Graphical Models 76 (2014) 214–223

Contents lists available at ScienceDirect

Graphical Models

journal homepage: www.elsevier.com/locate/gmod

A double layer method for constructing signed distance fields from triangle meshes ☆,☆☆

Yizi Wu [1], Jiaju Man *, Ziqing Xie

Key Laboratory of High Performance Computing and Stochastic Information Processing (HPCSIP), Ministry of Education of China, College of Mathematics and Computer Science, Hunan Normal University, Changsha, Hunan 410081, PR China

ARTICLE INFO

Article history:
Received 14 May 2013
Received in revised form 9 April 2014
Accepted 10 April 2014
Available online 24 April 2014

Keywords:
Distance field
Signed distance field
Triangle mesh
Double layer

ABSTRACT

A new algorithm is proposed for constructing signed distance fields (SDF) from triangle meshes. In our approach, both the internal and external distance fields for the triangle mesh are computed first. Then the desired SDF is computed directly from these two distance fields. As only points are used to generate the distance fields, some complicated operations, such as the computation of the distance from a point to a triangle, are avoided. Our algorithm is in a very simple form and is straightforward to parallelize. Actually, we have implemented it by use of the OpenCL API and a CPU-to-GPU speedup ratio of 10–40 is obtained. Further, this method is validated by our numerical results.

1. Introduction

A discrete signed distance field (SDF) of a geometric object is a scalar field defined on a 3D grid, where each grid point stores the shortest distance to the surface of the object. The distance is positive outside the object and negative otherwise. SDF has been widely used in computer graphics, such as rigid body simulation [1]. The popular level set method operates on a scalar field defined on a 3D-grid, and the best choice of this scalar field may be a SDF. Actually, level set based methods have been successfully applied in physically based animation, such as simulating smoke [2], fire [3], and liquids [4,5].

Since geometric objects are often represented by triangle meshes, the necessity of converting a triangle mesh into a SDF arises. There often exists difficulties in computing the distance and the sign for each grid point. Fortunately, an accurate SDF is not necessary in many applications. For example, level set methods [2–5] only need an approximate SDF. Even if the initial SDF is accurate, a re-initialization process is unavoidable after a few time steps. Furthermore, computing the accurate signed distance values for the entire 3D grid is not necessary for many applications [5,6]. That is to say, only the signed distance values near the triangle mesh are required and the values for grid points far away from the mesh can be safely labelled as infinity. Therefore, it is also meaningful to compute an approximate version of SDF from a triangle mesh. The double layer algorithm proposed in this paper is aimed at providing a simple strategy for computing an approximate SDF from a triangle mesh.

This paper is organized as follows. In Section 2, we discuss the related work and the motivation of our algorithm, then the proposed algorithm is described and analyzed in Sections 3 and 4. The numerical results are presented in Section 5. Finally a concluding remark is given in Section 6.

☆ Supported by the National Natural Science Foundation of China (Granted Nos.: 11171104, 10871066), the Construct Program of the Key Discipline in Hunan, and Supported by Scientific Research Fund of Hunan Provincial Education Department (Granted No.: 10K40).
☆☆ This paper has been recommended for acceptance by Peter Lindstrom.
* Corresponding author.
E-mail addresses: wuyzking@163.com (Y. Wu), manjiaju@hunnu.edu.cn (J. Man), ziqingxie@yahoo.com.cn (Z. Xie).
[1] Research supported by the Innovative Research Project for Postgraduates in Hunan Province (Granted No.: CX2012B200).

http://dx.doi.org/10.1016/j.gmod.2014.04.011
1524-0703/

2. Related work

It is well known that there exist challenges for computing both the distance value and its corresponding sign for grid points while constructing SDF for a triangle mesh. In the literature, these two factors are usually considered independently.

There are many approaches to compute the complete (unsigned) distance field. For each grid point, the brute force method computes the distance from the present grid point to all triangles of the triangle mesh and finally takes the minimum as the distance value. The brute force method is simple and accurate. Since its time complexity is $O(mn)$, with m the amount of triangles and n the amount of grid points, it is not applicable for high grid resolution or large number of mesh triangles. Hierarchical methods [7–9] build a hierarchy for the triangle mesh to speed up the computation, and reduce the time complexity to $O(n \log m)$. Nevertheless, the time necessary for initialization and queries of the hierarchy cannot be ignored, and additional memory is needed for saving the hierarchy as the number of triangles becomes large. Additionally, it is worthwhile to point out that it is not straightforward to build the hierarchy. Characteristic [10,11] methods build characteristics for triangle primitives (faces, edges, vertices) and compute the distance values in a small neighbourhood of the triangle mesh, then use a scan-conversion algorithm to obtain the entire distance field. The scan-conversion algorithm can be accelerated by using graphics hardware [11]. The distance transform methods only compute the distance values accurately for grid points near the triangle mesh, then use a transform to generate the distance values for other grid points. There exist some transform methods, such as chamfer distance transforms [12–14], vector distance transforms [15–17], fast marching methods [18] and fast sweep methods [19].

In recent years, with the development of graphic process unit (GPU) technology, a lot of GPU-based algorithms aimed at the computation of distance field are developed. A GPU-based linear facterization method was proposed in [20,21], they express the non-linear distance function of each primitive as a dot product of linear factors, linear terms are efficiently computed using texture mapping hardware. In [22], a constant time jump flooding algorithm to approximate the distance field is proposed and implemented on GPU. There exist many other GPU-based approach, such as parallelizing the Voronoi diagram methods [23,24], the vector propagation algorithms [25], and GPU algorithms on adaptive grids [26].

The computation of distance from a point to a triangle is needed during the construction of a distance field. In [7] a case analysis approach in which some optimization techniques were implemented was proposed. In this approach, the square distance instead of the actual distance is computed and the square root is taken only when the shortest distance is found. This topic has also been discussed or modified in some other papers [27,28,9,29]. However, it is still complicated and expensive.

Another topic is how to compute the sign of each grid point. Methods are mainly divided into two categories in the literature. One of them is the scan-conversion method [7,30,1,31]. A typical scan conversion means casting rays along rows of grid points. The grid points at which the ray has crossed the mesh an uneven number of times, are inside the mesh. However problems will occur when the ray intersects a vertex or an edge of the mesh rather than the interior or a triangle [32]. The second kind of methods make use of the surface normals of the mesh. In this approach, the sign of a grid point can be determined by evaluating the inner product of a normal and a directional vector. But sometimes this does not work since a triangle mesh is not $\mathbf{C}^1$-continuous. Actually, in many cases, we have the same distance to more than one triangle with different signs as described in [7]. A kind of pseudo-normal approach has been proposed in [32,33] to overcome such difficulty. However, an edge list (which cannot be constructed in linear time) of the triangle mesh is required in this method. A review of algorithms aimed to construct a signed distance field from a triangle mesh can be found in [34].

In the application of modern computer graphics, the meshes become more and more detailed in order to model more and more complicated geometric objects, and this may lead to very large number of small triangles in a mesh. It tends to take a lot of time and memory to compute SDFs. It is noted that if the mesh becomes fine, it can be sampled by points accurately. We found that it is possible to reduce the storage overhead by use of point-based representation and it is easy to overcome the difficulties of computing signs by introducing two layers of points. Therefore, this paper is aimed to develop a simple point-based double layer algorithm to approximate the SDF from a triangle mesh which has a large number of small triangles. The main contributions of our algorithm are listed as follows:

- It computes distance fields for an internal layer and an external layer of the triangle mesh instead of constructing the distance field of the triangle mesh directly. To generate the distance fields, only points are used to approximate the internal and external layers, and many complicated operations, such as building hierarchies, characteristics or computing the point-to-triangle distance, are avoided.
- The approximated signed distance values near the triangle mesh is constructed from the two distance fields directly by use of a simple strategy. Both the sign and the distance values are obtained simultaneously without any complicated configurations. The sign of grid points far away from the triangle mesh is computed by a very simple scan conversion.
- The proposed algorithm is easy to be parallelized. In fact, we can parallelize it straightforward on a triangle basis, i.e., treat all triangles in the mesh in parallel. In this paper, the parallel algorithm is implemented by use of the OpenCL API and a speed-up ratio of 10–40 is obtained.

3. The proposed algorithm

A triangle mesh $\mathbf{M}$ is just a union of triangles. In most cases, $\mathbf{M}$ is assumed to be a closed, orientable 2-manifold

in 3D Euclidean space. Thus the normal information of triangles can be used while constructing the SDF of **M**. In addition, to construct a well-defined SDF, the following three conditions must be satisfied [35].

- Triangles may share only vertices and edges. Otherwise, they are disjoint.
- Each edge must be adjacent to exactly two triangles.
- Triangles incident on a vertex must form a single cycle around that vertex.

For a closed, orientable 2-manifold **M**, we define its δ-internal layer L_I^δ and δ-external layer L_E^δ as

$$L_I^\delta = \{\mathbf{p} \in \mathbb{R}^3 : \mathbf{p} \textit{ is inside } \mathbf{M} \textit{ and } dist(\mathbf{p}, \mathbf{M}) = \delta\},$$
$$L_E^\delta = \{\mathbf{p} \in \mathbb{R}^3 : \mathbf{p} \textit{ is outside } \mathbf{M} \textit{ and } dist(\mathbf{p}, \mathbf{M}) = \delta\},$$

where δ is a small positive number and $dist(\mathbf{p}, \mathbf{M})$ is the distance from point $\mathbf{p}$ to the manifold **M**. If the distance field $D_I(D_E)$ of $L_I^\delta(L_E^\delta)$ is known, a signed distance field ϕ of **M** can be computed trivially. As shown in Fig. 1, for a point $\mathbf{p}$ inside **M** (i.e. $D_I(\mathbf{p}) \leqslant D_E(\mathbf{p})$), we have $|\phi| + \delta = D_E$, thus $\phi(\mathbf{p}) = -(D_E(\mathbf{p}) - \delta)$ since $\mathbf{p}$ is inside **M**. If $\mathbf{p}$ is outside **M**, $\phi(\mathbf{p})$ can be obtained similarly. In a word, we have

$$\phi(\mathbf{p}) = \begin{cases} -(D_E(\mathbf{p}) - \delta), & \text{if } D_I(\mathbf{p}) \leqslant D_E(\mathbf{p}) \\ D_I(\mathbf{p}) - \delta, & \text{otherwise.} \end{cases} \quad (1)$$

Though **M** is a triangle mesh and L_I^δ and L_E^δ are generated from it, L_I^δ or L_E^δ is no longer represented by a triangle mesh as **M** in our work. Instead, we sample each of them as a set of discrete points. Actually, there are several reasons for us to choose such a representation. Firstly, to construct a triangle mesh, vertices and vertex normals of **M** are needed whereas only the normals of all the triangle faces of **M** known. Secondly, points have no connectivity. Once a point is generated, we use it to update the distance field and then discard it. Hence it is not necessary to save it in memory during the whole computation. Finally, we avoid some complicated operations, such as building hierarchies or characteristics for the triangle mesh, or computing the point-to-triangle distance.

Based upon the above facts, the proposed algorithm can be divided into three sub steps as follows:

- Initialize all 3D grid fields, i.e., D_I, D_E and ϕ.
- Compute the internal and external distance field (D_I and D_E).
- Generate the final signed distance field ϕ for **M**.

3.1. Initialize all 3D grid fields

The 3D grid fields D_I, D_E and ϕ are maintained during the computing process. All the grids have the same resolution $N_x \times N_y \times N_z$ and their grid values are initially labelled as $+\infty$. This can be implemented by assigning all grid values with a very large positive number M, e.g., $M = 1.0e9$.

3.2. Compute internal and external distance field

From each triangle of **M**, internal points are generated to update internal distance field D_I and external points are generated to update the external distance field D_E.

For a triangle ABC of **M**, its outward normal $\mathbf{n}$ and area S can be computed as

$$\mathbf{n} = \frac{(\vec{AB} \times \vec{AC})}{\|\vec{AB} \times \vec{AC}\|}, \quad S = \frac{1}{2}\|\vec{AB} \times \vec{AC}\|.$$

Once a point $\mathbf{p}$ in triangle ABC is choosed, then the internal point $\mathbf{p}_I$ and the external point $\mathbf{p}_E$ corresponding to $\mathbf{p}$ can be computed as

$$\mathbf{p}_I = \mathbf{p} - \delta\mathbf{n}, \quad \mathbf{p}_E = \mathbf{p} + \delta\mathbf{n}. \quad (2)$$

The next question is how to choose points in a triangle. We demonstrate a strategy as follows. If the triangle is small enough (e.g. $S < \epsilon$, with ϵ a given small positive number), then the barycentre of the triangle should be chosen. If a triangle is too big, it should be subdivided into several small triangles and then the barycentres of these small triangles are chosen.

It is worthwhile to point out that a big triangle could be subdivided recursively (Fig. 2). If $\epsilon < S \leqslant 2\epsilon$, we could find the longest edge of ABC. Without loss of generality, suppose it is AB whose middle point is $M = \frac{1}{2}(A + B)$. Thus two smaller triangles ACM and BCM are obtained. If $S > 2\epsilon$, we find three middle points of all the edges of triangle ABC, i.e., $M = \frac{1}{2}(A + B)$, $N = \frac{1}{2}(B + C)$, $P = \frac{1}{2}(C + A)$. Then four smaller triangles APM, BMN, CPN and MNP are obtained. For the case of $S > 2\epsilon$, each subdivided triangle is considered recursively until its area is smaller than 2ϵ. Almost uniformly distributed internal or external particle layers are generated by using such a subdivision strategy.

Although the recursive algorithm above is easy to understand and implement, it is not supported by some parallel platforms, such as GPUs. To overcome such a kind of difficulty, we introduce another algorithm to subdivide a triangle. The algorithm is described in Algorithm 1 (see also Fig. 3).

Algorithm 1. Dividing a big triangle (non-recursive)

$n \leftarrow \left\lceil \sqrt{\frac{S}{\epsilon}} \right\rceil$;
if $n > n_{max}$ **then**
  $n \leftarrow n_{max}$;
end if
$\vec{s_1} \leftarrow \frac{1}{n}\vec{AB}$, $\vec{s_2} \leftarrow \frac{1}{n}\vec{BC}$, $\vec{s_3} \leftarrow \frac{1}{n}\vec{AC}$;
$P_s \leftarrow \frac{1}{3}[A + (A + \vec{s_1}) + (A + \vec{s_3})]$;
$choose(P_s)$;
$i \leftarrow 2$;
while $i \leqslant n$ **do**
  $P_s \leftarrow P_s + \vec{s_1}$; $choose(P_s)$;
  $P \leftarrow P_s$; $j \leftarrow 2$;
  while $j \leqslant i$ **do**
    $P \leftarrow P + \vec{s_2}$; $choose(P)$;
    $j \leftarrow j + 1$;
  end while
  $i \leftarrow i + 1$;
end while

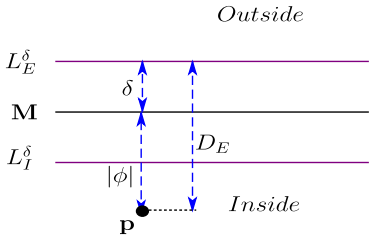

Fig. 1. Computing ϕ from D_E for point **p** inside **M**.

The integer n_{max} in the above algorithm is used to prevent the triangle from being divided into too many small triangles, e.g., $n_{max} = 16, 32$. Algorithm 1 not only has no recursive structure, but also runs faster than the recursive algorithm and adapts to parallel computing. Consequently, our algorithm can be parallelized straightforward by letting all triangles be processed at the same time.

Suppose that a point **p** in the triangle is chosen, with its corresponding internal point and external point computed by Eq. (2). Once an internal (external) point is generated, it is used to update the internal (external) distance field. Assume that the current generated internal (or external) point corresponding to **p** is $\mathbf{p}_I$ ($\mathbf{p}_E$). We update a neighbourhood of **p** (find its closest grid point (i,j,k) first, then a $(2K+1)\times(2K+1)\times(2K+1)$ neighbourhood of **p** in a 3D grid defined as $\{(i\pm i_x, j\pm j_y, k\pm k_z) : 0 \leqslant i_x, j_y, k_z \leqslant K\}$) in the internal (or external) distance field. Without loss of generality, we take the internal distance field D_I. For each grid point (x,y,z) in the neighbourhood of **p**, its distance to the internal point $\mathbf{p}_I$ is computed, if the new computed distance is smaller than $D_I(x,y,z)$, it is taken as the updated value of $D_I(x,y,z)$. Then the neighbourhood of **p** (not $\mathbf{p}_I$) is updated since we aimed to compute the signed distance for the triangle mesh **M** (not for the internal layer). Of course one can also take $\mathbf{p}_I$ as the centre of the neighbourhood, nevertheless, a wider neighbourhood is needed to obtain the same accuracy, as shown in Fig. 4.

3.3. Generate the signed distance field

It is time for us to construct the signed distance field in a neighbourhood of the triangle mesh **M**. Typically, the signed distance ϕ by use of Eq. (1) for each grid point **p**, which satisfies $D_I(\mathbf{p}) < \infty$ and $D_E(\mathbf{p}) < \infty$. In Eq. (1), $D_I(\mathbf{p}) \leqslant D_E(\mathbf{p})$ means that **p** is inside the mesh **M**, and then $D_I(\mathbf{p}) - \delta \leqslant 0$. However, occassionally this condition cannot be satisfied due to the numerical error. To overcome such a difficulty, we modify Eq. (1) slightly as:

$$\phi(\mathbf{p}) = \begin{cases} \min(0, -(D_E(\mathbf{p}) - \delta)), & \text{if } D_I(\mathbf{p}) \leqslant D_E(\mathbf{p}) \\ \max(0, D_I(\mathbf{p}) - \delta), & \text{otherwise.} \end{cases} \tag{3}$$

After the signed distance values in a neighbourhood of **M** have been computed, the signs of other grid points can be easily determined by a scan conversion. For this purpose, we choose a casting direction, for example, the axis *Z*. From $k = 0$ to $k = N_z - 2$, for each $0 \leqslant i \leqslant N_x - 1$ and $0 \leqslant j \leqslant N_y - 1$, if $\phi(i,j,k) < 0$ and $\phi(i,j,k+1) > -\infty$, we set $\phi(i,j,k+1) = -\infty$. Our algorithm ends here.

It is worthwhile to point out that the signed distance values far away from the triangle mesh are still labelled as infinity rather than the accurate signed distance values. Fortunately, the narrow band techniques are popular in many applications (e.g. fluid simulation [5] and image segmentation [6]), that is to say, the signed distance values far away from the triangle mesh can be safely labelled as infinity. Of course, if necessary, the entire signed distance field can be obtained by any distance transform method such as fast marching methods [18], fast sweeping methods [19], or some fast GPU based methods [20,25].

4. Analysis and optimizations

As our algorithm is aimed to construct the SDF approximately, an error analysis is presented in this section.

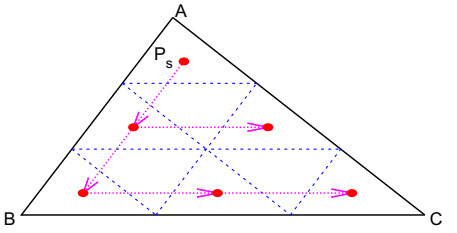

Fig. 3. Subdividing triangles (non-recursive).

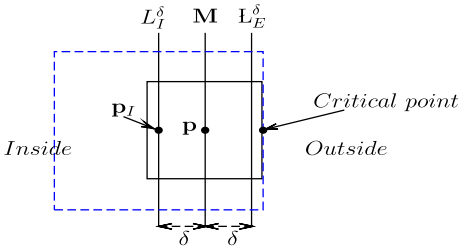

Fig. 4. To reach the same critical point, $\mathbf{p}_I$ needs a wider neighbourhood than **p**.

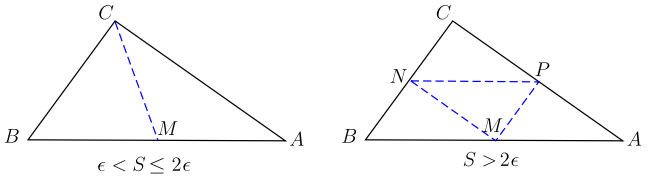

Fig. 2. Subdividing triangles (recursive).

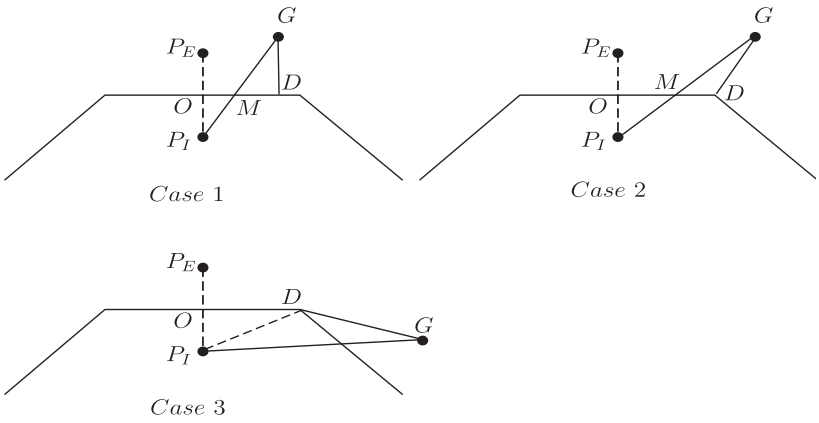

Fig. 5. Description of distancing error.

Furthermore, some optimization techniques are discussed to reduce the CPU time of our algorithm.

4.1. Analysis

For simplicity, we only consider the error bounds for the 2D case.

Theorem 1. *Assume there is no sign error. For each grid point, suppose that the approximated SDF value is d_e and the accurate SDF value is d. Then we have*

(1) *For 2D case,* $|d_e - d| < L_{max}/2$,
(2) *For 3D case,* $|d_e - d| < \frac{2}{3}L_{max}$,

where L_{max} is the maximum edge length of the mesh.

Proof.

(1) We only consider an arbitrary grid point G outside the mesh, since grid points inside the mesh can be handled similarly. As shown in Fig. 5, three cases should be considered according to the position of the grid point G.
In all cases, $d = GD$ and $d_e = GP_I - \delta = GP_I - P_IO$.
For *Case* 1 and *Case* 2:

$$\begin{aligned}|d_e - d| &= GP_I - P_IO - GD\\ &= GM + MP_I - P_IO - GD\\ &\leqslant |GM - GD| + |MP_I - P_IO|\\ &< MD + MO = OD\\ &\leqslant L_{max}/2.\end{aligned}$$

For *Case* 3:

$$\begin{aligned}|d_e - d| &= GP_I - P_IO - GD\\ &= GP_I - GD - P_IO\\ &< P_ID - P_IO < OD\\ &\leqslant L_{max}/2.\end{aligned}$$

(2) For the case of 3D, a similar discussion can be made. □

The algorithm may require that L_{max} should be consistent with the grid step Δ, that is, $L_{max} < C\Delta$ for some small constant C. Our algorithm has a linear accuracy $O(\Delta)$, that seems to be of low accuracy. Fortunately, for many applications in computer graphics applications [2–5], the accuracy is really not an issue. The key point is to obtain a SDF which looks like (after visualization) the original mesh. Actually, a geometrical model can be represented by either a triangle mesh or a set of discrete points, each of them is only an approximate version of the original model. What we have done is just to convert the triangle mesh into a point set and then construct SDF from it.

Our algorithm may cause sign errors if the particles are not distributed uniformly. As shown in Fig. 6 for a 2D analogue, where P is an internal particle, while Q is an external particle for another edge, MF is the midnormal of PQ, and $L_2 > L_1$ ($L_1 = P_0O, L_2 = Q_0O$). If a grid point happens to fall into the region enclosed by the triangle OGF, it may be identified as being inside the mesh since it is close to the internal particle P. However, the fact is that it is outside the mesh. It is noted that F is the error point furthest from the mesh. The following theorem shows that the length of OF is not too long, that is, the sign error will not spread far away from the triangle mesh, thus our algorithm is numerically stable.

Theorem 2. *As shown in* Fig. 6, *if* $L_2 \tan\theta \leqslant \delta$, *then* $OF \leqslant (L_2 - L_1)/2$.

Proof. The coordinate system is shown as in Fig. 6. For some points, it is simple to determine their coordinates, e.g., $P_0(-L_1\cos\theta, L_1\sin\theta)$, $Q_0(L_2\cos\theta, L_2\sin\theta)$, $P(-L_1\cos\theta + \delta\sin\theta, L_1\sin\theta + \delta\cos\theta)$, $Q(L_2\cos\theta + \delta\sin\theta, L_2\sin\theta - \delta\cos\theta)$. Thus

$$\vec{PQ} = ((L_1 + L_2)\cos\theta, (L_2 - L_1)\sin\theta - 2\delta\cos\theta)$$

and the middle point of PQ is $M(\frac{L_2-L_1}{2}\cos\theta + \delta\sin\theta, \frac{L_1+L_2}{2}\sin\theta)$.

Suppose $L = OP$ and the coordinate of F is $(L\cos\theta, -L\sin\theta)$, we have

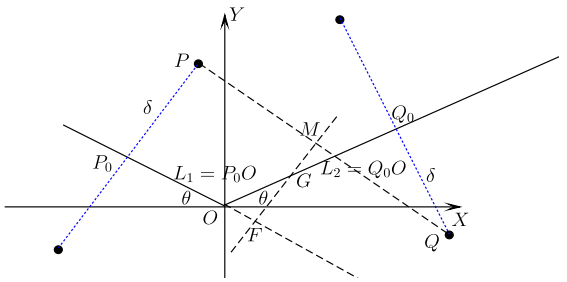

Fig. 6. Description of sign error.

$$\vec{MF} \cdot \vec{PQ} = 0$$

which yields that

$$L = \frac{(L_2 + L_1)(L_2 - L_1)}{2[L_2 + L_1 + \sin 2\theta(\delta - L_2 \tan\theta)]}$$

Since $0 \leqslant \theta \leqslant \pi/2$, we have

$$L \leqslant \frac{L_2 - L_1}{2}$$

if $L_2 \tan\theta \leqslant \delta$. Thus completes the proof. □

Because $L_2 \leqslant 1/2\Delta$ after mesh subdivision by our algorithm and we usually set $\delta \geqslant 1/2\Delta$, the condition of the above theorem can be satisfied for well-defined meshes which with θ small enough. The fact that this condition cannot be satisfied means the mesh has high curvature details, which remains to be a difficult topic for other SDF constructing methods also. From the above theorem, it is known that, if the points are uniformly distributed or the gaps between points become very small, the result SDF causes less noise. Actually, from most of our numerical results, our algorithm works well even if the above condition is not strictly satisfied. Consequently, our algorithm seems to be robust to various types of triangle meshes. Of course, for some extreme cases, the noise caused by sharp turns can be noticed by naked eyes. We'll discuss these limitations later with numerical results.

4.2. Optimizations

In our algorithm, the calculations mainly concentrated on the computation of the two layers of distance fields. Fortunately, there are several ways to improve the performance.

An obvious optimization technique is to avoid too many computations of square roots as proposed in [7]. That is to say, two distance square fields are computed first instead of two distance fields and the square roots are only taken at the final step.

Another fact is that saving two of the three 3D grids is enough, rather than saving all of them. For example, we can use ϕ to save either the external grid D_E or internal grid D_I. As a result, less memory is required and the time of initializing a 3D grid is saved.

Finally, we introduce a simple optimization strategy which greatly reduces the CPU time. To update the distance square field *ds* for each particle, a similar form to the following structure may appear in the code.

```
For i := 1 → N
  For j := 1 → N
    For k := 1 → N
      ds_ijk := d_i^2 + d_j^2 + d_k^2
```

here $N = 2K + 1$ is the size of the mesh's neighbourhood as mentioned in the Section 3. A slightly modified scheme is as follows:

```
For i := 1 → N
  ds_i := d_i^2
  For j := 1 → N
    ds_j := d_j^2
    For k := 1 → N
      ds_ijk := ds_i + ds_j + d_k^2
```

the modified scheme reduce the number of multiplication operations from $3N^3$ to $N^3 + N^2 + N$. For example, it is known that $N = 5$ is an appropriate size of neighbourhood in most situations. Then the modified scheme reduce the number of multiplication operations from 375 to 155 for each point. Such an optimization strategy is not possible for classical methods which compute the point-to-triangle distance. The fast algorithm for point-to-triangle distance computation usually requires about 40 multiplication operations. Thus about $40N^3$ multiplication operations are needed for each triangle in a neighbourhood of size *N*. Therefore, the computational time of our algorithm is comparable to the classical methods even if the triangles are subdivided for many times. Furthermore, the edge list which cannot be constructed in linear time (in terms of the number of triangles) when the triangle mesh represented by shared vertex scheme is not needed in our algorithm.

Table 1
The errors for the Stanford the bunny model with different grid resolution.

Grid res.	Grid step (Δ)	e_{max}	e_{avg}
$72 \times 72 \times 72$	0.06	0.04923	0.02701
$126 \times 126 \times 126$	0.04	0.03911	0.01838
$251 \times 251 \times 251$	0.02	0.02100	0.00816
$501 \times 501 \times 501$	0.01	0.01202	0.00427

5. Results and discussions

The proposed algorithm is implemented using C++ on our graphic workstation, on which a Suse 11sp2 Linux operating system is installed. The signed distance field generated by our algorithm is visualized by the marching cubes algorithm [36,37] and rendered using the OpenGL API. A parallel version of our algorithm is also implemented by using the OpenCL API. There are two OpenCL devices in our workstation. One of them is a Nvidia Tesla C2075 GPU, which has 5 GB global memory. The other is

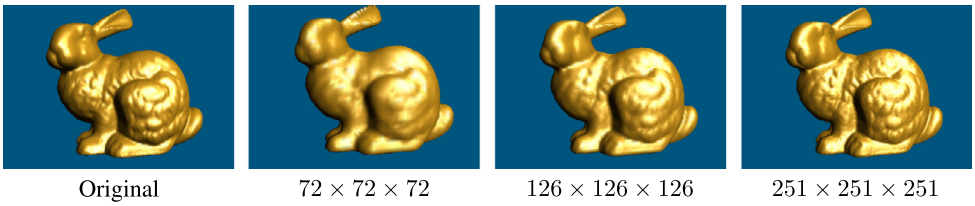

Fig. 7. The bunny example (with different grid resolutions).

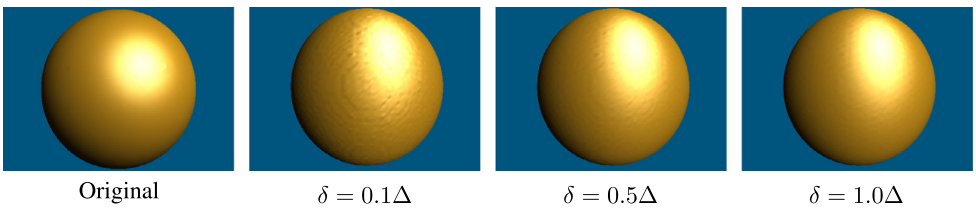

Fig. 8. The sphere example (with different δ values).

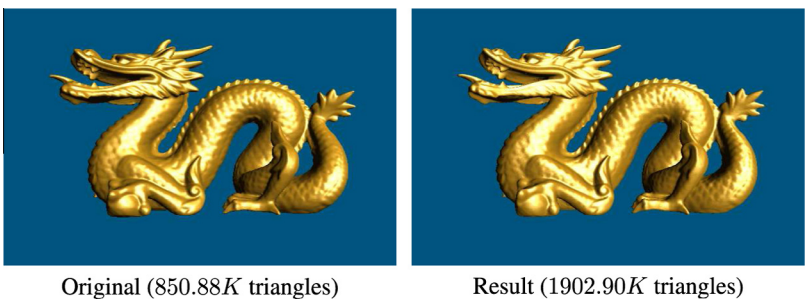

Fig. 9. The dragon example.

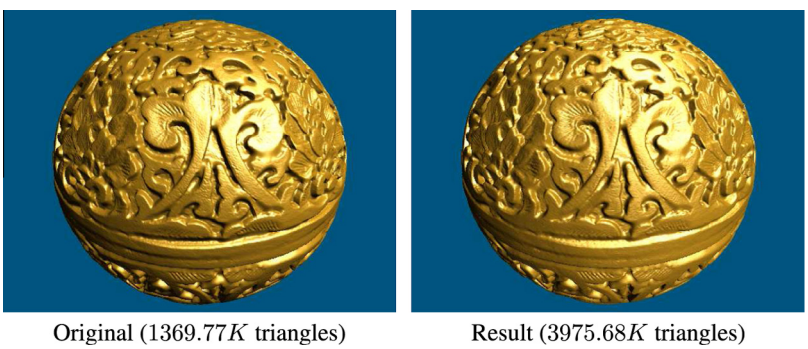

Fig. 10. The circular box example.

Table 2
CPU time for some models with grid resolution $251 \times 251 \times 251$ (seconds).

Model	Triangles	AWPN	Ours
Bunny	68.91*K*	0.62	0.68
Dragon	850.88*K*	3.60	1.59
Circular box	1369.77*K*	4.56	2.44

Table 3
OpenCL (on Intel Device) time for some models with grid resolution $251 \times 251 \times 251$ (seconds).

Model	Triangles	Method in [11]	Ours
Bunny	68.91*K*	0.13	0.11
Dragon	850.88*K*	0.37	0.21
Circular box	1369.77*K*	0.43	0.14

Table 4
Comparing the performance between the OpenCL version and No OpenCL version (seconds).

Grid Res.	$T_{NoOpenCL}$	T_{Tesla}	R_{Tesla}	T_{Intel}	R_{Intel}
$251 \times 251 \times 251$	2.44	0.08	30.05	0.14	17.43
$501 \times 501 \times 501$	4.67	0.27	17.30	0.29	16.10
$715 \times 715 \times 715$	10.53	0.59	17.85	0.63	16.71
$715 \times 834 \times 715$	14.80	0.85	17.41	0.74	20.00
$715 \times 1001 \times 715$	22.16	1.65	13.43	1.05	21.10
$715 \times 1251 \times 715$	41.47	–	–	1.38	30.05
$715 \times 1667 \times 715$	110.37	–	–	2.50	44.15

Intel(R) Xeon(R) CPU E5-2630 with 32 GB physical memory. We set $K = 2, \delta = 0.5\Delta$ and $\epsilon = \Delta^2$ in coding unless specified.

5.1. Error measurements

We test the Stanford bunny model with different grid resolutions (i.e. different grid step). Angle Weighted Pseudonormal (AWPN) method in [33] was proved to be an accurate approach in constructing signed distance. We define a set Ω containing all grid points whose signed distance values d satisfy $|d| < \infty$ in both AWPN and our method. For an arbitrary grid point $\mathbf{p}$, suppose the signed distance values computed by AWPN and our approach is denoted by $\bar{d}(\mathbf{p})$ and $d(\mathbf{p})$, respectively. The maximum and average norms can be defined as follows.

$$e_{max} = \max_{\mathbf{p}\in\Omega} |d(\mathbf{p}) - \bar{d}(\mathbf{p})|, \quad e_{avg} = \frac{1}{|\Omega|}\sum_{\mathbf{p}\in\Omega} |d(\mathbf{p}) - \bar{d}(\mathbf{p})|, \quad (4)$$

where $|\Omega|$ denotes the number of elements in Ω. The numerical results show that the proposed algorithm converges as the grid step becomes smaller (Table 1 and Fig. 7). Although it is only of linear accuracy, the reduced triangle mesh (constructed from the resulting SDF by Marching cubes method) looks just like the original mesh and may be good enough for many applications in computer graphics.

The numerical results may depend on the parameter δ which cannot be too large, whereas too small δ may also cause problems. For example, if M is a sphere (see Fig. 8), too small δ cannot separate the internal layer from the external counterpart well, and lead to noises just as Theorem 2 predicts. The quality of the resulting SDF also depends on the parameters ϵ and K. Obviously a small ϵ would yield better quality with lower performance. We set $\epsilon = \Delta^2$ in all of our numerical tests so that ϵ is consistent with the grid step Δ. From our numerical results, $K = 2$ is an appropriate choice. Actually, $K = 1$ may cause too much noise in a few examples when $K \geqslant 3$ there is no obvious improvement for accuracy. Since the run time for $K + 1$ is $\frac{(2K+3)^3}{(2K+1)^3}$ times of that for K, it is not a good idea to choose a large K.

Table 5
OpenCL time with different area distribution of triangles on $251 \times 251 \times 251$ grids (seconds).

Mesh level	Triangles	Variance	Time
level1	$3.21K$	0.0013	0.07
level2	$3.21K$	0.0052	0.19
level3	$3.21K$	0.0098	0.53

5.2. Performance

We test the proposed algorithm for triangle meshes with large number of triangles. Both the dragon example (Fig. 9) and circular box example (Fig. 10) are computed with $715 \times 834 \times 715$ grid resolution. In our algorithm, the main storage overhead is for saving the (unsigned and signed) distance fields. Thus it always works well even if the number of triangles of the mesh becomes very large. The time complexity is $O(n)$ if the average area of the triangles is on the order of Δ^2, where n is the number of triangles. The running time on CPU of some models are listed in Table 2. It is observed that the time is not strictly proportional to the number of triangles due to the fact that triangles in different models may have different sizes. We have also compared the running time of our algorithm with a totally-optimized implementation of the state-of-art method AWPN.

For the computation of SDFs on GPU, there is a novel algorithm proposed in [11], which has a CPU-to-GPU speedup ratio of 5–10. We have implemented this method by using the OpenCL API. The signed distance values are computed for grid points which are about 2Δ away from the triangle mesh, just as our method. A direct comparison of our algorithm and the algorithm in [11] is represented in Table 3.

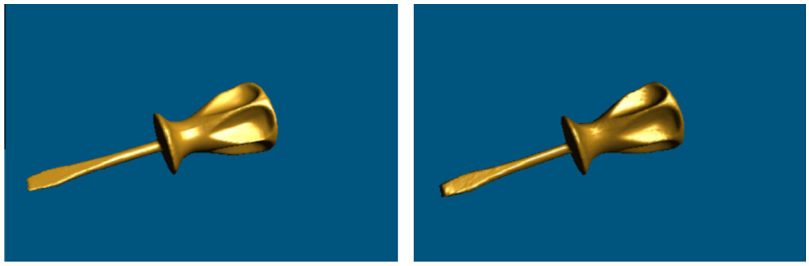

Fig. 11. The screwdriver example.

We have also compared the running time of our algorithm between the OpenCL version with the non-OpenCL version. We test the circular box model (1369.77K triangles) on both CPU and OpenCL devices. The results are presented in Table 4. The OpenCL running time (in seconds) T_{Tesla}, T_{Intel} is measured by use of the profile strategy of the OpenCL API. The CPU-to-GPU speedup ratio are computed, i.e., $R_{Tesla} = T_{NoOpenCL}/T_{Tesla}$ and $R_{Intel} = T_{NoOpenCL}/T_{Intel}$. Here $T_{NoOpenCL}$ denotes the running time without using OpenCL. In Table 4, The symbol "–" means out of memory. As the Tesla GPU has only 5 GB global memory, it runs out of memory as the grid resolution becomes high.

It is observed that our algorithm runs as fast as the state-of-art methods, and has a CPU-to-GPU speedup ratio about 10–40. Consequently, our GPU algorithm, obtained trivially from the CPU version, has excellent performance. It should be pointed out that the OpenCL running time may not only depend on the number of triangles, but also the area distribution of the triangles. We test the sphere model with different area distribution of triangles on the Intel OpenCL device. The numerical results are shown in Table 5. The variance in Table 5 is defined as follows:

$$Var = \frac{1}{n}\sum_{i=1}^{n}(s_i - m)^2,$$

where n is the number of triangles, s_i is the area of the ith triangle ($i = 1, 2, \ldots, n$) and m is the average area of all triangles in the mesh. A big variance indicates the areas of triangles distribute in a wide range. It could be observed that our algorithm likes triangle meshes with uniform distributed triangle areas.

6. Conclusions

In this paper, a very simple and efficient algorithm for constructing signed distance fields from triangle meshes is proposed. First, both an internal and an external field are constructed for the triangle mesh. Then they are used to generate the desired signed distance field by use of a simple formula. Since less memory is required, our algorithm has the ability to deal with meshes with a very large number of triangles. Although two distance fields are required, our algorithm can accomplish its work within reasonable time since the time-consuming operations are only performed in a small neighbourhood of the triangle mesh. Our algorithm constructs the SDF of the triangle mesh approximately as shown before. However, it is good enough for many applications in computer graphics. If we are allowed to view the internal and external layers as a new representation of the original geometry model, then our algorithm can be considered as an accurate method. Though our GPU algorithm is obtained trivially from the CPU version, it has a good CPU-to-GPU speed-up ratio.

It should be pointed out that, although our algorithm works well in most situations, it still has some shortages. Theorem 2 claimed that the condition $\delta \geqslant L_2 \tan\theta$ should be satisfied in order to prevent the sign errors from spreading far away from the triangle mesh. This means the internal and the external layers may intersect with each other at places with high curvature. In other words, the 3D grid cannot separate the triangle mesh well and noise may be generated. In fact, even if the triangle mesh has many high curvature details, the proposed algorithm still works well (Figs. 9 and 10). However, when the meshes have very high curvature structures, noise are produced. (As shown in Fig. 11, the head of the screwdriver is noisy.) Fortunately, in most situations, it is meaningless to construct SDFs for meshes with very high curvature structures. Actually, even if the accurate SDF is computed, small details may be smeared out. Another shortage is that our algorithm is only designed for high detailed meshes with lots of small triangles in some degree. If the areas of the triangles of the mesh have a wide-range distribution, the parallel version of the algorithm will slow down (Table 5) since different work items may have different workloads.

References

[1] E. Guendelman, R. Bridson, R. Fedkiw, Nonconvex rigid bodies with stacking, ACM Trans. Graph. (TOG) 22 (2003) 871–878.
[2] R. Fedkiw, J. Stam, H.W. Jensen, Visual simulation of smoke, in: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, ACM, 2001, pp. 15–22.
[3] D.Q. Nguyen, R. Fedkiw, H.W. Jensen, Physically based modeling and animation of fire, ACM Trans. Graph. (TOG) 21 (2002) 721–728.
[4] N. Foster, R. Fedkiw, Practical animation of liquids, in: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, ACM, 2001, pp. 23–30.
[5] D. Enright, S. Marschner, R. Fedkiw, Animation and rendering of complex water surfaces, ACM Trans. Graph. (TOG) 21 (2002) 736–744.
[6] J. Lie, M. Lysaker, X.-C. Tai, A variant of the level set method and applications to image segmentation, Math. Comput. 75 (2006) 1155–1174.
[7] B.A. Payne, A.W. Toga, Distance field manipulation of surface models, IEEE Comput. Graph. Appl. 12 (1992) 65–71.
[8] J. Strain, Fast tree-based redistancing for level set computations, J. Comput. Phys. 152 (1999) 664–686.
[9] A. Guezlec, "meshsweeper": dynamic point-to-polygonal mesh distance and applications, IEEE Trans. Visual. Comput. Graph. 7 (2001) 47–61.
[10] S. Mauch, A Fast Algorithm for Computing the Closest Point and Distance Transform, 2000. <http://www.acm.caltech.edu/seanm/software/cpt/cpt.pdf>.
[11] C. Sigg, R. Peikert, M. Gross, Signed distance transform using graphics hardware, in: Visualization, 2003. VIS 2003, IEEE, 2003, pp. 83–90.
[12] G. Borgefors, Chamfering: a fast method for obtaining approximations of the Euclidean distance in N dimensions, in: Proc. 3rd Scand. Conf. on Image Analysis (SCIA3), 1983, pp. 250–255. <http://www.scia2015.org/>
[13] G. Borgefors, Distance transformations in arbitrary dimensions, Comput. Vis. Graph. Image Process. 27 (1984) 321–345.
[14] S. Svensson, G. Borgefors, Digital distance transforms in 3D images using information from neighbourhoods up to $5 \times 5 \times 5$, Comput. Vis. Image Underst. 88 (2002) 24–53.
[15] P.-E. Danielsson, Euclidean distance mapping, Comput. Graph. Image Process. 14 (1980) 227–248.
[16] J.C. Mullikin, The vector distance transform in two and three dimensions, CVGIP: Graph. Models Image Process. 54 (1992) 526–535.
[17] R. Satherley, M.W. Jones, Vector-city vector distance transform, Comput. Vis. Image Underst. 82 (2001) 238–254.
[18] J.A. Sethian, A fast marching level set method for monotonically advancing fronts, Proc. Natl. Acad. Sci. 93 (1996) 1591–1595.
[19] H. Zhao, A fast sweeping method for eikonal equations, Math. Comput. 74 (2005) 603–627.
[20] A. Sud, N. Govindaraju, R. Gayle, D. Manocha, Interactive 3D distance field computation using linear factorization, in: Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games, ACM, 2006, pp. 117–124.
[21] A. Sud, N. Govindaraju, R. Gayle, E. Andersen, D. Manocha, Surface distance maps, in: Proceedings of Graphics Interface 2007, ACM, 2007, pp. 35–42.

[22] G. Rong, T.-S. Tan, Jump flooding in GPU with applications to Voronoi diagram and distance transform, in: Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games, ACM, 2006, pp. 109–116.
[23] A. Sud, M.A. Otaduy, D. Manocha, DiFi: fast 3D distance field computation using graphics hardware, Computer Graphics Forum, vol. 23, Wiley Online Library, 2004, pp. 557–566.
[24] N. Cuntz, A. Kolb, Fast hierarchical 3D distance transforms on the GPU, in: Proc. Eurographics, Short-Paper, 2007, pp. 93–96. <https://www.eg.org/>
[25] J. Schneider, M. Kraus, R. Westermann, GPU-based real-time discrete Euclidean distance transforms with precise error bounds, in: International Conference on Computer Vision Theory and Applications (VISAPP), 2009, pp. 435–442. <http://visapp.visigrapp.org/>
[26] T. Park, S.-H. Lee, J.-H. Kim, C.-H. Kim, CUDA-based signed distance field calculation for adaptive grids, in: 2010 IEEE 10th International Conference on Computer and Information Technology (CIT), IEEE, 2010, pp. 1202–1206.
[27] M.W. Jones, 3D Distance from a Point to a Triangle, Department of Computer Science, University of Wales Swansea Technical Report CSR-5, 1995.
[28] F.D. IX, A. Kaufman, Incremental triangle voxelization, in: Proceedings of Graphics Interface, 2000, pp. 205–212. <http://www.graphicsinterface.org/>
[29] J. Huang, Y. Li, R. Crawfis, S.C. Lu, S.Y. Liou, A complete distance field representation, in: Proceedings of the Conference on Visualization'01, IEEE Computer Society, 2001, pp. 247–254.
[30] M.W. Jones, The production of volume data from triangular meshes using voxelisation, Comput. Graph. Forum 15 (1996) 311–318.
[31] T. Ju, Robust repair of polygonal models, ACM Trans. Graph. (TOG) 23 (2004) 888–895.
[32] J.A. Bærentzen, H. Aanæs, Generating Signed Distance Fields from Triangle Meshes, Informatics and Mathematical Modeling, Technical University of Denmark, DTU 20, 2002.
[33] J.A. Baerentzen, H. Aanaes, Signed distance computation using the angle weighted pseudonormal, IEEE Trans. Visual. Comput. Graph. 11 (2005) 243–253.
[34] M.W. Jones, J.A. Baerentzen, M. Sramek, 3D distance fields: a survey of techniques and applications, IEEE Trans. Visual. Comput. Graph. 12 (2006) 581–599.
[35] C.M. Hoffmann, Geometric and Solid Modeling: An Introduction, Morgan Kaufmann Publishers Inc., 1989.
[36] W.E. Lorensen, H.E. Cline, Marching cubes: a high resolution 3D surface construction algorithm, ACM Siggraph Computer Graphics, vol. 21, ACM, 1987, pp. 163–169.
[37] C. Montani, R. Scateni, R. Scopigno, A modified look-up table for implicit disambiguation of marching cubes, Vis. Comput. 10 (1994) 353–355.