Skeletonization and Distance Transforms of 3D Volumes Using Graphics Hardware

M.A.M.M. van Dortmont, H.M.M. van de Wetering, and A.C. Telea

Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, the Netherlands
m.a.m.m.van.dortmont@student.tue.nl, wstahw@win.tue.nl, alext@win.tue.nl

Abstract. We propose a fast method for computing distance transforms and skeletons of 3D objects using programmable Graphics Processing Units (GPUs). We use an efficient method, called distance splatting, to compute the distance transform, a one-point feature transform, and 3D skeletons. We efficiently implement 3D splatting on GPUs using 2D textures and a hierarchical bi-level acceleration scheme. We show how to choose near-optimal parameter values to achieve high performance. We show 3D skeletonization and object reconstruction examples and compare our performance with similar state-of-the-art methods.

1 Introduction

The skeleton of a three-dimensional object is the set of interior points that have at least two closest points on the object surface. Alternative definitions use the set of centers of maximal contained balls [1] or first order singularities of the object surface's distance transform (DT). The skeleton points, together with their distance to the 3D surface, define the Medial Surface Transform (MST), which can be used for volumetric animation [2], surface smoothing [3], or topological analysis used in shape recognition, registration, or feature tracking.

While 2D skeletonization of raster images is a well-studied problem, skeletonization of 3D volumes still has some open issues. First, 3D skeletons tend to be far more complex than their 2D counterparts. Second, there exist several 2D criteria used to detect and/or simplify the skeleton in a noise-resistant way, e.g. the collapsed boundary length criterion [4, 5, 6]. However, there are hardly any similar 3D criteria that comply with the same requirements, e.g. prune and/or detect the skeleton starting from its less important points inwards, prevent skeleton disconnection during pruning, and are robust to noise. Last but not least, computing skeletons for large 3D volumes like nowadays medical scans can be a time-consuming process.

In this paper, we show how to compute 3D skeletons and distance transforms by extending to the 3D case a recent 2D skeletonization method that uses a new idea of computing skeletons by splatting distance textures [7]. We show how to efficiently implement the non-trivial 3D distance splatting on GPUs. Next, we show how to integrate a well-known 3D skeletonization criterion [8] in our splatting approach in order to compute 3D skeletons fully on the graphics card. We

A. Kuba, L.G. Nyúl, and K. Palágyi (Eds.): DGCI 2006, LNCS 4245, pp. 617–629, 2006.

keep the attractive features of the original 2D method (speed, implementation simplicity, arbitrary distance metrics). We demonstrate our approach with examples of skeletonizing and surface smoothing of real-world complex 3D objects.

The structure of this paper is as follows. Section 2 briefly overviews related work. Section 3 outlines the 2D splatting proposed by [7]. Section 4 details how we extended splatting to compute 3D skeletons. Section 5 presents our results, discusses the method, and compares it with its main competitor [8]. Finally, Section 6 concludes this paper.

2 Background

The methods for computing medial axes and skeletons can be algorithmically classified into three groups: thinning [9], Voronoi-based methods [4], and distance field methods [3, 7, 6]. In 3D, many such methods still have limitations. First, there is no generally accepted skeleton detection and/or pruning criterion that yields noise-resistant and connected 3D skeletons. For example, the θ-SMA method [10] detects skeleton points by thresholding the angle between the so-called feature points, or anchor points. This can yield skeletons with holes or even disconnections and is sensitive to noise. Euclidean Skeletons [11] improves upon θ-SMA by using a combined angle and feature point distance criterion. Other local criteria, e.g. divergence-based (Siddiqi *et al.* [12]) and moment-based (Rumpf and Telea [3]) have the same problem, i.e. can yield disconnected skeletons, unless homotopy is explicitly enforced, e.g. as in [13]. In this paper, we do not consider homotopy preservation as this is not efficiently implementable on GPUs. A second problem of 3D skeletonization is its relatively low speed. Recent GPU-based methods are one up to two magnitude orders faster than CPU-based skeletonization methods. Sud *et al.* [8] extract 3D skeletons on the GPU using the θ-SMA detector and Voronoi-based clamping techniques to limit overdraw. A related method [14] computes 3D signed distance transforms on the GPU, but not 3D skeletons. Strzodka and Telea [7] use the GPU to compute 2D skeletons using the collapsed boundary length, or anchor point distance, detector [4, 5, 6]. The skeleton and the boundary's distance transform (DT) are computed by a simple idea, called distance splatting, which is efficiently implemented on GPUs. Besides being simple, this method allows using any L_p metric, like Manhattan or (an)isotropically weighted Euclidean. Finally, we mention the important class of 3D thinning methods that compute skeletons by iteratively removing voxels from the object boundary in a given order [9]. Although simple to implement, and yielding connected skeletons, such methods can generate ill-centered and/or noisy skeletons, unless voxel removal is done in a true distance-to-boundary order, e.g. as proposed by [15].

3 Distance Splatting in 2D

Our aim is to generalize the 2D method described in [7] to perform 3D DT computation and skeletonization on the GPU, preserving its attractive points:

simplicity, accomodation of several distance metrics, and efficiency. The extension is not trivial, as the 3D case introduces specific difficulties, not present in 2D. We detail these (and our solution) in the following, starting with some definitions. Given an object $\Omega \in \mathbb{R}^3$ with surface $\partial\Omega$, the distance transform $DT : \Omega \to \mathbb{R}$ of $\partial\Omega$ can be defined as

$$DT(p) = \min_{q \in \partial\Omega} (dist(p, q)) \tag{1}$$

where $dist(p, q)$ is a distance metric (e.g. Euclidean or Manhattan). For a $p \in \Omega$, the feature transform $FP(p)$ yields the boundary points at distance $DT(p)$ from p

$$FP(p) = \{q \in \partial\Omega | dist(p, q) = DT(p)\} \tag{2}$$

The skeleton of Ω can be defined as

$$S(\Omega) = \{p \in \Omega | \exists q, r \in \partial\Omega, q \neq r : dist(p, q) = dist(p, r) = DT(p)\} \tag{3}$$

The tuples $(p, DT(p))$ with $p \in S(\Omega)$ form the medial surface transform (MST). Using the MST, one can reconstruct the surface $\partial\Omega$. To allow us to easily measure distances at any point $q \in \Omega$ from a given point $p \in \partial\Omega$, we introduce the Point Distance Function (PDF)

$$PDF_p(q) = dist(p, q) \tag{4}$$

For typical distances, we also have that

$$PDF_p(q) = PDF_0(q - p), \tag{5}$$

i.e. we can compute PDF_p by translating the PDF centered at the origin, PDF_0.

The 2D splatting method [7] we shall extend to 3D works on a discrete (image) sampling (V, V_S) of $(\Omega, \partial\Omega)$. Splatting computes 2D skeletons on the GPU in two steps. First, $DT(V_S)$ is computed by drawing PDF_0, sampled in a 2D texture, centered on all pixels $p \in V_S$. The actual distance minimization (Eqn. 1) is done during the drawing, by assigning the luminance-encoded distance values to the depth channels of the drawn pixels, and using the depth (Z buffer) test to mask pixels with greater distance values. The implementation takes a single texture draw with the pixel shaders functions of modern GPUs. Besides distance, splatting also propagates a second signal U, which encodes an arc-length boundary parameterization, so the method effectively computes a one-point feature transform of V_S. Next, the (pruned) skeleton $S(V, \tau)$ is computed as

$$S(V, \tau) = \{(i, j) \in \Omega | max(U_{i+1,j} - U_{i,j}, U_{i,j+1} - U_{i,j}) > \tau\} \tag{6}$$

The above gives the so-called collapsed boundary length at every pixel [4, 5, 6], i.e. all skeleton points where more than τ boundary units have collapsed. Increasing τ values prune the skeleton inward from its outer branches, yielding a connected, noise-free skeleton.

4 Distance Splatting in 3D

4.1 New Algorithm

A first problem of extending the above 2D algorithm to 3D is finding a suitable 3D replacement for the collapsed boundary length. A 'collapsed surface area' criterion would be a good candidate. However, we do not know how to (easily) compute such a measure. Hence, we use some simpler, though arguably less robust, local skeletonization criteria. Unlike global criteria, like the collapsed boundary length, local criteria, e.g. the θ-SMA angle [10], the divergence-based [12] or the moment-based criterion [3] use only information in a small neighbourhood of the considered point. These are more vulnerable to noise and can yield gaps or even disconnections in the skeleton. However, local criteria are simple and very efficient to implement on GPUs. After several experiments, we found the combined measure of angle between feature points and distance between feature points [11] the most robust in 3D and chose it as basis for our GPU skeletonization. A second problem is how to efficiently extend the 2D distance splatting [7] to 3D. In 2D, splatting could directly implement Eqn. 1, as explained in Sec. 3. However, though modern GPUs have 3D (volumetric) textures, they cannot render 3D primitives. To perform 3D splatting, we must find efficient ways to render volumetric primitives as a set of 2D (polygonal) primitives.

In our algorithm, we first generate the DT similarly to the 2D algorithm [7]. For all points p in the discretely sampled (voxelized) volume V counterpart of Ω, we compute the distance $DT(p)$ to the voxelized surface V_S counterpart of $\partial\Omega$, as well as *one* of its feature points $FP(p)$

```
1 Initialize DT to ∞
2 forall p in V_S
3   forall q in V
4     if (PDF_p(q) < DT(q))
5       DT(q) = PDF_p(q)
6       FP(q) = p
```

Listing 1.1. *Splatting-based DT computation*

This yields a one-point feature transform of V_S [16]. Next, we compute a skeleton detector $f(p)$ similar to [11]. In detail, we use

$$f(p) = angle(q)^a * DT(q)^b \tag{7}$$

where a=1, b=3/2, $angle(q)$ is the maximum angle between feature vectors $r-p$ at p, where $r \in FP(p)$ and $dist(q)$ is the maximum distance between feature points $FP(p)$ at p. Since we compute a single feature point $FP(p)$ instead of all potentially many feature points, we actually compute $angle(q)$ and $dist(q)$ using the neighbours $n(p)$ of p. Indeed, if p is near or on the skeleton, it will have

a neighbour $n(p)$ that has a feature point $FP(n(p))$ in a significantly different location than $FP(p)$, yet with a similar DT as p (see Eqn. 3). Another property to check for skeleton points is whether they are centers of maximal balls. If q is such a point, no ball centered at a neighbor p of q, of radius $DT(p)$, can completely contain a ball centered at q with radius $DT(q)$, i.e. $\forall p, q \in \Omega : p \in n(q) : DT(q) + \|q - p\| > DT(p)$. This property holds, among others, for the city block, chessboard, D^6 and D^{26} distance metrics. If a neighbour p of q fails this test, q is not the center of a maximal ball, so is not part of the skeleton. The complete detector computation is shown in Listing 1.2.

```
1  forall q in V
2    detector(q) = dist(q) = angle(q) = 0
3  forall q in V
4    forall p in n(q)∩V
5      if (DT(p) ≤ DT(q) + ‖q − p‖)
6        angle(q) = max(∠ (FP(p)−p,FP(q)−q), angle(q))
7        dist(q) = max(‖FP(q)−FP(p)‖, dist(q))
8      else
9        angle(q) = dist(q) = 0
10       break out of loop
11   detector(q) = f(angle(q), dist(q))
```

Listing 1.2. *Pseudocode for angle and distance-based skeleton detector*

For $n(p)$, we use the 6-neighbour set. [11] states that this suffices for accurately computing the detector in Eqn. 7. The skeleton $S(p, \tau, \alpha, \beta) = \{p \in \Omega | f(p) > \tau \wedge angle(q) > \alpha \wedge DT(q) > \beta\}$ is obtained by thresholding the detector f as well as the maximal feature angle *angle* and maximal inter-feature distance *dist*. Similar to [11], typical thresholds values are $\tau \approx 180$, $\alpha \in [45, 100]$ degrees, and $\beta \in [0.05\mathcal{D}, 0.15\mathcal{D}]$ where $\mathcal{D} = 2\max(DT)$ is the object diameter.

4.2 Implementation

We implemented our method in C++ using OpenGL and Cg (C for graphics) [17] as our shader language. We splat the 3D PDF texture (Listing 1.1, Sec. 4.1) using only 2D rendering primitives. We splat several 2D textures on an xy-axis-aligned, slice-by-slice basis, as described next (see also Fig. 1; line numbers refer to Listing 1.1). For every xy slice, the initialization (line 1) is done by clearing the depth and color buffers. We implement the loops in lines 2 and 3 by drawing quadrilaterals on the current slice (the thick vertical line in Fig. 1), textured with a 2D slice from the 3D PDF function (Eqn. 5). The distance minimization (line 4) is done by assigning the PDF value from the texture to the depth value of the drawn pixels, using a pixel shader. We use the depth test, so this implicitly does the minimization and yields the minimal DT value in the depth (Z) buffer. We save the DT (line 5) by copying it to the alpha channel of the drawn pixel.

Finally, we store the feature point (line 6) by writing the splatted point p's coordinates to the RGB color channels of the drawn pixel. The drawn image thus holds the DT in the alpha channel and the one-point feature transform FP in the RGB channels. The efficiency of our implementation depends critically on the PDF texture size. We store the 3D PDF as 2δ 2D texture slices of size $(2\delta)^2$, where δ is the PDF radius. Such a slice is shown in Fig. 1 with gray values. We do not use 3D textures as these lack the high numerical precision needed and also do not allow non-power-of-two sizes, which would increase δ unnecessarily. When splatting, we do not iterate over the entire set V_S, but over the smaller 'band' V_S' (thick line in Fig. 1), which includes the points on slices at most δ pixels from the current slice, since these are the only ones that can influence the DT result on the current slice.

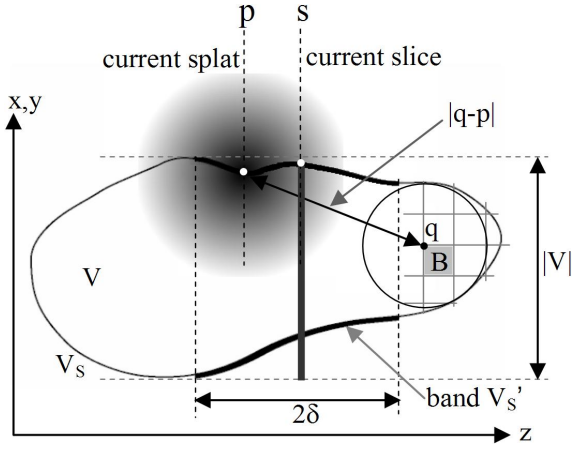


Fig. 1. 3D distance splatting principle

A (naive) upper bound for δ is $|V|/2$, i.e. half of the shortest axis of V's bounding box. However, this leads to many draw operations that do not affect the final DT. We reduce the overdraw by two techniques: a hierarchical optimization and a tighter upper bound estimation for δ, as follows. We implement a 3D version of the adaptive hierarchical optimization proposed by [7], as follows. We divide V in equally sized blocks B of c^3 voxels. We construct a coarse-scale version V_c of V, where V_c contains one sample of every block in V. We compute (also by GPU splatting) the coarse-scale distance transform DT_c of V_c, where the distance between two samples in V_c is is the maximal distance between any two voxels from their blocks $B \subset V$. For every block B, we splat only those boundary voxels that can affect its DT. These are all $p \in V_S$ that are closer to B than $DT_c(B)$. This bi-level hierarchical scheme has three advantages. First, we can quickly skip splatting the blocks B which are outside V. Second, we check if the minimal distance from B to the surface point p undergoing splatting ($|p-q|$ in Fig. 1) exceeds $DT_c(B)$ (shown by the radius of the circle centered at B).

If so, p cannot affect the DT of any voxel in B, so we skip splatting p over B. Finally, DT_c upper-bounds the radius at which a surface point p can influence the DT, so we use it as a tighter upper bound for the PDF size d than $|V|/2$. Our improved PDF size δ' is

$$\delta' = \min\left(\frac{|V|}{2}, \max_{B \subset V_c}(DT_c(B)) + 1\right). \tag{8}$$

This is a globally optimized PDF size (GPDF). We also tried a locally optimized PDF size (LPDF) that changes for every block B. However, this was slower than the GPDF, as detailed further in Sec. 5.

The second stage of the algorithm (Listing 1.2) is also implemented by rendering xy-aligned slices. The initializations (lines 1,2) are done by clearing the color buffer before drawing a slice. Next, we draw a rectangle for every volume slice (loop at line 3). The inner loop (line 4) is done using a vertex shader to generate the texture coordinates of the neighbours $n(q)$ so that the pixel shader can use these to access the relevant textures. If the fragment fails the ball containment test (line 5), it is discarded, since not part of the skeleton (lines 9,10). If the fragment passes the test, the maximum distance and angle are calculated (lines 6,7). We then use these to evaluate the detector f (Eqn. 7) and store it in a texture (line 11). Finally, we threshold this texture on-the-fly with the user-chosen values τ, α, β (Sec. 4.2), yielding the desired pruned 3D skeleton.

5 Discussion

We tested our method on both synthetic volumes and volumes segmented from real 3D scans (see Fig. 3). We used an Athlon 3.4GHz PC with 1 GB RAM and tested on two different GPUs, i.e. a GeForce 6800 with 128 MB and a GeForce 6600 with 256 MB graphics memory. We first compared our results with a software-only implementation based on the Euclidean Feature Transform method [16], which efficiently computes a feature transform (Table 1,column SW) and uses the same skeleton detector (Eqn. 7). Both methods yielded identical skeletons. We also used the pruned skeletons to reconstruct smoothed objects, by splatting the skeleton voxels with PDF functions equal to their corresponding MST values. It is well known that this replaces small-scale boundary details, corresponding to pruned skeleton points, with spherical surface segments. Figure 4 shows reconstructions for several objects. Our skeletons are indeed exact, as shown by the cube reconstructed from a non-pruned skeleton (Fig. 4 b), which is identical to the original cube (Fig. 3 f). We can easily handle noisy objects with highly complex 3D skeletons, e.g. the CT-scanned frog intestin (Fig. 3 a) or the MRI-scanned colon (Fig. 3 h). Reconstructing the colon from a highly pruned skeleton yields the smooth shape shown in Fig. 4 d.

We stress that our 3D distance splatting is exact by construction. Splatting propagates the distance from a boundary point directly, thus exactly, to the interior points. The depth test guarantees that the minimal distance is always correctly kept. This is not the case for incremental methods, e.g. level-set based [3,6],

Table 1. Benchmarks of splatting-based skeletonization

model	volume size	PDF size	object voxels	surface voxels	SW time	6600 time	6800 time	skel. voxels	recon. time	voxels/ sec.
cube	128x128x128	45	91125	11618	18	3.8	2.5	4961	0.8	4647
box	151x101x101	37	67392	9592	11.1	2.3	1.7	4032	0.5	5642
sphere 1	128x128x128	85	324157	18642	145.2	23.2	10.0	1	0.5	1864
sphere 2	256x256x256	171	2627271	75942	N/A	452.7	199.3	1	2.0	381
cylinder 1	51x51x51	31	61590	8138	5.9	1.7	1.2	4303	0.7	6781
cylinder 2	129x129x213	61	1674880	72043	781	188.1	19.2	37461	42.7	3752
cow	165x107x64	53	190041	21152	30	13.3	6.7	6402	2.6	3157
ellipse	100x100x100	25	23094	4164	3.2	0.9	0.7	288	0.1	6940
spring	100x100x100	15	38978	14013	2.2	1.7	1.5	2289	1.2	9342
ice 2	80x80x80	23	29880	5948	3.6	1.2	1.0	1255	0.6	5948
ice 3	80x80x80	29	41964	8104	4.7	2.3	1.7	1551	1.3	4767
rings	100x100x100	33	264784	28272	28.4	9.1	6.1	3222	1.6	4634
duo	72x69x90	23	36931	8636	3.1	1.7	1.4	2261	1.2	6168
intestin	60x71x94	17	13599	5724	3	0.9	0.8	1611	0.6	7155
colon	256x256x311	43	653170	81308	350.7	42.4	26.1	65120	35.7	3115
bent	150x150x150	49	429307	34211	92.9	21.8	11.6	10706	7.9	2949

that propagate information (e.g. distance, feature points) from point to point. Unless special measures are taken, such methods accumulate errors yielding visibly incorrect DTs and skeletons [16].

We would like to compare the performance of our GPU-based skeletonization with other methods, e.g. [3], [10], [11], [13], [8], and [12]. Unfortunately, this is far from trivial. These methods use different input and/or skeleton data models and skeleton detectors; have non-trivial, non-available implementations and/or test datasets; and performance is reported for different platforms. For example, we use a voxel-based model for both the input object and the computed skeleton, just as [11] and [12]. In contrast, [13], [10] and [3] use polygonal surface models for either or both.

The most interesting method to compare against is probably DiFi [8]. DiFi also uses GPUs to compute a DT and skeletons, and has a very similar skeleton detector (θ-SMA). DiFi handles both polygonal and volumetric objects. Since we do not have a DiFi implementation, nor its test objects, we shall compare our method with DiFi using the number of input object surface points processed per second. Comparing Table 1 (rightmost column) with Table 2), we see a large performance overlap between our method and DiFi. Our method skeletonizes objects at a rate of [3157..9342] surface voxels/second, with an average of 5356 (we left out the two spheres from this benchmark, since they are special absolute worst-case situations for any skeletonization method, also not present in DiFi's benchmarks). For DiFi, these figures are [1500..10500] voxels/second, with an average of 5516. As our method, DiFi can also handle many distance metrics, e.g. all L_p norms, if the Voronoi regions of the surface elements are connected. However, it is much easier to change the distance metric with our method than

with DiFi. We can use a specific distance metric by providing its sampled version as a 3D PDF texture. We can do this globally, but also locally. Every surface point can use another PDF function just by using another texture. For example, we can easily compute the so-called Johnson-Mehl or Apollonius diagrams [18], also called generalized skeletons, using additively, respectively multiplicatively, weighted Euclidean PDF functions, by scaling or multiplying the PDF texture at every point [7]. Doing this with DiFi appears to be significantly more complex [8].

Table 2. Skeletonization performance, DiFi method (from [8])

model	surface (voxels)	time (sec.)	voxels/sec.
octahedron	4862	0.85	5720
brain 1	18944	1.82	10408
brain 2	4988	0.64	7793
sinus 1	34507	22.1	1561
sinus 2	104154	49.7	2095

As Table 3 c shows, using our globally optimized PDF size (GPDF) calculation (Sec. 4.2) has a major performance impact for relatively elongated objects (e.g. 'bent', 'colon', 'intestin') where it massively reduces the amount of GPU overdraw during splatting. For objects tightly fitting their bounding-box, e.g. 'sphere' or 'cube', the optimization has no impact. Since the optimization itself does not cost extra time, it is always an efficient, valuable mechanism. Finally, we see that reconstruction is clearly faster than skeletonization (Table 1, column 'recon'). This is as expected, since a (pruned) skeleton has less points than the surface it comes from, and its MST values are exactly equal to the distance-to-boudary at every point, i.e. they match the absolute optimal PDF size value (Sec. 4.2).

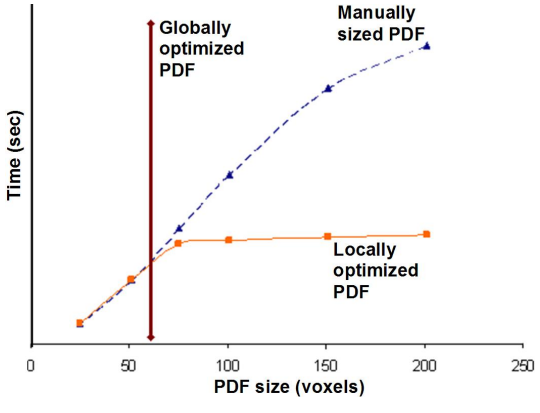


Fig. 2. Performance of local versus global PDF size choices

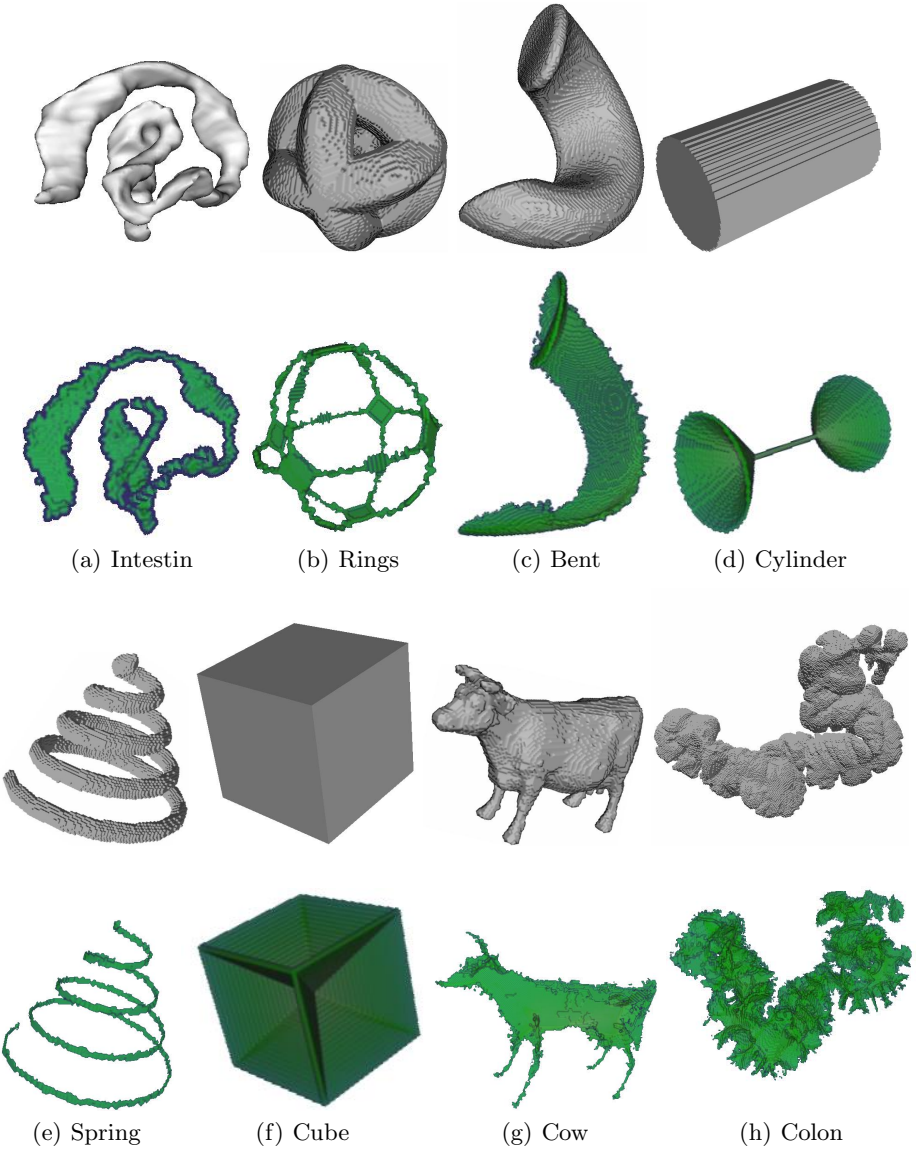
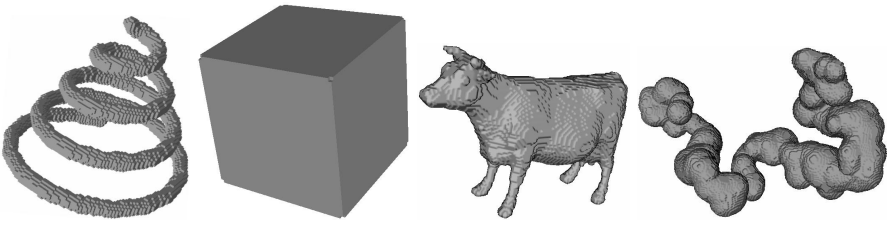


Fig. 3. Examples of 3D splatting-based skeletonization

Table 3 (a,b) shows the effect of using different coarse grid block sizes c in our bi-level hierarchical acceleration (Sec. 4.2). Increasing c means less CPU overhead, but more GPU overdraw. Decreasing c has the opposite effect. Varying c also implicitly affects the PDF size (Table 3). An optimal PDF size estimation would be obtained for the minimal block size $c = 1$. However, decreasing c increases the time needed to compute DT_c as well as the GPDF (Eqn. 8). For $c \leq 9$



(a) Spring (b) Cube (c) Cow (d) Colon

Fig. 4. Reconstruction of smoothed objects by splatting pruned skeletons

Table 3. Benchmarks for variable coarse grid size (a,b); Naive versus globally-optimized PDF size performance (c)

(a)

model	grid size	time (sec)	PDF size
rings	6	10.0	49
	7	9.9	53
	8	10.9	55
	9	12.8	63
	10	12.7	61
	11	14.6	67
	100	23.3	97
cow	6	7.4	53
	7	6.3	53
	8	6.5	53
	9	6.7	53
	10	6.7	53
	11	6.8	53
	100	6.8	53
bent	6	19.7	65
	7	16.1	65
	8	17.1	73
	9	17.4	74
	10	18.2	75
	11	21.0	83
	100	49.4	131

(b)

model	grid size	time (sec)	PDF size
spring	6	3.4	37
	7	4.0	43
	8	4.6	49
	9	5.2	55
	10	5.9	61
	11	6.6	67
	100	8.7	83
duo	6	2.0	37
	7	2.4	43
	8	2.6	49
	9	2.9	55
	10	3.3	61
	11	3.3	61
	100	3.5	61
colon	6	76.0	55
	7	57.7	57
	8	47.9	61
	9	45.9	67
	10	44.4	69
	11	51.9	77
	100	112	119

(c)

	naive		globally optimized	
model	PDF size	time	PDF size	time
cube	45	2.5	45	2.5
box	37	1.7	37	1.7
sphere 1	85	10.0	85	10.0
sphere 2	171	199.3	171	199.3
cylinder 1	31	1.2	31	1.2
cylinder 2	129	72.3	61	19.2
cow	53	6.7	53	6.7
ellipse	25	0.7	25	0.7
spring	83	7.4	15	1.5
ice 2	43	1.6	23	1.0
ice 3	79	4.5	29	1.7
rings	97	19.7	33	6.1
duo	61	3.3	23	1.4
intestin	43	1.5	17	0.8
colon	119	179.9	43	26.1
bent	131	50.1	49	11.6

voxels, this cost is no longer negligible. After extensive testing on several models, we found the optimal coarse block size c to lie between 7 and 10, so we chose 10 as a default value. Finally, we compared the efficiency of local (per-block) and global PDF size optimizations (see Sec. 4.2). We timed our method using the locally optimized PDF size (LPDF), globally optimized PDF size (GPDF), and also, for comparison purposes, a fixed-size PDF (FPDF) manually set to values ranging from 25 to 201. As the graph in Fig. 2 shows, GPDF picks a PDF size δ' for which the FPDF (ascending graph) and LPDF (leveled graph)

have the same, roughly linear, performance. For PDF sizes slightly larger than δ' (around 60 voxels in our graph), LPDF clearly beats FPDF. However, GPDF picked a size below this range for any configuration (3D shape) we availed of, so we settled with GPDF, which is simpler to compute than LPDF.

6 Conclusion

We have presented a flexible and efficient, yet very simple to program, algorithm to compute 3D skeletons on the GPU. We generalize the 2D distance splatting idea presented in [7] to the 3D case, and combine it with a different skeleton detector. Similar to [7], we use a bi-level hierarchical scheme to speed up our method by reducing the overdraw amount. Additionally, we use the coarse-scale distance transform (DT) to estimate an optimal size for our splat radius (PDF size), and thus reduce the overdraw even further. Since the optimal PDF size is highly object-dependent, and the GPU drawing performance is at least linearly dependent on the PDF size, this optimization can drastically improve the overall performance, as shown by our experiments. We performed extensive testing to evaluate our method on a range of volumetric objects, deduce optimal parameter values, and validated our results by performing (smoothed) object reconstructions from the skeleton. Overall, our simple splatting-based DT computation and skeletonization is as efficient as more complex methods, such as DiFi [8], and also lets one quite easily customize the distance metric used just by defining a 3D texture.

A more challenging subject, however, is finding efficient global criteria for noise-resistant detection and hole-free pruning of 3D skeletons. What such criteria might be, and whether they can efficiently be implemented on GPUs, is a subject for further research.

References

1. Blum, H.: A Transformation for Extracting New Descriptors of Shape. In: Models for the Perception of Speech and Visual Form. MIT Press (1967) 362–380
2. Gagvani, N., Kenchammana-Hosekote, D., Silver, D.: Volume animation using the skeleton tree. Proc. IEEE Volume Visualization (1998) 47–53
3. Rumpf, M., Telea, A.: A Continuous Skeletonization Method Based on Level Sets. Proc. VisSym (2002) 151–158
4. Ogniewicz, R.L., Kübler, O.: Hierarchic Voronoi skeletons. Pattern Recognition **28**(3) (1995) 343–359
5. Costa, L., Cesar, R.: Shape Analysis and Classification: Theory and Practice. CRC Press, Inc. (2000)
6. Telea, A., van Wijk, J.J.: An Augmented Fast Marching Method for Computing Skeletons and Centerlines. Proc. IEEE VisSym (2002) 251–258
7. Strzodka, R., Telea, A.: Generalized Distance Transforms and Skeletons in Graphics Hardware. Proc. VisSym (2004) 221–230
8. Sud, A., Otaduy, M.A., Manocha, D.: DiFi: Fast 3D Distance Field Computation Using Graphics Hardware. Computer Graphics Forum **23**(3) (2004) 557–566

9. Palágyi, K., Kuba, A.: Directional 3D Thinning Using 8 Subiterations. Proc. DGCI (1999) 325–336
10. Foskey, M., Lin, M.C., Manocha, D.: Efficient Computation of a Simplified Medial Axis. Proc. ACM Symp. Solid Modeling (2003) 96–107
11. Malandain, G., Fernández-Vidal, S.: Euclidean Skeletons. Image and Vision Computing **16**(5) (1998) 317–327
12. Siddiqi, K., Bouix, S., Tannenbaum, A., Zucker, S.: Hamilton-Jacobi Skeletons. IJCV **48**(3) (2002) 215–231
13. Sud, A., Foskey, M., Manocha, D.: Homotopy-Preserving Medial Axis Simplification. Proc. ACM Symp. Solid Modeling (2005) 39–50
14. Sigg, C., Peikert, R., Gross, M.: Signed Distance Transform Using Graphics Hardware. Proc. IEEE Visualization (2003) 83–90
15. Pudney, C.: Distance-ordered homotopic thinning: A skeletonization algorithm for 3d digital images. Computer Vision and Image Understanding **72**(3) (1998) 404–413
16. Reniers, D., Telea, A.: Quantitative comparison of tolerance-based distance transforms. Proc. VISAPP'06 (2006) 57–65
17. Pharr, M., Fernando, R.: GPU Gems 2: Programming Techniques for High-Performance Graphics. Addison-Wesley (2005)
18. Aurenhammer, F.: Voronoi diagrams: A survey of a fundamental geometric data structure. SIAM J. Comp. (27) (1998) 654–667