GPU-based Approaches for Shape Diameter Function Computation and Its Applications Focused on Skeleton Extraction

Article *in* Computers & Graphics · July 2016

DOI: 10.1016/j.cag.2016.06.006

CITATIONS
6

READS
1,225

7 authors, including:

Martin Madaras
Comenius University Bratislava
29 PUBLICATIONS **29** CITATIONS
SEE PROFILE

Adam Riečický
Comenius University Bratislava
7 PUBLICATIONS **14** CITATIONS
SEE PROFILE

Roman Ďurikovič
Comenius University Bratislava
108 PUBLICATIONS **455** CITATIONS
SEE PROFILE

Andrea Baldacci
Italian National Research Council
5 PUBLICATIONS **9** CITATIONS
SEE PROFILE

GPU-based Approaches for Shape Diameter Function Computation and Its Applications Focused on Skeleton Extraction

Abstract

In this paper two approaches for the computation of the shape diameter function (SDF) on the GPU are outlined and compared. The SDF is a scalar function describing the local thickness of an object. It can be used for consistent mesh partitioning and skeletonization. In the first approach, we have reorganized the tracing of the rays to be well suited for the rasterization hardware. To the best of our knowledge, this is the first method to show how to compute the SDF using only the rasterization hardware and without the need of any acceleration data structures. The second approach uses parallel ray casting and an octree traversal using OpenCL. We demonstrate that the first method achieves similar results as the ray casting using OpenCL. In addition, it is faster for large meshes and it is simpler to implement. Furthermore, we extend the SDF computation by fast post-processing using texture-space diffusion. The fast SDF computation can be used in many applications such as the automatic skeleton extraction as we demonstrate in the article.

Keywords:
shape diameter function, OpenCL, depth peeling, skeleton extraction, skeleton texture mapping

1. Introduction

The shape diameter function (SDF) is a scalar function defined on the mesh surface which expresses a measure of the diameter of the object's volume in the neighborhood of each point on the surface. See Figure 1 for some examples. The original idea was proposed by Shapira et al. [1] and it is related to the concept of Medial Axis Transform (MAT) [2].

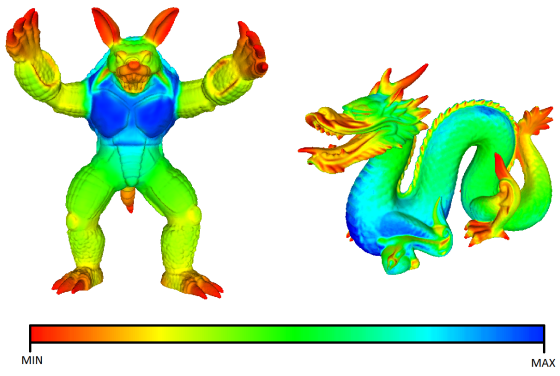


Figure 1: The shape diameter function (SDF) expresses a measure of the local thickness around each point on the surface. The SDF values are colored in a scale from low (red) to high (blue).

The SDF computation is based on creating a cone centered around the inward-normal direction of every point and sending several rays inside this cone to the other side of the mesh (see Figure 3), measuring the distance at the point of intersection. This process, despite being highly parallelizable, is computationally expensive. If processed on the CPU, the task becomes extremely inefficient. Therefore, this paper presents two algorithms to accelerate the calculation of the SDF. In the first approach, instead of tracing multiple rays per vertex at once, we launch one ray for each vertex multiple times. In fact, our method is based on iteratively peeling two or three successive depth layers of the mesh from multiple views around the mesh. From the review of the current literature, to the best of our knowledge, we can state that our depth peeling approach is the first one that computes the SDF exploiting only the rasterization hardware and without the need of any acceleration data structures. In the second approach, we propose a parallel method for computing the SDF that is performed on the GPU using OpenCL, exploiting the independence of rays. Our approach starts by transferring the data containing indices of triangles and our acceleration structure to the GPU. We then calculate a diameter value for each ray independently. Finally, the computation of the final SDF value is performed for each vertex gathering its computed rays diameters.

Given the above tools for SDF computation, we illustrate a few practical applications of the fast SDF computation. The SDF values have to be computed repeatedly during an iterative process, thus we have to compute the SDF as fast as possible. Proposed applications are the computation of skeleton sheets for mesh parameterization and an improvement of mesh contraction weights to extract finer skeletons with small scale details. Additionally, we illustrate how the SDF can be used for the fast extraction of skeletons with linear topology.

2. Related Work

Shape diameter function The shape diameter function (SDF) was introduced by Shapira et al. [1]. Given a point on the mesh surface, the SDF is defined as the weighted average of

the lengths of rays traced from the given point inside the cone centered at its inward-normal. For each ray, the normal at the point of intersection is checked against the normal at the starting point. If the angle between the two normals is less than 90 degrees the ray is rejected. From a practical point of view, this test avoids taking into account rays pointing directly to the outside of the mesh. To make this method robust and also immune to non-watertight meshes, outliers removal is performed preserving only those rays whose lengths fall within one standard deviation from the median of all lengths. In [3], it is pointed out that the outliers removal technique proposed in [1] generates counter-intuitive results in some cases. For example, in the case of a mesh composed of two parallel (infinite) planes, the SDF value obtained by the Shapira et al. method is given by the average of one correct value that is the length of the ray cast along the normal and the lengths of all the other rays thrown inside the cone that systematically overestimates the correct diameter. The same problem occurs at the bifurcation of a Y-shape. In this case, the correct value would be the minimum of all rays lengths, thus increasing the opening of the cone has the only effect of adding noise to the diameter estimation when taking an average ray length. In [3], in order to resolve the dilemma between a small or large cone, a more conservative estimation of the SDF is introduced by using an adaptive cone size. In particular, the algorithm starts with a large cone aperture that gets decreased as long as the measure of the absolute growth of the minimum ray length remains within a certain threshold. For example, in the case of the infinite parallel planes this method converges to a small cone size giving a correct SDF value equivalent to the distance between the two planes. Another application of SDF was introduced in [4], where SDF is used for mesh segmentation based on strokes drawn by the user. Essentially, the main operation in the SDF calculation is given by tracing rays. In [1], such operation is performed on the CPU using an octree as an acceleration structure. As Table 1 shows, ray tracing on the CPU is costly and the need for an acceleration structure is mandatory. A fast approximation of the SDF is proposed in [5] where the SDF value is computed only for some Poisson-distributed points over the mesh surface and then the computed values are propagated across the whole mesh using the Poisson equation. The latter paper reports only the timing for a single low poly model of a horse where a performance in the order of seconds is achieved but at the cost of some approximation. Instead, our GPU method based on depth peeling computes the SDF values for each vertex without approximations and without the need of acceleration structures. Moreover, we achieve performance in the order of few seconds even for meshes with hundreds of thousands of vertices.

Depth peeling The depth peeling technique, introduced in [6], starts by rendering the scene getting the nearest fragment to the camera. In the subsequent passes, the previous depth buffer is bound to a texture in a fragment shader. The scene is rendered again and all fragments whose depth is less or equal to the corresponding depth from the previous pass are discarded. To avoid read-modify-write hazards, the technique ping-pongs the depth values between two different buffers. Depth peeling is a robust image-based solution to different kinds of problems, specifically: order-independent transparency, layered depth image and ray tracing. The most severe drawback of this technique is that it requires the geometry to be drawn multiple times causing a strong limitation for GPU-bound applications. A more efficient implementation of the depth peeling algorithm has been proposed in [7].

Parallel ray casting There are several ways to perform ray casting. Carr et al. [8] proposed to generate rays and perform traversal of acceleration structures on the CPU, then store the results and perform the ray-triangle intersections on the GPU. Purcell el al. [9] proposed to store the scene geometry and the acceleration structure on the GPU and to use the same device to perform both traversal and intersection. Recent efforts to optimize the algorithms for GPUs have demonstrated to obtain better results on traversal of acceleration structures than a single-core execution on CPU. Therefore, the second option is more suitable for the SDF algorithm.

3. GPU-based Computation of the Shape Diameter Function

We propose two different methods for computing the SDF using the GPU. The first approach exploits the OpenGL programmable pipeline via shaders. The second approach relies on OpenCL to parallelize ray casting for each vertex. We provide a comparison of two methods in terms of their computed values and performances.

3.1. Parallelized Ray Casting using Depth Peeling

The SDF calculation using depth peeling exploits the inherently parallel task of tracing independent rays in a different way than standard ray tracing. In particular, we start by generating a set of cameras by uniformly sampling a sphere around the 3D model and placing an orthogonal camera at each sampled position. The camera view direction is set to look down along the sphere normal at each sampled point. We call the set of cameras $\mathbf{C}$. As a preprocessing step, we also pack all vertices in a single texture that we call $\mathbf{T}$. Now, for each camera, we alternate drawing front or back faces performing a depth peeling until no more fragments are written or after a maximum amount of iterations is reached. We keep track of the result of the last 3 render operations using a circular array of 3 framebuffers: $\mathbf{F_0}$, $\mathbf{F_1}$ and $\mathbf{F_2}$. At every odd iteration, we bind the last three generated depth textures (two at the first pass) and we perform the computation of one diameter value along the current view direction. We accumulate the diameters computed from all directions in a texture array. From this array, we calculate the final SDF value by first removing outliers and then by performing a weighted sum of the inliers. As in the original algorithm, the weight of each ray is given by the inverse of the angle between the ray and the center of the cone around the inward-normal of each vertex.

To simplify the comprehension of the algorithm, we refer to Figure 2 during our explanation. An outline of this method is reported in Algorithm 1 and Algorithm 2. Given a camera $c \in \mathbf{C}$, we bind $\mathbf{F_0}$ and we render the front-most part of the model as depicted in Figure 2(a). We write to both the color and the

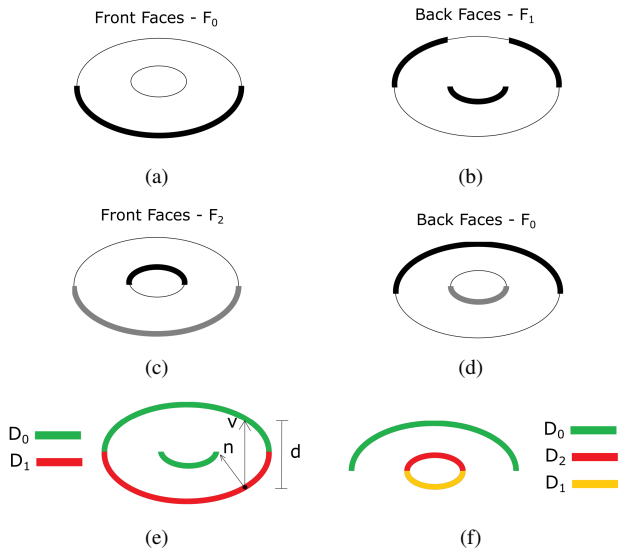


Figure 2: Figures (a), (b), (c) and (d) show the first four iterations of our depth peeling ray tracing applied to a torus. The images show the front-most surfaces as bold black lines, hidden surfaces as thin black lines, and peeled away surfaces as light grey lines. Figures (e) and (f) illustrate the diameter calculation launched at each odd iteration.

depth buffer, using a 32 bit floating point texture for the RGB color channel and a 32 bit depth buffer. In the color channel we store the normals of the vertices. At the next iteration, we draw the back faces of the model to $\mathbf{F_1}$ as depicted in Figure 2(b). Given $\mathbf{F_0}$ and $\mathbf{F_1}$, we can now calculate the diameter value for the vertices that belong to the first depth layer $\mathbf{D_0}$, which is the one stored in $\mathbf{F_0}$. The depth layer $\mathbf{D_0}$ is highlighted in red in Figure 2(e). We bind a framebuffer of the same dimension of the texture $\mathbf{T}$ and we draw a full-screen quad to launch a fragment program for each vertex. The fragment program fetches the position for each vertex from $\mathbf{T}$. The vertex position is transformed by the current camera matrix $c \in \mathbf{C}$ and its projection in the image space $\mathbf{v_p}$ is found along with its distance $\mathbf{v_z}$ with respect to the camera. At this point, we obtain the depth values $\mathbf{z_0}$ and $\mathbf{z_1}$ by sampling respectively the textures $\mathbf{D_0}$ and $\mathbf{D_1}$ (red and green lines in Figure 2(e), respectively) at the position $\mathbf{v_p}$. If $\mathbf{v_z}$<$\mathbf{z_1}$, we conclude that the vertex must belong to the first depth layer $\mathbf{D_0}$ and thus its diameter value along this view direction is given by: d = abs($\mathbf{v_z}$–$\mathbf{z_1}$). Otherwise, if $\mathbf{v_z}$>$\mathbf{z_1}$ we discard the fragment.

In the third iteration, we render to $\mathbf{F_2}$ the front faces of the 3D model but discarding all fragments whose depth is less than the one stored in $\mathbf{D_1}$. See Figure 2(c) for the resulting depth buffer $\mathbf{D_2}$. At the fourth iteration, we render to $\mathbf{F_0}$ (overwriting all data previously stored) the back faces of the 3D model discarding all fragments whose depth is less than the one stored in $\mathbf{D_2}$. See Figure 2(d) for the resulting depth buffer $\mathbf{D_0}$. At this point, we can calculate the diameter value for the depth layer $\mathbf{D_2}$ depicted in red in Figure 2(f). The computation of the diameter value proceeds exactly as in the previous pass except for the depth comparison performed between successive layers. As a matter of fact, at this pass we obtain the depth values $\mathbf{z_0}$, $\mathbf{z_2}$ and $\mathbf{z_1}$ by sampling respectively the textures $\mathbf{D_0}$, $\mathbf{D_2}$ and $\mathbf{D_1}$ at the projected vertex position $\mathbf{v_p}$, where the order of the values is consistent with the use of our circular array.

Algorithm 1 Depth Peeling Ray Tracing

```
Require: set of cameras C
Require: circular array of 3 framebuffers F
Require: diameters texture array A
Require: maximum iterations N
Ensure: SDF value at each vertex
  i ← 0
  j ← 0
  k ← 0
  for each c in C do
    while i < N do
      setCamera(c)
      if i == 0 then
        bindShader(nopeeling)
      else if j > 0 then
        bindShader(peeling)
        bindDepthTexture(F[j − 1])
      else
        bindShader(peeling)
        bindDepthTexture(F[2])
      end if
      if i%2 == 0 then
        F[j] ←renderModel(GL_FRONT)
      else
        F[j] ←renderModel(GL_BACK)
      end if
      if i%2! = 0 then
        if i > 0 then
          pB ← (j + 1)%3
          pF ← (j == 0)?2 : (j − 1)
          A[k] ←calcDiameter(F[pF], F[j], F[pB])
        else
          A[k] ←calcDiameter(F[j − 1], F[j], NULL)
        end if
        k ← (k + 1)
      end if
      j ← (j + 1)%3
      i ← (i + 1)
    end while
  end for
  SDF ←weightedSum(A)
```

Algorithm 2 Calculate Diameter

```
Require: front faces buffer FRONT
Require: back faces buffer BACK
Require: back faces from previous step PREVBACK
Require: texture pack with vertices normals Tn
Require: texture pack with vertices positions Tv
Require: current view direction D
Ensure: diameter value at current vertex
  N ←texelFetch(Tn, fragCoord.xy)
  angle ←acos(N.dot(D))
  if angle > 60 then
    discard fragment and return
  end if
  V ←texelFetch(Tv, fragCoord.xy)
  P ←project(V)
  if isFalseIntersection() then
    discard fragment and return
  end if
  zFront ←texture2D(FRONT , P.xy)
  zBack ←texture2D(BACK , P.xy)
  zPrevBack ←texture2D(PREVBACK, P.xy)
  if zPrevBack < P.z  and  P.z < zBack then
    return zBack − zFront
  else
    discard fragment and return
  end if
```

In the fragment shader for SDF computation, we thus accept only vertices whose depth $\mathbf{v_z}$ satisfies the following inequality: $\mathbf{z_1}$<$\mathbf{v_z}$<$\mathbf{z_0}$. The comparison using the three layers is necessary

because vertices whose depth satisfies the condition $\mathbf{v_z}<\mathbf{z_0}$ but not the condition $\mathbf{z_1}<\mathbf{v_z}$ already got their values computed from the previous passes. The algorithm described proceeds until no more fragments get rasterized or after a fixed number of iterations. To test if any fragment is drawn to the framebuffer, we simply start an occlusion query before each depth peeling iteration. We now point out some implementation details neglected until now for the sake of clarity. When evaluating the SDF function, the corresponding fragment program is early discarded if the current view direction does not form an angle of less than 60 degrees with the inward-normal of the vertex under consideration. Moreover, to perform the false intersection test from [1], we also pack all the vertex normals in a texture and we test the vertex normal against the normal at the point of intersection stored in the color texture drawn during the depth peeling pass. Finally, once a single diameter value is computed for a vertex, it is stored using the *imageStore* command in a texture array bound to the SDF fragment program as an *image2DArray*. Similarly, a counter contained in an unsigned integer texture is increased using the *imageAtomicAdd* function. At the end of the whole process, we have an unsigned integer texture containing the number of rays traced per vertex and a texture array containing all the ray lengths for each vertex. To gather the final SDF value, we bind a framebuffer of the same dimension of the texture **T** and we draw a fullscreen quad. For each vertex, we now loop over the diameter values contained in the texture array and we sort them using a naive bubble sort. Then, we calculate mean and variance of the values and we calculate the weighted sum of all rays lengths that are less than one standard deviation away from the median (which is taken as the middle element of the sorted values). We can perform SDF calculation both for vertices and faces. The aforementioned explanation assumes that values are calculated per vertex. For the face case, we trace rays through the barycenter of the triangle.

3.2. Parallelized Ray Casting using OpenCL

Our second approach for the computation of the SDF extends the original implementation of the algorithm by porting it to the GPU using OpenCL. The computation of the SDF is a three-step process that consists of ray generation, ray casting and traversal of the chosen acceleration structure and finally the computation of the distance at the point of intersection.

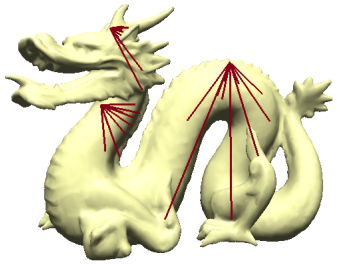


Figure 3: Rays are cast in cones from each vertex. The cast rays are displayed as red lines.

In the first step, we have to generate rays in a cone centered around the inward-normal of a given point and their associated weights (Figure 3). As in the original algorithm, the weight of each ray is given by the inverse of the angle between the ray and the center of the cone. The generation of rays on the CPU and their transfer to the GPU create unnecessary overhead because we must store every ray and its associated weight. Therefore, the rays should be generated on the GPU. However, random generation of rays would imply having a pseudo-random generator in the OpenCL. Rolland [3] has tackled this problem by defining a cone sampling strategy consisting of random rays that are uniformly generated inside the cone. Our algorithm further expands the former method by evenly distributing the rays in a cone using the spherical Fibonacci point set [10]. Furthermore, a uniform distribution of rays helps prevent unnecessary bias in the values on the mesh. A comparison of random ray generation with respect to our approach can be seen in Figure 4. The algorithm generates rays restricted to a given sphere cap.

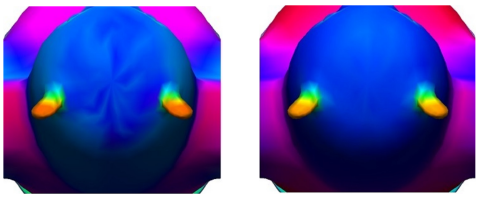


Figure 4: (Left) random ray generation, (right) uniform ray generation using spherical Fibonacci point set.

Each ray is then transformed into the world coordinates by multiplying it by the tangent space matrix specified by the point's inward-normal and by the two orthogonal unit vectors spanning the tangent plane of the point. It is important to note that only valid points are used to generate the rays, where a point is valid if it has a non-zero normal, tangent and binormal vectors. In the case of non-manifold models, the use of triangle barycenters instead of vertices is more reliable as the normal of some points cannot be properly determined. As a side effect, this also helps reduce unnecessary data transfer thanks to the fact that normal, tangent and binormal can be easily calculated.

In the second step, for each ray, we traverse our acceleration structure. The acceleration structure is one of the most important parts of ray tracing. For the purpose of comparison with the original article, we built an octree structure, even though a BVH or kDTree could be traversed faster. Our octree implementation is based on Laine's stack-based approach [11].

In the third step, we compute ray-triangle intersections between the cast ray and the triangles contained in the found octree node. For this purpose, we use the Moller and Trumbore algorithm [12] because it mainly uses the dot and cross products which can be evaluated very efficiently on current graphics hardware. Once the values per ray are calculated, they are transferred back to the CPU, where the final outliers removal takes place. The rays with lengths that do not fall within one standard deviation from the median of all lengths are removed. The inliers are then averaged using the weights calculated in the first step. To overcome the errors in the measure caused by pose changes, a smoothing operation is necessary. Therefore, we perform a k-ring neighborhood Gaussian smoothing, where

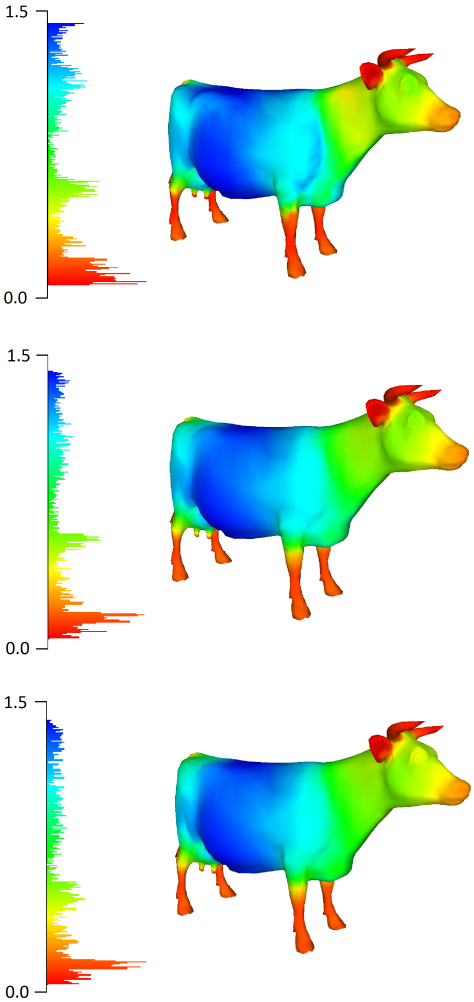


Figure 5: (From top to bottom) the SDF values of Shapira's method, our OpenGL depth peeling and our OpenCL raycasting. The SDF values are colored in a scale from low (red) to high (blue) and the histogram of computed SDF values is on the left side of the image.

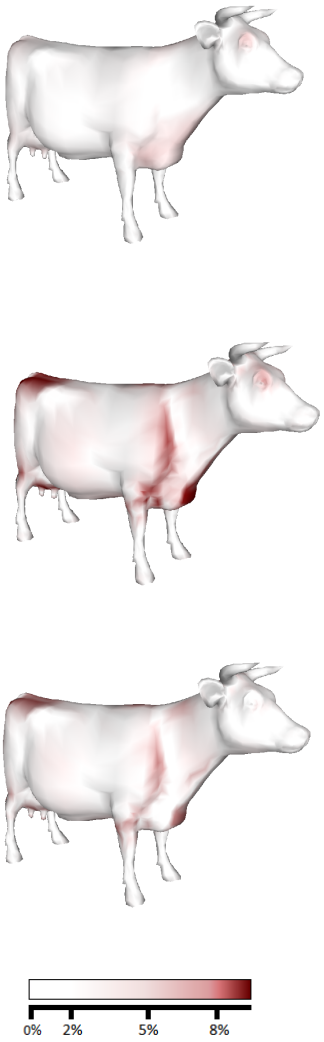
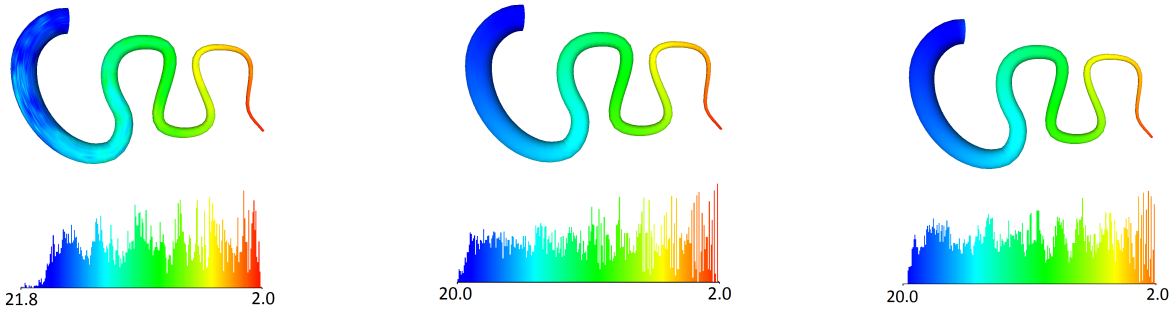


Figure 6: (From top to bottom) the difference of SDF values of OpenGL depth peeling vs. OpenCL raycasting, Shapira vs. OpenCL raycasting and Shapira vs. OpenGL depth peeling. (Bottom) the color scale of differences from white to red.

k specifies the blur radius in terms of edges in neighbourhood radius.

4. Results and Comparison

We now compare the results from both our methods for parallel computation of the SDF. Finally, at the end of this section, a texture-space diffusion method is proposed in order to smooth the values across the surface. A performance comparison of the two approaches is presented in Table 1. Furthermore, we have compared the values of OpenGL depth peeling, OpenCL raycasting and the original Shapira's implementation. The results of all three sets of SDF values with their related histograms can be seen in Figure 5. The differences among the methods mapped into a red scale are visualized in Figure 6. To ensure the correctness of our implementation, several basic tests have been performed on different shapes with known diameters. Figure 7 shows all three methods applied on a circle swept along a bezier curve with a diameter linearly interpolated between 2 and 20 units. All three methods give almost the same results and the highest deviation (up to 10%) from the real diameter has been measured in the thickest parts of the model using the original Shapira's method. Tests on all 3D models were performed using 30 rays for each point and with a cone aperture of 120. For the depth peeling method, we used 256 camera positions around each model. The differences between the SDF values of our two methods can be seen in Figure 9. All tests have been run on a desktop PC with the following configuration: Intel Core i5, 2,67GHz with 4GB RAM and AMD Radeon R9 290. As you can see in Table 1, the OpenCL raycasting is more efficient for 3D models with a modest number of vertices. For 3D models with a high vertex count (above 150k), the depth peeling approach is instead definitely faster. The processing times for the CPU approach using Shapira's method [1] and the processing times for the smoothing phase are also presented in Table 1. To perform the smoothing, each vertex and his neighbors are



(a) (b) (c)

Figure 7: The SDF computed on a model with a known diameter: a circle swept along a bezier curve, where the diameter is linearly interpolated between 2 and 20 units. Results using (a) Shapira's method, (b) OpenGL depth peeling and (c) OpenCL raycasting. The values are visualized from low (red) to high (blue). The histogram on the left of each method shows the distribution of the computed diameter values.

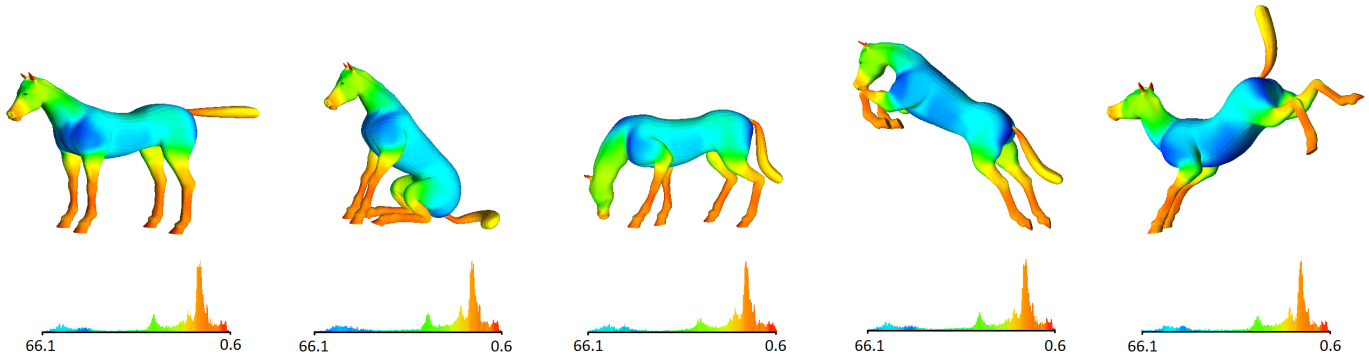


Figure 8: The SDF computed on a model in different poses using OpenGL depth peeling. The histogram below the images show that the SDF values computed using our technique are stable under pose changes.

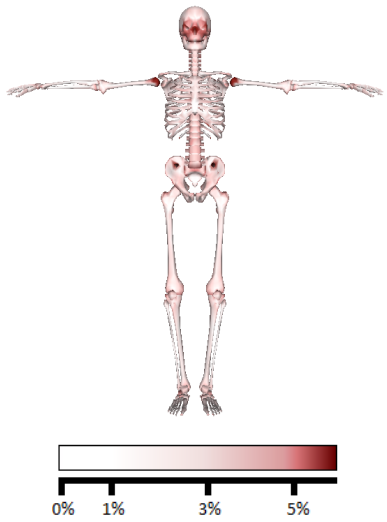


Figure 9: Comparison of computed values using our two GPU approaches. (Top) absolute difference of values between approaches. White vales are below 1%, a high difference is around 5%. (Bottom) the color scale of differences from white to red.

Mesh	Vertices	Faces	OGL	OCL	CPU	Smooth
Cow	2903	5804	392	31	878	43
C-shape	3554	7104	297	32	1017	46
S-shape	10034	20064	485	141	1984	140
Skeleton	8168	16040	687	78	735	141
Bunny	35947	69451	661	624	10376	530
Budha	144647	293232	1796	2262	39082	2808
Armadil.	172974	345944	2515	2901	89004	3557
Candela.	249976	499942	2418	4027	130324	4665
Gargoyle	500002	1000000	3957	8206	262695	7301

Table 1: Processing time comparisons of our two approaches and original Shapira's approach. The "OGL" are times for the depth peeling technique, "OCL" are times for the parallel ray casting, "CPU" are times for the Shapira's implementation and "Smooth" are times for the postprocessing step that makes Gaussian diffusion of SDF values on the mesh surface. For this smoothing step, we used neighborhood radius parameter $k = 2$. Times are in milliseconds [ms].

taken into account and the average of their values is calculated. Smoothing is computed on the CPU, because it cannot be implemented easily on the GPU as we need the k-ring connectivity information of the neighborhood of each vertex. As you can see in Table 1, this is the slowest step in our implementation.

4.1. Postprocessing using Texture-space Diffusion

We can use the extracted skeleton and a skeleton texture mapping (STM) [13] to blur the SDF values on the surface of the object. The idea is to map the SDF values into texture space, make the Gaussian diffusion there and map the smoothed values back to the model surface, as shown in Figure 10. Texture-

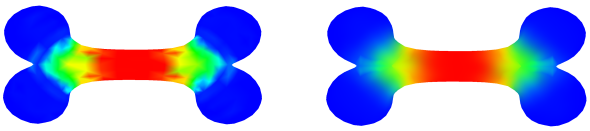


Figure 10: Computed SDF values (left) are stored in the texture and blurred (right) using texture-space diffusion with STM parameterization. Here, a kernel with radius 10px was used.

space diffusion using STM guarantees that local neighborhood of each point on the model surface is homotopic to local neighborhood in the texture space. Comparison of smoothing on the mesh using various k-ring areas and smoothing in the texture using Gaussian filter with various radius settings can be seen in Table 2.

Radius	k-ring	CPU Smoothing	STM Smoothing
64	6	14,882	0,140
32	5	10,764	0,093
16	4	7,566	0,057
8	3	5,024	0,051
4	2	3,120	0,048
2	1	1,762	0,041

Table 2: Smoothing in a 2048 x 2048 texture using various radius settings. Times shown are in seconds [s].

5. Further Applications and Extensions

In this section, we present some applications of computed SDF values focusing on skeleton extraction. These applications need to compute SDF values at each step during an iterative process. Therefore, these applications of SDF are a motivation for fast parallel SDF computation on the GPU. In the skeleton texture mapping approach presented in [13], we need to compute an internal skeleton from which every mesh vertex is visible. There are automatic techniques for curve-skeleton extraction [14, 15, 16], but extracted curve-skeletons using these techniques do not satisfy the above mentioned condition. Because of this, we have extended the Laplacian-based smoothing method presented in [14] with an SDF term that moves skeleton vertices near the medial axis transform (MAT). In addition, the SDF values can be used to modify parameters as weights and grouping distances during Laplacian-based skeleton extraction to obtain better results with skeleton extraction. For further details, we refer the reader to the original Laplacian-based mesh contraction [14]. Finally, we show an application of the SDF for the fast linear skeleton extraction.

5.1. Proposed SDF Modifications for Laplacian-based Skeleton Extraction

In order to parameterize a mesh using a skeleton texture map (STM) [13], we have to guarantee that the skeleton is reliable. Reliability refers to the property of the curve-skeleton that every boundary point (i.e., the point on object's surface) is visible from at least one skeleton location [17]. Some basic ideas behind the modifications of skeleton extraction algorithm are shown here. However, the in-depth discussion of this topic is out of the scope of this paper. For a more detailed description, we refer the reader to [18].

5.1.1. Skeleton Sheets using SDF

In most cases, a mesh can be parameterized by a curve-skeleton, but not in general. As a matter of fact, in some cases, the skeleton must contain sheets, similarly to the medial axis transform (MAT). A skeleton consisting of a mixture of curves and surface sheets was defined in [19] and called a "meso-skeleton". In [19], authors used Voronoi poles to guide the surface toward the medial axis. Here we show that this is the place where the SDF can be efficiently used. Shifted vertices into the central sheet using SDF are less noisy than Voronoi poles (or MAT) and SDF can be computed faster using our GPU approaches. Furthermore, our method creates skeleton sheets only in the regions where the curve-skeleton reliability is not satisfied. Meso-skeletons created using Voronoi poles [19] present sheet surfaces also in the regions which could be parameterized using just the curve-skeletons. This is clearly not efficient for STM because skeleton sheet takes much more space in the final texture. Therefore, we want to have the skeleton sheets only in the regions where they are inevitable. We extended the mesh

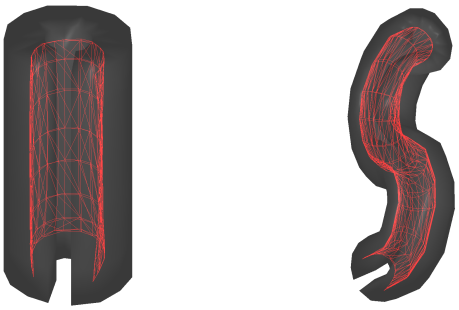


Figure 11: Examples of skeleton sheets of C-shape and S-shape models. Classical Laplacian-based skeleton extraction fails here because of non-convex cross-sections along the skeleton.

contraction method for skeleton extraction [14] with an SDF centering term as follows:

$$\begin{bmatrix} \mathbf{W}_L\,\mathbf{L} \\ \mathbf{W}_H \\ \mathbf{W}_C \end{bmatrix} \mathbf{V}' = \begin{bmatrix} \mathbf{0} \\ \mathbf{W}_H\,\mathbf{V} \\ \mathbf{W}_C\,\mathbf{C} \end{bmatrix}, \tag{1}$$

where matrix $\mathbf{C}$ is composed of vertices shifted by the SDF values in the opposite of normal direction $\mathbf{c_i} = \mathbf{v_i} - (SDF_i/2) \cdot \mathbf{n_i}$. $\mathbf{v_i}$, $\mathbf{n_i}$ and SDF_i are respectively the position, normal and SDF value of the vertex i. For a more detailed description of

the original contraction weights ($\mathbf{W}_L$, $\mathbf{W}_H$) and the introduced weights $\mathbf{W}_C$ we refer reader to [14] and to [18], respectively.

During the iterative contraction process, we tend to collapse the vertices into the medial axis composed of vertices shifted by half of the SDF values in the opposite of the normal direction. Using this extended version, we are able to create the curve-skeleton in the parts of the model where this is reliable and skeleton sheets where the curve-skeleton would not satisfy the reliability condition (see Figure 11).

5.2. Modification of Iterative Parameters

At each iteration of the iterative contraction process, the SDF values are computed and used to drive the modification of the contraction weights and grouping distance parameters. This results in skeletons which reflect better small-scale details of the 3D model (see Figure 14).

5.2.1. Updating of Grouping Distance using SDF

Here we show how SDF values can be used to create a skeleton from a not fully contracted mesh. During the simplification of the contracted mesh, we add another condition upon which a half-edge collapse is applied. This new condition is satisfied if an Euclidean distance between two nodes is less than a distance threshold dt_i. The distance threshold for each node is computed as: $dt_i = ggt \cdot diag + (SDF_i/2) \cdot gmt$, where ggt is a global grouping tolerance term, SDF_i is an SDF value for vertex i, gmt is global multiplication term and $diag$ is a diagonal of the model's bounding box. We obtained the best results with values set to $ggt = 0.05$ and $gmt = 1.25$. These parameters enable all the uncontracted parts of the mesh to collapse while the contracted parts of the mesh remain unchanged (see Figure 12).

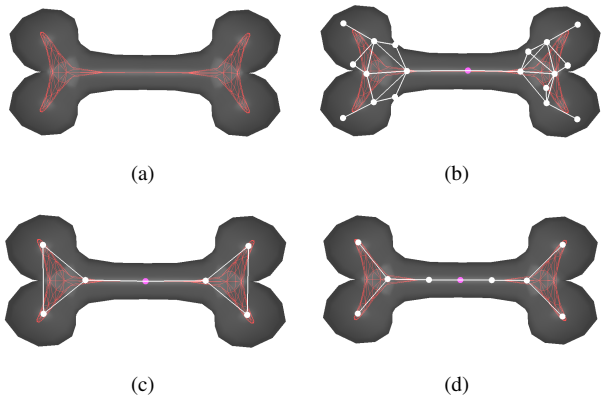


Figure 12: Grouping distance using SDF. (a) Contraction of the mesh when the volume is too big for skeleton extraction, (b) using lower global grouping distance, (c) using higher global grouping distance, (d) using local grouping distance based on SDF.

5.2.2. Updating of Contraction Weights

During the iterative contraction process, we change the holding weight $\mathbf{W}_{H_i}$ after each iteration based on the SDF values. We scale $\mathbf{W}_{H_i}$ according to the normalized SDF value $SDFnorm_i$. First, we update $\mathbf{W}_{H,i}$ according to the surface change of one ring areas as $\mathbf{W}^{t+1}_{H,i} = \mathbf{W}^{0}_{H,i}\sqrt{A^0_i/A^t_i}$, where A^0_i and A^t_i are the original and current areas of adjacent faces for vertex i, respectively. Then we normalize the SDF values to the interval $\langle 0, 1\rangle$ as $SDFnorm_i = (SDF_i - minSDF)/(maxSDF - minSDF)$, where $minSDF$ and $maxSDF$ are the minimal and the maximal SDF values. Afterwards, we change $\mathbf{W}_{H_i}$ according to the volume change using SDF as $\mathbf{W}_{H_i} = \mathbf{W}_{H_i}/(SDFnorm_i * (1.0 - bias) + bias)$, where $bias$ is an offset value because we want to avoid division by zero. We set bias as 0.01 in our tests, which produces high $\mathbf{W}_{H_i}$ values for $SDFnorm_i = 0$, while avoiding a division by zero. Using our iterative modification of contraction weights (Figure 13), we were able to preserve small-scale details also in cases where the thickness of the 3D model changes rapidly (see Figure 14). More details concerning this application can be found in [18].

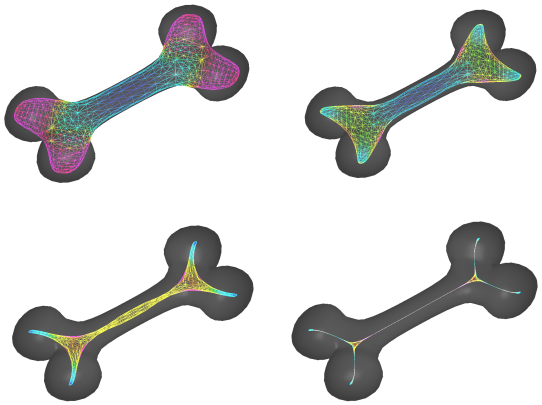


Figure 13: Contraction weights using SDF. The mesh is colored according to the SDF values. Red regions with higher SDF values are more contracted and blue regions with lower SDF value are less contracted.

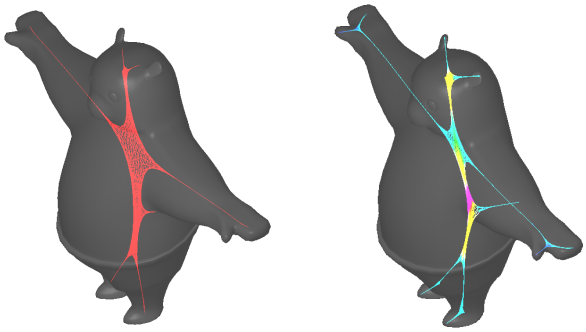


Figure 14: Difference of small-scale details as hands, ears and tail without SDF (left) and with the SDF term (right).

5.3. Fast Linear Skeleton Extraction

The SDF values have been used for the fast linear skeleton extraction from particle simulation. We used our fast parallel GPU implementation for linear skeleton extraction from particle simulation of the worm body [20]. The extracted skeleton

was used to compress the simulation and to visualize it using a skinning algorithm.

In order to compute the SDF of an input point cloud, a surface must be reconstructed. The reconstructed surface must be closed, but it must not be 2D-manifold. Therefore, we construct a local Delaunay triangulations for the neighborhood of each vertex in its tangent plane. In the next step, all edges in the local triangulations are copied into the global triangulation. This results in a graph that is a non-manifold orientable closed global triangulation.

Having unoriented point clouds, a normal can be estimated using PCA. However, the orientation of the normal remains unknown posing a serious problem for a correct SDF calculation. Therefore, we orient the normals in two steps. First, we choose a vertex whose normal orientation can be easily decided. For this purpose, we choose the outermost vertex in one direction, for example the topmost vertex. For this vertex, we are able to choose the normal orientation, because we know that its y coordinate has to be positive, facing out of the mesh bounding box. We then propagate the orientation over the mesh surface, by comparing normal orientations between neighboring vertices. At the end of this process we obtain an oriented point cloud.

Figure 15: The algorithm for skeleton extraction from the worm cuticle based on SDF. (a) Mesh of cuticle, (b) SDF half-vectors. (c) Shifted mesh vertices, (d) poly line obtained by agglomerative clustering and (e) extracted skeleton structure.

The cuticle of the worm has a simple topology without branching and a fast skeleton extraction algorithm is needed. The algorithm first computes the SDF values and then constructs the final skeleton using agglomerative clustering with the distance threshold t (see Figure 15). From clustered polyline a more subdivided skeleton can be easily constructed. The method is faster than all known robust methods and works well for the worm topology.

In order to match skeletons from all the timesteps and calculate the rotations between different timesteps, we need to ensure that each skeleton has the same number of segments. Thus, we find a clustering threshold t using binary search between the minimal and maximal SDF values computed on the surface. The computed threshold t is then used for agglomerative clustering during the skeleton construction.

6. Discussion

The way in which the parallelized OpenCL raycasting proceeds is similar to Shapira's approach [1]. The OpenGL depth peeling method proposes another point of view to approach the SDF calculation. Both proposed approaches achieved similar results in average, even though the depth peeling method is a bit faster in comparison to the parallelized version of ray casting technique, especially when the input model has more than 150k vertices. This is due to the fact that the whole SDF processing is calculated in a buffer space. Therefore, after the rendering phase, the calculation is independent of the mesh complexity. Both GPU methods are faster than former method as can be seen in Table 1. Unfortunately, we were not able to compare times with the Rolland's method [3], because computational times were not mentioned in their paper. The comparison of our two implementations against the Shapira's version shows that the results obtained are very similar. For this test, we set up our implementations to follow the details outlined in the original paper. The differences in the calculated values are very likely due to implementation details which were not mentioned in the original paper. Finally, as demonstrated in Figure 8, our depth peeling method is robust to pose changes with essentially no differences to the original Shapira's method with given a sufficient sampling of the view sphere around the model.

7. Conclusions and Future Work

We have implemented two different approaches for the computation of the SDF and compared their results and processing times. Both approaches are implemented on the GPU. The first one is calculated using shaders and depth peeling while the second one is based on parallelized ray casting using OpenCL. As mentioned in the discussion, we found out that it is more efficient to use OpenCL raycasting for relatively small meshes and OpenGL depth peeling for relatively large ones.

Furthermore, we presented several applications of the SDF. We have extended Laplacian smoothing-based skeleton extraction by an SDF term to obtain skeletons with finer details. We also extended curve skeletons to skeleton sheets which are contained in models with non-convex cross-sections. Finally, we used our customized SDF linear skeleton extraction for the fast compression of particle simulation based on skinning.

In both of our GPU implementations for the SDF computation, we maintain the original outliers removal approach proposed in [1], leaving the one proposed by [3] for future work. As a matter of fact, the adaptive cone size proposed in [3] helps with the estimation of the SDF only in some special cases, but it is computationally more expensive than original method as the ray casting must be iterated several times to find the correct cone size.

References

[1] L. Shapira, A. Shamir, D. Cohen-Or, Consistent mesh partitioning and skeletonisation using the shape diameter function, Vis. Comput. 24 (2008) 249–259.

[2] H. I. Choi, S. W. Choi, H. P. Moon, Moon: Mathematical theory of medial axis transform, Pacific J. Math.
[3] X. Rolland-Nevière, G. Doërr, P. Alliez, Robust diameter-based thickness estimation of 3d objects, Graphical Models 75 (6) (2013) 279–296.
[4] L. Fan, L. Lic, K. Liu, Paint mesh cutting, Computer Graphics Forum 30 (2) (2011) 603–612.
[5] M. Kovacic, F. Guggeri, S. Marras, R. Scateni, Fast approximation of the shape diameter function, in: GraVisMa 2010 Proceedings, 2010, p. 1724.
[6] C. Everitt, Interactive order-independent transparency, White paper, nVIDIA 2 (6) (2001) 7.
[7] L. Bavoil, K. Myers, Order independent transparency with dual depth peeling, Tech. rep., NVIDIA Developer SDK 10 (Feb. 2008).
[8] N. A. Carr, J. D. Hall, J. C. Hart, The ray engine, in: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, HWWS '02, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2002, pp. 37–46.
[9] T. J. Purcell, I. Buck, W. R. Mark, P. Hanrahan, Ray tracing on programmable graphics hardware, ACM Trans. Graph. 21 (3) (2002) 703–712.
[10] R. Marques, C. Bouville, M. Ribardire, L. P. Santos, K. Bouatouch, Spherical fibonacci point sets for illumination integrals, Computer Graphics Forum 32 (8) (2013) 134–143.
[11] S. Laine, T. Karras, Efficient sparse voxel octrees, in: Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, I3D '10, ACM, New York, NY, USA, 2010, pp. 55–63.
[12] T. Möller, B. Trumbore, Fast, minimum storage ray-triangle intersection, J. Graph. Tools 2 (1) (1997) 21–28.
[13] M. Madaras, R. Ďurikovič, Skeleton texture mapping, in: Proceedings of the 28th Spring Conference on Computer Graphics, SCCG '12, ACM, New York, NY, USA, 2013, pp. 121–127.
[14] O. K.-C. Au, C.-L. Tai, H.-K. Chu, D. Cohen-Or, T.-Y. Lee, Skeleton extraction by mesh contraction, ACM Trans. Graph. 27 (3) (2008) 44:1–44:10.
[15] G. Aujay, F. Hétroy, F. Lazarus, C. Depraz, Harmonic skeleton for realistic character animation, in: Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, SCA '07, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2007, pp. 151–160.
[16] T. K. Dey, J. Sun, Defining and computing curve-skeletons with medial geodesic function, in: Proceedings of the 4th EG symposium on Geom. processing, 2006, pp. 143–152.
[17] N. D. Cornea, D. Silver, P. Min, Curve-skeleton properties, applications, and algorithms, IEEE Transactions on Visualization and Computer Graphics 13 (3) (2007) 530–548.
[18] R. Ďurikovič, M. Madaras, Mathematical Progress in Expressive Image Synthesis II: Extended and Selected Results from the Symposium MEIS2014, Springer Japan, Tokyo, 2015, Ch. Controllable Skeleton-Sheets Representation Via Shape Diameter Function, pp. 79–90.
[19] A. Tagliasacchi, I. Alhashim, M. Olson, H. Zhang, Mean curvature skeletons, Comp. Graph. Forum 31 (5) (2012) 1735–1744.
[20] M. Piovarči, M. Madaras, R. Ďurikovič, Physically inspired stretching for skinning animation of non-rigid bodies, in: Proceedings of the 31st Spring Conference on Computer Graphics, SCCG '15, ACM, New York, NY, USA, 2015, pp. 47–53.