results with the largest relative differences of Lion (genus 0) and Sculpture (genus 3): the Voronoi diagram of the initial sites, and the CCVTs generated using the GPU and the CPU of Lloyd's algorithm and the L-BFGS algorithm, with the same number of iterations for each algorithm. We note that the GPU results have larger energy values than their CPU counterparts due to its approximation nature.

In addition to the comparison in terms of visual inspection and energy values, we may also measure the geometric uniformity of the sites and their Voronoi cells in the CCVT results. For every site $\mathbf{x}_i$, we define the radius r_i of its Voronoi cell Ω_i, the distance d_i to its nearest neighboring site, and the area a_i of its Voronoi cell Ω_i as follows:

$$r_i = \max_{\mathbf{x}\in\Omega_i} ||\mathbf{x}-\mathbf{x}_i||, d_i = \min_{j\neq i} ||\mathbf{x}_i-\mathbf{x}_j||, a_i = Area(\Omega_i).$$

The standard deviations of r_i, d_i, and a_i are used to measure the uniformity of a set of sites. Table 6 lists the standard deviations for initial sites and the sites in the CCVT results on Lion. The uniformity of the initial sites is greatly improved in all the results. Again, the GPU results are overall still not as good as the CPU ones.

The quality of the CCVT we computed with the GPU is heavily affected by the distortion factors. If the distortion factors are very large in a certain part of a surface, the Voronoi cells in this part are mapped to a very small region in the geometry image. Then the resolution of the geometry image will not be adequate for accurate computation in this part of the surface, resulting in larger errors in the computation of centroids (for Lloyd's algorithm) or energy value and gradients (for the L-BFGS algorithm). To illustrate this, we compare the results on David Head using two different parameterizations (see Fig. 12). Clearly, the second parameterization has larger distortion, which leads to more artifacts than the first parameterization. This problem could be alleviated by using a geometry image of higher resolution (see the example in Fig. 12(e) and (h)), or using multiple geometry images based a multi-chart surface parameterization [45].

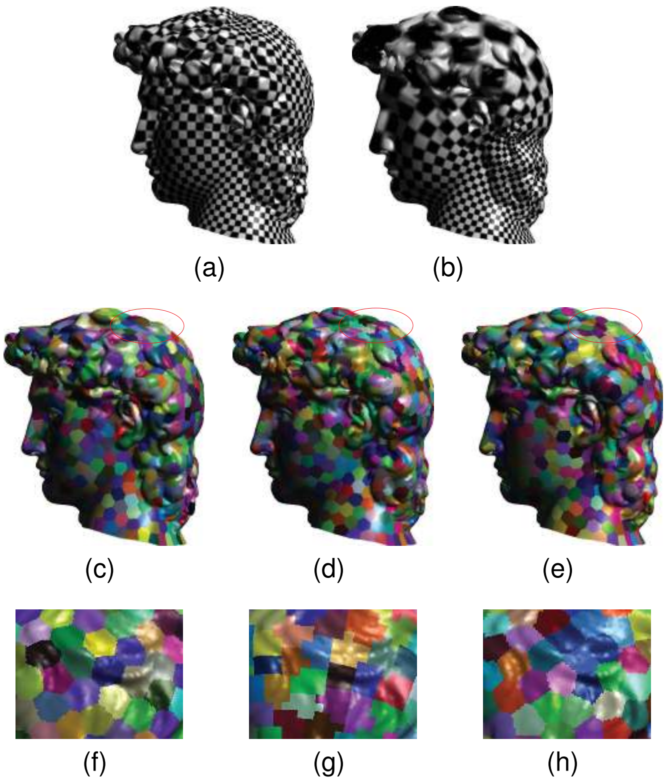


Fig. 12. (a)&(b) Comparison of two different parameterizations of David Head. (c) the CVTs computed using the parameterization in (a) with a 512×416 geometry image; (d) the CVTs computed using the parameterization in (b) with a 512×191 geometry image; and (e) the CVTs computed using the parameterization in (b) with a 2048×765 geometry image. (f)-(h) are enlarged top views of the circled part.

The CCVT can be used for surface remeshing by computing a well-shaped triangulation of the surface as the dual mesh of a CCVT. Fig. 13 shows an example of remeshing the Body surface with 1,000 sites.

5.3 JFA Errors

Ideally, the CVT energy should decrease monotonically during the iterations in both Lloyd's and the L-BFGS algorithm, if implemented accurately. However, since some pixels may be misclassified by the JFA into other Voronoi cells, the partial CVT energy values computed for those Voronoi cells are slightly different to the accurate values. Despite this, the sites move greatly and the

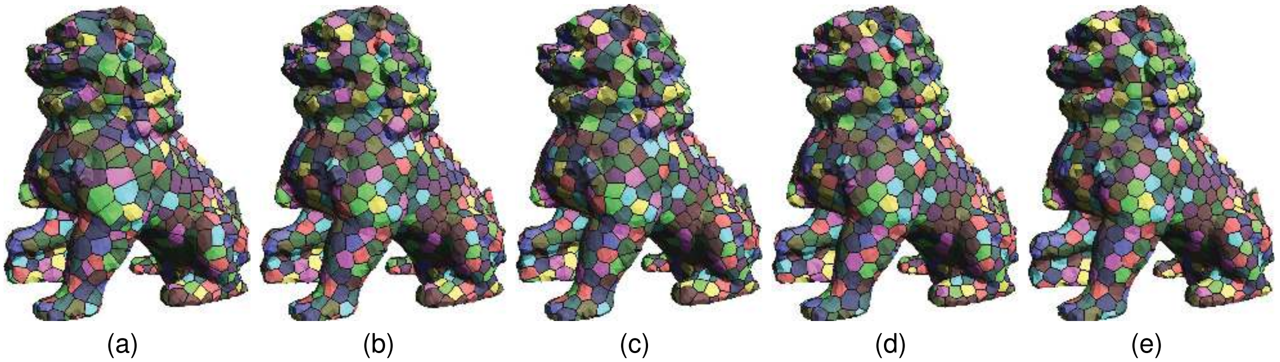


Fig. 10. Comparison of (a) the Voronoi diagram of initial sites and CCVT results on the surface of Lion generated by Lloyd's (GPU) (b), L-BFGS (GPU) (c), Lloyd's (CPU) (d) and L-BFGS (CPU) (e) algorithms. The relative differences of the GPU results are 5.07% for Lloyd's algorithm ((b) and (d)) and 6.17% for the L-BFGS algorithm ((c) and (e)). As a reference, the CVT energy of the initial sites is 2.538×10^{-2}, and the relative differences of the initial sites (before optimization) are 82.85% for Lloyd's algorithm and 83.12% for the L-BFGS algorithm.

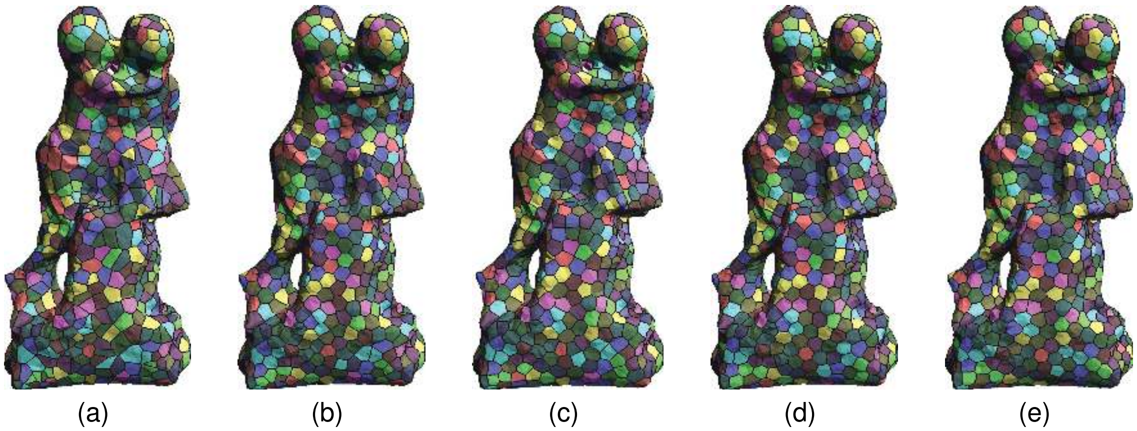


Fig. 11. Comparison of (a) the Voronoi diagram of initial sites and CCVT results on the surface of Sculpture generated by Lloyd's (GPU) (b), L-BFGS (GPU) (c), Lloyd's (CPU) (d) and L-BFGS (CPU) (e) algorithms. The relative differences of the GPU results are is 7.88% for Lloyd's algorithm ((b) and (d)) and 8.36% for the L-BFGS algorithm ((c) and (e)). As a reference, the CVT energy of the initial sites is 8.726×10^{-3}, and the relative differences of the initial sites (before optimization) are 91.02% for Lloyd's algorithm and 90.44% for the L-BFGS algorithm.

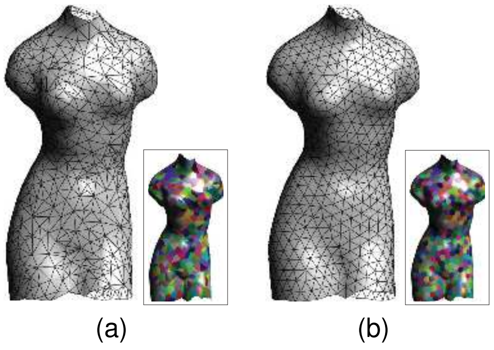


Fig. 13. (a) The dual triangle mesh of 1,000 initial sites; and (b) dual triangle mesh of 1,000 final sites of the CCVT. The insets show the corresponding Voronoi diagrams on the surface.

total CVT energy keeps decreasing in early iterations. However, in later iterations, when most of the sites are no longer moving, the error of JFA may cause the CVT energy to fluctuate. When this happens, most sites would remain unchanged but a small number of sites may oscillate between some positions.

To evaluate the consequence of this oscillation, we compared the JFA with an implementation on the GPU which computes the distance from every pixel to every site accurately by brute force, and show the results in Fig. 14. We see that the energy values only begins to oscillate in very late iterations due to the errors of the JFA.

In practice, this oscillation is very small and so does not affect the quality of the CVT for most applications in graphics. One may handle this nonconvergent behavior by terminating the computation when the CVT energy is found to increase.

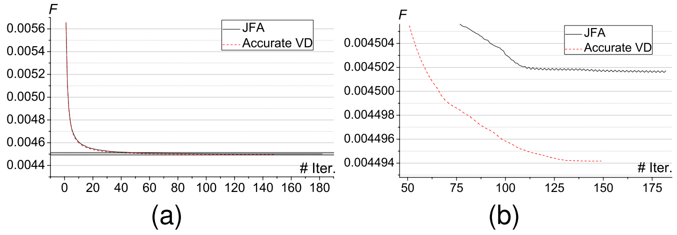


Fig. 14. Energy values of Lloyd's algorithm using the JFA and an accurate method to compute the Voronoi diagram of 1,000 sites on Sculpture. (b) A zoom-in view of energy plots in the shaded region in (a).

TABLE 7
Comparison of Cg and CUDA running time (in seconds) of Lloyd's algorithm for 1,000 sites in the 2D domain of $[-1,1] \times [-1,1]$. All steps are executed 243 times.

Step	Cg Time	CUDA Time
JFA	0.637	0.775
Compute Centroids	0.237	0.413
Test Stop Condition	0.015	0.107
Draw New Sites	0.041	0.032
Total Time	0.931	1.327

5.4 Shader Language VS. CUDA

CUDA is a relatively new and popular C-like language for general purpose computation on the GPU. Since CUDA has some features not available in traditional shader languages (such as accessing the shared memory), it is much faster for applications which can benefit from the new features. However, Cg is better suited than CUDA for implementing CVT algorithms. To compare Cg and CUDA programs, we list detailed time breakdown for each step in Lloyd's algorithm and the L-BFGS algorithm for a 2D case in Table 7 and Table 8 (the time is the total time for all iterations). It is clear that the JFA and the regional reduction (computing centroid for Lloyd's algorithm, and computing CVT energy and its gradient for the L-BFGS algorithm) are the two steps dominating the total running time. The Cg version is faster for both steps than the CUDA version. In total, the CUDA versions are more than 40% slower than the Cg versions.

The better efficiency of the Cg implementation can be explained as follows. The JFA spends most of its time on accessing memory rather than computation. For every pixel, it needs to read information from nine pixels and write to one pixel (itself). The reading addresses required by the JFA are non-coalesced [46] and change in every pass according to different step lengths. So it is very difficult, if not impossible, to make this step efficient with CUDA.

Computing the centroids in Lloyd's algorithm (as well as the energy value and gradients computation in the L-BFGS algorithm) is essentially a regional reduction problem. The regional reduction is known to be difficult for CUDA, since it requires many threads writing to a same memory address. This is usually implemented using *atomic operations* [46]. Currently, CUDA only supports atomic operations on integers. So for floating point numbers, we have to mimic the atomic operations by tagging the five least significant bits of the thread ID (see [47], [48] for details). This is inefficient and hardware-dependent, since the warp size must be known in advance.

TABLE 8
Comparison of Cg and CUDA running time (in seconds) of the L-BFGS algorithm for 1,000 sites in the 2D domain of $[-1,1] \times [-1,1]$. All steps are executed 92 times.

Step	Cg Time	CUDA Time
Draw Sites	0.046	0.057
JFA	0.259	0.291
Compute F and ∇F	0.117	0.261
Total GPU Time	0.443	0.664
Total Time	0.582	0.867

In conclusion, our algorithms for computing the CVT fit the traditional shader languages better than CUDA at this moment. However, as CUDA is fast evolving, we believe that efficient atomic operations on floating point numbers will be available soon. That will be likely to make CUDA faster than Cg for implementing our GPU algorithms in the near future.

6 Conclusion and Future Work

We have presented new techniques that use the GPU to compute the centroidal Voronoi tessellation both in 2D and on a surface. We proposed a novel algorithm to directly compute the Voronoi diagram on a surface, and also presented a new method using the vertex program to perform the regional reduction. Equipped with these two tools, we implemented two algorithms on the GPU – Lloyd's algorithm and the L-BFGS algorithm. For Lloyd's algorithm, the entire procedure is performed on the GPU; and for the L-BFGS algorithm, the major computational work is performed on the GPU. Our GPU implementations of the two methods have shown significant speedup over their CPU counterparts. Although our results are discrete approximations to the true CVTs, we believe that many applications can benefit from the fast computation of the CVT made possible by our GPU-based algorithms. Our algorithms can be directly extended to 3D CVT with the help of 3D textures, but the detailed analysis of the performance and result quality in 3D is out of the scope of this paper.

Sharp features are essential to model remeshing, especially for artificial models. How to incorporate sharp features in our algorithm will be an important future work. Integrating sharp edges into geometry image [49] is a possible solution of this problem.

In our current implementation of the L-BFGS algorithm, only the computation of CVT energy values and gradients, which is the most time-consuming part, is performed on the GPU. Currently, these values still need to be read back to the CPU for computing the new sites, thus slowing down the overall computation. If we could migrate this task onto the GPU, the speedup would be even more significant.

The number of sites of the CVT is currently limited by the size of the 2D texture in the GPU (e.g. for a 512×512 texture, the number of sites should not exceed 10,000; otherwise, the results would be very poor due to the large approximation errors). Furthermore, when using the geometric image to represent a surface of complex shape, the surface parameterization often has a large distortion and that leads to large discretization error in computation. In the future we will consider applying our GPU-base method to a multiple-chart representation of a surface of complex shape in order to compute the CVT with a much larger number of sites or on a surface of arbitrary shape.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive comments. We would like to thank Feng Sun and Dongming Yan for their help on CPU programs for the L-BFGS algorithm, and Vasconcelos for providing her source code.

Guodong Rong and Xiaohu Guo are partially supported by the National Science Foundation under Grant No. CCF-0727098. Yang Liu is supported by the European Research Council (GOODSHAPE FP7-ERC-StG-205693). Wenping Wang is partially supported by the General Research Funds (718209, 717808) of Research Grant Council of Hong Kong, NSFC-Microsoft Research Asia co-funded project (60933008), and National 863 High-Tech Program of China (2009AA01Z304). Xiaotian Yin and Xianfeng David Gu are partially supported by NSF CAREER CCF-0448399, DMS-9626223, DMS-0523363, CCF-0830550, and ONR N000140910228.

References

[1] F. Aurenhammer, "Voronoi diagrams — a survey of a fundamental geometric data structure," *ACM Computing Surveys*, vol. 23, no. 3, pp. 345–405, 1991.

[2] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu, *Spatial tessellations: concepts and applications of Voronoi diagrams*, 2nd ed. John Wiley & Sons, 1999.

[3] A. Gersho, "Asymptotically optimal block quantization," *IEEE Transactions on Information Theory*, vol. 25, no. 4, pp. 373–380, 1979.

[4] G. F. Tòth, "A stability criterion to the moment theorem," *Studia Scientiarum Mathematicarum Hungarica*, vol. 38, no. 1-4, pp. 209–224, 2001.

[5] Q. Du and D. Wang, "The optimal centroidal Voronoi tessellations and the Gersho's conjecture in the three-dimensional space," *Computers and Mathematics with Applications*, vol. 49, no. 9-10, pp. 1355–1373, 2005.

[6] K. A. Lyons, H. Meijei, and D. Rappaport, "Algorithms for cluster busting in anchored graph drawing," *Journal of Graph Algorithms and Applications*, vol. 2, no. 1, pp. 1–24, 1998.

[7] O. Deussen, S. Hiller, C. van Overveld, and T. Strothotte, "Floating points: A method for computing stipple drawings," *Computer Graphics Forum*, vol. 19, no. 3, pp. 41–50, 2000, (Proceedings of Eurographics 2000).

[8] A. Hausner, "Simulating decorative mosaics," in *Proceedings of ACM SIGGRAPH 2001*. New York, NY, USA: ACM Press / ACM SIGGRAPH, 2001, pp. 573–580.

[9] L.-P. Fritzsche, H. Hellwig, S. Hiller, and O. Deussen, "Interactive design of authentic looking mosaics using Voronoi structures," in *Proceedings of 2nd International Symposium on Voronoi Diagrams in Science and Engineering*, 2005, pp. 82–92.

[10] Q. Du and M. Gunzburger, "Grid generation and optimization based on centroidal Voronoi tessellations," *Applied Mathematics and Computation*, vol. 133, no. 2-3, pp. 591–607, 2002.

[11] Q. Du and X. Wang, "Centroidal Voronoi tessellation based algorithms for vector fields visualization and segmentation," in *Proceedings of IEEE Visualization*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 43–50.

[12] A. McKenzie, S. V. Lombeyda, and M. Desbrun, "Vector field analysis and visualization through variational clustering," in *Proceedings of EUROGRAPHICS - IEEE VGTC Symposium on Visualization*, 2005, pp. 29–35.

[13] Q. Du and D. Wang, "Tetrahedral mesh generation and optimization based on centroidal Voronoi tessellations," *International journal for numerical methods in engineering*, vol. 56, no. 9, pp. 1355–1373, 2003.

[14] S. Valette and J.-M. Chassery, "Approximated centroidal Voronoi diagrams for uniform polygonal mesh coarsening," *Computer Graphics Forum*, vol. 23, no. 3, pp. 381–389, 2004, (Proceedings of Eurographics 2004).

[15] P. Alliez, É. C. de Verdière, O. Devillers, and M. Isenburg, "Centroidal Voronoi diagrams for isotropic surface remeshing," *Graph. Models*, vol. 67, no. 3, pp. 204–231, 2005.

[16] S. Valette, J.-M. Chassery, and R. Prost, "Generic remeshing of 3D triangular meshes with metric-dependent discrete Voronoi diagrams," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 2, pp. 369–381, 2008.

[17] J. Dardenne, S. Valette, N. Siauve, and R. Prost, "Medial axis approximation with constrained centroidal Voronoi diagrams on discrete data," in *Proceedings of Computer Graphics International*, 2008, pp. 299–306.

[18] Q. Du, V. Faber, and M. Gunzburger, "Centroidal Voronoi tessellations: Applications and algorithms," *SIAM Review*, vol. 41, no. 4, pp. 637–676, 1999.

[19] Q. Du, M. D. Gunzburger, and L. Ju, "Constrained centroidal Voronoi tessellations for surfaces," *SIAM Journal on Scientific Computing*, vol. 24, no. 5, pp. 1488–1506, 2003.

[20] S. P. Lloyd, "Least squares quantization in PCM," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[21] Y. Liu, W. Wang, B. Lévy, F. Sun, D.-M. Yan, L. Lu, and C. Yang, "On centroidal Voronoi tessellation — energy smoothness and fast computation," *ACM Transactions on Graphics*, vol. 28, no. 4, pp. 1–17, 2009.

[22] G. Rong and T.-S. Tan, "Jump flooding in GPU with applications to Voronoi diagram and distance transform," in *Proceedings of the Symposium on Interactive 3D Graphics and Games*. ACM Press, 2006, pp. 109–116.

[23] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the fifth Berkeley Symposium on Mathematical Statistics and Probability*. University of California Press, 1967, pp. 281–297.

[24] L. Ju, Q. Du, and M. Gunzburger, "Probabilistic methods for centroidal Voronoi tessellations and their parallel implementations," *Parallel Computing*, vol. 28, no. 10, pp. 1477–1500, 2002.

[25] Q. Du and M. Emelianenko, "Acceleration schemes for computing centroidal Voronoi tessellations," *Numerical Linear Algebra with Applications*, vol. 13, no. 2-3, pp. 173–192, 2006.

[26] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[27] K. E. Hoff III, T. Culver, J. Keyser, M. Lin, and D. Manocha, "Fast computation of generalized Voronoi diagrams using graphics hardware," in *SIGGRAPH '99*, 1999, pp. 277–286.

[28] M. O. Denny, "Algorithmic geometry via graphics hardware," Ph.D. dissertation, Universität des Saarlandes, 2003.

[29] I. Fischer and C. Gotsman, “Fast approximation of high order Voronoi diagrams and distance transforms on the GPU,” *Journal of Graphics Tools*, vol. 11, no. 4, pp. 39–60, 2006.

[30] T.-T. Cao, K. Tang, A. Mohamed, and T.-S. Tan, “Parallel banding algorithm to compute exact distance transform with the GPU,” in *Proceedings of the Symposium on Interactive 3D Graphics and Games*, 2010, to appear.

[31] C. Sigg, R. Peikert, and M. Gross, “Signed distance transform using graphics hardware,” in *Proceedings of IEEE Visualization*, 2003, pp. 83–90.

[32] A. Sud, M. A. Otaduy, and D. Manocha, “DiFi: Fast 3D distance field computation using graphics hardware,” *Computer Graphics Forum*, vol. 23, no. 3, pp. 557–566, 2004, (Proceedings of Eurographics 2004).

[33] A. Sud, N. Govindaraju, R. Gayle, and D. Manocha, “Interactive 3D distance field computation using linear factorization,” in *Proceedings of ACM Symposium on Interactive 3D Graphics and Games*, 2006, pp. 117–124.

[34] G. Rong and T.-S. Tan, “Variants of jump flooding algorithm for computing discrete Voronoi diagrams,” in *Proceedings of the 4th International Symposium on Voronoi Diagrams in Science and Engineering (ISVD’07)*, 2007, pp. 176–181.

[35] O. Weber, Y. S. Devir, A. M. Bronstein, M. M. Bronstein, and R. Kimmel, “Parallel algorithms for approximation of distance maps on parametric surfaces,” *ACM Transactions on Graphics*, vol. 27, no. 4, pp. 1–16, 2008.

[36] C. N. Vasconcelos, A. Sá, P. C. Carvalho, and M. Gattass, “Lloyd’s algorithm on GPU,” in *Proceedings of the 4th International Symposium on Visual Computing*, 2008, pp. 953–964.

[37] E. F. Bollig, “Centroidal Voronoi tessellation of manifolds using the GPU,” Master’s thesis, Florida state university, 2009.

[38] I. Buck and T. Purcell, “A toolkit for computation on GPUs,” in *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, R. Fernando, Ed. Addison-Wesley, 2004, ch. 37, pp. 621–636.

[39] G. Rong, T.-S. Tan, T.-T. Cao, and Stephanus, “Computing two-dimensional Delaunay triangulation using graphics hardware,” in *Proceedings of the Symposium on Interactive 3D Graphics and Games*. ACM Press, 2008, pp. 89–97.

[40] D. C. Liu and J. Nocedal, “On the limited memory BFGS method for large scale optimization,” *Mathematical Programming*, vol. 45, no. 3, pp. 503–528, 1989.

[41] X. Gu and S.-T. Yau, “Global conformal surface parameterization,” in *Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2003, pp. 127–137.

[42] X. Gu, S. J. Gortler, and H. Hoppe, “Geometry images,” *ACM Transactions on Graphics*, vol. 21, no. 3, pp. 355–361, 2002.

[43] Y. Liu, “HLBFGS,” 2009, http://www.loria.fr/~liuyang/software/HLBFGS/.

[44] D.-M. Yan, B. Lévy, Y. Liu, F. Sun, and W. Wang, “Isotropic remeshing with fast and exact computation of restricted Voronoi diagram,” *Computer Graphics Forum*, vol. 28, no. 5, pp. 1445–1454, 2009, (Proceedings of Symposium on Geometry Processing 2009).

[45] P. Alliez, M. Meyer, and M. Desbrun, “Interactive geometry remeshing,” *ACM Transactions on Graphics*, vol. 21, no. 3, pp. 347–354, 2002.

[46] NVIDIA Corporation, “NVIDIA CUDA$^{\text{TM}}$ programming guide,” http://www.nvidia.com/object/cuda_home.html, July 2009.

[47] V. Podlozhnyuk, “Histogram calculation in CUDA,” NVIDIA Corporation, White Paper, 2007.

[48] R. Shams and R. A. Kennedy, “Efficient histogram algorithms for NVIDIA CUDA compatible devices,” in *Proceedings of International Conference on Signal Processing and Communications Systems (ICSPCS)*, 2007, pp. 418–422.

[49] M. Gauthier and P. Poulin, “Preserving sharp edges in geometry images,” in *Graphics Interface 2009*, May 2009, pp. 1–6.

Guodong Rong received his B.Eng. and M.Eng. degree both in computer science from Shandong University in 2000 and 2003 respectively, and his Ph.D. degree in computer science from National University of Singapore in 2007. He is currently a research scholar at Department of Computer Science, University of Texas at Dallas. His research interests include computer graphics, computational geometry, visualization and image processing, especially on using the GPU to accelerate geometry-related problems.

Yang Liu received the B.S. and M.S. degree in mathematics from the University of Science and Technology of China, Anhui, China, in 2000 and 2003, respectively, and the Ph.D degree in computer science from the University of Hong Kong, Hong Kong in 2008. He is currently a post-doc in Loria/Inria, France. His research interests include geometric computation and optimization, computer aided geometric design and computer graphics.

Wenping Wang is a Professor of Computer Science at The University of Hong Kong. His research covers computer graphics, visualization, and geometric computing. He got his B.Sc. (1983) and M.Eng. (1986) at Shandong University, China, and Ph.D. (1992) at University of Alberta, Canada, all in Computer Science. He is currently Associate Editor of the Springer journal Computer Aided Geometric Design and IEEE Transactions on Visualization and Computer Graphics. He is program co-chair of several international conferences, including Geometric Modeling and Processing (GMP 2000), Pacific Graphics 2003, ACM Symposium on Physical and Solid Modeling (SPM 2006), and International Conference on Shape Modeling (SMI 2009).

Xiaotian Yin received the BS degree in computer science from Peking University, China, in 2001, and worked for Bell Labs Research China from 2001 to 2004. He is currently a PhD candidate in computer science at Stony Brook University, and a visiting scholar in the Mathematics Department of Harvard University. His research interests are in the broad area of computational differential geometry and computational topology. For more information, see http://www.cs.sunysb.edu/~xyin.

Xianfeng David Gu received his PhD degree in computer science from Harvard University in 2003. He is an associate professor of computer science and the director of the 3D Scanning Laboratory in the Department of Computer Science in the State University of New York at Stony Brook University. His research interests include computer graphics, vision, geometric modeling, and medical imaging. His major works include global conformal surface parameterization in graphics, tracking and analysis of facial expression in vision, manifold splines in modeling, brain mapping and virtual colonoscopy in medical imaging, and computational conformal geometry. He won the US National Science Foundation CAREER Award in 2004. He is a member of the IEEE.

Xiaohu Guo is an assistant professor of computer science at the University of Texas at Dallas. He received the PhD degree in computer science from the State University of New York at Stony Brook in 2006. His research interests include computer graphics, animation and visualization, with an emphasis on geometric and physics-based modeling. His current researches at UT-Dallas include: spectral geometric analysis, deformable models, centroidal Voronoi tessellation, GPU algorithms, 3D and 4D medical image analysis, etc. For more information, please visit http://www.utdallas.edu/~xguo. He is a member of the IEEE.