Parallel algorithms for approximation of distance maps on parametric surfaces

Ofir Weber[1], Yohai S. Devir[2], Alexander M. Bronstein[3], Michael M. Bronstein[4], and Ron Kimmel[5]

We present an efficient $\mathcal{O}(n)$ numerical algorithm for first-order approximation of geodesic distances on geometry images, where n is the number of points on the surface. The structure of our algorithm allows efficient implementation on parallel architectures. Two implementations on a SIMD processor and on a GPU are discussed. Numerical results demonstrate up to four orders of magnitude improvement in execution time compared to the state-of-the-art algorithms.

Categories and Subject Descriptors: Numerical Analysis [**G.1.0**]: Parallel algorithms

Additional Key Words and Phrases: eikonal equation, geodesic distances, fast marching, geometry image, multiple charts, parallel algorithms, GPU, SIMD

1. INTRODUCTION

Approximation of geodesic distances on curved surfaces is an important computational geometric problem, appearing in many computer graphics applications. For example, several surface segmentation and editing methods are based on cutting the surface along geodesic paths [Katz and Tal 2004; Funkhouser et al. 2004]. Function interpolation on meshes requires the knowledge of geodesic distances, and has numerous uses such as skinning [Sloan et al. 2001] and mesh watermarking [Praun et al. 1999]. Isometry-invariant shape classification [Elad and Kimmel 2001; Hilaga et al. 2001; Mémoli and Sapiro 2005; Bronstein et al. 2006b], minimum-distortion parametrization [Zigelman et al. 2002; Zhou et al. 2004; Peyré and Cohen 2003], and non-rigid correspondence techniques [Bronstein et al. 2006a] require the matrix of all pair-wise geodesic distances on the surface. Other fields where the need to compute geodesic distance maps arises are medical imaging, geophysics[Sethian and Popovici 2006], and robot motion planning [Hershberger and Suri 1999] and navigation to mention a few.

The problem of distance map computation can be formulated as the viscosity solution of the *eikonal equation*,

$$\|\nabla t\| = 1, \quad t(S) = 0, \tag{1}$$

where S is a set of source points on the surface. In optics and acoustics, the eikonal equation governs the propagation of waves through a medium. The solution of the eikonal equation demonstrates that light or acoustic waves traverse the path between two points, which takes the least time, a physics law known as *Fermat's principle*.

In [1996], Sethian proposed an $\mathcal{O}(n \log n)$ algorithm for first-order approximation of weighted distance maps on domains with weighted Euclidean metric, known as *fast marching*. A similar algorithm based on a different discretization of the eikonal equation was developed independently by Tsitsiklis [1995]. The main idea of fast marching is to simulate a wave front advancing from a set of source points S. The propagating front can be thought of as a "prairie fire" evolution towards directions where the grid has not yet been "burnt out". At time $t = 0$, the fire starts at the source points, and the algorithm computes the time values t for each vertex at which the advancing fire front reaches it.

Algorithm 1 outlines the fast marching method. Solution of the eikonal equation starts by setting initial (usually zero) distance to the set of source points S and updating the neighboring points by simulating an advancing wavefront. The

algorithm is constructed similar to Dijkstra's algorithn for finding shortest paths in graphs. It maintains a set of fixed vertices S, for which the time of arrival has already been computed, and a priority queue Q of all other vertices sorted by their times of arrival. The basic operation of the fast marching algorithm is the *update* step, which computes the time of arrival of the wavefront to a grid point based on the times of arrival to its neighbor points.

By construction, the updated value cannot be smaller than the values of the supporting vertices. This monotonicity property ensures that the solution always propagates outwards by fixing the vertex with the smallest t. The latter implies that the values of grid points in S vertices are never recomputed. Since the *update step* has constant complexity, the overall complexity of the fast marching algorithm is determined by the procedure that finds the smallest t in the priority queue Q. Heap sorting-based priority queue allows to implement this task in $\mathcal{O}(\log n)$, where n is the number of grid vertices. Since each vertex is removed from Q and inserted to S only once, the overall complexity is $\mathcal{O}(n \log n)$.

Over the last decade, the fast marching algorithm was generalized to arbitrary triangulated surfaces [Kimmel and Sethian 1998], unstructured meshes [Sethian and Vladimirsky 2000], implicit unorganized surfaces [Mémoli and Sapiro 2001], and parametric surfaces [Spira and Kimmel 2004]. Higher-order versions of fast marching were also proposed [Sethian and Vladimirsky 2000]. Besides fast marching, there exist other families of numerical algorithms for approximate and exact computation of geodesic distances on surfaces, among which the most notable one is the Mount-Mitchel-Papadimitriou (MMP) algorithm [1987], whose most recent approximate implementation by Surazhsky *et al.* [2005] appears to be the fastest distance computation code available in public domain. Another recent algorithm is presented in [Jeong and Whitaker 2007].

In this paper, we explore the problem of geodesic distance map approximation on regularly sampled parametric surfaces (often referred to as geometry images), a representation becoming growingly popular as an alternative to unordered triangular meshes [Gu et al. 2002]. The paper is organized as follows. In Section 2, we formulate the eikonal equation on parametric surfaces. Section 3 is dedicated to the update step. We show a compact expression in matrix-vector form for a first-order update step on geometry images based on the planar wavefront model. We show that the scheme is numerically stable, which allows its use with low-precision arithmetics. We also study the update step based on the spherical wavefront model proposed by Novotni and Klein [2002] and indicate its numerical difficulties. Section 4 presents a raster scan algorithm for approximate distance map computation on geometry images. The proposed algorithm can be thought of as a generalization of Danielsson's raster scan method [1980] to geometry images, or as a raster-scan version of the parametric fast marching algorithm [Spira and Kimmel 2004]. We show that the raster scan algorithm converges with a bounded number of iterations, which enables its use for geodesic distance map computation. In Section 5, we discuss two parallel implementations of the raster scan algorithm on a SIMD processor and a GPU, which we refer to as the *parallel marching method* or PMM for short. Graphics hardware has been previously used for computation of distance maps and Voronoi diagrams on the plane or in the three-dimensional Euclidean space [Sigg et al. 2003; Hoff et al. 1999; Fischer and Gotsman 2005; Sud et al. 2006]. However, the use of vector processors for computation of geodesic distance maps is a different and significantly more complex problem, which to the best of our knowledge, has not been yet addressed in the literature, perhaps, with the exception of [Carr et al.]. We also discuss the extension of the proposed algorithm to geometry images represented using multiple charts, which is of critical importance in many practical applications. In Section 7, we present numerical tests and performance benchmarks for our algorithms. Parallel marching methods outperform the state-of-the-art distance computation algorithms by up to four orders of magnitude on commodity hardware, making feasible real-time implementation of many applications, where the complexity of geodesic distance computation has been so far prohibitively high. Section 8 concludes the paper.

Algorithm 1: Fast marching method.

Input: Numerical grid $\mathbf{U}$, set of source points $S \subset \mathbf{U}$ with the corresponding initial values $t(s)$
Output: The distance map $t : \mathbf{U} \mapsto \mathbb{R}^+$.

Initialization

```
1  Q ⟵ ∅
2  foreach point u ∈ U \ S do t(u) ⟵ ∞
3  foreach point u ∈ S do
4      Q ⟵ Q ∪ N(u)
5  end
   Iteration
6  while Q ≠ ∅ do
7      u ⟵ ExtractMin(Q)
8      S ⟵ S ∪ {u}
9      foreach point v ∈ N(u) do Update (v)
10 end
```

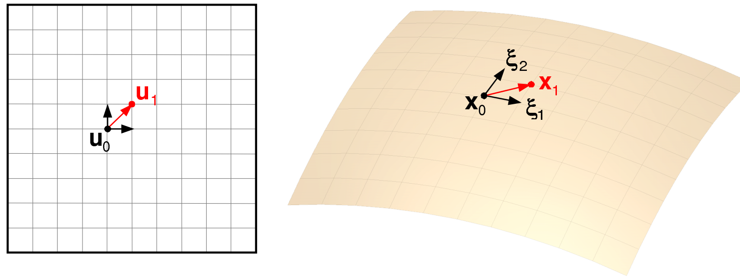

Fig. 1. A system of coordinates in the parametrization domain (left) and the corresponding local system of coordinates on the surface (right).

2. EIKONAL EQUATION ON GEOMETRY IMAGES

For the largest part of the discussion in this paper, we focus our attention on parametric two-dimensional manifolds, i.e. surfaces that can be represented by a single smooth mapping $\mathbf{x} : \mathbf{U} \to \mathbb{R}^3$, where $\mathbf{U} \subset \mathbb{R}^2$ is a parametrization domain. The topology of $\mathbf{U}$ depends on the topology of the surface. The derivatives

$$\xi_i = \frac{\partial \mathbf{x}}{\partial u^i} \tag{2}$$

with respect to the parametrization coordinates constitute a local system of coordinates on the surface (Figure 1). Distances on the surface are measured according to the differential arclength element,

$$ds^2 = d\mathbf{u}^{\mathrm{T}}\mathbf{G}d\mathbf{u}, \tag{3}$$

where $d\mathbf{u} = (du^1, du^2)$ and $\mathbf{G}$ is a 2×2 *metric* matrix, whose elements are given by $g_{ij} = \xi_i^{\mathrm{T}} \xi_j$. The local system of coordinates is *orthogonal* if and only if $\mathbf{G}$ is diagonal (note that orthogonality of the coordinate system in the parametrization domain does not imply orthogonality of the coordinate system on the surface).

A distance map on the surface is computed by solving the eikonal equation, expressed in our notation as

$$\|\nabla_{\mathbf{G}} t\|^2 \ = \ \nabla_{\mathbf{u}}^{\mathrm{T}} t \ \mathbf{G}(\mathbf{u})^{-1} \nabla_{\mathbf{u}} t \ = \ 1 \tag{4}$$

on a discrete grid obtained by sampling the parametrization domain $\mathbf{U}$. A special case, usually referred to as a *geometry image*, is obtained by discretizing the parametrization domain on a regular Cartesian grid with equal steps, which for convenience are henceforth assumed to be 1 in each direction. The origin of the term stems from the fact that the surface can be represented as tree matrices holding the coordinates of the sampled surface $\mathbf{x}(\mathbf{U})$.

In a geometry image, a grid point $\mathbf{u}_0$ can be connected to its neighbors $\mathbf{u}_0 + \mathbf{m}$ according some grid connectivity. The simplest grid connectivity is based on four neigbors: $\mathbf{m} = (\pm 1, 0)^{\mathrm{T}}, (0, \pm 1)^{\mathrm{T}}$. Another possible grid connectivity is the eight-neigbor connectivity, where $\mathbf{m} = (\pm 1, 0)^{\mathrm{T}}, (0, \pm 1)^{\mathrm{T}}, (\pm 1, \pm 1)^{\mathrm{T}}, (\pm 1, \mp 1)^{\mathrm{T}}$. The former two grid connectivity patterns create four and eight triangles, respectively, supporting the grid point $\mathbf{u}_0$. Let us examine a triangle created by $\mathbf{x}_0 = \mathbf{x}(\mathbf{u}_0)$, $\mathbf{x}_1 = \mathbf{x}(\mathbf{u}_0 + \mathbf{m}_1)$, and $\mathbf{x}_2 = \mathbf{x}(\mathbf{u}_0 + \mathbf{m}_2)$; without loss of generality we will henceforth assume that $\mathbf{x}_0 = 0$. In local coordinates, we can write

$$\mathbf{x}_i \ = \ \mathbf{x}_0 + m_i^1 \xi_1 + m_i^2 \xi_2, \tag{5}$$

or $\mathbf{X} = \mathbf{TM}$, where $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2)$, $\mathbf{T} = (\xi_1, \xi_2)$, and $\mathbf{M} = (\mathbf{m}_1, \mathbf{m}_2)$. The matrix $\mathbf{E} = \mathbf{M}^{\mathrm{T}}\mathbf{G}\mathbf{M}$ describes the geometry of the triangle. If $e_{12} > 0$ is positive, the angle $\sphericalangle \mathbf{x}_1 \mathbf{x}_0 \mathbf{x}_2$ on the surface is acute.

3. UPDATE STEP

The fast marching algorithm can be formulated for parametric surfaces as shown in [Spira and Kimmel 2004]. All computations are performed on the grid in the parametrization domain, though the distances are computed with respect to the surface metric $\mathbf{G}$. In the numerical core of this algorithm lies the update step, which given a grid point $\mathbf{u}_0$ and the times of arrival of its neighbors, computes the time of arrival $t(\mathbf{u}_0)$. Since $\mathbf{u}_0$ is shared by several triangles (the exact number of triangles depends on the grid connectivity), $t(\mathbf{u}_0)$ is computed in each triangle and the smallest value is selected to update the time of arrival at $\mathbf{u}_0$.

Let $\mathbf{u}_0$ be updated from its two neighbors $\mathbf{u}_1 = \mathbf{u} + \mathbf{m}_1$ and $\mathbf{u}_2 = \mathbf{u}_0 + \mathbf{m}_2$, whose times of arrival are $t_1 = t(\mathbf{u}_0 + \mathbf{m}_1)$ and $t_2 = t(\mathbf{u}_0 + \mathbf{m}_2)$. We denote $\mathbf{x}_i = \mathbf{x}(\mathbf{u}_i)$ and assume without loss of generality that $\mathbf{x}_0 = 0$. Our goal is to compute $t_0 = t(\mathbf{u}_0)$ based on t_1, t_2 and the geometry of the triangle $\mathbf{x}_1\mathbf{x}_0\mathbf{x}_2$. The update of $\mathbf{x}_0$ has to obey the following properties:

(1) *Consistency*: $t_0 > \max\{t_1, t_2\}$.

(2) *Monotonicity*: an increase of t_1 or t_2 increases t_0.

(3) *Upwinding*: the update has to be accepted only from a triangle containing the characteristic direction (characteristics of the eikonal equation coincide with minimum geodesics on the surface).

(4) *Numerical stability*: a small perturbation in t_1 or t_2 results in a bounded perturbation in t_0.

3.1 Planar wavefront approximation

In the original fast marching algorithm, a vertex is updated by simulating a planar wavefront propagating inside the triangle [Kimmel and Sethian 1998]; the values of the two supporting vertices allow to compute the front direction. The same update scheme was used in [Spira and Kimmel 2004]. Here, we develop a similar scheme, expressing it more compactly and without the use of trigonometric functions, which allow more efficient computation. We model the wavefront as a planar wave propagating from a virtual planar source described by the equation $\mathbf{n}^{\mathrm{T}}\mathbf{x} + p = 0$, where

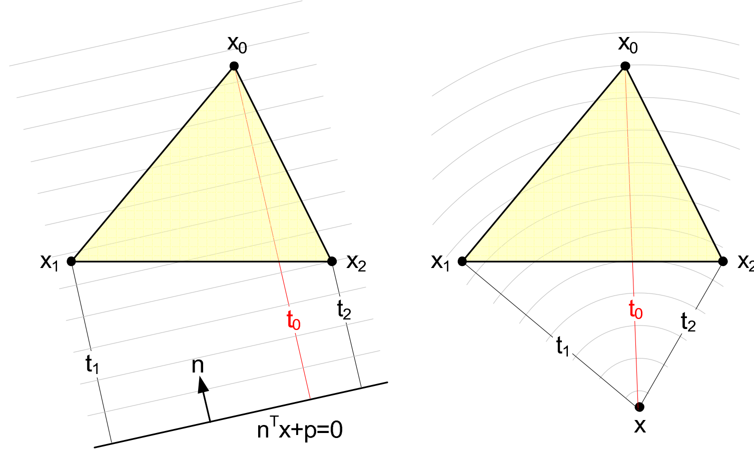

Fig. 2. Update schemes based on the planar (left) and spherical (right) wavefront propagation models.

$\mathbf{n}$ is the propagation direction (Figure 2). Demanding that the supporting vertices $\mathbf{x}_1$, $\mathbf{x}_2$ of the triangle lie at distances t_1 and t_2, respectively, from the source, we obtain

$$\mathbf{X}^{\mathrm{T}}\mathbf{n} + p \cdot \mathbf{1} \;=\; \mathbf{t}, \tag{6}$$

where $\mathbf{X}$ is a matrix whose columns are $\mathbf{x}_1$ and $\mathbf{x}_2$, $\mathbf{1} = (1,1)^{\mathrm{T}}$, and $\mathbf{t} = (t_1, t_2)^{\mathrm{T}}$. The wavefront time of arrival to the updated vertex $\mathbf{x}_0$ is given by its distance from the planar source,

$$t_0 \;=\; \mathbf{n}^{\mathrm{T}}\mathbf{x}_0 + p \;=\; p. \tag{7}$$

Assuming that the mesh is non-degenerate, $\mathbf{x}_1$ and $\mathbf{x}_2$ are linearly independent, and we can solve (6) for $\mathbf{n}$, obtaining

$$\mathbf{n} \;=\; \mathbf{X}(\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}(\mathbf{t} - p \cdot \mathbf{1}). \tag{8}$$

Invoking the condition $\|\mathbf{n}\| = 1$ yields

$$\begin{aligned} 1 \;&=\; \mathbf{n}^{\mathrm{T}}\mathbf{n} \\ &=\; (\mathbf{t} - p \cdot \mathbf{1})^{\mathrm{T}}(\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-\mathrm{T}}\mathbf{X}^{\mathrm{T}}\mathbf{X}(\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}(\mathbf{t} - p \cdot \mathbf{1}) \\ &=\; (\mathbf{t} - p \cdot \mathbf{1})^{\mathrm{T}}(\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}(\mathbf{t} - p \cdot \mathbf{1}) \\ &=\; p^2 \cdot \mathbf{1}^{\mathrm{T}}\mathbf{Q}\mathbf{1} - 2p \cdot \mathbf{1}^{\mathrm{T}}\mathbf{Q}\mathbf{t} + \mathbf{t}^{\mathrm{T}}\mathbf{Q}\mathbf{t}, \end{aligned} \tag{9}$$

where $\mathbf{Q} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1} = \mathbf{E}^{-1}$. Hence, t_0 can be found as the largest solution of the quadratic equation

$$t_0^2 \cdot \mathbf{1}^{\mathrm{T}}\mathbf{Q}\mathbf{1} - 2t_0 \cdot \mathbf{1}^{\mathrm{T}}\mathbf{Q}\mathbf{t} + \mathbf{t}^{\mathrm{T}}\mathbf{Q}\mathbf{t} - 1 \;=\; 0 \tag{10}$$

(the smallest solution corresponds to the opposite propagation direction, where the wavefront arrives to $\mathbf{x}_0$ *before* it arrives to $\mathbf{x}_1$ and $\mathbf{x}_2$ and therefore has to be discarded). To speed the solution up, the terms $\mathbf{1}^{\mathrm{T}}\mathbf{Q}\mathbf{1}$ and $\mathbf{1}^{\mathrm{T}}\mathbf{Q}$ depending on the grid geometry only are pre-computed.

The consistency condition can be written as $p \cdot \mathbf{1} > \mathbf{X}^{\mathrm{T}}\mathbf{n} + p \cdot \mathbf{1}$ or simply $\mathbf{X}^{\mathrm{T}}\mathbf{n} < 0$, which can be interpreted geometrically as a demand that the direction $-\mathbf{n}$ must form acute angles with the triangle edges. In order to impose monotonicity, we demand that

$$\nabla_{\mathbf{t}}\, t_0 \;=\; \left(\frac{\partial t_0}{\partial t_1}, \frac{\partial t_0}{\partial t_2}\right)^{\mathrm{T}} \;>\; 0. \tag{11}$$

Differentiating (10) with respect to $\mathbf{t}$, we obtain

$$t_0 \cdot \nabla_{\mathbf{t}}\, t_0 \cdot \mathbf{1}^{\mathrm{T}}\mathbf{Q}\mathbf{1} - \nabla_{\mathbf{t}}\, t_0 \cdot \mathbf{1}^{\mathrm{T}}\mathbf{Q}\mathbf{t} - t_0 \cdot \mathbf{Q}\mathbf{1} + \mathbf{Q}\mathbf{t} \;=\; 0, \tag{12}$$

from where

$$\nabla_{\mathbf{t}}\, t_0 \;=\; \frac{\mathbf{Q}(\mathbf{t} - p \cdot \mathbf{1})}{\mathbf{1}^{\mathrm{T}}\mathbf{Q}(\mathbf{t} - p \cdot \mathbf{1})}. \tag{13}$$

Substituting (8), we can write

$$\mathbf{Q}(\mathbf{t} - p \cdot \mathbf{1}) \;=\; (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{n} \;=\; \mathbf{Q}\mathbf{X}^{\mathrm{T}}\mathbf{n}. \tag{14}$$

Observe that the monotonicity condition $\nabla_{\mathbf{t}} t_0 > 0$ is satisfied when either $\mathbf{Q}\mathbf{X}^{\mathrm{T}}\mathbf{n} > 0$, or $\mathbf{Q}\mathbf{X}^{\mathrm{T}}\mathbf{n} < 0$, that is, both coordinates of $\mathbf{Q}\mathbf{X}^{\mathrm{T}}\mathbf{n}$ have the same sign. However, since the consistency of the solution requires $\mathbf{X}^{\mathrm{T}}\mathbf{n}$ to be negative, and $\mathbf{Q}$ is positive semi-definite, $\mathbf{Q}\mathbf{X}^{\mathrm{T}}\mathbf{n}$ cannot have both coordinates positive. We therefore conclude that the solution has to satisfy $\mathbf{Q}\mathbf{X}^{\mathrm{T}}\mathbf{n} = \mathbf{Q}(\mathbf{t} - p \cdot \mathbf{1}) < 0$. This yields $\mathbf{1}^{\mathrm{T}}\mathbf{Q}(\mathbf{t} - p \cdot \mathbf{1}) < 0$. The latter condition can be rewritten as

$$0 \;>\; \mathbf{Q}(\mathbf{t} - p \cdot \mathbf{1}) \;=\; (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{n}, \tag{15}$$

where the inequality is interpreted coordinate-wise. Observe that the rows of the matrix $(\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}$ are orthogonal to $\mathbf{x}_1$, $\mathbf{x}_2$, or in other words, are normal to the triangle edges. This gives the following geometric interpretation of the monotonicity condition: the direction $-\mathbf{n}$ must come from within the triangle. Since the update direction also obeys the consistency condition, any direction coming from within the triangle must form acute angles with the triangle edges, leading to the demand that the angle $\sphericalangle \mathbf{x}_1\mathbf{x}_0\mathbf{x}_2$ is acute (or, equivalently, $e_{12} > 0$).

Consistency and monotonicity conditions should guarantee that the update is performed only from a triangle that contains the characteristic direction, which makes the update scheme upwind [Sethian and Vladimirsky 2000]. However, since $\mathbf{n}$ is only an approximation of the characteristic direction, it may happen that the conditions are not satisfied although the true characteristic lies inside the triangle. For a sufficiently small triangle, this can happen only if any of the two inner products $\mathbf{n}^{\mathrm{T}}\mathbf{x}_1$, $\mathbf{n}^{\mathrm{T}}\mathbf{x}_2$ is sufficiently close to zero. This corresponds to the situation in which t_0 can be updated from one of the triangle edges (one-dimensional simplices) $\mathbf{x}_0\mathbf{x}_1$, $\mathbf{x}_0\mathbf{x}_2$. In this case, the simple Dijkstra-type update,

$$t_0 \;=\; \min\{t_1 + \|\mathbf{x}_1\|, t_2 + \|\mathbf{x}_2\|\}, \tag{16}$$

is performed.

In order to ensure that the update formula is numerically stable, we assume that t_i is affected by a small error ε, which, in turn, influences the computed time of arrival t_0. Using first-order Taylor expansion, we have

$$\tilde{t}_0 \;\approx\; t_0 + \varepsilon \cdot \frac{\partial t_0}{\partial t_i} \;\le\; t_0 + \varepsilon \cdot \left(\left| \frac{\partial t_0}{\partial t_1} \right| + \left| \frac{\partial t_0}{\partial t_2} \right| \right). \tag{17}$$

Under the monotonicity condition $\nabla_{\mathbf{t}}\, t_0 > 0$, we can write

$$\tilde{t}_0 \;\approx\; t_0 + \varepsilon \cdot \mathbf{1}^{\mathrm{T}}\nabla_{\mathbf{t}}\, t_0 \;=\; t_0 + \varepsilon \cdot \frac{\mathbf{1}^{\mathrm{T}}\mathbf{Q}(\mathbf{t} - p \cdot \mathbf{1})}{\mathbf{1}^{\mathrm{T}}\mathbf{Q}(\mathbf{t} - p \cdot \mathbf{1})} \;=\; t_0 + \varepsilon. \tag{18}$$

The error in t_0 is also bounded in the one-dimensional Dijkstra-type update, which makes the update formula stable.

The planar wavefront update scheme is summarized in Algorithm 2. Note that it is valid only for acute triangulations; when some triangles have obtuse angles ($e_{12} < 0$), they have to be split by adding connections to additional neighbor grid points, as proposed by Spira and Kimmel in [2004].

Algorithm 2: Planar update scheme for acute triangulation.

1 Set $t_0^{\text{new}} \longleftarrow t_0$.
2 **foreach** *triangle* $\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2)^{\mathrm{T}}$ **do**
3 Solve the quadratic equation (10) for t_0.
4 **if** $\boldsymbol{Q}(\boldsymbol{t} - t_0 \cdot \boldsymbol{1}) > 0$ *or* $t_0 < \max\{t(\boldsymbol{x}_1), t(\boldsymbol{x}_2)\}$ **then** compute t_0 according to (16).
5 Set $t_0^{\text{new}} \longleftarrow \min\{t_0^{\text{new}}, t_0\}$.
6 **end**

3.2 Spherical wavefront approximation

A different update scheme was proposed by Novotni and Klein [Novotni and Klein 2002]. They update a vertex with its Euclidean distance from a virtual point source, whose coordinates are estimated from the times of arrival to the two supporting vertices. This approach is similar in its spirit to the Mitchel-Mount-Papadimitriou algorithm [Mitchell et al. 1987; Surazhsky et al. 2005], and should apparently be more accurate that its planar counterpart for computing distance maps from point sources. Here, we show that this scheme can be inconsistent and numerically unstable.

According to Novotni and Klein, the wavefront is modeled as a spherical (circular) wave propagating from a virtual point source $\mathbf{x}$ (Figure 2, right). Demanding that the supporting vertices $\mathbf{x}_1$, $\mathbf{x}_2$ of the triangle lie at distances t_1 and t_2, respectively, from the source, we obtain for $i = 1, 2$

$$t_i^2 \;=\; (\mathbf{x}_i - \mathbf{x})^{\mathrm{T}}(\mathbf{x}_i - \mathbf{x}) \;=\; \mathbf{x}_i^{\mathrm{T}}\mathbf{x}_i - 2\mathbf{x}_i^{\mathrm{T}}\mathbf{x} + \mathbf{x}^{\mathrm{T}}\mathbf{x}. \tag{19}$$

The time of arrival of the wavefront to the updated vertex $\mathbf{x}_0$ is given by its distance from the point source,

$$t_0^2 \;=\; (\mathbf{x}_0 - \mathbf{x})^{\mathrm{T}}(\mathbf{x}_0 - \mathbf{x}) \;=\; \mathbf{x}^{\mathrm{T}}\mathbf{x}. \tag{20}$$

Denoting $s_i = t_i^2$, and $\mathbf{q} = (s_1 - \mathbf{x}_1^{\mathrm{T}}\mathbf{x}_1, s_2 - \mathbf{x}_2^{\mathrm{T}}\mathbf{x}_2)^{\mathrm{T}}$, we obtain

$$s_0 \cdot \mathbf{1} - 2\mathbf{X}^{\mathrm{T}}\mathbf{x} \;=\; \mathbf{q}. \tag{21}$$

Assuming the mesh to be non-degenerate,

$$\mathbf{x} \;=\; \frac{1}{2}\mathbf{X}(\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}(s_0 \cdot \mathbf{1} - \mathbf{q}) \;=\; \frac{1}{2}\mathbf{X}\mathbf{Q}(s_0 \cdot \mathbf{1} - \mathbf{q}). \tag{22}$$

Plugging the later result into (20), we have

$$\begin{aligned} s_0 \;&=\; \mathbf{x}^{\mathrm{T}}\mathbf{x} \;=\; \frac{1}{4}(s_0 \cdot \mathbf{1} - \mathbf{q})^{\mathrm{T}}\mathbf{Q}(s_0 \cdot \mathbf{1} - \mathbf{q}) \\ &=\; \frac{1}{4}\left(s_0^2 \cdot \mathbf{1}^{\mathrm{T}}\mathbf{Q}\mathbf{1} - 2s_0 \cdot \mathbf{1}^{\mathrm{T}}\mathbf{Q}\mathbf{q} + \mathbf{q}^{\mathrm{T}}\mathbf{Q}\mathbf{q}\right). \end{aligned} \tag{23}$$

Consequently, t_0 is given as the largest positive solution of the following bi-quadratic equation

$$t_0^4 \cdot \mathbf{1}^{\mathrm{T}}\mathbf{Q}\mathbf{1} - 2t_0^2 \cdot (\mathbf{1}^{\mathrm{T}}\mathbf{Q}\mathbf{q} + 2) + \mathbf{q}^{\mathrm{T}}\mathbf{Q}\mathbf{q} \;=\; 0. \tag{24}$$

In order to enforce consistency, we require

$$\begin{aligned} s_0 \;>\; s_i \;&=\; (\mathbf{x}_i - \mathbf{x})^{\mathrm{T}}(\mathbf{x}_i - \mathbf{x}) = \mathbf{x}_i^{\mathrm{T}}\mathbf{x}_i - 2\mathbf{x}_i^{\mathrm{T}}\mathbf{x} + \mathbf{x}^{\mathrm{T}}\mathbf{x} \\ &=\; \mathbf{x}_i^{\mathrm{T}}(\mathbf{x}_i - 2\mathbf{x}) + s_0, \end{aligned} \tag{25}$$

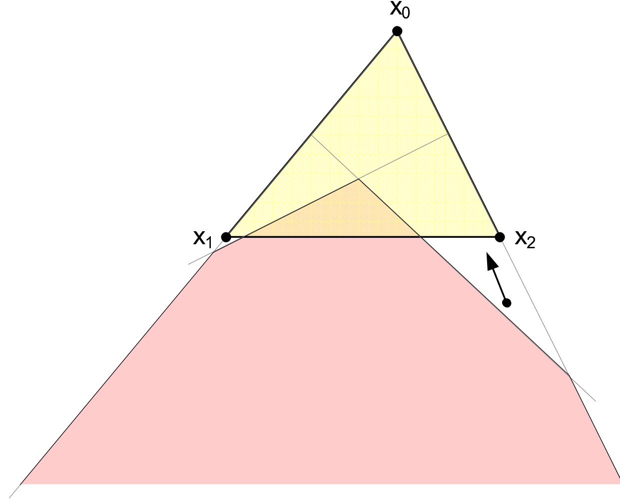

Fig. 3. Consistency and monotonicity conditions of the spherical update scheme require that the virtual source lie inside the region shaded in red. Some update directions coming from within the triangle are outside that region.

or, alternatively,

$$\mathbf{x}_i^{\mathrm{T}} \left(\mathbf{x} - \frac{1}{2} \mathbf{x}_i \right) \; > \; 0. \tag{26}$$

The geometric interpretation of the former condition is that the source point $\mathbf{x}$ lies at the "positive" sides of the two perpendicular bisectors to the edges $\mathbf{x}_0\mathbf{x}_1$ and $\mathbf{x}_0\mathbf{x}_2$.

To enforce monotonicity, we differentiate (24) with respect to $\mathbf{s} = (s_1, s_2)^{\mathrm{T}}$,

$$2s_0 \cdot \nabla_{\mathbf{s}}\, s_0 \cdot \mathbf{1}^{\mathrm{T}}\mathbf{Q}\mathbf{1} - 2\nabla_{\mathbf{s}}\, s_0 \cdot (\mathbf{1}^{\mathrm{T}}\mathbf{Q}\mathbf{q} + 2) - 2s_0 \cdot \mathbf{Q}\mathbf{1} + 2\mathbf{Q}\mathbf{q} = 0, \tag{27}$$

from where

$$\nabla_{\mathbf{s}}\, s_0 \;=\; \frac{\mathbf{Q}(s_0 \cdot \mathbf{1} - \mathbf{q})}{\mathbf{1}^{\mathrm{T}}\mathbf{Q}(s_0 \cdot \mathbf{1} - \mathbf{q}) - 2}. \tag{28}$$

Requiring $\nabla_{\mathbf{s}}\, s_0 > 0$ in conjunction with the consistency condition yields $\mathbf{Q}(s_0 \cdot \mathbf{1} - \mathbf{q}) > 0$, or

$$\mathbf{Q}\mathbf{X}^{\mathrm{T}}\mathbf{x} > 0, \tag{29}$$

which can be interpreted geometrically as a demand that $\mathbf{x}$ lies inside the angle $\sphericalangle \mathbf{x}_1\mathbf{x}_0\mathbf{x}_2$.

Figure 3 shows that some characteristic directions lying inside the triangle violate consistency or monotonicity. Therefore, the spherical wavefront update scheme is likely to introduce errors that will propagate with the computed front. Also note that unlike its planar counterpart, the spherical wavefront propagation model is not numerically stable. Observe that $\|\nabla_{\mathbf{s}}\, s_0\| > 1$ for any $\mathbf{x}$, and for $\mathbf{x}$ lying on the edge $\mathbf{x}_1\mathbf{x}_2$, the gradient is infinite, meaning that roundoff errors can potentially explode, invalidating the numerical solution.

These disadvantages of the spherical wavefront scheme become less pronounced for $\mathbf{t} \gg \ell$, where ℓ is the largest triangle edge length. However, for large values of the arrival times, the spherical and the planar models converge and produce nearly identical solutions. Due to these difficulties, in this paper we use the planar wavefront update scheme.

4. RASTER SCAN ALGORITHM

One of the disadvantages of the fast marching algorithm is that it is inherently sequential, thus allowing no parallelization. In addition, the order of visiting the grid points depend on the shape of the propagating wavefront and is therefore data-dependent. This results in irregular memory access that is unlikely to utilize the caching system efficiently. These drawbacks call for searching for alternative grid traversal orders.

In his classical paper, Danielsson [1980] observed that since the geodesics on the Euclidean plane are straight lines, all possible characteristic directions of the eikonal equation fall into one of the four quadrants of a Cartesian grid and can be therefore covered by traversing the grid in four directed raster scans. Danielsson's raster scan spirit (commonly referred to as *fast sweeping*) was adopted in [Zhao 2004] for solving the eikonal equation on weighted Euclidean domains, and in Bornemann and Rasch [Bornemann and Rasch 2006]; similar ideas date back to Dupuis and Oliensis' studies on shape from shading [Dupuis and Oliensis 1994].

Raster scan traversal has linear complexity in the grid size, and is characterized by regular access to memory, which increases the efficiency of caching. Since the order of visiting of the grid points is independent of the data and is known in advance, one can use the pre-caching mechanism, supported in many modern processors. In addition, unlike its priority queue-based counterpart, raster scan can be efficiently parallelized as will be shown in Section 5.

Here, we use the raster scan order to traverse the Cartesian grid in the surface parametrization domain, as summarized in Algorithm 3. As in the priority queue-based traversal order, all computations are done in the parametrization domain, taking into account the metric on the surface. Since each directed raster scan covers only 90° of possible characteristic directions, the update of a point on the grid can be done only from the triangles containing that direction. For example, if the eight-neighbor grid connectivity is used, only two triangles formed by three neighbors are absolutely required in the update (Figure 4, first row).

Observe that unlike the Euclidean case where the characteristics are straight lines, on a general geometry image, the characteristics in the parametrization domain are usually curved. This implies that the four raster scans may cover only a part of a characteristic, and have to be repeated more times in order to produce a consistent distance map. As a consequence, the complexity of the raster algorithm for geometry images is $\mathcal{O}(N_{\text{iter}} \cdot n)$, where n is the grid size, and N_{iter} is the data-dependent number of iterations. In what follows, we present a bound on the maximum number of iterations required before the algorithm stops.

PROPOSITION 1. *Algorithm 3 applied to a geometry image $\boldsymbol{x}(\mathbf{U})$ will stop after at most*

$$N_{\text{iter}} \leq \left\lceil \frac{2D\,\lambda^{\boldsymbol{G}}_{\max}}{\pi \lambda^{\boldsymbol{G}}_{\min}} \sqrt{(\lambda^{\boldsymbol{H}^1}_{\min})^2 + (\lambda^{\boldsymbol{H}^2}_{\min})^2 + (\lambda^{\boldsymbol{H}^3}_{\min})^2} \right\rceil + 1$$

iterations, where D is the surface diameter, $\lambda^{\boldsymbol{H}^i}_{\min}$ is the smallest eigenvalue of the Hessian matrix $\boldsymbol{H}^i = \nabla^2_{\boldsymbol{uu}} x^i$ of x^i with respect to the parametrization coordinates $\boldsymbol{u}$, and $\lambda^{\boldsymbol{G}}_{\max}/\lambda^{\boldsymbol{G}}_{\min}$ is the condition number of the metric $\boldsymbol{G}$.

For proof, see Appendix A. When the surface is given as a graph of a function $z(x,y)$, the bound can be simplified as

$$N_{\text{iter}} \leq \left\lceil \frac{2D\,\lambda^{\mathbf{G}}_{\max}}{\pi \lambda^{\mathbf{G}}_{\min}} \lambda^{\mathbf{H}}_{\min} \right\rceil + 1, \tag{30}$$

where $\mathbf{H} = \nabla^2 z$.

The main significance of this bound is that the maximum number of iterations does not depend on the discretization of $\mathbf{U}$ and is a constant regardless of the grid size. Note, however, that the bound depends both on the properties of the surface expressed in terms of the metric $\mathbf{G}$ and the diameter D, and those of the parametrization expressed in terms

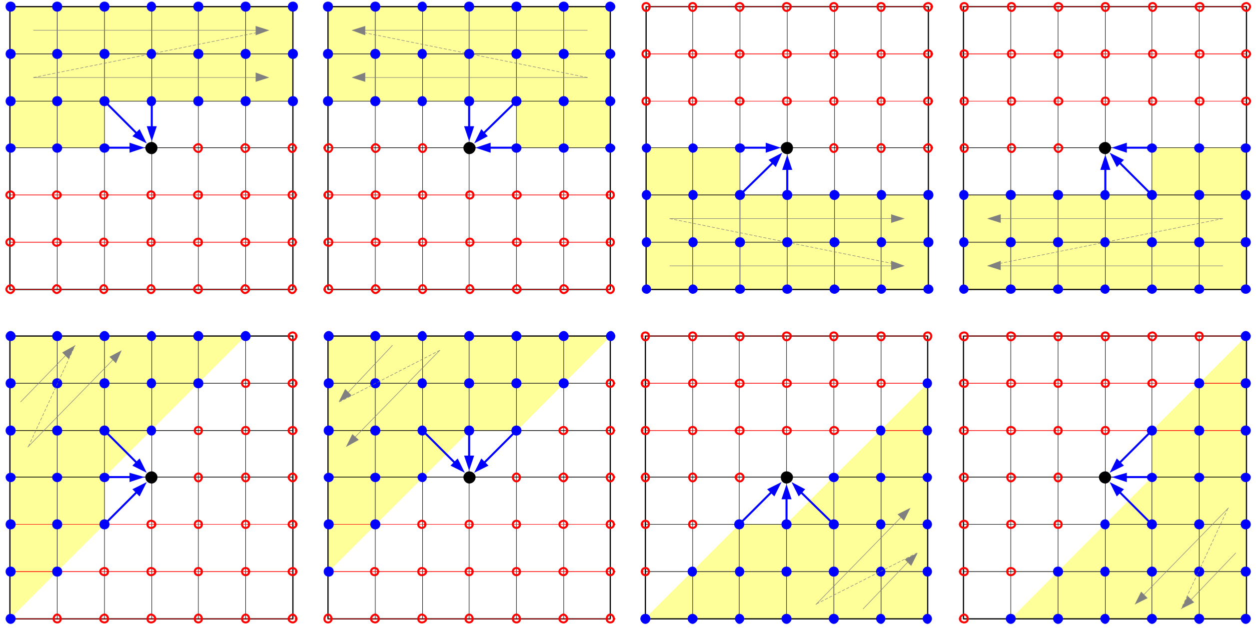

Fig. 4. Update of a point on a grid with eight-neighbor connectivity using the raster scan algorithm. First row: four directed raster scans; second row: the same raster scans rotated by 45°.

of the Hessian $\mathbf{H}^i$. This means that some parametrizations of the same surface may be less favorable for the raster scan algorithm. For example, in the parametrization $\mathbf{x} = (u^1 \cos u^2, u^1 \sin u^2, 0)^{\mathrm{T}}$ of a flat disc, the characteristics in the parametrization domain are curved and require multiple iterations to be covered.

Note that the bound is a worst case bound; in practice the number of iterations required for convergence may be smaller. Adding another triangle to the grid update such that every grid point is updated from four "causal" (in the raster scan order) neighbors rather than from three causal neighbors as shown in Figure 4 may reduce the number of iterations. It is important to emphasize that in the worst case N_{iter} will remain unchanged.

The main disadvantage of the raster scan algorithm is the lack of flexibility in controlling the tradeoff between the algorithm complexity and accuracy, unlike some other distance approximation algorithms, like the phase flow [Ying and Candès 2006] and the approximate MMP algorithm [Surazhsky et al. 2005].

5. PARALLELIZATION

The structure of the raster scan algorithm gives much opportunity for exploiting data independence to compute some of the grid updates concurrently on a set of parallel computation units. To demonstrate the parallelism, let us consider for example the right-down raster scan, starting from the top leftmost grid point t_{11}. After t_{11} has been updated, the points t_{12} and t_{21} can be updated concurrently, since their updates do not depend on each other. Next, the points t_{31}, t_{22} and t_{13} are updated concurrently, and so on (Figure 5, left). Assuming the number of available computation units is $P \geq \min\{M,N\}$, the right-down raster scan can be performed in $M+N-1$ steps, where at each step k the points along the line $i+j=k+1$ are updated. If the number of processors is smaller, every step is serialized into $\lceil (k+1)/P \rceil$ sub-steps. The other three directed raster scans are parallelized in the same manner.

Algorithm 3: Raster scan algorithm on a single-chart geometry image.

Input: Numerical $M \times N$ grid $\mathbf{U}$, set of source points $S \subset \mathbf{U}$ with the corresponding initial values $t(s)$
Output: The distance map $t : \mathbf{U} \mapsto \mathbb{R}^+$.

```
   Initialization
1  Pre-compute the update equation coefficients for each triangle.
2  foreach point u ∈ U \ S do t(u) ⟵ ∞
   Iteration
3  for iter = 1,2,... do
4      for i = 1,2,...,M do
           Right-up scan
5          for j = 1,2,...,N do Update (u_ij)
           Right-down scan
6          for j = N,N−1,...,1 do Update (u_ij)
7      end
8      for i = M,M−1,...,1 do
           Left-up scan
9          for j = 1,2,...,N do Update (u_ij)
           Left-down scan
10         for j = N,N−1,...,1 do Update (u_ij)
11     end
12     if ||t^(n) − t^(n−1)|| = 0 then stop
13 end
```

An obvious disadvantage of such a parallelization is the lack of data coherence in the memory, which may deteriorate performance on many architectures such as GPUs. Another disadvantage is the fact that the number of operations in each step is not constant and the benefit from the parallelization is obtained only on sufficiently long diagonals. A way to overcome these two difficulties is to rotate the direction of all raster scans by 45° (Figure 4, second row). Using the rotated raster scans, rows or columns of the grid can be updated concurrently (Figure 5, right). This allows coherent access to memory and provides better parallelization with a speedup factor of P. Since the same operations are performed to update all the grid points, the algorithm is suitable for implementation on a SIMD processor. We refer to this parallelized scheme as to *parallel marching*.

5.1 Extensions to multi-chart geometry images

The approach presented so far is limited to geometry images represented as a single chart, though the latter can be of arbitrarily complex topology (such a topology usually introduces "holes" in the parametrization domain, which can be handled efficiently by "masking" the update in those regions). This may be a major limitation in many practical application, where due to the varying level of detail on the surface, the representation as a single-chart geometry image is either inaccurate or inefficient.

Here, we discuss a generalization of the raster scan algorithm to *multi-chart geometry images*, i.e. surfaces represented as an atlas of overlapping charts. Formally, we are given a collection of K maps $\mathbf{x}^k : \mathbf{U}^k \rightarrow \mathbb{R}^3$, $k = 1,...,K$, where each $\mathbf{U}^k$ is sampled on a regular Cartesian grid, usually, with different sampling density, according to the detail level of the underlying surface. For simplicity, we assume that the charts overlap only at the boundaries, i.e. for two neighboring charts i and j, $\mathbf{x}^i(\mathbf{U}^i) \cap \mathbf{x}^j(\mathbf{U}^j) \subseteq \mathbf{x}^i(\partial \mathbf{U}^i), \mathbf{x}^j(\partial \mathbf{U}^j)$. We denote by $\mathscr{P}_{ij}$ an operator projecting the values of the distance

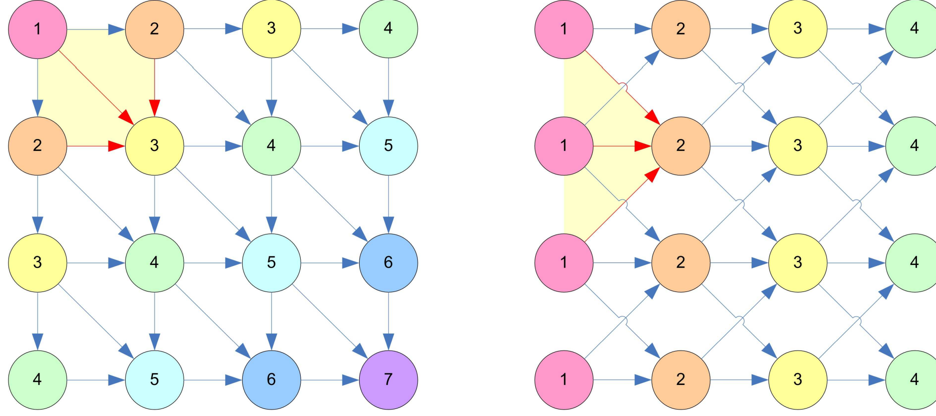

Fig. 5. Dependency graph in the right-down (left) and the rotated up-left-down (right) raster scan updates. Grid point updates that can be computed concurrently are numbered and shaded with different colors.

map t^i from $\partial\mathbf{U}^i$ onto the shared portion of $\partial\mathbf{U}^j$.

The charts are organized as an undirected adjacency graph with K vertices, and edges (i, j) corresponding to each pair of adjacent charts i and j. We denote by $\mathscr{N}_i$ the collection of all the neighboring charts of the chart i. The problem of distance computation on a multi-chart geometry image can be thought of as distance computation in such an adjacency graph, in which each vertex represents a chart, and can be therefore solved using a generalization of the Dijkstra's algorithm (Algorithm 4).

Each chart is associated with a quadruplet (k,t,S_0,t_0), where $k = 1,...,K$ is a chart index, and t is a scalar distance value, whose assignment is discussed in the sequel. t_0 denotes a set of fixed values on $S_0 \subseteq \mathbf{U}^k$ serving as the boundary conditions. Additionally, a distance map $t^k : \mathbf{U}^k \to \mathbb{R}^+$ is maintained for each chart. The algorithm maintains a priority queue Q holding as its entries the quadruplets (k,t,S_0,t_0). Similarly to the standard Dijkstra's algorithm, the queue is sorted according to t. Initially, the queue is filled with the source values given as the input. For example, when computing the distance map from a single point $u \in \mathbf{U}^k$, Q is set to $\{(k,0,u,0)\}$ (note, however, that the source is not necessarily limited to a single point, and may span across multiple charts). The algorithm proceeds by removing the quadruplet (k,t,S_0,t_0) with the minimum t from the queue, and running the single-chart raster scan algorithm on $\mathbf{U}^k$ with S_0 and $t_0(S_0)$ serving as the source. This produces the distance map $t^k : \mathbf{U}^k \to \mathbb{R}^+$. Next, the values of t^k on the chart boundary are projected onto the neighbor charts $\mathbf{U}^i$, $i \in \mathscr{N}_k$, using the operators $\mathscr{P}_{ki}$. In order to guarantee monotonicity, the minimum between the extrapolated value $\mathscr{P}_{ki}t^k$ and the current value of t^i is used. We denote by δ^i the maximum difference between the previous and the new value of t^i on the shared portion of boundary $\partial\mathbf{U}^i$. A non-trivial δ^i implies that the distance map t^i is not up-to-date. In such a case, points on the boundary $\partial\mathbf{U}^i$ where the distance map value has decreased are added to the initial set of source points, and the chart is added to the priority queue with the distance value t set to be the minimum value of the updated points. The algorithm terminates when the queue becomes empty, implying by construction that all distance maps are up-to-date.

Unlike the standard Dijkstra's algorithm, the described procedure is no more a single-pass algorithm. In fact, a geodesic can pass back and forth from one chart to another, resulting in multiple updates of both. However, the number of such repetitions is bounded under conditions similar to those stated in Proposition 1.

Algorithm 4: Raster scan algorithm on a multi-chart geometry image.

Input: K grids $\mathbf{U}^1,...,\mathbf{U}^K$; projection operators $\mathscr{P}_{ij}$; sources with corresponding initial values $\{(S^1_{\text{init}},t^1_{\text{init}}),...,(S^K_{\text{init}},t^K_{\text{init}})\}$ on one or more charts.
Output: The distance maps $t^k:\mathbf{U}^k\mapsto\mathbb{R}^+$ for $k=1,...,K$.

1 Initialize the priority queue to $Q=\emptyset$.
2 **for** $k=1,...,K$ **do**
3   **if** $S^k_{\text{init}}\neq\emptyset$ **then** set $t_{\min}=\min\{t^k_{\text{init}}(u):u\in S^k_{\text{init}}\}$, and add $(k,t_{\min},S^k_{\text{init}},t^k_{\text{init}})$ to the queue.
4 **end**
5 **while** $Q\neq\emptyset$ **do**
6   Set $(k,t_{\min},S_0,t_0)=\arg\min\{t_{\min}:(k,t_{\min},S_0,t_0)\in Q\}$, and remove it from the queue.
7   Run the single-chart raster scan algorithm to compute the distance map t^k on $\mathbf{U}^k$ using $t_0(S_0)$ as the source.
8   **forall** $i\in\mathscr{N}_k$ **do**
9   Set $\bar{t}_i=\mathscr{P}_{ki}t^k$.
10   **forall** $u\in\partial\mathbf{U}^i\cap\partial\mathbf{U}^k$ **do**
11   Set $t^i(u)=\min\{\bar{t}_i(u),t_i(u)\}$ for all $u\in\partial\mathbf{U}^i$, and
12   **end**
13   Set $S^i_{\text{upd}}=\{u\in\partial\mathbf{U}^i\cap\partial\mathbf{U}^k:t_i(u)-\bar{t}_i(u)>\varepsilon\}$, and update $t^i(u)=\bar{t}_i(u)$ on $u\in S^i_{\text{upd}}$.
14   **if** $S^i_{\text{upd}}\neq\emptyset$ **then** compute $t^i_{\min}=\min\{t^i(u):u\in S^i_{\text{upd}}\}$;**else** set $t^i_{\min}=\infty$.
15   **end**
16   Find $i=\arg\min\{t^i_{\min}:i\in\mathscr{N}_k\}$.
17   **if** $t^i_{\min}<\infty$ **then** set $S_0=S^i_{\text{init}}\cup\partial\mathbf{U}^i$, $t_0=t^i_{\text{init}}(S^i_{\text{init}})\cup t^i(\partial\mathbf{U}^i)$, and add $(i,t^i_{\min},S_0,t_0)$ to the queue.
18 **end**

6. DISTANCE MAP COMPUTATION ON A GPU

Modern GPUs are extremely powerful processors, capable of performing near trillions of operations (teraflop) per second. The reason for such high performance originates from the computation-hungry computer graphics applications, such as rendering and texture mapping. Though the first GPUs were designed exclusively for these applications, the availability of so powerful architectures led to numerous attempts to employ graphics hardware for computationally-demanding applications besides computer graphics, e.g. scientific computing. This has evolved into a trend of general-purpose computing on GPUs [GPG], to which the manufacturers of graphics processors responded with developing a new generation of programmable GPUs. NVIDIA [CUD] and AMD [CTM], the two major GPU vendors, released their first GPGPU environments in early 2007. The new platforms completely hide the low level graphics functionality of the GPU and exposes the GPU as a general massively parallel machine capable of running thousands of threads concurrently. With the new environment, the developers do not need to have prior computer graphics knowledge and the programming is done by using high-level languages such as C.

6.1 CUDA

In this paper, we used the Compute Unified Device Architecture (CUDA) platform developed by NVIDIA for the implementation of the PMM algorithm. Similar results could be obtained by using the AMD platform [CTM]. For the sake of completeness, we briefly overview the most important features of CUDA; for a comprehensive review, refer to the CUDA programming guide [CUD].

The G8X series GPUs supporting CUDA have multiple independent processing units. When programmed using

CUDA, the GPU is viewed as a processing unit capable of executing thousands of threads in parallel. Both the CPU (referred to as *host*) and the GPU (referred to as *device*) maintain their own memory. Data can be transferred from one memory to another over the PCIe bus, yet the bandwidth of this bus (4 GB/sec) is significantly smaller compared to the bandwidth of internal buses on the GPU (100 GB/sec for the latest NVIDIA GPUs).

CUDA provide access to device DRAM memory either through global memory or texture memory. In addition, CUDA features on-chip shared memory with extremely fast general read/write access, used to share data between threads. Texture memory is a read-only memory with a cache optimized for 2D spatial locality. The global memory is a read/write non-cached memory space. Access to device memory is relatively slow compared to the speed of arithmetic calculations, making GPUs especially suitable for programs with high *arithmetic intensity* (ratio between ALU and memory access operations) and potentially inefficient for those with a low one. For example, the NVIDIA 8800GTX GPU can theoretically perform 345 G floating points operations/sec, while having the memory bandwidth of only 86 GB/sec = 22G floats/sec (when latency is completely hidden). In order to get better utilization of the computational power of the GPU and avoid the memory bandwidth bottleneck, memory access to device memory should be minimized. One way of doing so is by fetching a large portion of data from the global memory into shared memory (access to shared memory can be as fast as reading from a register) followed by as much as possible computations, and finally writing back the result to the global memory. The shared memory in this case can be thought of as user-managed cache.

Access to global memory should be coherent in a way that subsequent threads should access subsequent addresses in linear memory. There is no obligation to use such an access pattern. However, incoherent accesses will lead to extremely slow memory bandwidth. Texture memory is more flexible in the sense that coherence is two-dimensional. However, this is a read-only memory and there is no manual control over the caching scheme.

The architecture of the computational units in the GPU is *single-program-multiple-data* (SPMD), allowing to execute the same function independently on different data. Functions executed in this way are called *kernels*; the execution of a kernel is organized as a grid of *thread blocks*. A thread block is a batch of *threads* that can cooperate together by efficiently sharing data through fast shared memory and synchronizing their execution to coordinate memory accesses. The maximum number of threads in a single thread block is limited, however, many thread blocks can be batched together into a grid of blocks that can be processed by the device in parallel. It is important to note that communication and synchronization between threads of different thread blocks on the same grid is impossible. The only way to impose a global synchronization point on all threads is to divide the work into separate kernel invocations.

A serious limitations of the device memory is the memory latency ($400-600$ cycles). Much of this latency can be hidden by the GPU thread scheduler if there are sufficient independent arithmetic instructions that can be issued while waiting for memory accesses to complete. This means that while some threads are stalled by memory latency, others can progress with ALU computations. This is only possible if there are enough threads waiting for execution which implies that a grid of thread blocks should contain as many as possible threads.

6.2 Algorithm

Although CUDA is a significant step towards general purpose computing on GPUs, mapping a sequential algorithm from CPU to GPU is not trivial. Besides requiring a parallel version of the algorithm, certain restrictions should be fulfilled in order for the implementation to be efficient. In this section, we describe how to efficiently map PMM to GPU architecture.

As we mentioned in Section 5, the update of an entire $M \times N$ grid can be done in 4 subsequent scans (*up*, *down*, *left* and *right*). Each scan is further serialized into smaller parallel steps. For example, the *up* scan is composed of M serialized steps. In each each step, N vertices in a row are updated in parallel. The distance map is allocated in the

global memory space as a read/write linear array of 32-bit floating point values. The geometrical properties of the underlying surface are stored in the read-only texture memory space. We can pre-compute the coefficients used by the numerical update scheme at the pre-processing stage and store them into the texture memory instead of the actual locations of the vertices.

Straightforward implementation is done by mapping each row or column of the grid to a single kernel call. Each thread in each thread block updates the distance at a single vertex according to the previous distance at that vertex and three distances of the vertices at the previous row/column (Figure 6). Kernel invocations serve as global synchronization point so we can be sure that the next row/column will be processed only after the previous row/column is fully updated. The memory access for the *up* and *down* scans are coherent, yet, the *left/right* scans use an incoherent memory access pattern, since the addresses of elements in a single column are far from each other in linear memory.

In order to overcome this limitation, we propose to organize the data in the following way. We allocate 2 different arrays to hold the distance maps. The first map is used solely for the *up* and *down* scans, while the second map is used solely for the *left* and *right* scans. The *left/right* map is stored in a transposed manner so we access both *up/down* and *left/right* distance maps on a row-by-row basis. Since each scan depends on the result of the previous scan, we must copy the results obtained by the *up* and *down* scans from the *up/down* map to the *left/right* map. The task of copying the map in a transposed manner can be done efficiently with a single kernel invocation and without violating the coherence.[6] The basic idea is to first decompose the matrix into smaller blocks that can fully reside in the shared memory. Each block is transposed separately and is written in a coherent manner back to the global memory.

The proposed memory organization results in a better performance, but suffers from a different bottleneck. Invoking a single kernel for each row/column in the grid leads to $2M+2N$ kernel invocations. A kernel invocation consumes a fixed overhead regardless of how many computations are done in that kernel (up to 20 μsec per kernel). To demonstrate the severity of the problem, consider a grid with 3000×3000 points. The total time for the kernel invocations alone will be approximately $(2 \times 3000 + 2 \times 3000) \times 20\ \mu\text{sec} = 240$ msec. This time alone exceeds the total time of our optimized kernel computation by nearly an order of magnitude (see Table I).

A possible remedy is using a kernel that processes a batch of rows rather than a single row at a time. Each batch is composed of a strip of Ω consecutive rows, such that the total number of kernel invocations is reduced by a factor of Ω, to $\frac{2M+2N}{\Omega}$. For each row in the strip, each thread fetches the former distance at that vertex from the distance map into the shared memory. The thread then calculates the updated distance at that vertex and writes the result back to the distance map. All threads are then synchronized. Once all the threads reach the synchronization point, each thread can start working on the calculation of the next row in the strip. Besides having the advantage of reduced number of kernel invocations, this access pattern also leads to higher arithmetic intensity, since for a large enough Ω, a single fetch from the distance map per vertex is required (instead of four), since we can keep the computed distances of the previous row in the shared memory and do not need to read them again from global memory as we advance to the next row.

On the other hand, a new problem arises. While communication through shared memory and synchronization between threads of the same thread block is possible, there is no synchronization mechanism between threads of different thread blocks. Since the maximum number of threads in a thread block is limited (512 on latest NVIDIA GPU), we are limited to small grids only. Moreover, modern GPUs have several independent multiprocessors, working in parallel (16 on latest NVIDIA GPU) and since each thread block can be processed by a single multiprocessor, the utilization of the hardware will be poor.

Figure 6 shows a small portion of the distance map that is handled by a single thread block with 32 threads and $\Omega = 8$ rows. Note that thread 0 is guaranteed to produce valid result only at row 0 since at any other row, the update depends

[6]Refer to the "matrix transpose" example in [CUD] for further details.

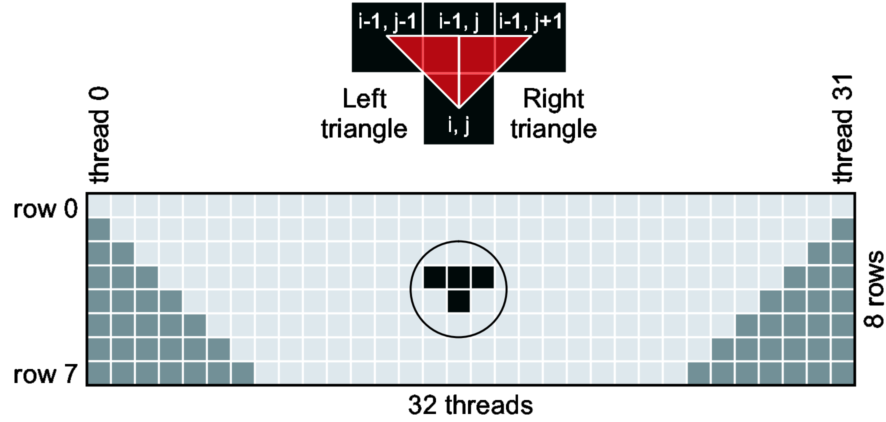

Fig. 6. Top: vertex (i, j) is updated based on two neighboring triangles. Bottom: a small portion of the distance map that is handled by a single thread block with 32 threads and $\Omega = 8$ rows. The light gray area is safe, while the dark gray one might contains errors.

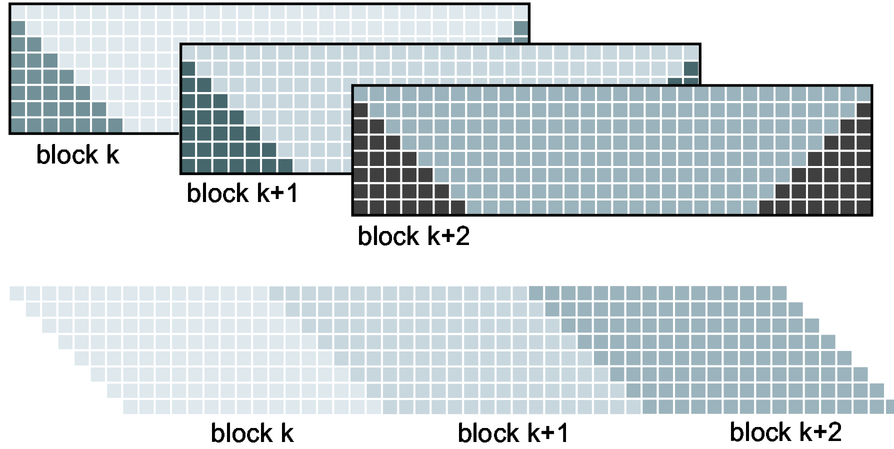

Fig. 7. Top: three overlapping blocks with corresponding "safe zones". Vertices in the overlapping region are computed twice by two threads belongs to adjacent thread blocks. Bottom: write access pattern. Each thread belongs to exactly one parallelogram, hence writes a single value to the distance map.

on vertices which are located left to row 0. Values that comes from the left cannot be trusted since they belongs to a different thread block. The potential errors on the first column may lead to errors on the second column as well (rows 2 to 7). In general, only the area shown in light gray in Figure 6 is a "safe zone".

An important observation is that we can still maintain consistent updates if we allow some partial overlap between subsequent thread blocks, such that each thread block updates vertices only in the "safe zone". The blocks should overlap in such a way that each vertex of the grid belongs to at least one "safe zone". Figure 7 shows a small portion of the distance map covered by three overlapping blocks. Note that some vertices belong to two "safe zones". In order to avoid write conflicts, we demand that in such a case, only the block on the right will write the updated distance to the global memory. This write access pattern is illustrated in Figure 7, where each vertex is updated by exactly one thread belonging to one of the parallelograms.

The amount of redundancy that is introduced due to repetitions depends on the ratio between the number of threads in a single block and Ω. In order to reduce the redundancy, we would like to keep Ω small for a fixed block size. On the other hand, larger value of Ω decrease the number of kernel invocations and improve the arithmetic intensity. We conclude that a correct balance between the block size and Ω is a key to achieving good hardware utilization.

Another way to reduce the number of kernel invocations and increase the parallelism is to combine the four grid scans into a single scan. Performing several scans in parallel does not change the correctness of the algorithm since in the worse case, it will have to perform four times more iterations in order to converge. When a geodesic in the parametrization domain changes its direction smoothly from a zone affected by the *up* scan to a zone affected by the *down* scan, it must cross the *left* or *right* zones first. For non-smooth geodesics with abrupt angles at some points, this will not hold and additional iterations might be needed. Since this case is rare, we can safely combine the *up* and *down* scans into a single scan followed by another scan combining the *left* and *right* scans.

7. NUMERICAL RESULTS

In order to demonstrate the efficiency of the proposed algorithms, we consider its two particular implementations. In the first implementation, the PMM algorithm was implemented in C on an Intel Pentium platform, with the update step written in inline assembly and taking advantage of the SSE2 extensions (Intel SIMD architecture). [7]

The second implementation was developed on an NVIDIA 8800GTX GPU with 768 MB memory using the CUDA environment.[8] This GPU has 16 independent multiprocessors, each containing 8 processors. During fine-tuning of our code, we ran it on variable grid sizes $\{64k \times 64k : k \in 1 \dots 47\}$. For each grid, we measured performance on several block sizes and different values of Ω and recorded the configurations which minimized the running time. In most runs, $\Omega = 16$ produced the best results. For relatively small grids (up to 768×768), a block size of 64 produced the best results. The reason is that the use of large blocks leads to a small number of active blocks at any given time, hence, resulting in not all the GPU multiprocessors being active. On large grids, a block size of 256 resulted in the best performance, reducing the amount of waisted computations to approximately 20% (note that even though the maximum block size is 512, configurations with too large blocks may lead to slower performance due to internal synchronization between threads in the same block or may even result in a non valid configuration for the hardware due to elimination of all hardware resources (registers, shared memory, etc.).

32-bit (single precision) floating point representation was used in both implementations. Pre-computation of geometry coefficients were excluded from time measurements (pre-computation on the GPU took around 9*ms* of preprocessing time on the largest grid with 3008×3008 vertices).

7.1 Performance benchmarks

Table I presents the execution times of the SSE2 and GPU implementations of the parallel marching algorithm on the sphere surface, with the number of vertices ranging from four thousand to nine million. In all cases, PMM converges in one iteration. Grid construction time (taking less than 10% of a single iteration time) was not measured. For comparison, execution time of the exact and approximate MMP algorithms implemented in [Surazhsky et al. 2005] are presented.

As appears from the table, the GPU outperforms its rivals on all grid sizes but the gap becomes more pronounced on large grids, where it outperforms the SSE2 implementation by nearly two orders of magnitude, achieving up to

[7] A packaged library is available from `http://www.cs.technion.ac.il/~weber/Shared/PMM/PMM_SSE2.rar`.
[8] A packaged library is available from `http://www.cs.technion.ac.il/~weber/Shared/PMM/PMM.rar`.

Vertices	MMP (Exact)	MMP (Approx.)	PMM SSE2	PMM GPU
4.1×10^3	0.4406	0.0982	0.0011	0.0006
16.4×10^3	4.4566	0.3898	0.0042	0.0009
65.5×10^3	54.886	1.6503	0.0172	0.0015
0.49×10^6	—	13.647	0.1308	0.0045
0.86×10^6	—	25.639	0.2306	0.0059
2.56×10^6	—	—	0.6791	0.0133
4.19×10^6	—	—	1.1301	0.0287
9.04×10^6	—	—	—	0.0389

Table I. Execution time (in seconds) of different geodesic distance computation algorithms on the sphere surface. For the exact and approximate MMP algorithms, the code by Surazhsky *et al.* was used. Execution time of PMM is given for one iterations required for algorithm convergence.

	MMP Approx.			PMM		
Grid step	Mean abs err	Mean rel err	Max abs err	Mean abs err	Mean rel err	Max abs err
1.56×10^{-2}	2.47×10^{-3}	1.79×10^{-3}	9.99×10^{-3}	7.11×10^{-3}	6.49×10^{-3}	1.26×10^{-2}
7.81×10^{-3}	1.17×10^{-3}	8.52×10^{-4}	4.88×10^{-3}	4.67×10^{-3}	4.34×10^{-3}	8.15×10^{-3}
3.91×10^{-3}	5.73×10^{-4}	4.16×10^{-4}	2.41×10^{-3}	2.91×10^{-3}	2.74×10^{-3}	5.05×10^{-3}
1.42×10^{-3}	2.05×10^{-4}	1.49×10^{-4}	8.71×10^{-4}	1.38×10^{-3}	1.31×10^{-3}	2.37×10^{-3}
1.08×10^{-3}	1.55×10^{-4}	1.13×10^{-4}	6.60×10^{-4}	1.08×10^{-3}	1.03×10^{-3}	1.86×10^{-3}
6.25×10^{-4}	—	—	—	7.23×10^{-4}	6.92×10^{-4}	1.24×10^{-3}
4.88×10^{-4}	—	—	—	5.92×10^{-4}	5.68×10^{-4}	1.01×10^{-3}
3.22×10^{-4}	—	—	—	4.36×10^{-4}	4.16×10^{-4}	7.38×10^{-4}

Table II. Accuracy of the approximate MMP and PMM on the sphere surface, measured in terms of the mean absolute (L_1) error, maximum absolute (L_∞) error, and mean relative error as a function of the grid sampling step. In both cases, the error depends approximately linearly on the sampling step, in accordance with the theoretical first-order accuracy.

240 million distance computations per second. For the same grid size, the SSE2 and the GPU PMM outperform the state-of-the-art approximate MMP algorithm by three and four orders of magnitude, respectively.

The data transfer rates between the CPU and the GPU are limited by the bus bandwidth (theoretical 4 GB/sec, observed 2.6 GB/sec). For example, for the largest grid with nine million vertices, the download time was $53ms$ (4 floats per vertex) and the upload time was $17ms$ (one float per vertex). In most typical scenarios, the geometry image is transferred to the GPU only once and several distance maps are computed, making this preprocessing time negligible. Moreover, modern GPUs are capable of performing kernels in asynchronous manner leading to better hiding of data transfer overheads.

Table II presents the accuracy of different geodesic distance computation algorithms, quantified in terms of the mean absolute (L_1) error, maximum absolute (L_∞) error, and mean relative error for different grid sampling steps. This comparison gives a fairly conservative bound on the accuracy of PMM, as the latter depends on the specific surface parametrization. In Figure 8, the complexity is presented as a function of the algorithm accuracy and the grid size. The SSE2 PMM achieves the accuracy of the approximate MMP algorithm at less than 10% of its complexity, whereas the GPU implementation requires about 0.3% complexity.

Figure 9 presents the number of distances per second computed by the GPU. Best utilization of the computational units is achieved for meshes exceeding five million vertices, where the rate reaches about 240 million distances per second. For the largest grid containing 9.04 million vertices, the peak GPU memory consumption is around 364 megabytes.

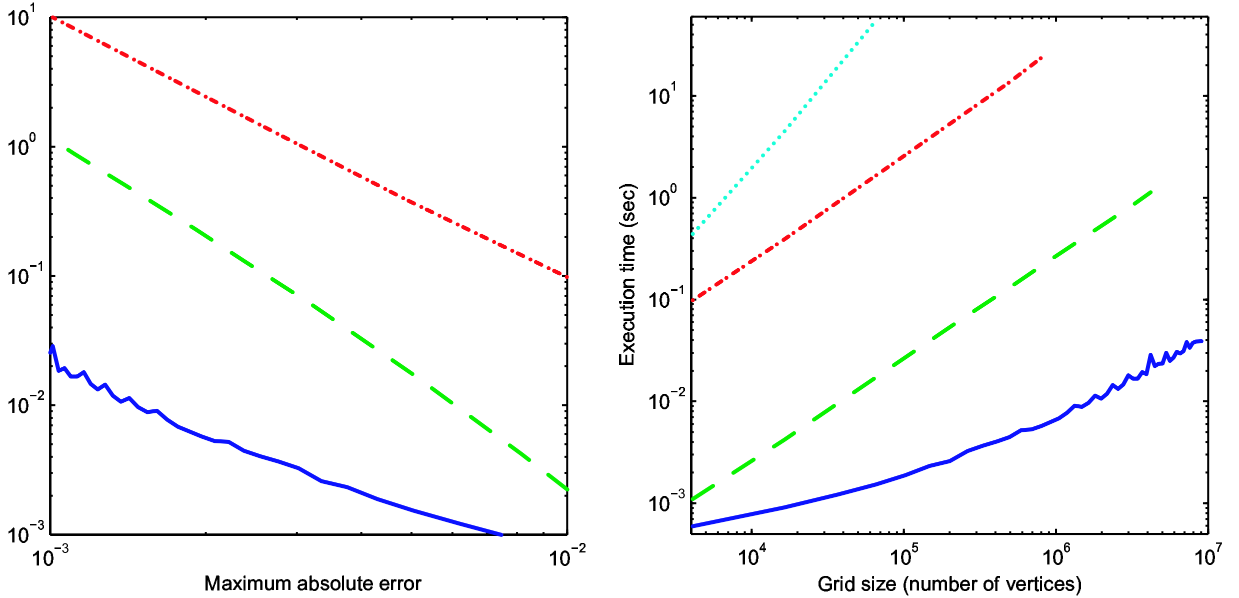

Fig. 8. Left: execution time vs. accuracy of different geodesic distance computation algorithms: GPU and SSE2 implementations of PMM (solid blue and dashed green lines, respectively), and approximate MMP algorithm (dash-dotted red). Right: execution time vs. grid size; same legend with the addition of the approximate MMP (dash-dotted red), and exact MMP (dotted cyan) algorithms.

7.2 Convergence

The dependence of the distance map accuracy on the number of iterations N_{iter} is visualized in Figure 10, which shows the distance map computed from a single point source on the "maze" surface with complicated spiral characteristics. As it appears from the figure, the algorithm converges in six iterations, achieving the mean absolute error of 0.0024 of the shape diameter, and the mean relative error of 0.0058.

While multiple iterations are required to compute a faithful distance map on the "maze" surface, it is important to stress that in general our practice shows that very few iterations are sufficient to obtain accurate distance map on most surfaces.

7.3 Geodesic paths, offset curves, and Voronoi diagrams

Figure 11 shows several computational geometric operations requiring the knowledge of a distance map on a surface. For this visualization, a face surface from the Notre Dame University database was used [Chang et al. 2003]. The surface contained 21,775 vertices and 42,957 faces. In the first two examples, a distance map was computed from a point source located at the tip of the nose. Equi-distant contours were computed using the marching triangle technique in the parametrization domain and then projected back onto the surface. Minimum geodesic paths were computed by backtracking the curve from some starting point along the gradient of the distance map t in the parametrization domain. Formally, geodesic computation can be thought of as solution of the ordinary differential equation

$$\dot{\gamma} = -\mathbf{G}^{-1}\nabla_{\mathbf{u}} t, \tag{31}$$

where $\gamma(s)$ is the geodesic path in the parametrization domain and $\Gamma(s) = \mathbf{x}(\gamma(s))$ is the geodesic on the surface. A first-order integration technique was used to compute $\gamma(s)$.

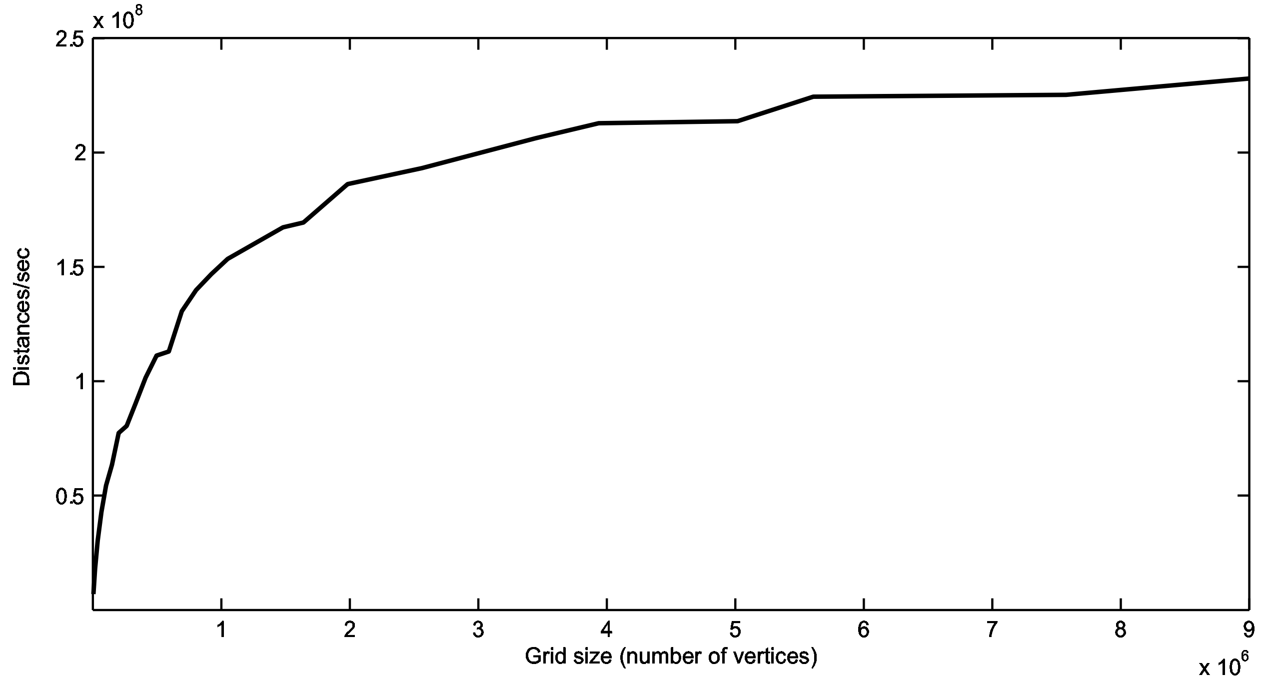

Fig. 9. Number of distance computations per second for the GPU implementation of PMM. Best utilization is achieved for grids exceeding five million vertices.

In the third example, a distance map from 20 random points on the surface was computed and a geodesic Voronoi diagram was found using marching triangles. In the fourth example, the distance map was computed from two disconnected curves and marching triangles were used to trace the geodesic offset curves.

7.4 Multi-chart geometry image

We demonstrate the distance computation algorithm for multi-chart on a "bumped torus" surface (Figure 12), represented using four 100×100, 150×150, 250×250, and 500×500 charts, each spanning a fourth of the surface and having the sampling density adjusted to the level of detail. Bilinear interpolation was used as the projection operators $\mathscr{P}_{ij}$. Figure 13 depicts the progress of Algorithm 4, in this example terminating after a single pass.

8. CONCLUSION

We presented a raster scan-based version of the fast marching algorithm for computation of geodesic distances on geometry images. The structure of the algorithm allowed its efficient parallelization on SIMD processors and GPUs, which have been considered in this paper. Numerical experiments showed that the proposed method outperforms state-of-the-art methods for first-order distance map approximation by one or two orders of magnitude, thus allowing real-time implementation of applications involving intensive geodesic distance computations. In our sequel works, we are going to demonstrate some of such applications. We also showed a generalization of the presented approach to multi-chart geometry images, which is important in many practical applications.

Acknowledgment

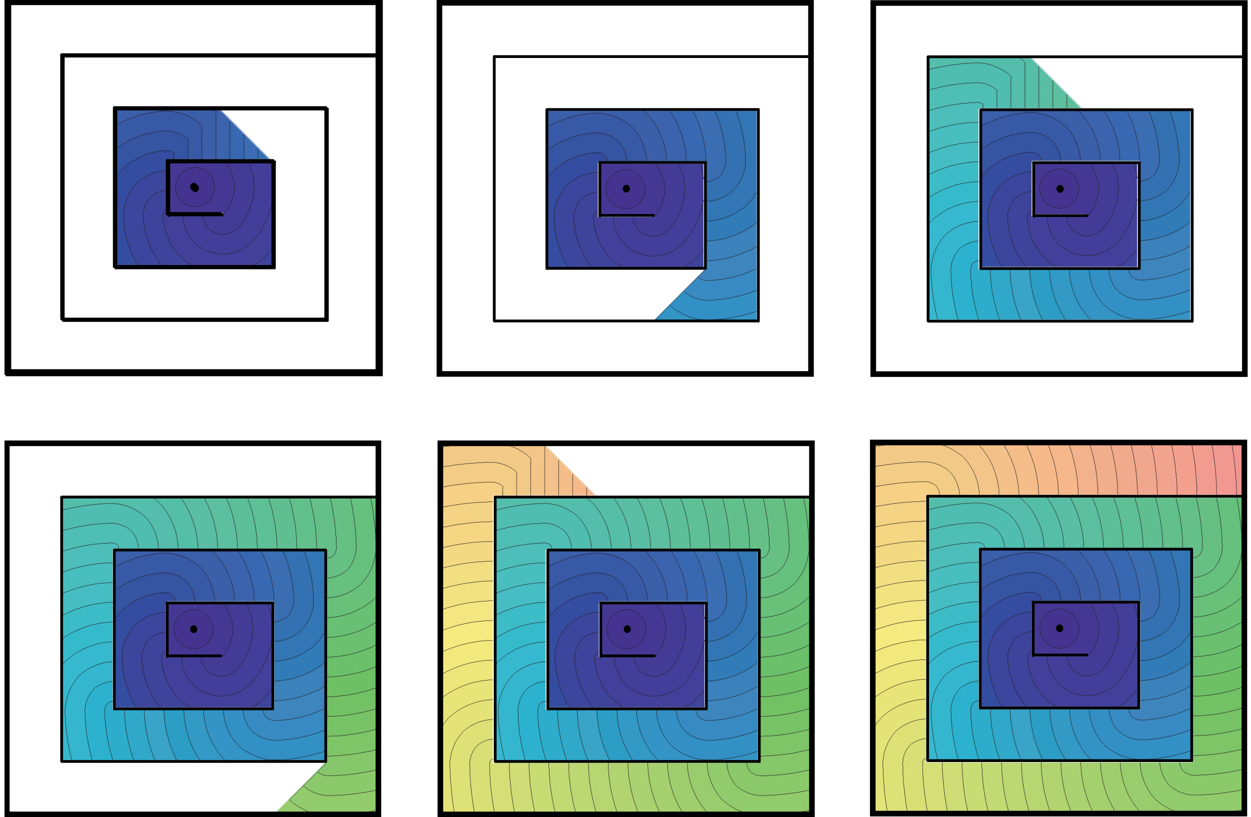

Fig. 10. Left-to-right, top-down: progress of PMM on the "maze" surface, initialized with the source point in the middle. Equidistant contours in the parametrization domain are shown. White regions stand for infinite distance. The algorithm converges after six iterations.

The authors thank Tania Surazhsky for the MMP code, Mark Silberstein for insightful discussions, and the anonymous reviewers for their valuable comments. We also wish to thank NVIDIA for their contribution of the GeForce 8800GTX graphics card. This research was partly supported by United States–Israel Binational Science Foundation grant No. 2004274 and by the Ministry of Science grant No. 3-3414, and in part by Elias Fund for Medical Research.

A. PROOF OF PROPOSITION 1

The algorithm will stop after N iterations, if the distance map remains unchanged between iteration $N-1$ and N. This, in turn, happens when $N-1$ iterations are sufficient to cover any characteristic in the parametrization domain. The number of iterations can therefore be bounded by bounding the total variation of the tangential angle of a characteristic. Our proof generalizes [Qian et al. 2006], where a similar result was shown for the Euclidean case.

Let $\Gamma(s)$ be the characteristic curve with on the surface, s its arclength, and $\gamma(s) = (u^1(s), u^2(s))^{\mathrm{T}}$ its parametrization in $\mathbf{U}$. Since $\Gamma(s) = \mathbf{x}(\gamma(s))$, using the chain rule we obtain

$$\begin{aligned} \dot{\Gamma} &= \mathbf{T}\dot{\gamma} \\ \ddot{\Gamma} &= \mathbf{T}\ddot{\gamma} + \mathbf{r}, \end{aligned} \tag{32}$$

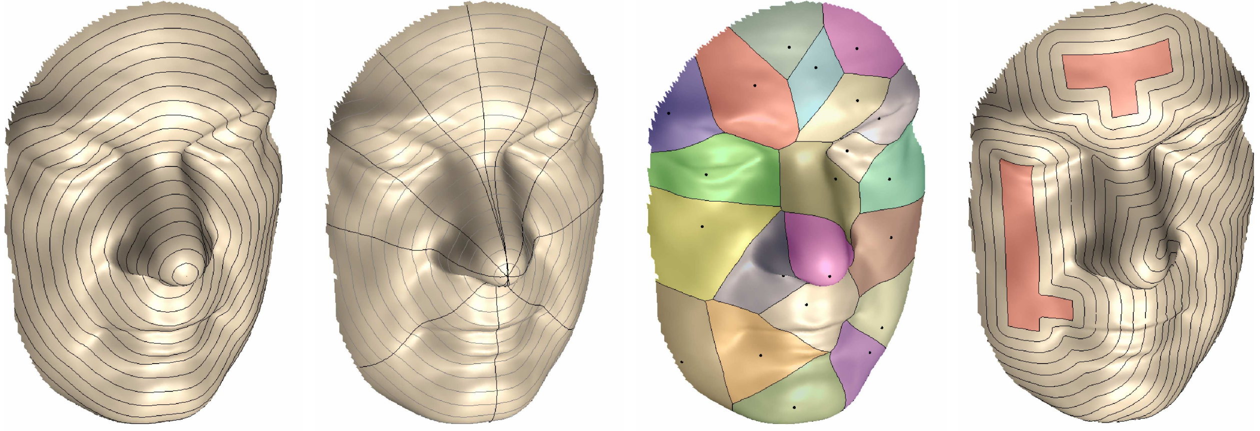

Fig. 11. Computation of distance maps on a geometry image (21,775 vertices, 42,957 faces). Left-to-right: equi-distant contours; minimum geodesic paths; geodesic Voronoi diagram; offset curves.

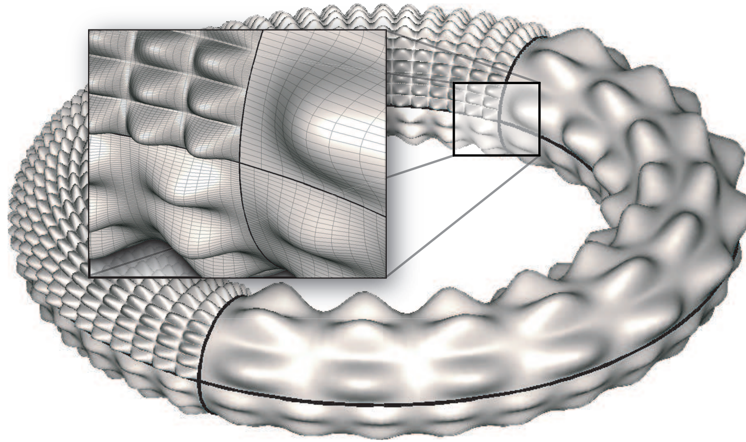

Fig. 12. A four-chart geometry image of a torus with bumps. Chart boundaries are plotted as bold black lines. Note that each chart has a different level of details and, consequently, a different sampling density.

where $\mathbf{r} = (\dot{\gamma}^{\mathrm{T}} \mathbf{H}^1 \dot{\gamma}, \dot{\gamma}^{\mathrm{T}} \mathbf{H}^2 \dot{\gamma}, \dot{\gamma}^{\mathrm{T}} \mathbf{H}^3 \dot{\gamma})^{\mathrm{T}}$ and $\mathbf{H}^i = \nabla^2_{\mathbf{uu}} x^i$ are the Hessian matrices of x^i with respect to the parametrization coordinates $\mathbf{u}$. Since Γ is a geodesic, $\ddot{\Gamma}$ is normal to the surface and hence

$$0 = \mathbf{P_T} \ddot{\Gamma} = \mathbf{T} \ddot{\gamma} + \mathbf{P_T r}, \tag{33}$$

where $\mathbf{P_T}$ denotes the projection on the tangent space.

Hence,

$$\begin{aligned} \|\mathbf{T}\ddot{\gamma}\| &= \|\mathbf{P_T r}\| \leq \|\mathbf{r}\| \\ &\leq \sqrt{(\lambda^{\mathbf{H}^1}_{\min})^2 + (\lambda^{\mathbf{H}^2}_{\min})^2 + (\lambda^{\mathbf{H}^3}_{\min})^2} \cdot \|\dot{\gamma}\|, \end{aligned} \tag{34}$$

Fig. 13. Left-to-right bottom-down: the progress of the multi-chart fast marching algorithm on the bumped torus geometry image from Figure 12. First, the distance map from the source point is computed on the upper left chart. The distance values on the boundaries are extrapolated onto the neighboring charts, in which the distance maps are computed subsequently in the order of the minimum distance value on the boundary. Color map depicts the level sets of the distance map. In this example, a single pass of the algorithm gives an accurate distance map.

where $\lambda_{\min}^{\mathbf{H}^i}$ is the smallest eigenvalue of the Hessian $\mathbf{H}^i$.

Since Γ is a geodesic, $\|\dot{\Gamma}\| = 1$. From (32) we have

$$1 \;=\; \|\dot{\Gamma}\|^2 \;=\; \dot{\gamma}^{\mathrm{T}}\mathbf{T}^{\mathrm{T}}\mathbf{T}\dot{\gamma} \;=\; \dot{\gamma}^{\mathrm{T}}\mathbf{G}\dot{\gamma} \;\geq\; \lambda_{\min}^{\mathbf{G}} \cdot \|\dot{\gamma}\|^2. \tag{35}$$

Hence, $1/\lambda_{\max}^{\mathbf{G}} \leq \|\dot{\gamma}\|^2 \leq 1/\lambda_{\min}^{\mathbf{G}}$. In a similar manner,

$$\|\mathbf{T}\ddot{\gamma}\|^2 \;=\; \ddot{\gamma}^{\mathrm{T}}\mathbf{T}^{\mathrm{T}}\mathbf{T}\ddot{\gamma} \;=\; \ddot{\gamma}^{\mathrm{T}}\mathbf{G}\ddot{\gamma} \;\geq\; \lambda_{\min}^{\mathbf{G}} \cdot \|\ddot{\gamma}\|^2 \tag{36}$$

Combining the above results, yields a bound on the curvature of γ

$$\begin{aligned} \kappa \;&=\; \frac{\|\ddot{\gamma} \times \dot{\gamma}\|}{\|\dot{\gamma}\|^3} \;\leq\; \frac{\|\ddot{\gamma}\|}{\|\dot{\gamma}\|^2} \\ &\leq\; \frac{\lambda_{\max}^{\mathbf{G}}}{\lambda_{\min}^{\mathbf{G}}} \sqrt{(\lambda_{\min}^{\mathbf{H}^1})^2 + (\lambda_{\min}^{\mathbf{H}^2})^2 + (\lambda_{\min}^{\mathbf{H}^3})^2}. \end{aligned} \tag{37}$$

Therefore, the total variation of the tangential angle of γ is bounded by

$$\mathrm{TV}(\phi) \;=\; \int_\gamma \kappa ds \;\leq\; \max\kappa\cdot\int_\Gamma ds \;\leq\; \max\kappa\cdot D. \tag{38}$$

In the worst case, an iteration is required for every $\pi/2$ in $\mathrm{TV}(\phi)$ to consistently cover the characteristic γ, which completes the proof.

REFERENCES

ATI CTM Guide : Technical reference manual. Website: `http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf`.

NVIDIA CUDA : Compute unified device architecture. Website: `http://developer.nvidia.com/cuda`.

SIGGRAPH 2007 GPGPU COURSE. Website: `http://www.gpgpu.org/s2007`.

BORNEMANN, F. AND RASCH, C. 2006. Finite-element discretization of static Hamilton-Jacobi equations based on a local variational principle. *Computing and Visualization in Science 9,* 2, 57–69.

BRONSTEIN, A. M., BRONSTEIN, A. M., AND KIMMEL, R. 2006a. Calculus of non-rigid surfaces for geometry and texture manipulation. *IEEE Trans. Visualization and Comp. Graphics*. to appear.

BRONSTEIN, A. M., BRONSTEIN, M. M., AND KIMMEL, R. 2006b. Generalized multidimensional scaling: a framework for isometry-invariant partial surface matching. *Proc. National Academy of Sciences 103,* 5 (January), 1168–1172.

CARR, N., HOBEROCK, J., CRANE, K., AND HART, J. Rectangular multi-chart geometry images. *Geometry Processing 2006*.

CHANG, K., BOWYER, K., AND FLYNN, P. 2003. Face recognition using 2D and 3D facial data. *ACM Workshop on Multimodal User Authentication*, 25–32.

DANIELSSON, P.-E. 1980. Euclidean distance mapping. *Computer Graphics and Image Processing 14*, 227–248.

DUPUIS, P. AND OLIENSIS, J. 1994. Shape from shading: Provably convergent algorithms and uniqueness results. In *Proc. ECCV*. 259–268.

ELAD, A. AND KIMMEL, R. 2001. Bending invariant representations for surfaces. In *Proc. CVPR*. 168–174.

FISCHER, I. AND GOTSMAN, C. 2005. Fast approximation of high order Voronoi diagrams and distance transforms on the GPU. Technical report CS TR-07-05, Harvard University.

FUNKHOUSER, T., KAZHDAN, M., SHILANE, P., MIN, P., KIEFER, W., TAL, A., RUSINKEWICZ, S., AND DOBKIN, D. 2004. Modeling by example. In *Proc. SIGGRAPH*. 652–663.

GU, X., GORTLER, S., AND HOPPE, H. 2002. Geometry images. *ACM Transactions on Graphics 21,* 3, 355–361.

HERSHBERGER, J. AND SURI, S. 1999. An optimal algorithm for Euclidean shortest paths in the plane. *SIAM J. Computing 28,* 6.

HILAGA, M., SHINAGAWA, Y., KOMURA, T., AND KUNII, T. L. 2001. Topology matching for fully automatic similarity estimation of 3D shapes. In *Proc. SIGGRAPH*. 203–212.

HOFF, K., CULVER, T., KEYSER, J., LIN, M., AND MANOCHA, D. 1999. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proc. ACM SIGGRAPH*. 277–286.

JEONG, W.-K. AND WHITAKER, R. 2007. A fast eikonal equation solver for parallel systems. In *Proc. SIAM Conference on Computational Science and Engineering*.

KATZ, S. AND TAL, A. 2004. Hierarchical mesh decomposition using fuzzy clustering and cuts. *ACM Trans. on Graphics 22,* 3 (July), 954–961.

KIMMEL, R. AND SETHIAN, J. A. 1998. Computing geodesic paths on manifolds. *Proc. of National Academy of Sciences 95,* 15, 8431–8435.

MÉMOLI, F. AND SAPIRO, G. 2001. Fast computation of weighted distance functions and geodesics on implicit hyper-surfaces. *Journal of Computational Physics 173,* 1, 764–795.

MÉMOLI, F. AND SAPIRO, G. 2005. A theoretical and computational framework for isometry invariant recognition of point cloud data. *Foundations of Computational Mathematics 5,* 3, 313–347.

MITCHELL, J. S. B., MOUNT, D. M., AND PAPADIMITRIOU, C. H. 1987. The discrete geodesic problem. *SIAM Journal of Computing 16,* 4, 647–668.

NOVOTNI, M. AND KLEIN, R. 2002. Computing geodesic distances on triangular meshes. In *Proc. Intl. Conf. in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG'2002)*.

PEYRÉ, G. AND COHEN, L. 2003. Geodesic re-meshing and parameterization using front propagation. In *Proc. VLSM'03*.

PRAUN, E., HOPPE, H., AND FINKELSTEIN, A. 1999. Robust mesh watermarking. In *Proc. SIGGRAPH*. 49–56.

QIAN, J., ZHANG, Y., AND ZHAO, H. 2006. Fast sweeping methods for eikonal equations on triangulated meshes. *SIAM Journal on Numerical Analysis*. to appear.

SETHIAN, J. AND POPOVICI, A. 2006. 3-d traveltime computation using the fast marching method. *Geophysics 64,* 2, 516–523.

SETHIAN, J. A. 1996. A fast marching level set method for monotonically advancing fronts. *Proc. of National Academy of Sciences 93,* 4, 1591–1595.

SETHIAN, J. A. AND VLADIMIRSKY, A. 2000. Fast methods for the Eikonal and related Hamilton-Jacobi equations on unstructured meshes. *PNAS 97,* 11, 5699–5703.

SIGG, C., PEIKERT, R., AND GROSS, M. 2003. Signed distance transform using graphics hardware. In *Proc. IEEE Visualization*. 83–90.

SLOAN, P.-P. J., ROSE, C. F., AND COHEN, M. F. 2001. Shape by example. In *ACM Symposium on Interactive 3D Graphics*. 133–144.

SPIRA, A. AND KIMMEL, R. 2004. An efficient solution to the eikonal equation on parametric manifolds. *Interfaces and Free Boundaries 6,* 4, 315–327.

SUD, A., GOVINDARAJU, N., GAYLE, R., AND MANOCHA, D. 2006. Interactive 3D distance field computation using linear factorization. In *Proc. ACM Symposium on Interactive 3D Graphics and Games*. 117–124.

SURAZHSKY, V., SURAZHSKY, T., KIRSANOV, D., GORTLER, S., AND HOPPE, H. 2005. Fast exact and approximate geodesics on meshes. In *Proc. SIGGRAPH*. 553–560.

TSITSIKLIS, J. N. 1995. Efficient algorithms for globally optimal trajectories. *IEEE Transactions on Automatic Control 40,* 9, 1528–1538.

YING, L. AND CANDÈS, E. J. 2006. The phase flow method. *Journal of Computational Physics 220*, 184–215.

ZHAO, H. 2004. A fast sweeping method for eikonal equations. *Mathematics of computation 74,* 250, 603–627.

ZHOU, K., SNYDER, J., GUO, B., AND SHUM, H.-Y. 2004. Iso-charts: Stretch-driven mesh parameterization using spectral analysis. In *Symposium on Geometry Processing*.

ZIGELMAN, G., KIMMEL, R., AND KIRYATI, N. 2002. Texture mapping using surface flattening via multi-dimensional scaling. *IEEE Trans. Visualization and computer graphics 9,* 2, 198–207.