The Visual Computer (2019) 35:961–971
https://doi.org/10.1007/s00371-019-01683-w

ORIGINAL ARTICLE

Generating signed distance fields on the GPU with ray maps

Bastian Krayer[1] · Stefan Müller[1]

Published online: 7 May 2019

Abstract
Signed distance fields represent objects as distances to the closest surface points with a sign differentiating inside and outside. We present an algorithm to compute a signed distance field from triangle meshes. All data are kept on the GPU, making it ideal for any pure graphics-based context. We split the algorithm into a fast parallel distance transform and a new method of computing the sign. To determine the sign, we compute the winding number for any point using a ray map, a ray-based data structure that preserves geometric meaning while reducing the amount of work to be done for ray tests. Based on that structure, we devise a simple parallel algorithm to sample an exponentially growing number of rays to cope with meshes having deficiencies such as holes or self-intersections. We demonstrate how our method is both fast and able to handle imperfect meshes.

Keywords Signed distance fields · Geometric algorithms · Object representation · GPGPU

1 Introduction

Distance fields are a common way to represent objects in two or three dimensions. For any object, the distance field at a point is the distance to the point closest on that object. For three-dimensional bodies, a natural representation is their boundary. If that boundary is closed and orientable, the distance field may be augmented so that the value is negative when the point is inside the object and positive when outside. Due to this addition, the signed distance field (SDF) for differentiable surfaces is differentiable at the surface boundary, whereas for unsigned distance fields, this is only true for non-boundary points. Various properties can be derived from an SDF, such as the normal or curvature of the isosurface passing through a point or the object's skeletonization. There are many applications of signed distance fields. Volumetric ambient occlusion or soft shadows [24] for rendering purposes can be approximated using the distance information of the surroundings. In complicated environmental scenarios, SDFs can be used for potential-based path planning algorithms [15]. The definition of a distance field is exactly a nearest neighbor problem, which commonly comes up in many applications, for example in collision detection [10,13]. The additional sign can be used to determine if one object is inside of another and therefore collided with it. The amount by which both objects interpenetrate is then found by taking the absolute distance from the SDF. Special rendering techniques such as sphere tracing [11] exist to efficiently display a distance field. Figure 1 shows two signed distance fields created by our method and then rendered interactively.

In this paper, we present a system for efficiently generating sampled signed distance fields directly on the GPU. We split the construction into creating a distance field and signing it afterward. The distance field is computed using a distance transform algorithm that takes into account initial distances for each cell. We have augmented the original algorithm to utilize fast local memory available to compute shaders. Signs are found by computing the winding number at each sampling location. We present a ray-based data structure called ray map. A ray map contains for each location information about the intersections of a ray originating at that point. This structure allows us to efficiently create many virtual paths through the grid, without losing geometric information. We use that property to approximate the sign of deficient meshes. To compensate for problems such as holes, intersections or duplicate faces, we sample the space of all paths through the grid. The average of all rays measures how much a point can

✉ Bastian Krayer
bastiankrayer@uni-koblenz.de

Stefan Müller
stefanm@uni-koblenz.de

[1] University Koblenz-Landau, Koblenz, Germany

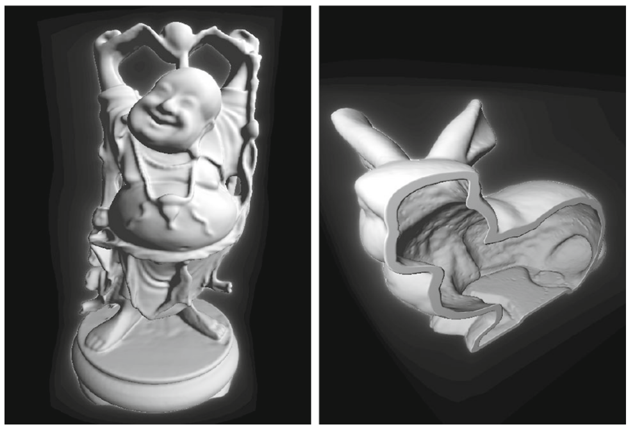

Fig. 1 The Buddha (left) and a bunny model (right) are converted to an SDF for each frame during runtime and then rendered interactively with a sphere tracing algorithm. The bunny is traced with distance field modifiers to show only half of the model and an outer wall with a thickness

be considered inside or outside. That operation is equal to the winding number for any non-deficient mesh.

2 Related work

This section presents an overview of the existing literature for distance field generation, both with and without sign. While distance fields are continuous in general, we will in the following be referring to discrete fields only. One of the most general algorithms is based on solving a special form of the partial differential eikonal equation given by

$$\begin{aligned} \|\nabla \mathrm{s}(\mathbf{p})\| &= 1, \\ \mathrm{s}|_{\partial S} &= 0 \end{aligned}$$

∂S denotes the boundary of the object. Intuitively, this can be thought of as a wave propagating from the surface at unit speed. Any point t units away from the closest point will be hit by the wave after exactly t time units. Techniques such as the fast marching method [21] can be used to solve this equation, which has a complexity of $O(n \log n)$, where n is the number of cells. A newer version by Yatziv et al. [26] even makes it possible in $O(n)$ by using a special kind of priority queue.

Another class of solutions is distance transforms. These can be grouped into approximate and exact variants. Examples of approximate solutions are propagation methods, such as the one described in Schneider et al. [20]. They use a vector propagation scheme together with GPU acceleration to compute a distance transform for a binary input with a precise error bound. An overview of different exact distance transforms can be found in a survey by Fabbri et al. [7], where they find the Meijster distance transform [18] algorithm to be most consistently well performing for various input configurations. The basic principle is the minimization of a number of quadratic functions with their loci at the feature points. The Meijster algorithm and similar ones are easily parallelizable, as an n-dimensional transform can be decomposed into n successive one-dimensional problems, each running in parallel for the corresponding $n-1$ dimensions. These algorithms process binary data. A more general algorithm, which we implemented efficiently, is described by Felzenszwalb et al. [8]. It uses a very similar approach to the Meijster algorithm, but also incorporates sampled distance values at grid locations. All of these methods do not take signs into account. Specialized algorithms have been developed to utilize the GPU, such as DiFi by Sud et al. [23]. They compute the distance field per slice in the z direction by determining only those primitives that would contribute to that slice, thus reducing the overall workload. Their follow-up technique [22] is also based on a slice approach, but computes distances based on a linear factorization of the distance function on primitives. Cuntz et al. [6] use a propagation method to update an initial binary grid with a push and pull scheme. Each step uses neighboring information from the previous step, an operation well supported by textures on GPUs, to approximate the correct distance and sign. For signing a distance field, thus determining inside and outside, multiple algorithms have been proposed. Many methods rely on either a correct initial guess or more generally a clean, oriented and closed mesh, since otherwise the sign is not uniquely defined. Intuitively, the sign for a point should be determined by checking whether the vector pointing from that point to the closest surface point lies in the same half-space as the normal at the closest point. If they point in the same space, the point lies inside, otherwise outside. This simple method fails in some cases. To overcome that, Baerentzen et al. [2] introduced angle weighted pseudo-normals. For any vertex, it is computed as the sum of all adjacent triangle normals weighted by their incident angle at that vertex. Baerentzen et al. proved that using this normal with the idea described above results in the correct sign. For deficient meshes, this test may still fail. Xu et al. [25] try to overcome such problems by first considering an unsigned distance transform. An isosurface with a given width is used as an initial region around the surface that defines an inside unambiguously. The sign for other sample points is found by a special traversal of the volume. Recently, another method was presented by Jacobson et al. [16]. They generalize winding numbers to solid angles and show how to efficiently compute that number for triangular meshes. This winding number field has the important property of smoothly degrading in the presence of mesh deficiencies. In a follow-up paper Barill et al. [3] speed up that process significantly by utilizing the fact that the winding number field is harmonic and thus varies smoothly in the distance. Thereby, groups of objects farther

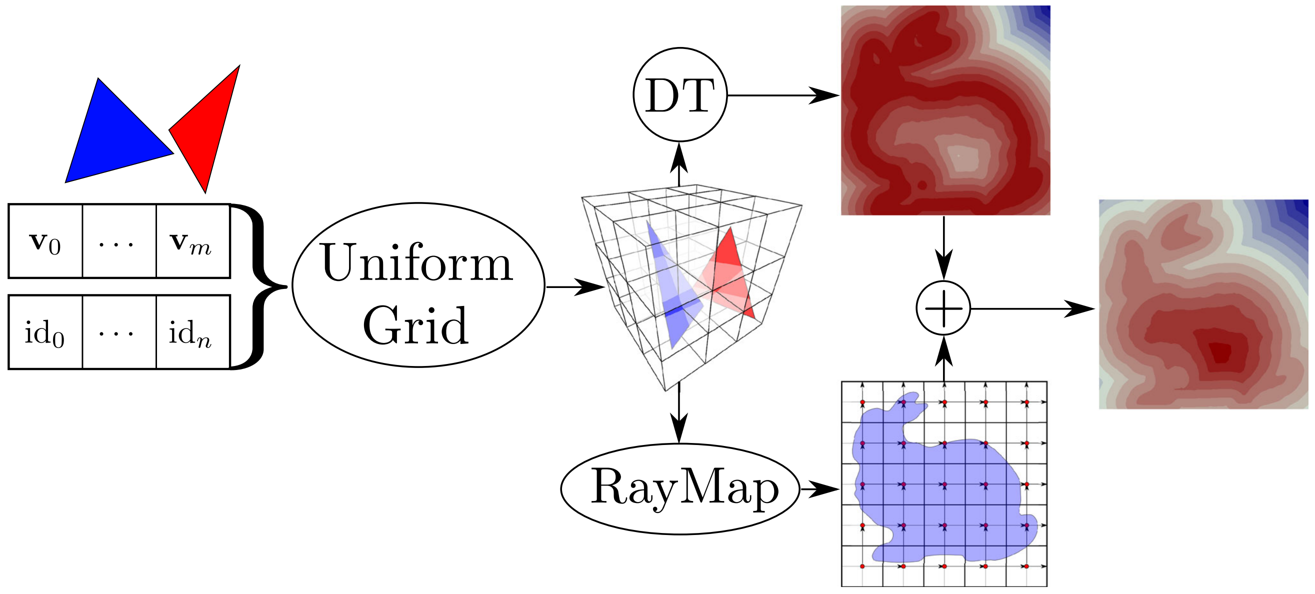

Fig. 2 Overview of our method. We start with arbitrary triangle data, usually specified by one array for vertices and one for indices, representing the triangle faces. These are then inserted into the uniform grid using a voxelization-based GPU algorithm (Sect. 3.1). Using the grid information, a distance transform (Sect. 3.3) is performed to create an unsigned distance field. A special data structure called ray map is created with the same grid. This structure is then used to sign the distance field (Sect. 3.5)

away can be replaced by a single combined representative for fast processing.

3 Method

Our method operates on raw triangle data, either static or dynamic. Dynamic changes such as animations or varying transformations are handled by pre-transforming all vertices and storing them in a dedicated buffer. This can easily be implemented using GPU compute operations. All other data structures will be created on the fly from that. This is a useful property, as all data stay on the GPU and no transfer from or to the CPU is needed, making it a good fit for any other GPU algorithm operating on the SDF.

An overview of our system can be seen in Fig. 2. The following sections will discuss the components in more detail.

3.1 Uniform grid generation

We use a uniform grid as a base for other steps. It is created with a conservative rasterization, utilizing the normal rendering pipeline. First, triangles are projected along the axis for which the largest amount of fragments will be generated. A geometry shader determines that axis by the largest normal component and switches the vertex components accordingly. Triangles are extended so that they cover the midpoints of all pixels they touch in order to generate necessary fragments. For more details, we refer to the GPU gems article [12]. One additional step we take for each fragment is to intersect the triangle with cells in the z direction using the Akenine-Möller method [1]. Otherwise, each x, y fragment would only fill one cell in the z direction, even though the triangle could possibly intersect multiple cells that project on the same pixel.

This process is repeated twice. The first time, a count of the objects per cell is generated. An integer buffer is initialized with zero for each cell. The fragment shader atomically increments each touched cell. Afterward, we perform a parallel prefix sum pass on all counts. Our implementation is based on the description by Blelloch [5]. This provides us with the starting position of each cell, if we were to sequentially store each cell's triangles, including duplications for any triangle being in many cells. As each triangle can become arbitrarily large, it is not known at the beginning how many entries this will result in, but the total number is automatically computed during the prefix sum algorithm. That total is used to resize an index buffer, if necessary, that will contain all triangle IDs per cell.

The second pass again performs the voxelization, but this time increments the counter and then inserts the current triangle's index at the previously stored count value in the index buffer. We use the same buffer for this as with the counts. Both the starting position and length of the cell entries can

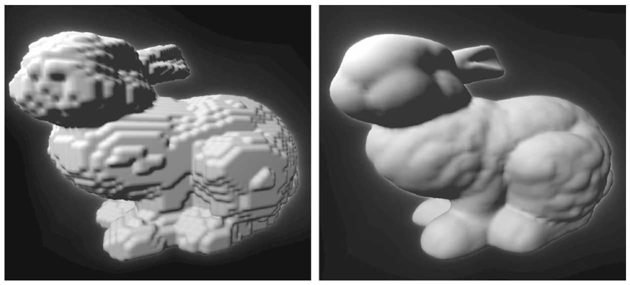

Fig. 3 A binary distance transform reproducing the voxel structure on the left. The right shows our distance transform, which does not suffer from the same problem, as it samples directly from the geometry

be recovered using two consecutive values from the accumulated buffer.

3.2 Generating distance fields

While there are specialized sampled distance fields such as adaptively sampled distance fields [4,9], complete distance fields [14] or more recently hierarchical *hp*-adaptive SDFs [17], we use a uniformly sampled field. We use a variant of the distance transform for sampled functions described by Felzenszwalb et al. [8]. We have also tried a binary distance transform, but have found it to be slightly less accurate, even with the extension of storing the nearest neighbor cell. This is due to the triangles being oriented themselves in the cells, so the binarization loses too much information. This still happens with the transform we chose, but to a lesser degree, so we generally can ignore it. A solution is to loop over neighboring cells in the final distance aggregation pass described in Sect. 3.3. Figure 3 shows the result of only using a pure distance transform with binary data, which reproduces the binary voxels, not the actual surface.

3.3 Distance transform

We have implemented a parallel version of the distance transform for sampled functions with the extension of also finding the index of the closest cell for each cell. Algorithm 1 describes the general procedure of dispatching the distance transform in parallel, which is performed by the GPU driver during a compute dispatch operation. It reserves shared memory for three temporary arrays used in the actual transform. These arrays can then be used by every work item. These address their part of the memory by their local index in their work group. As the maximum amount of shared memory is fixed by the hardware, we compute the work group size to be as large as possible while still being able to reserve enough memory.

Algorithm 2 shows the distance transform by Felzenszwalb et al. [8] augmented to use the shared memory and provide nearest neighbor information. All input data are copied into the shared memory once at the beginning. The procedure consists of two steps. The first step computes the lower envelope of parabolas at the grid points with the sampled distance values. In a second step, the envelope is used to find the smallest parabola and thereby the smallest distance. This also yields the index of the closest cell, which we save in order to use it in the final refinement step. This algorithm actually outputs the squared distance but can be used as input for the transform in the other two directions. As an initialization step, we find the smallest distance per cell to the triangles contained in it using the index buffer created by the uniform grid voxelization. If the cell is empty, a number higher than the largest possible distance in the grid is inserted, which serves as a representation of infinity but avoids numerical problems. The distance transform is trivially parallelizable as it can be represented by three consecutive one-dimensional transforms, one for each coordinate axis. Our implementation uses OpenGL compute shaders for this operation. We utilize the shared memory by querying the maximum size provided by the hardware and then generating shader code with the maximum work group size to still be within that memory budget. We have found that this greatly improves the running time over a global buffer approach. For a resolution of 50^3, this increase was nearly twofold on the dataset used for testing in Sect. 4. The average global buffer transform took 0.32 ms, while the average shared implementation took 0.17 ms .

Each one-dimensional execution finds the closest cell in that dimension for each other cell. In the first step, this is just the index in the same line. Each following step will inherit the index currently stored at the closest cell in the corresponding dimension. This results in a structure similar to the complete distance field introduced by Huang et al. [14]. After the transform is completed, we use the nearest grid cell information to find the closest distance to the triangles therein.

It should be noted that computing only this distance transform without the added accuracy can be done more efficiently with only one rasterization pass, where a triangle directly computes its distance to the touched voxels. The final distance is the minimum of all triangle distances. This requires an atomic minimum operation, which is generally only implemented for integers, not floats. This can be easily overcome by either a simple linear mapping or by a bijection from signed integers to floats.

Algorithm 1 Pseudocode for the x pass dispatch for the distance transform. This is equivalent to the work the GPU driver does when dispatching work groups. Inputs are the resolution $\mathbf{r}$, the initial grid g and the nearest neighbor field $\mathrm{nnf}_{\mathrm{in}}$. $\mathrm{nnf}_{\mathrm{out}}$ is the output nearest neighbor field.

```
 1: function DT($\mathbf{r}$, g, $\mathrm{nnf}_{\mathrm{in}}$, $\mathrm{nnf}_{\mathrm{out}}$)
 2:     $n \leftarrow \mathbf{r}_x$                          ▷ Length of the line
 3:     ▷ Corresponds to a compute dispatch call
 4:     ParallelFor each work group with 2D size $\mathbf{w}$
 5:         ▷ Reserve memory to be shared in workgroup
 6:         v, vals ← reserve_shared(n, $\mathbf{w}$)
 7:         z ← reserve_shared(n + 1, $\mathbf{w}$)
 8:         ParallelFor each workitem in workgroup
 9:             $\mathbf{i}$ ← global_index()
10:             $l$ ← local_index()
11:             ▷ See algorithm 2
12:             kernel($n, \mathbf{i}, l, \mathbf{r}$, g, $\mathrm{nnf}_{\mathrm{in}}$, $\mathrm{nnf}_{\mathrm{out}}$, v, z, vals)
13:         EndParallelFor
14:     EndParallelFor
15: end function
```

3.4 Generating signed distance fields

We aim to preserve geometric precision, allow for small deficiencies in meshes but still be very fast, so that SDFs, even for animated data, can be created immediately. Under these conditions, angle weighted pseudo-normals do not provide the robustness against mesh errors or the means to overcome those. Techniques such as [25] lose geometric precision and require an additional parameter that guides the amount of that precision. Winding numbers, while being fast, still require precomputation, which is slower than our chosen method.

We have opted for another common approach to compute the winding number: Ray casting. Instead of computing the revolutions of a curve or surface around a point directly, we count the signed intersections of a curve starting at the point going to infinity. Signed intersection means that if we hit the triangle from the back, we count −1, and otherwise 1. A point lies outside if the resulting number is 0, and inside otherwise. As with 2D, this also applies to the solid angle winding numbers. If we consider a closed solid, shooting rays in all possible directions from the inside will result in a signed intersection of −1 for each ray. For a closed orientable mesh, one ray would be enough to determine inside and outside.

Algorithm 2 Pseudocode for the x pass of the parallel distance transform based on [8].

```
16: function KERNEL($n, \mathbf{i}, l, \mathbf{r}$, g, $\mathrm{nnf}_{\mathrm{in}}$, $\mathrm{nnf}_{\mathrm{out}}$, v, z, vals)
17:     $\mathit{off}_0 \leftarrow nl$                     ▷ Offset with $n$ elements
18:     $\mathit{off}_1 \leftarrow (n+1)l$                 ▷ Offset with $n+1$ elements
19:     $\mathrm{v}[\mathit{off}_0] \leftarrow 0$
20:     $\mathrm{z}[\mathit{off}_1] \leftarrow -\infty$
21:     $\mathrm{z}[\mathit{off}_1 + 1] \leftarrow \infty$
22:     $k \leftarrow 0$
23:     ▷ Copy global data into faster shared memory
24:     for $i \leftarrow 0, n-1$ do
25:         $\mathrm{vals}[\mathit{off}_0 + i] \leftarrow \mathrm{g}[i, \mathbf{i}_x, \mathbf{i}_y]$
26:     end for
27:     ▷ The original transform
28:     $q \leftarrow 1$
29:     while $q \le n-1$ do
30:         $f_q \leftarrow \mathrm{vals}[\mathit{off}_0 + q]$
31:         $v_k \leftarrow \mathrm{v}[\mathit{off}_0 + k]$
32:         $f_{vk} \leftarrow \mathrm{vals}[\mathit{off}_0 + vk]$
33:         $s \leftarrow \frac{f_q + q^2 - (f_{vk} + v_k^2)}{2q - 2v_k}$
34:         if $s \le \mathrm{z}[\mathit{off}_1 + k]$ then
35:             $k \leftarrow k - 1$
36:             goto 30
37:         else
38:             $k \leftarrow k + 1$
39:             $\mathrm{v}[\mathit{off}_0 + k] \leftarrow q$
40:             $\mathrm{z}[\mathit{off}_1 + k] \leftarrow s$
41:             $\mathrm{z}[\mathit{off}_1 + k + 1] \leftarrow \infty$
42:             $q \leftarrow q + 1$
43:         end if
44:     end while
45:     $k \leftarrow 0$
46:     for $q \leftarrow 0, n-1$ do
47:         while $\mathrm{z}[\mathit{off}_1 + k + 1] < q$ do
48:             $k \leftarrow k + 1$
49:         end while
50:         $v_k \leftarrow \mathrm{v}[\mathit{off}_0 + k]$
51:         $d \leftarrow (q - v_k)^2 + vals[\mathit{off}_0 + v_k]$
52:         $\mathbf{p}_{\mathrm{out}} \leftarrow (q\ \mathbf{i}_x\ \mathbf{i}_y)^T$
53:         $\mathrm{g}[\mathbf{p}_{\mathrm{out}}] \leftarrow d$
54:         $\mathbf{p}_{\mathrm{nn}} \leftarrow (v_k\ \mathbf{i}_x\ \mathbf{i}_y)^T$        ▷ Nearest neighbor in line
55:         $\mathrm{nnf}_{\mathrm{out}}[\mathbf{p}_{\mathrm{out}}] \leftarrow \mathrm{nnf}_{\mathrm{in}}[\mathbf{p}_{\mathrm{nn}}]$  ▷ Carry over neighbor
56:     end for
57: end function
```

The winding number handles closed surface self-intersection correctly, but does not on its own work with open or broken meshes. One approach for this is selecting a number of sample rays, average the signed intersections and then threshold that number to determine inside or outside. This process is generally very slow, as it requires many rays. This is a problem we try to solve with our algorithm, which uses the same information as the previous steps and is highly parallelizable, thus being ideal for GPU work. It could also be applied to similar problems, such as rendering Boolean combinations of surfaces, as in [19].

3.5 The ray map data structure

We construct a grid that stores for each point the intersection count of a ray with a direction starting at that point and going to infinity. First, we observe that the final intersection count of a ray is just the sum of all signed intersections encountered along the way. Therefore, it is independent of the order in which the triangles are processed. This is important, as the order of processing is not specified due to the GPU's parallel execution model. We start by considering a 1D grid. Let i be the index of a cell. The local intersection count $\mathrm{w}_l(i)$ is the number of intersections of a ray connecting the cell's corresponding world space center to the one of cell $i+1$. The intersection number of a line segment from cell i to $j \geq i$ is the number of intersections of that segment and all triangles on that way. This is just the sum of all w_l along the way.

For the opposite direction, we have to negate the w_l, since only the direction changes but not the geometry. Summation is done up to, but not including the actual final cell, as that would extend the ray one cell further.

$$\begin{aligned}\mathrm{w}(i,j) &= \begin{cases}\sum_{n=i}^{j-1} \mathrm{w}_l(n) & j \geq i,\\ \sum_{n=j}^{i-1} -\mathrm{w}_l(n) & j \leq i\end{cases}\\ &= \sum_{n=\min(i,j)}^{\max(i,j)-1} \operatorname{sign}(j-i)\,\mathrm{w}_l(n)\\ &= \operatorname{sign}(j-i) \sum_{n=\min(i,j)}^{\max(i,j)-1} \mathrm{w}_l(n)\end{aligned} \tag{1}$$

From this last formulation, it can be seen directly that w is antisymmetric, as only the factor in front of the sum changes.

$$\mathrm{w}(i,j) = -\,\mathrm{w}(j,i) \tag{2}$$

Let m be the maximum index and define the ray r_{in} from a cell i in the grid to ∞ as

$$\mathrm{r}_{\mathrm{in}}(i) = \mathrm{w}(i, m+1) \tag{3}$$

As there are no intersections outside of the grid, the value of w will not change after the last cell. All rays in the positive direction of all cells can be computed in one scan pass from m to 0 with

$$\mathrm{r}_{\mathrm{in}}(m) = \mathrm{w}_l(m), \tag{4}$$
$$\mathrm{r}_{\mathrm{in}}(i) = \mathrm{r}(i+1) + \mathrm{w}_l(i) \tag{5}$$

This process is similar to summed area tables or cumulative distribution functions. Importantly, this allows us to easily compute the intersection counts of the arbitrary line segments from before.

$$\mathrm{w}(i,j) = \mathrm{r}_{\mathrm{in}}(i) - \mathrm{r}_{\mathrm{in}}(j) \tag{6}$$

It can be seen by simple substitution that this represents Eq. 1 regardless of the order of i and j. With a slight modification, we can also define rays partially or fully inside or outside of the grid. A ray starting at a nonexistent cell i after the volume ($i > m$) will intersect nothing, so $\mathrm{r}(i) = 0$. For cells to the left ($i < 0$), there are no intersections until the volume starts, so we can just use the first cell as a starting point instead.

$$\mathrm{r}(i) = \begin{cases}0 & \text{if } i > m,\\ \mathrm{r}_{\mathrm{in}}(\max(0,i)) & \text{otherwise}\end{cases} \tag{7}$$

This allows us to compute one ray in the positive ($\mathrm{r}(i)$) and one in the negative direction ($\mathrm{w}(i,0) = \mathrm{r}(i) - \mathrm{r}(0)$) with only two precomputed values. Most importantly, this does not lose any geometric precision. It will also come in handy later, since paths between points can be composed of line segments along the coordinate axes. By the definition of w_l, the connection requires evaluating intersections in the two cells crossed. To avoid this, we compute the intersections in one cell and order them depending on whether they occur to the left or right of the cell's world space center, which is just a check of the computed intersection parameter. The sum of left intersections will be added to the current number stored in the left cell. The right sum will be added to the current cell's number. Cells are not processed all at once, since otherwise synchronization or atomic operations between neighboring cells would be necessary. Instead, every second cell is processed at once in a first step. That way, it is safe to write to the left cell asynchronously. A second step will process every second cell with an offset of one. With this scheme, all intersections in a cell are computed only once. Figure 4 illustrates this process in 2D.

After the local intersection counts have been evaluated, we find the values of r with a simple scan from right to left over all entries using Eq. 5. We tried speeding this up by using the distance values computed previously to quickly move over free regions where there will be no change in the intersection count. This can be seen as a discrete form of sphere tracing [11]. While we found that often times the number of cells needed to find the correct sign went down significantly, it did not accelerate the procedure.

We define the n-dimensional ray map as a function

$$\begin{aligned}&\mathrm{r}_D : \mathbf{N}^n \to \mathbf{Z}^k,\\ &D = \{\mathbf{d}_0, \mathbf{d}_1, \ldots \mathbf{d}_k\} \subset \mathbf{Z}^n \backslash \mathbf{0}^n\end{aligned} \tag{8}$$

which assigns each grid point the intersection counts for a set D consisting of k rays starting at that cell specified by their discrete directions $\mathbf{d}_i$, $i = 1...k$. The i-th entry of r_D is the intersection count of the i-th ray starting from that point in direction $\mathbf{d}_i$. In 1D, there is only one direction, disregarding multiples, while higher dimensions allow for infinite ones. The construction procedure from before only works for adjacent cells, since the ray would otherwise cross other cells on its way. This leaves 4 directions in 2D and 13 in 3D, which

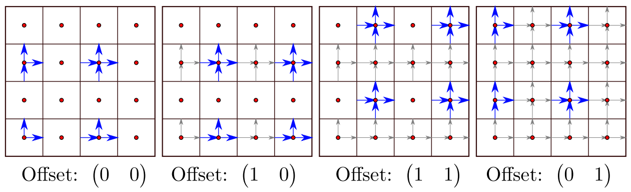

Fig. 4 Distribution scheme of the ray map construction. Every second cell in each direction is accessed in one step. This allows the cell to compute the intersections along the ray directions and distribute the values that belong to neighboring cells without synchronization

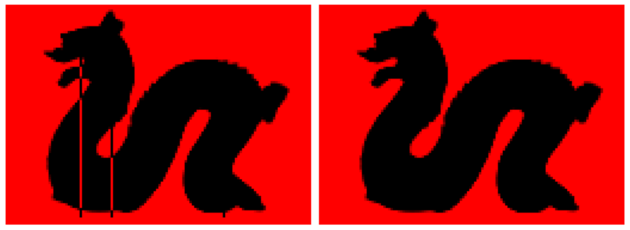

Fig. 5 Slice from the distance field of the dragon model. Using only one direction (left) may lead to wrong inside–outside classifications due to imperfect models or edge cases in the intersection routines. Using six rays can fix these small errors (right)

comes from the 8- and 26-neighborhood around a point and discarding the inverse directions, as these can be computed from the ray map without further processing.

For our 3D ray map, we use $D = \{\left(\begin{smallmatrix}1\\0\\0\end{smallmatrix}\right), \left(\begin{smallmatrix}0\\1\\0\end{smallmatrix}\right), \left(\begin{smallmatrix}0\\0\\1\end{smallmatrix}\right)\}$, allowing us to compute six rays from any point using Eq. 6. Computing the local intersection count is done just as in the 1D case, but with every second cell in each direction in each pass to avoid synchronization hazards in any of the three directions. This results in eight passes, each processing $\frac{1}{8}$th of cells. The accumulation of counts to compute the actual values of r_D also stays the same, but follows the procedure from the n-dimensional distance transforms. Each direction requires a scan along that direction. For example, the x direction requires one scan for each (y, z) coordinate pair in the grid resolution.

With six rays, errors such as a small hole can be fixed since most rays will still hit the correct geometry. As we only see a winding number of 0 as outside, we take the absolute average of all ray intersection counts and check whether it is close enough to 0, using some threshold t. We use $t = \frac{1}{2}$. An example of a wrong inside–outside classification using only one direction can be seen in Fig. 5.

This procedure so far is susceptible to slight changes in geometry. In the most extreme case, very small holes in an otherwise closed model, all aligned with the rays, will result in a misclassification. To counter this problem, we use the fact that the winding number does not depend on a ray, but can be an arbitrary curve going through the point to infinity. This can be thought of as follows: If we smoothly deform the surface without crossing the query point, the winding number must stay the same. For the number to change, the query point would need to cross a surface boundary. For a ray that counts intersections, this procedure would only possibly change the number of intersections encountered, not the sum of signed intersections itself as that would mean the winding number changed. We can construct the same intersection configuration by not deforming the surface but the ray instead. Figure 6 illustrates this.

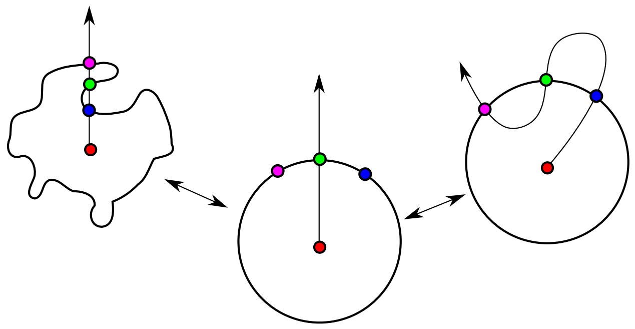

Fig. 6 Illustration of the intuition behind using curves instead of rays. The object on the left can be smoothly deformed into the middle circle. Deforming the ray to the one on the right side results in the same local intersections as for the left object

We can construct a path from one point $\mathbf{p}$ to any other point $\mathbf{q}$. Using a more relaxed notation, we denote $\mathrm{w}(\mathbf{p}, \mathbf{q})$ as the signed intersection count of a chosen path between the two points. Let $b_i(\mathbf{q}), i = 1, \dots, 6$ be the intersection counts of the six rays computed before. Their sum is $c(\mathbf{q}) = \sum_{i=1}^{6} b_i(\mathbf{q})$. The connecting path can be combined with any of those six rays to produce a curve to infinity with count $\mathrm{w}(\mathbf{p}, \mathbf{q}) + b_i(\mathbf{q})$. The sum of those curve intersections is then

$$\begin{aligned} c(\mathbf{p}) &= \sum_{i=1}^{6} w(\mathbf{p}, \mathbf{q}) + b_i(\mathbf{q}) \\ &= 6w(\mathbf{p}, \mathbf{q}) + \sum_{i=1}^{6} b_i(\mathbf{q}) \\ &= 6w(\mathbf{p}, \mathbf{q}) + c(\mathbf{q}) \end{aligned} \tag{9}$$

which reduces the computation to calculating $w(\mathbf{p}, \mathbf{q})$ and then combining it with the already found value $c(\mathbf{q})$.

This can be extended further by considering multiple paths to the next point. One example is to decompose the path into segments along the coordinate axes. This results in six possible paths for each permutation, i.e., *xyz*, *xzy*, *yxz*, *yzx*, *zxy* and *zyx*. These can be efficiently constructed with the ray map. Each of these contributes to six paths. For l connecting paths, this allows us to virtually construct $6l$ curves. That way a number of connection paths could be computed using the minimum amount of data accesses and then combined with a single data access to produce a multitude of virtual curves.

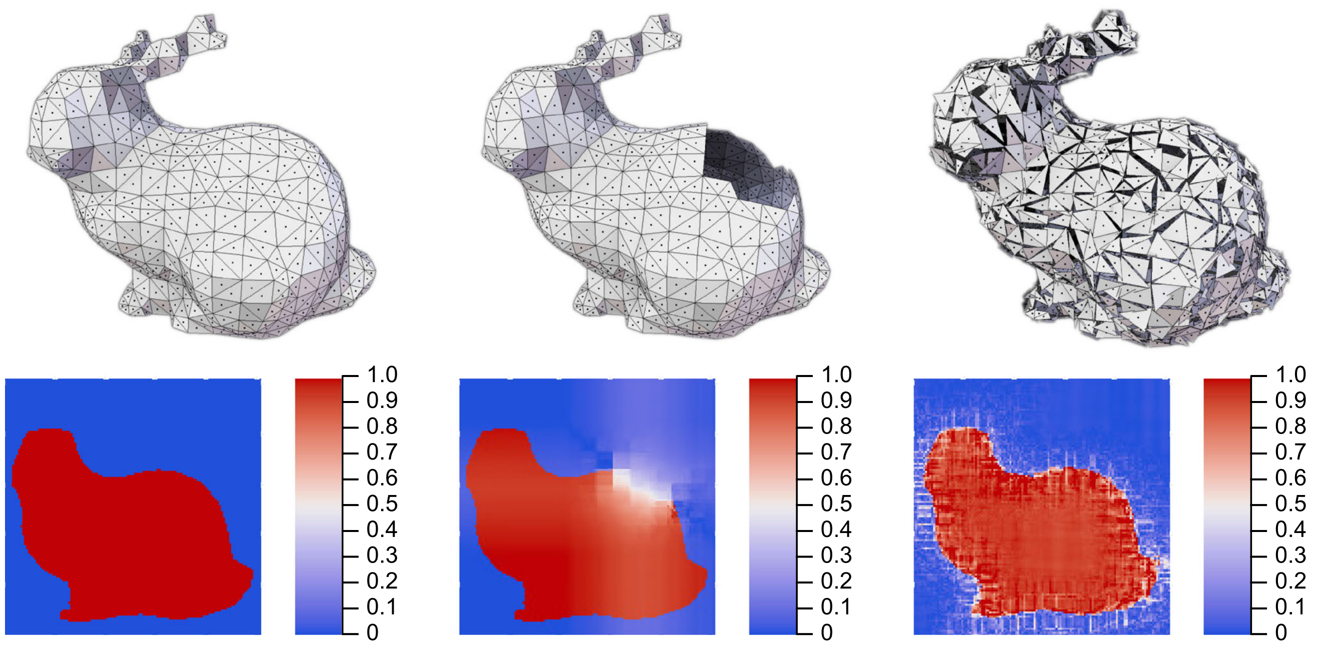

Fig. 7 Examples of the absolute averaged winding number fields for a simplified bunny model and two versions broken in different ways. The fields still find a correct inside–outside classification. For the middle version, where the error is concentrated in one region, we see a smooth change in field values around that region and clear edges away from it. The more chaotic bunny on the right is not closed and contains various intersecting triangles. The inside is still clearly segmented

This makes it possible to efficiently sample a huge number of all paths through the grid with relatively few samples. Additionally, the same reasoning can be applied not just to the six initial rays, but to any number n of curves starting at $\mathbf{q}$ resulting in ln virtual curves.

Taken even further, we consider applying this procedure iteratively. Let $c_t(\mathbf{p})$ be the signed intersection count with $n_t(\mathbf{p})$ curves at position $\mathbf{p}$ for step t. Initially, c_0 is the result of the six initial rays and $n_0 = 6$. Now, consider a number of sample points $\{\mathbf{q}_i\}_{i=1}^{v}$. $\mathbf{p}$ may be connected to one $\mathbf{q}_i$ by a number l_i of paths. Putting these thoughts together results in the evolution equation for c and n.

$$c_{t+1}(\mathbf{p}) = c_t(\mathbf{p}) + \sum_{i}^{v} n_t(\mathbf{q}_i)(\sum_{j}^{l_i} \mathrm{w}_j(\mathbf{p}, \mathbf{q}_i)) + l_i c_t(\mathbf{q}_i), \tag{10}$$

$$n_{t+1}(\mathbf{p}) = n_t(\mathbf{p}) + \sum_{i}^{v} l_i n_t(\mathbf{q}_i) \tag{11}$$

Equation 10 adds the counts of the virtual paths to the current count, while Eq. 11 updates the number of rays computed at each point. If the neighbors and paths are fixed the total number of paths can be computed once for all points. As this is an exponential growth, a lot of paths can be spawned with very few steps. For example, using eight neighbors with six paths each will result in 14,406 virtual curves after the second step. This can be a problem; as after a few steps, numbers may become too large for some data types. This could be optimized by using fewer well-spaced samples per pass and then doing more passes.

A simpler technique that still worked well for meshes with small deficiencies that we tested only counts intersections and uses the ray map. The only difference from before is that we do not aggregate from previous steps, but instead only use the six initial rays and different offset points for each iteration step. Thus we can add six times the number of offset points connected with different paths per step.

Regardless of the chosen method, when a specified number of passes are finished, the final result is averaged using the sum and total number of paths stored for each cell and then thresholded as before. Figure 7 shows the resulting averaged winding number field for different models.

4 Results

Our test setup has 32 GB of RAM, an Intel Core i7-6800K CPU and a GeForce GTX 1080 GPU. Our implementation is made in C++ using OpenGL. We have tested the performance of our system with a wide variety of models. The Thingi10k [27] database contains about 10.000 meshes of various sizes and qualities. It was created to reflect models encountered during real applications for 3D printing. Many of those con-

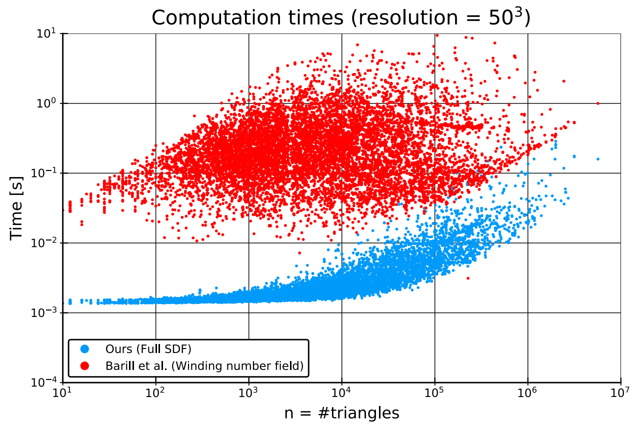

Fig. 8 Average runtimes for the models in the Thingi10k [27] database with a resolution of 50^3. Shown are our times for the full SDF computation in blue and the times for the fast winding number algorithm by Barill et al. [3] on the same system. We used their publicly available implementation, which runs multi-threaded on the CPU. It should be noted that they report smaller times using their system. Our system still performs the complete transform at faster times than their reported precomputation; however, their reported times are closer to ours

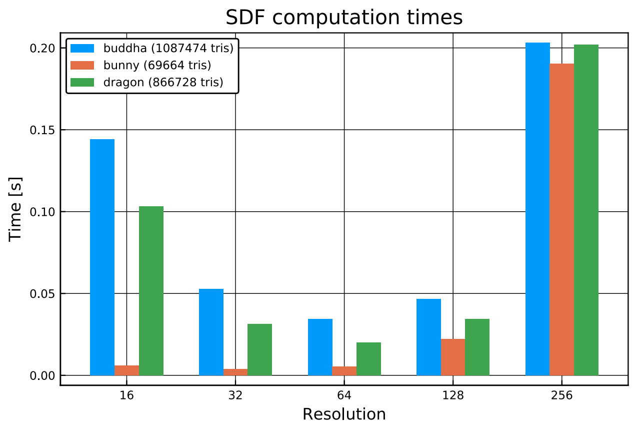

Fig. 9 Times for computing the SDF for three models of varying complexity for increasing resolutions

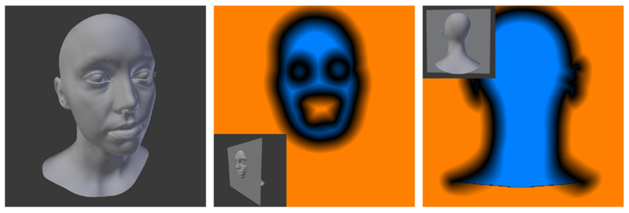

Fig. 10 SDF of the Emily scan from the Wikihuman Project. It contains inner geometry to model a hole in the mouth region. Eyes are separate objects that intersect the outer shell. The mesh is not closed at the bottom. Our method correctly classifies the mouth and eye regions as outside and inside respectively. The overall inner region of the model is found despite it not being closed

tain some amount of error with duplicated or missing faces, non-manifold geometry, self-intersections and others. Aside from the geometric properties, meshes range from 4 triangles to over 5 million. Results of that evaluation can be seen in Fig. 8. Our algorithm performs competitively with other work. The full SDF computation including signs is faster than the state of the art fast winding number algorithm by Barill et al. [3]. The sign computation takes up most of the time if more than 3 passes are used. Sign computations with 5 additional sign gathering passes averaged 55% for the 50^3 resolution and 51% for 100^3 of the complete runtime. This step warrants some implementation improvement.

The runtime is influenced both by the triangles to process and the resolution, but not in a simple linear fashion. Figure 9 shows average timings for three example models and some resolutions.

These results seem counterintuitive at first with lower resolutions taking more time than higher ones, but can be explained.

One obvious slowdown occurs when too many triangles fall in one cell, effectively negating the spatial partition of the grid. This can be seen with the Buddha model that has a spike in computation time at a resolution of 16. Another effect can be seen at higher resolutions. As we have optimized our distance transform to use fast local memory, we compute the maximum work group size that can be used to still fit in that memory. Increasing the resolution to 256 allows for less parallel items for more work, thus slowing down the computation considerably. A better way to utilize the shared memory in this scenario could speed up the execution a lot. A different approach would be to compute the distance transform at a lower resolution, for example 64 or 128. This would require the final distance gathering to search the neighborhood of the closest cell. We have found that in most models we have tested triangles do not on average appear in more than two cells for higher resolutions, thus leaving most cells relatively empty and the neighborhood query not too complex. The intermediate resolutions are not as affected by triangle number per voxel or the huge number of cells to process in the distance transform and thus allow for lower computation times.

In Fig. 7, we have shown that the averaged winding number approximation can accurately determine inside from outside even in deficient models. Figure 10 shows some additional results with a scanned and edited model that contains problematic elements such as inner geometry, intersecting geometry and holes.

We compared the signs computed by the method of Barill et al. with ours. For models without geometric deficiencies, our method produces the same result. A more interesting point of comparison is with deficient models.

Figure 11 shows a comparison of the results for two deficient meshes. The Emily scan contains intersecting triangles, internal geometry and a large hole. The distorted bunny is

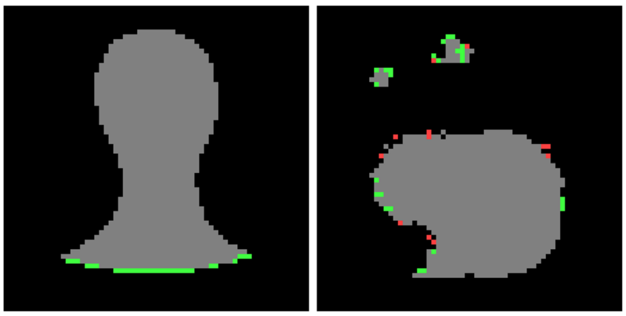

Fig. 11 Visualization of the differences between the method of Barill et al. and ours for Emily scan and the strongly distorted bunny of Fig. 7. The chosen slices contain the largest amount of differences out of all the slices for those models. Displayed in gray are all voxels for which the methods agree, red are the voxels classified as inside by our method only and green by theirs only. It is noteworthy that differences only occur at the undefined edges of the model

the same model as in Fig. 7. Its triangles do not share edges, intersect and are not consistently oriented in general. While there are differences for both meshes, they are less than half of one percent, for the Emily scan 0.15% and 0.24% for the distorted bunny. For the bunny and similar extremely deficient meshes, their method is less noisy at the boundary due to the smoothness of their approximation. Noteworthy is that the differences generally only occur at the undefined object boundaries. For the scan, this is the lower part of the shoulders, and for the bunny, it is the whole boundary. Both methods agree on voxels on the inside. Since a correct boundary is not uniquely and clearly definable in these cases, we believe these differences are justifiable for the increased speed, especially since the techniques agree for correct meshes and are nearly similar for only slightly broken models. One problem is that after determining the sign, the distance values do not correctly reflect the zero-iso surface at the hole. One solution could be to generate the sign first and then check for cells without any triangles in them. If there is a change in sign around one, the initial distance can be set to zero or to some approximated distance fitting the signs of the neighborhood. That way, surfaces could be automatically closed and still be valid distance fields.

5 Conclusion and future work

Utilizing localized properties such as cell distances and intersection counts can be used to efficiently generate signed distance fields on the GPU. We have shown that this method performs well for a wide variety of meshes. All data can be fully processed on the GPU so no slow data transfers between host and GPU are needed. Approximating the winding number using our ray map data structure is simple to implement and provides efficient sampling of the full discrete curve space. More directions could be used to further improve that sampling. Choosing resolutions based on the distribution and size of triangles could further speed up the method for desired distance field resolutions that do not match a mesh well. A hierarchical method could be used instead of a uniform grid, for example fast linear bounding volume hierarchies or octrees, which could also be created on the GPU.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Akenine-Möller, T.: Fast 3d triangle-box overlap testing. In: ACM SIGGRAPH 2005 Courses, SIGGRAPH '05. ACM, New York, NY, USA (2005). https://doi.org/10.1145/1198555.1198747
2. Baerentzen, J.A., Aanaes, H.: Signed distance computation using the angle weighted pseudonormal. IEEE Trans. Vis. Comput. Graph. **11**(3), 243–253 (2005). https://doi.org/10.1109/TVCG.2005.49
3. Barill, G., Dickson, N.G., Schmidt, R., Levin, D.I.W., Jacobson, A.: Fast winding numbers for soups and clouds. ACM Trans. Graph. **37**(4), 43:1–43:12 (2018). https://doi.org/10.1145/3197517.3201337
4. Bastos, T., Celes, W.: Gpu-accelerated adaptively sampled distance fields. In: 2008 IEEE International Conference on Shape Modeling and Applications, pp. 171–178 (2008). https://doi.org/10.1109/SMI.2008.4547967
5. Blelloch, G.E.: Prefix Sums and Their Applications. School of Computer Science, Carnegie Mellon University, Pittsburgh, Tech. rep. (1990)
6. Cuntz, N., Kolb, A.: Fast Hierarchical 3D Distance Transforms on the GPU. In: P. Cignoni, J. Sochor (eds.) EG Short Papers. The Eurographics Association (2007). https://doi.org/10.2312/egs.20071042
7. Fabbri, R., Costa, L.D.F., Torelli, J.C., Bruno, O.M.: 2d euclidean distance transform algorithms: a comparative survey. ACM Comput. Surv. **40**(1), 2:1–2:44 (2008). https://doi.org/10.1145/1322432.1322434
8. Felzenszwalb, P.F., Huttenlocher, D.P.: Distance transforms of sampled functions. Theory Comput. **8**(19), 415–428 (2012). https://doi.org/10.4086/toc.2012.v008a019
9. Frisken, S.F., Perry, R.N., Rockwood, A.P., Jones, T.R.: Adaptively sampled distance fields: A general representation of shape for computer graphics. In: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '00, pp. 249–254. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (2000). https://doi.org/10.1145/344779.344899
10. Fuhrmann, A., Groß, C.: Distance fields for rapid collision detection in physically based modeling. In: GraphiCon'2003, pp. 58–65 (2003)
11. Hart, J.C.: Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. Vis. Comput. **12**(10), 527–545 (1996). https://doi.org/10.1007/s003710050084
12. Hasselgren, J., Akenine-Möer, T., Ohlsson, L.: Conservative rasterization. In: Pharr, M. (ed.) GPU Gems 2, pp. 677–690. Addison-Wesley, Reading (2005)
13. Hoff, K., Zaferakis, A., Lin, M., Manocha, D.: Fast 3d geometric proximity queries between rigid and deformable models using

graphics hardware acceleration. UNC-CH Technical Report TR02-004 (2002)
14. Huang, J., Li, Y., Crawfis, R., Lu, S.C., Liou, S.Y.: A complete distance field representation. In: Proceedings of the Conference on Visualization '01, VIS '01, pp. 247–254. IEEE Computer Society, Washington, DC, USA (2001). http://dl.acm.org/citation.cfm?id=601671.601709
15. Hwang, Y.K., Ahuja, N.: A potential field approach to path planning. IEEE Trans. Robot. Autom. **8**(1), 23–32 (1992). https://doi.org/10.1109/70.127236
16. Jacobson, A., Kavan, L., Sorkine-Hornung, O.: Robust inside-outside segmentation using generalized winding numbers. ACM Trans. Graph. **32**(4), 33:1–33:12 (2013). https://doi.org/10.1145/2461912.2461916
17. Koschier, D., Deul, C., Bender, J.: Hierarchical hp-adaptive signed distance fields. In: Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation, SCA '16, pp. 189–198. Eurographics Association, Goslar Germany, Germany (2016). http://dl.acm.org/citation.cfm?id=2982818.2982844
18. Meijster, A., Roerdink, J.B.T.M., Hesselink, W.H.: A General Algorithm for Computing Distance Transforms in Linear Time, pp. 331–340. Springer US, Boston, MA (2000). https://doi.org/10.1007/0-306-47025-X_36
19. Rossignac, J., Fudos, I., Vasilakis, A.: Direct rendering of boolean combinations of self-trimmed surfaces. Computer-Aided Design **45**(2), 288–300 (2013). https://doi.org/10.1016/j.cad.2012.10.012. http://www.sciencedirect.com/science/article/pii/S0010448512002175. Solid and Physical Modeling 2012
20. Schneider, J., Kraus, M., Westermann, R.: GPU-based real-time discrete euclidean distance transforms with precise error bounds. In: International Conference on Computer Vision Theory and Applications (VISAPP), pp. 435–442 (2009)
21. Sethian, J.A.: A fast marching level set method for monotonically advancing fronts. Proc. Natl. Acad. Sci. **93**(4), 1591–1595 (1996). https://doi.org/10.1073/pnas.93.4.1591
22. Sud, A., Govindaraju, N., Gayle, R., Manocha, D.: Interactive 3d distance field computation using linear factorization. In: Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games, I3D '06, pp. 117–124. ACM, New York, NY, USA (2006). https://doi.org/10.1145/1111411.1111432
23. Sud, A., Otaduy, M.A., Manocha, D.: Difi: Fast 3d distance field computation using graphics hardware. Comput. Graph. Forum **23**(3), 557–566 (2004). https://doi.org/10.1111/j.1467-8659.2004.00787.x
24. Wright, D.: Dynamic occlusion with signed distance fields. In: ACM SIGGRAPH (2015)
25. Xu, H., Barbič, J.: Signed distance fields for polygon soup meshes. In: Proceedings of Graphics Interface 2014, GI '14, pp. 35–41. Canadian Information Processing Society, Toronto, Ont., Canada, Canada (2014). http://dl.acm.org/citation.cfm?id=2619648.2619655
26. Yatziv, L., Bartesaghi, A., Sapiro, G.: O(n) implementation of the fast marching algorithm. J. Comput. Phys. **212**(2), 393–399 (2006). https://doi.org/10.1016/j.jcp.2005.08.005
27. Zhou, Q., Jacobson, A.: Thingi10k: A dataset of 10,000 3d-printing models. arXiv preprint arXiv:1605.04797 (2016)

Bastian Krayer received his M.Sc degree in computational visualistics from the University of Koblenz-Landau (Germany) where he is now a Ph.D student working in the Computer Graphics Research Group. His research interests include geometry representations, visualization methods, and GPU-based acceleration techniques.

Stefan Müller is a professor at the University of Koblenz-Landau (Germany) and head of Computer Graphics Research Group since 2002. His main research area is photorealistic computer graphics, real-time rendering, virtual reality and augmented reality.