Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization (2004)
O. Deussen, C. Hansen, D.A. Keim, D. Saupe (Editors)

Generalized Distance Transforms and Skeletons in Graphics Hardware

R. Strzodka[1] and A. Telea[2]

[1] Centre of Advanced European Studies and Research (caesar), Bonn, Germany
[2] Department of Mathematics and Computer Science, Eindhoven University of Technology, Netherlands

Abstract

We present a framework for computing generalized distance transforms and skeletons of two-dimensional objects using graphics hardware. Our method is based on the concept of footprint splatting. Combining different splats produces weighted distance transforms for different metrics, as well as the corresponding skeletons and Voronoi diagrams. We present a hierarchical acceleration scheme and a subdivision scheme that allows visualizing the computed skeletons with subpixel accuracy in real time. Our splatting approach allows one to easily change all the metric parameters, treat any 2D boundaries, and easily produce both DTs and skeletons. We illustrate the method by several examples.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Curve, surface, solid, and object representation; Picture/Image Generation - Bitmap and frame buffer operations

1. Introduction

Distance transforms and skeletons are well-known shape representation tools with many applications in collision detection, geometric simplification and reconstruction, robot motion planning, mesh generation, and animation. Given an object boundary $\delta\Omega$, the distance transform (briefly DT) of $\delta\Omega$ is defined as

$$\mathrm{DT}(p) = \min_{q\in\delta\Omega}(dist(p,q)) \qquad (1)$$

for all points $p \in \Omega$. The distance metric is usually the Euclidean one $dist(p,q) = \|p-q\|_2$. The above assigns to every point p the distance to the closest boundary point q. This definition can be extended to the so-called complete or feature distance transform (FDT), as follows

$$\mathrm{FDT}(p) = \{\mathrm{DT}(p), \{q\}\},\ q = \arg\min_{q\in\delta\Omega}(dist(p,q)) \qquad (2)$$

The FDT labels every point with its DT value and the boundary points $\{q\}$ for which this value gets realized.

Skeletons, or medial axes, are defined as the set of centers of maximal balls contained in $\delta\Omega$, or the locus of points at equal distance from at least two boundary points:

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

Skeletons are useful as they characterize an object by a structure one dimension lower: The skeleton of a 3D volume is a 2D manifold, and the skeleton of a 2D surface is a 1D curve. Skeletons can be used to easily perform topological reasoning about the object, which is useful in shape analysis, registration, and recognition. If the DT is stored for every skeleton point, one obtains the medial axis transform (MAT). The entire boundary representation can be reconstructed from the MAT [TVW02, PSS*03]. By removing (pruning) less important parts of the skeleton, one can reconstruct simplified versions of the original object. If $\delta\Omega$ consists of several disconnected components, or sites, the set $S(\delta\Omega)$ corresponds to the Voronoi diagram of the sites. Different types of metrics in Eqns. 1-3 lead to the so-called generalized Voronoi diagrams [TVW01]. For example, by using a multiplicative weighted metric $dist(p,q) = K_p\|p-q\|$, where $p \in \delta\Omega$ and $q \in \Omega$, one obtains the Apollonius diagrams, describing the growth of plant cells and areas of best received transmitters [Aur91]. By using additive weights for the sites in the metric $dist(p,q) = K_p + \|p-q\|$, we obtain the Johnson-Mehl diagrams which describe the growth of crystals from a given seed set [Aur91]. Yet other generalizations exist, such as using the L_1, or Manhattan distance metric [TVW01].

Computing DTs and skeletons for objects given on an uniform pixel (in 2D) or voxel (in 3D) grid is a special case. Such objects frequently arise from digitized images or volumetric scanning, and are characterized by a large number of noisy boundary points. Given the large sizes Ω to be considered (e.g. mega-pixel images in 2D), as well as the inherent support for pixel and voxel data in graphics hardware, accelerating FDT and skeleton computation in graphics hardware has recently gained increased attention.

In this paper, we present a graphics hardware framework for computing generalized complete distance transforms and skeletons of two-dimensional objects. Regarding the DTs, our framework directly supports any distance metric, computes a pixel-exact complete distance transform, and supports boundary representations given as a set of possibly disconnected pixels. For the skeletons, we fully support the following quality requirements:

- produce one pixel thin, connected skeletons (R_1)
- produce skeletons for all generalized DT metrics (R_2)
- robustly handle noisy objects (R_3)
- allow an intuitive skeleton simplification (R_4)
- allow reconstructing objects from skeletons (R_5)

In Section 2, we review the existing graphics hardware methods for FDT and skeleton computation. Section 3.1 introduces generalized distance splatting, the simple key idea of our framework. Section 3 shows how splatting can be used to compute FDTs as well as skeletons. In Section 4, we refine the basic splatting with a hierarchical acceleration scheme that enhances the performance by an order of magnitude. We demonstrate our approach by several applications in Section 5. In Section 5.3, we show how to compute sub-pixel resolution skeletons that respect the mentioned quality requirements. Finally, we conclude the paper with a discussion and future research directions in Section 6.

2. Related Work

Since we are interested in computing both the DT and skeleton, we overview here only those (hardware based) methods that produce both. Concerning discretization, most hardware based methods represent $\delta\Omega$ as a sequence of linear primitives, i.e. lines in 2D and polygons in 3D. Virtually all methods represent the DT as a 2D pixel or 3D voxel image. Graphics hardware is used to efficiently compute the primitive DTs and to combine those into the complete DT field.

Skeletons can be computed either *after* DT computation, by postprocessing this field, or *during* the DT computation. Postprocessing methods, the most numerous, define and detect the skeleton as the singularity set of the DT. For example, the θ-SMA method detects the skeleton points p by computing the maximal angle θ formed by p and the boundary points q that p is closest to [FLM03, SM03]. Keeping the points with large θ values yields a simplified skeleton. However, as the authors explain, the θ-SMA method does not produce connected skeletons, and is sensitive to boundary noise such as sharp details. Conceptually, one could combine any local skeleton detector (such as divergence based [SBTZ99] or moment-based [RT02] detectors) with any DT computation. However, local detectors which classify a point as being skeleton or not using its immediate surroundings, have two problems. If the boundary and DT are not discretized on a very fine grid, singularity detectors using local derivation or integration fail producing a *one pixel thin* skeleton (R_1 in Sec. 1). Fine grids imply considerably higher computational costs. Secondly, a pruning criterion not based on some global quantity always disconnects the skeleton at some point, i.e. fails R_1.

Most acceleration schemes employ some Voronoi partition of the space around linear sites to limit the effect of a site on the complete DT to its vicinity [SP03, Mau03]. However, in our case every boundary point is a site, so computing such a Voronoi diagram is equivalent to the initial problem itself.

The second class of methods computes the skeleton *together* with the DT. Boundary information is propagated which allows both skeleton detection and pruning. The only graphics hardware based method which explicitely propagates boundary information is described by Hoff et al. [HCK*99] for generalized Voronoi diagrams. For a set of point and line sites, the DT graph of every site is encoded as a polygonal height mesh. Next, all DT graphs are drawn using depth testing, yielding the complete DT in the depth buffer. If the DT graphs are drawn in different colors, the color buffer will hold the sites' Voronoi regions colored by the sites' colors. Voronoi edges are found by applying edge detection on the frame buffer. The method has two related drawbacks. First, the DT graphs are only (linear) approximations of the exact DTs. This causes no problems for a few, large sites, or when interested just in the DT. However, we consider boundaries where every pixel is a separate site. When not using the *exact* DT graphs of the sites, even small DT errors are sufficient to propagate the wrong site color and yield wrong skeletons (see Fig. 1). This inevitably leads to noisy, disconnected, thus incorrect, skeletons. Exact point site DT graphs require a mesh of pixel-thin polygons. As explained in [HCK*99, SM03], this would dramatically increase computation time. In particular for point sites, the direct use of non-linear point site DT functions is therefore more efficient [SP03].

Few software skeletonization methods match all requirements listed in Sec. 1. For an overview, see [PSS*03, TVW02]. A good candidate in this class is the AFMM (Augmented Fast Marching Method) [TVW02]. Essentially, the AFMM computes the DT on a pixel grid Ω by solving the Eikonal equation $\nabla(\mathrm{DT}) = 1$ starting with $\mathrm{DT} = 0$ on $\delta\Omega$. Besides the distance, every pixel gets the id U of the closest boundary point, so the AFMM computes the FDT. U is initialized on $\delta\Omega$ to an arc length parameterization. That is, a point's id equals the distance along $\delta\Omega$ to some fixed start point). The simplified skeleton of $\delta\Omega$ is given by a thresholded derivative

$$S(\delta\Omega, U) = \{(i,j) \mid \max(U_{i+1,j} - U_{i,j}, U_{i,j+1} - U_{i,j}) > \tau\} \tag{4}$$

of the signal U on Ω. Objects with non compact boundaries (e.g. holes) are treated by transporting two ids U^1 and U^2 initialized from two different boundary start points. The skeleton is given by $\min(S(\delta\Omega, U^1), S(\delta\Omega, U^2))$. The AFMM is based on the fast marching method. Recently, level set methods have been implemented in graphics hardware [RS01, LKHW03]. But level set methods suffer from numerical diffusion, which causes problems when computing skeletons, as shown in Sec. 3.5. For an overview of related computational methods using graphics hardware, see also [Har].

Given the above, we advocate a combination of the AFMM boundary length skeleton detector, a signal propagation as in [HCK*99] and direct use of non-linear point site DT functions similar to [SP03]. Our combination results in a simple and efficient hardware based scheme and fulfills the quality requirements of the AFMM scheme.

3. Distance and Skeleton Construction

We assume an object boundary $\delta\Omega$ given as a set of image pixels. We lay no further constraints on $\delta\Omega$, such as computability of quantities like curvature or gradients, as other methods do [SBTZ99, SM03, RT02]. Our model fits well data coming with no high level (e.g. continuous) object model. Typical examples are scanned, possibly noisy, digital images, from which digital boundaries are extracted by image segmentation methods.

3.1. Distance Splatting

Since our boundary representation is just a set of discrete points, we proceed from this level on with the DT construction. Both the boundary and the DT are discretized on the same pixel grid $G = \{(i * \Delta x, j * \Delta y)\}$. Denote by $\mathrm{PDF}_0(q-p)$ the distance between any two points p and q . This depends on some supplied *point distance function (PDF)* $\mathrm{PDF}_0 : \mathbb{R}^2 \to \mathbb{R}$, e.g. the Euclidean norm $\mathrm{PDF}_0(v) = \|v\|_2$. We do not put any assumptions on this function. To conveniently measure distances form a point p we define

$$\mathrm{PDF}_p(q) = \mathrm{PDF}_0(q-p).$$

With the above, the generation of the complete discrete FDT of a given boundary $\delta\Omega$ can be summarized in a simple algorithm.

```
1 initialize DT to max_value
2 initialize U on boundary
3 for all boundary points p
4   for every point q
5     if (PDF_p(q) < DT(q))
6         DT(q)  = PDF_p(q)
7         U(q)   = U(p)
```

The skeleton can now be obtained by applying the thresholded derivative in Eqn. 4 to the computed signal U. For stable skeletons, it is essential to compute the derivative on the U values and *not* on the DT. The parameter τ simplifies the skeleton by removing those points which correspond to boundary details shorter than τ pixels.

If we ignored the transport of the signal U, i.e omit line 7, then lines 5 and 6 would reduce to a simple minimization ($\mathrm{DT}(q) = \min(\mathrm{DT}(q), \mathrm{PDF}_p(q))$) and we could trivially implement the DT computation in graphics hardware using standard OpenGL. For this, we encode the PDF as a single channel (luminance) texture of size D^2 pixels. A conservative estimate should set D to Ω's diameter, such that all object points get affected by all boundary points. Consecutively drawing the texture centered at every boundary point with the blending mode set to `GL_MIN` yields the DT. A similar technique is used by [YW03] for performing hardware based morphological operations.

3.2. Accuracy

Choosing the right per-pixel resolution for the representation of the PDF is essential for producing exact DTs and skeletons. If we use fixed point textures of b bits per pixel, we could encode the distance values $v \in [0..R]$ of a splat of R pixels radius as $\tilde{v} = [2^b(v/R)]$. For different distances $v_1 \neq v_2$, we want different encodings $\tilde{v_1} \neq \tilde{v_2}$. Given an R, we can evaluate the smallest occurring difference between two distance values on the pixel grid:

$$\Delta_{min} v = \min_{i,j,k,l \in 0..R} \left(\sqrt{i^2 + j^2} - \sqrt{k^2 + l^2} \right)$$

$\Delta_{min} v$ must be distinguishable in fixed point, i.e. $2^b(\Delta_{min}/R_{max})$ must be greater than 1. For 8 bit textures, we obtain $R_{max} = 11$ pixels. For 16 bit textures, $R_{max} = 180$ pixels, which allows objects of 360 pixels maximal diameter, whereas 24 bits permit R_{max} to exceed 3000 pixels. If we allocate a s23e8 floating point texture, we have more than enough mantissa bits. In case we do not avail of high precision textures, we can exactly encode a 24 bit number by the 8 bit color channels of a RGB texture. A dot product of this texture value with the constant $(256^2, 256, 1)$ then reconstructs the original 24 bit value.

3.3. Computing the FDT

Adding the U propagation to the DT splatting presented in the previous section cannot, however, be done in standard OpenGL. Specifically, standard OpenGL texturing and imaging cannot efficiently implement the U propagation conditioned on the DT test (lines 5-7 of the pseudocode in Sec 3.1). The three efficient comparisons we could use are `GL_MIN` type blending functions, the alpha test, and the depth test. Blending is the very last operation in the pipeline, so we cannot further process its results in one pass. Moreover, blending can only be performed on 8 up to, in some implementations, 16 bit colors, which would result in insufficient precision for the DT (see Sec. 3.2). Alpha testing only works against a fixed value for all pixels. The only standard OpenGL buffer which can be altered by rendering with 24 bit precision is the depth buffer. However, depth testing works efficiently only for linear primitives such as polygons, which limit the DT precision (Sec. 2). There is no OpenGL primitive allowing efficient drawing of non-linear depth images in the depth buffer.

However, the functionality of DX8 and DX9 graphics hardware allows per fragment replacement of depth values by e.g. the fragment's texture value. We use this mechanism to construct two simple and accurate implementations of the FDT splatting. These are discussed next.

3.4. Depth Replace

Depth replace refers to the ability to replace a fragment's depth value by other fragment data or any computed data. This allows direct implementation of the splatting algorithm in Sec. 3.1. The main algorithm (lines 1-5) stays the same, i.e. draws the footprint of the PDF texture centered at every boundary point. A fragment receives the distance value from the PDF texture and the two signals U^1 and U^2, encoded in fixed point in the RG, respectively BA channels of the fragment's RGBA color. The depth replace operation uses then the distance value to replace the depth value, while the color value stays unchanged. In this way, the depth test performs the minimization (line 6) and also conditionally transports the signals U^1 and U^2 (line 7). We obtain the desired signals U^1 and U^2 in the color buffer and the DT in the depth buffer. The corresponding pseudo-code simply reads

```
1 clear depth buffer to 1
2 set fragment processing to depth replace
3 for all boundary points p
4   set color to U(p)
5   draw PDF texture centered at p
```

As mentioned, there are two possibilities to implement the depth replace: fragment programs and texture shaders. These are presented next.

3.4.1. Fragment Programs

The standardized extension `ARB_fragment_program` allows implementing the depth replace on any current DX9 hardware, as the depth value is one of the modifiable results. Line 2 in the above pseudocode becomes the following fragment program:

```
1 !!ARBfp1.0
2 TEMP R0;
3 TEX R0.x, fragment.texcoord[0],texture[0],
  RECT;
4 MOV result.depth.z, R0.x;
5 MOV result.color, fragment.color.primary;
6 END
```

Line 3 reads the texture value, which is output by line 4 as depth. Line 5 sets the U signal encoded in color. Since writing to other components than `z` in `result.depth` has no effect, one could merge lines 3 and 4 into one. However, we'll need line 4 for an additional operation later (Sec. 5.1).

3.4.2. Texture Shaders

The older DX8 API also allows per-fragment replacement of depth by texture values. To our knowledge, this has been exposed in OpenGL only by the proprietary `NV_texture_shader(1,2,3)` extensions. All extensions implement depth replace, but this is simplest done via `NV_texture_shader3`. The setup of texture shaders reads

```
1 TS0: GL_TEXTURE_RECTANGLE_NV
2 TS1: GL_DOT_PRODUCT_AFFINE_DEPTH_REPLACE_NV
     input: result of TS0
```

The first shader (TS0) simply samples the value of the PDF texture, which is passed to the second shader (TS1). TS1 computes the dot product of the texture value RGB with the texture coordinates STR and replaces the fragment's depth. The dot product is computed in s23e8 floating point precision, so we can use it to reconstruct a 24 bit precise distance value from the 8 bit RGB colors, as described in Sec. 3.2.

3.5. Result Comparison

Both implementations deliver the same result, i.e. the signals U^1 and U^2 in a color pbuffer and the DT in the depth buffer. The simplified skeleton is computed by sampling the values U^1 and U^2 and implementing Eqn. 4 as a simple 2x2 filter in a separate fragment program or in the fixed fragment pipeline of DX8 graphics hardware. On our GeForce FX 5800 Ultra chip, the fragment program implementation is about 10% faster than the texture shader. This happens, however, because the graphics driver could not execute the shader setup from Sec. 3.4.2 correctly. Instead, we had to insert an 'empty' shader between the two operations to restore functionality. With a correct driver, the texture shaders would have been probably slightly faster than the fragment program.

An important aspect of our method is that it extracts *pixel-level correct* skeletons for all boundaries and threshold values, whereas the original AFMM, or similar methods based on incremental propagation, such as many thinning methods, do not. Figure 1 shows several problems such methods exhibit. First, double parallel skeleton branches are created in the one-hole plate in Fig. 1 c where single branches should appear, as correctly shown in Fig. 1 d. Secondly, branches passing close to high boundary curvature variations, such as the one exiting the leaf's twig, get incorrect angles (Fig 1 a

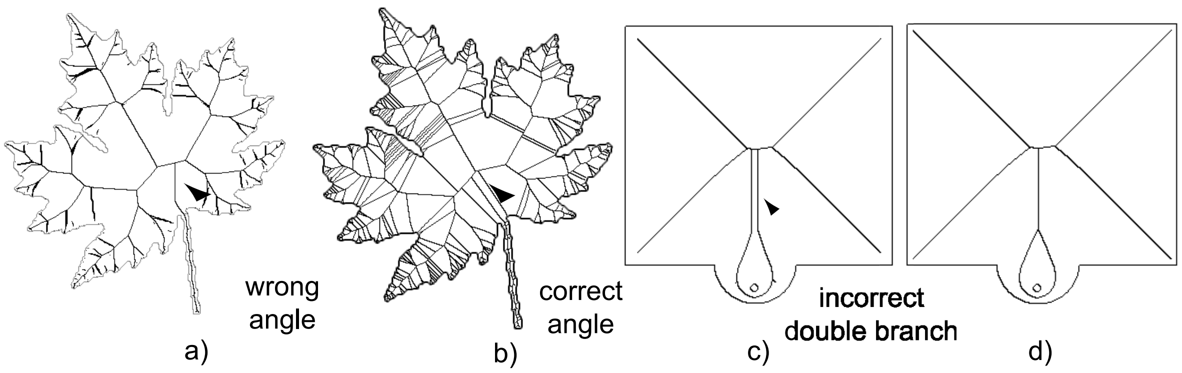

Figure 1: *Diffusion error (a,c). Correct skeletons (b,d)*

vs the correct angle in Fig. 1 b). Finally, for thresholds lower than 10 pixels, skeletons are noisy, disconnected, or have branches thicker than one pixel (Fig. 1 a). Incremental propagation, such as level set methods, suffer from a finite diffusion error accumulation. This is not visible in the relatively smooth distance field. However, such errors may translate into incorrect closest boundary point labeling in the U field, and given its discrete nature, thus to incorrect skeletons. The method presented here has no such problems, as it propagates information directly, and not incrementally, from the boundary. Skeletons are correctly computed regardless of the grid size or boundary noisiness. The threshold τ in Eqn. 4 can be set as low as the minimally possible value of 2 pixels. Remark that a threshold of 1 would yield a skeleton equal to the object, given that the U difference between two boundary neighbors is 1.

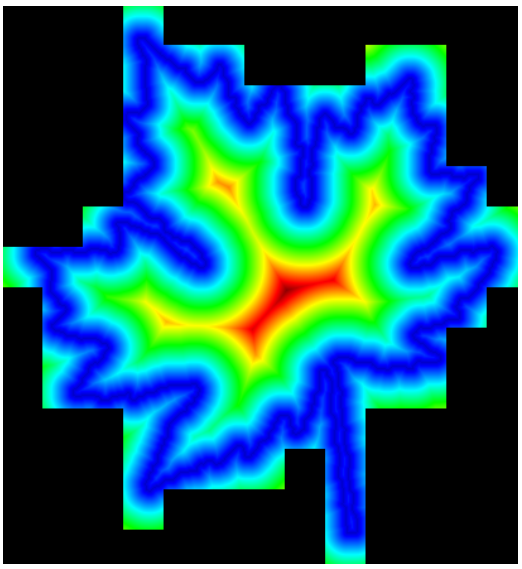

Figure 2: *Tiling in the hierarchical acceleration scheme*

4. Adaptive Hierarchical FDT Computation

The hardware based skeletonization method presented in the previous section is essentially limited by the pixel fill rate. For an object Ω of maximal diameter D and boundary $\delta\Omega$ of B pixels, the method needs to draw BD^2 pixels. In the worst case, D is unknown, so one must consider $D = \max(\Omega_W, \Omega_H)$, where (Ω_W, Ω_H) is the size of the object's bounding box, i.e. for every image pixel each boundary point is considered in the minimization of the distance. We can improve on the above by reducing the number of boundary points relevant for the minimization of the distance depending on the image region. The implementation of this adaptivity into the graphics hardware algorithm bears some resemblance to [LKHW03], where also a tiled representation is used to cull unnecessary computation.

First, we produce a coarse scale version Ω_c of the original object image Ω. For every pixel tile in Ω of size (W,H) pixels, the corresponding pixel in Ω_c is set to interior if any tile pixel is an interior one, otherwise is set to exterior. Next, we compute the distance transform DT_c on Ω_c using the method described in Sec. 3.3, where the distance between two pixels in Ω_c is the maximal distance between any two pixels from the corresponding tiles in Ω.

For the full scale image, we proceed as follows. For every boundary point p with coordinates (x_p, y_p), we construct a quad of coordinates $(0,0), (W,0), (W,H), (0,H)$ and texture coordinates $(x_S - x_p, y_S - y_p), (x_S - x_p + W, y_S - y_p), (x_S - x_p + W, y_S - y_p + H), (x_S - x_p, y_S - y_p + H)$, where (x_S, y_S) is the center of the PDF splat. All quad coordinates are assembled in an OpenGL vertex array A. Next, we iterate over all image tiles Q and reject those for which $\Omega_c(Q)$ is zero, i.e. the tiles outside the boundary, since one typically wants the DT and skeleton for the inside only. If this test passes, the tile Q is splat from all boundary points $p \in \delta\Omega$, as follows. For every p, we compute $d_{\min}$, the minimal distance between p and the tile Q (see Fig. 3). If $d_{\min}$ is larger than $\mathrm{DT}_c(Q)$, p cannot have any impact on the tile Q, so we skip

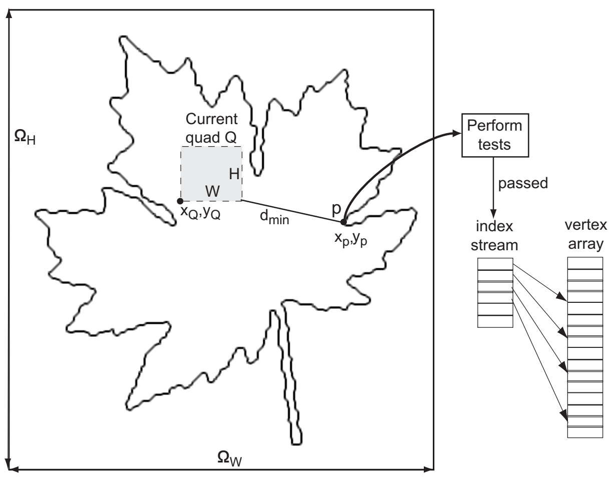

Figure 3: *Hierarchical acceleration scheme*

splatting it. If not, we add p's index in A to an index stream. After all boundary points for the current tile are processed, we set the coordinate and texture transforms to the translations $T_{coord} = (x_Q, y_Q)$ and $T_{tex} = (x_Q, y_Q)$, such that the coordinates from A will map to the current tile and the correct splat texture coordinates respectively. Finally, we draw the quads in the index stream and proceed with the next tile (the complete pseudocode is given in Fig. 4). Figure 2 illustrates the above scheme: the DT is visible only on the computed tiles, the rejected tiles are black.

```
for all quads Q of the image
{
  if $\Omega_c$(Q) is not background
  {
    set transforms $T_{tex}$ and $T_{coord}$
    for all points p of boundary $\delta\Omega$
    {
      compute $d_{min}$ between p and Q
      if $DT_c$(Q) > $d_{min}$
        write index of p to stream
    }
    draw stream
  }
}
```

Figure 4: *Hierarchical acceleration algorithm*

The efficiency of this hierarchical scheme depends on the choice of the coarse scale. Choosing a small tile size decreases the amount of overdraw, but increases the overhead caused by constructing the vertex array, computing the coarse DT$_c$, and drawing the quads. Choosing a too large tile size reduces the overhead, but also quadratically diminishes the savings. In practice, we have obtained optimal results by using a fixed tile size of 32 by 32 pixels. For this tile size, the first test typically eliminates about half of the quads. The second test eliminates between 80 and 97 percent of the quads (and thus pixels) to be drawn, as compared to the algorithm given in Sec. 3.1. The overhead caused by constructing and rendering the quads is about 10 percent of the total time, whereas computing the coarse scale FDT has practically no overhead. Overall, the FDT splatting is accelerated by 8 to 9 times.

Figure 10 show a number of skeletons computed by our accelerated methods from real data. The image in Fig. 10 c is obtained by segmenting a 1800 by 1800 pixels digital photograph of the roots of a rice plant, grown in a semi transparent jelly. Skeletonization allows extracting a root data model on which geometrical and topological measurements can be performed to assess the plant growth (Fig. 10 d). Since a large number of such images must be taken for different plants, growth phases, and viewing angles, the speedup offered by our method is important for this application. Similar measurements can be performed after skeletal extraction on the multipolar neuron image (Figs. 10 a,b).

Dataset	Time (AFMM)	Boundary pixels	Interior pixels	Total image pixels
Leaf 1	4.81	2160	67393	182040
Leaf 2	5.36	2864	70315	247401
Leaf 3	8.16	4134	110791	291250
Plate	4.19	1006	59799	90000
Roots	56.10	44657	371804	3258000
Room	16.78	9997	199740	208978
Neuron	16.11	14820	82632	613309

Table 1: *Software FDT Timings*

Table 1 shows, for a number of datasets, the timings given by a software AFMM implementation. Table 2 shows the timings of the simple brute force hardware method (BF) from Sec. 3.4 and the hierarchical variant (H), the percentage of tiles passing the two tests, and the speedup with respect to the AFMM software method. The brute force variant is slightly faster than the AFMM. However, the hierarchical variant gives a speedup of one order of magnitude. The above scheme works for more than two resolution levels too, however, for images up to thousands of pixels squared, two levels suffice.

5. Applications

5.1. Generalized Voronoi diagrams and skeletons

The presented method directly supports computing most versions of generalized Voronoi diagrams. One can choose the distance metric by simply sampling its footprint, for a point, in a texture splat. Additionally, the metric for every boundary point may be changed independently. For example, to implement the classical additive and multiplicative Euclidean metrics, we let every boundary point p_i have two weights

Dataset	Time (BF)	Time (H)	Test 1 pass (%)	Test 2 pass (%)	Speedup H/AFMM
Leaf 1	1.20	0.14	54.9	13.6	34.3
Leaf 2	2.10	0.20	46.0	11.1	26.8
Leaf 3	4.75	0.42	42.9	7.2	19.4
Plate	0.28	0.09	77.0	23.9	46.5
Roots	41.23	3.79	46.1	3.9	14.8
Room	3.31	0.64	100.0	9.9	26.1
Neuron	18.92	2.5	45.4	3.6	6.44

Table 2: *Hardware FDT Timings*

a_i and b_i. This translates to replacing line 4 in the fragment program in Sec. 3.4.1 with a MAD instruction that performs scaling and biasing with a_i and b_i respectively. Also, the DT$_c$ and the estimate $d_{\min}$ in the hierarchical algorithm (Fig. 4) have to be computed according to the new metric. Figure 5 shows, for a given point set, the additive (Johnson-Mehl) (b), classical (c), and multiplicative (Apollonius) (d) diagrams. Color intensities indicate the weight values. Figure 6 shows the Voronoi diagrams and DTs for the Euclidean and Manhattan distance metrics respectively on the more complex building floor dataset used by Hoff et al. in [HCK*99]. Such diagrams are often used in rigid motion path planning. An unexpected result we discovered by this visualization is that the diagrams for the two considered metrics are largely similar in the area of interest, i.e. close to the sites (inside the floor drawing), The differences become large only far away from the sites (outside the drawing). This result could be used to substitute more expensive by cheaper metrics in computationally intensive path planning simulations, given the diagrams of the two are similar.

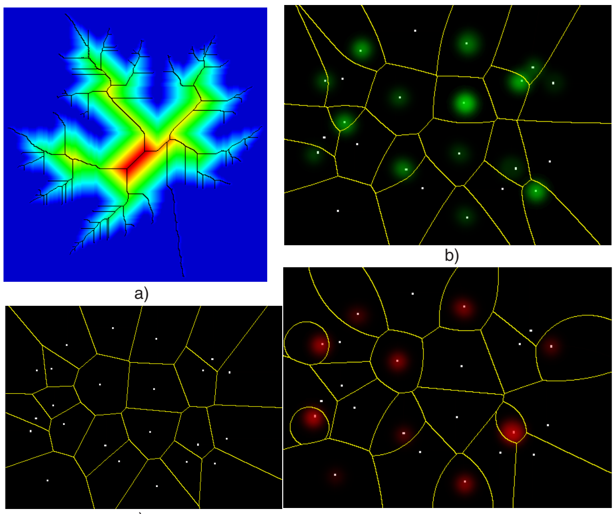

Figure 5: *Generalized Voronoi diagrams and skeletons*

An interesting related result is visualizing *generalized skeletons*, i.e. skeletons of other metric than the Euclidean. Figure 5 a shows such a skeleton, drawn in black over the distance transform, for the Manhattan metric (compare with Fig. 1 b for the Euclidean one). As for the Voronoi diagrams, we notice a large similarity of the two skeletons close to the boundary. Such generalized skeletons may open new possibilities for object simplification and recognition to the classical Euclidean ones.

5.2. Feature preserving evolution

A related application of weighted distance transforms is computing feature preserving evolutions. Given an object Ω and its DT, the set $\Omega_\tau = \{p \in \Omega | \mathrm{DT}(p) > \tau\}$ represents increasingly 'shrunk', or simplified, versions of the original object. The parameter τ is the time of an evolution in which the boundary $\delta\Omega$ advances with constant speed in its normal direction, as in a classical level set formulation [Set99]. In some applications it is desirable to preserve certain details in the above evolution while removing others. This translates to a low speed in the areas to be preserved and a high speed in the areas to be simplified. We can use the weighted DTs produced by our method to simulate the above. In the example in Fig. 7 a, the user has marked the leaf's twig and tips to be preserved with red. The color intensity is interpreted as a multiplicative weight for the Euclidean distance. Once the DT is computed (Fig. 7 d), the parameter τ is interactively set and used in a simple fragment program to threshold the DT to deliver the evolved images Ω_τ. Figure 7 b and c show how the marked details are preserved during two instants of the level set evolution. Figure 7 d shows the resulting distance transform overlaid with the generalized skeleton. One notices how the skeletal branches bend in regions of high DT gradients (close to the weighted areas) to follow the shortest path to the boundary. Also, we notice the apparition of skeletal branches close to and parallel with the weighted boundary features (leaf tips). In some sense, the skeleton gives now a continuous transition from the object's inside to its boundary.

5.3. Subpixel Resolution Skeletons

Since we compute the Voronoi diagrams and skeletons in a per pixel fashion, we can exploit this to visualize these structures with subpixel accuracy. We make use of the result that for any L_k norm, any compact site has a compact and bounded Voronoi region [SM03], as follows. Consider a 4 by 4 pixel area for which we have already computed the FDT at input image resolution (Fig. 8). We call this resolution the *computation* resolution. The closest boundary points $q_1..q_4$ to the pixel centers $p_1..p_4$ are labeled by the ids $U_1..U_4$. For U, we may consider here any of the two signals U^1 and U^2. Given the convex Voronoi region properties mentioned above, the closest boundary point to a point p situated in the square $p_1p_2p_3p_4$ belongs to the set $\{q_1, q_1, q_3, q_4\}$. Given

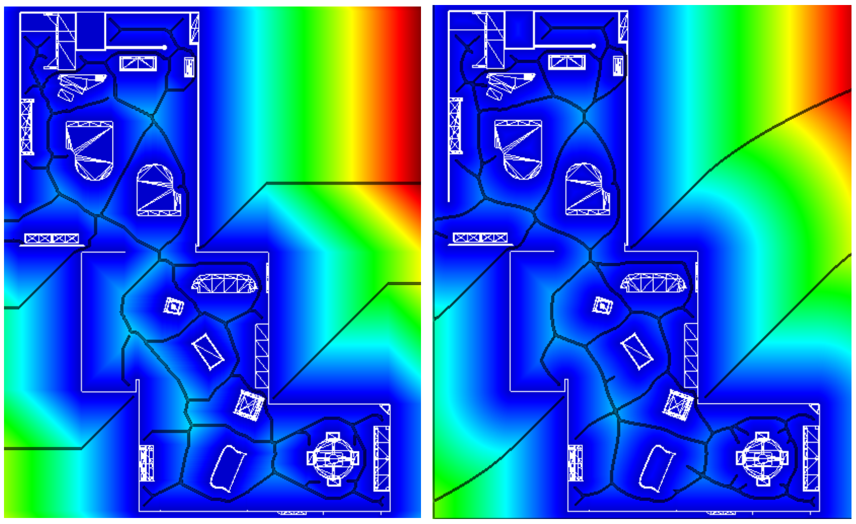

Figure 6: *Euclidean versus Manhattan distances in Voronoi diagrams*

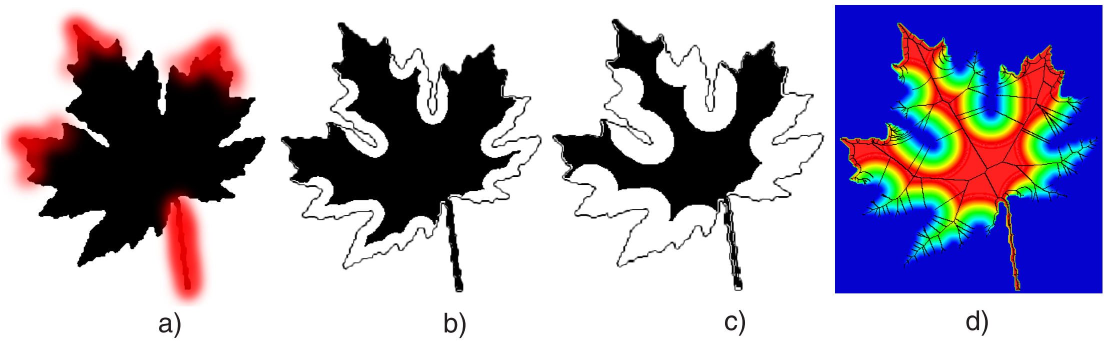

Figure 7: *Feature preserving evolution*

an actual *display* resolution larger than the computation resolution, we explicitly compute for every display pixel the closest boundary point by minimizing the four distances to the computation resolution pixels $p_1..p_4$. This assigns the closest point ids U to the display resolution pixels. Next, we apply the same thresholded derivative (Eqn. 4) and obtain the skeleton at the display resolution.

Figure 9 shows the skeleton on a zoomed region of the image in Fig. 10 b computed with and without the subpixel scheme. In the left image, the gray area shows the computation resolution, the subpixel skeleton being the thin black line in the middle. The subpixel scheme produces skeletons obeying the *same* quality requirements as the original ones. For Euclidean metrics, the subpixel skeletons consist of line segments over the computation pixel area. For other metrics, they represent display pixel accurate approximations of the actual skeletal curves. In all cases, the actual location of the skeleton points is determined up to the user selected resolution.

The four point minimization scheme described above is implemented as a fragment program. The actual boundary point coordinates are determined by building a 256^2 lookup texture that encodes the boundary coordinates for every id U. The two 8 bit components of the U^1 value serve as lookup indices into this texture. Although the complete subpixel skeleton program has over two hundred assembly operations, zooming the skeletons still occurs in real time.

6. Conclusions

We have presented a framework for computing distance transforms, Voronoi diagrams, and skeletons of generalized metrics using graphics hardware. When compared to the most similar software-based method, the AFMM, our hardware method exhibits a number of advantages:

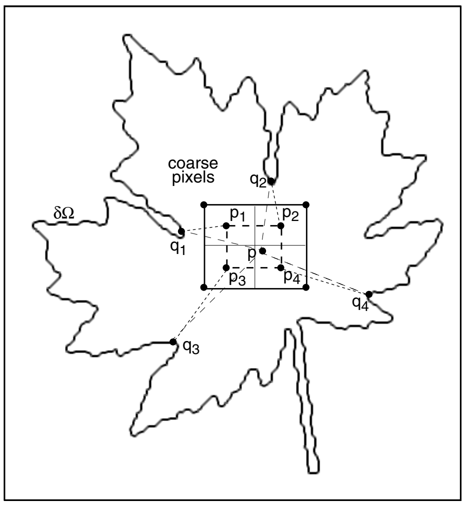

Figure 8: *Subpixel resolution scheme*

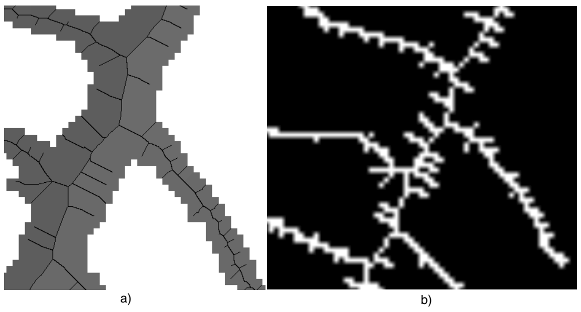

Figure 9: *Subpixel versus pixel resolution skeletons*

- delivers pixel-accurate, correct results for all images and simplification thresholds, which the AFMM did not.
- delivers a performance increase of about one order of magnitude.
- easily supports different metrics and different site weighting.
- delivers skeleton localization with subpixel precision.

The presented method is not a hardware-based reimplementation of the AFMM. As discussed in Sec. 2, the AFMM is essentially a level set evolution, whereas our splatting performs *direct* information propagation from the boundary to its inside. The only direct overlap of the two is the skeletal simplification scheme based on collapsed boundary length. Also, it is interesting to compare our performance with the 3D distance transform in graphics hardware proposed by [SP03]. For an equal amount of rendered voxels (vs pixels in our case) and voxels on the 3D input surface (vs pixels on the 2D input boundary in our case), our method has about 1.8 times less 'pixel throughput'. However, we solve a far more general problem with arbitrary distance functions, which neither allow a-priori bounds on the Voronoi cells nor a direct computation of distances in the fragment programs. Moreover, we compute the skeleton, so we propagate three signals (DT, U^1, U^2) instead of one.

We envisage a number of extensions of the presented method. Interactive visualization and exploration of skeletons for different metrics will give more insight in the similarities and differences thereof. New metrics could be found that allow more effective skeletal shape representation and visualization than the classical Euclidean one. For example, anisotropic distance metrics may open new ways for shape modeling and visualization. Computing k order Voronoi diagrams which record, for every point, the k^{th} closest site, should be easy, by storing k distance texture values instead of one. Finally, we consider extending the method to handle generalized FDTs and robust skeletons in 3D.

References

[Aur91] AURENHAMMER F.: Voronoi diagrams: A survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23 (1991), 345–405. 1

[FLM03] FOSKEY M., LIN M. C., MANOCHA D.: Efficient computation of a simplified medial axis. *Proc. Shape Modeling* (June 2003), 67–76. 2

[Har] HARRIS M. J.: General purpose computation using graphics hardware. http://www.gpgpu.org/. 3

[HCK*99] HOFF K., CULVER T., KEYSER J., LIN M., MANOCHA D.: Fast computation of generalized voronoi diagrams using graphics hardware. *Proc. SIGGRAPH* (1999), 277–286. 2, 3, 7

[LKHW03] LEFOHN A., KNISS J., HANDEN C., WHITAKER R.: Interactive visualization and deformation of level set surfaces using graphics hardware. *Proc. Visualization* (2003), 73–82. 3, 5

[Mau03] MAUCH S.: Efficient algorithms for solving static hamilton-jacobi equations. *Ph.D. Thesis, California Institute of Technology, Purdue, CA* (2003). 2

[PSS*03] PIZER S., SIDDIQI K., SZELEKY G., DAMON J., ZUCKER S.: Multiscale medial loci and their properties. *IJCV 55*, 2-3 (2003), 155–179. 1, 2

[RS01] RUMPF M., STRZODKA R.: Level set segmentation in graphics hardware. In *Proceedings ICIP* (2001), vol. 3, pp. 1103–1106. 3

[RT02] RUMPF M., TELEA A.: A continuous skele-

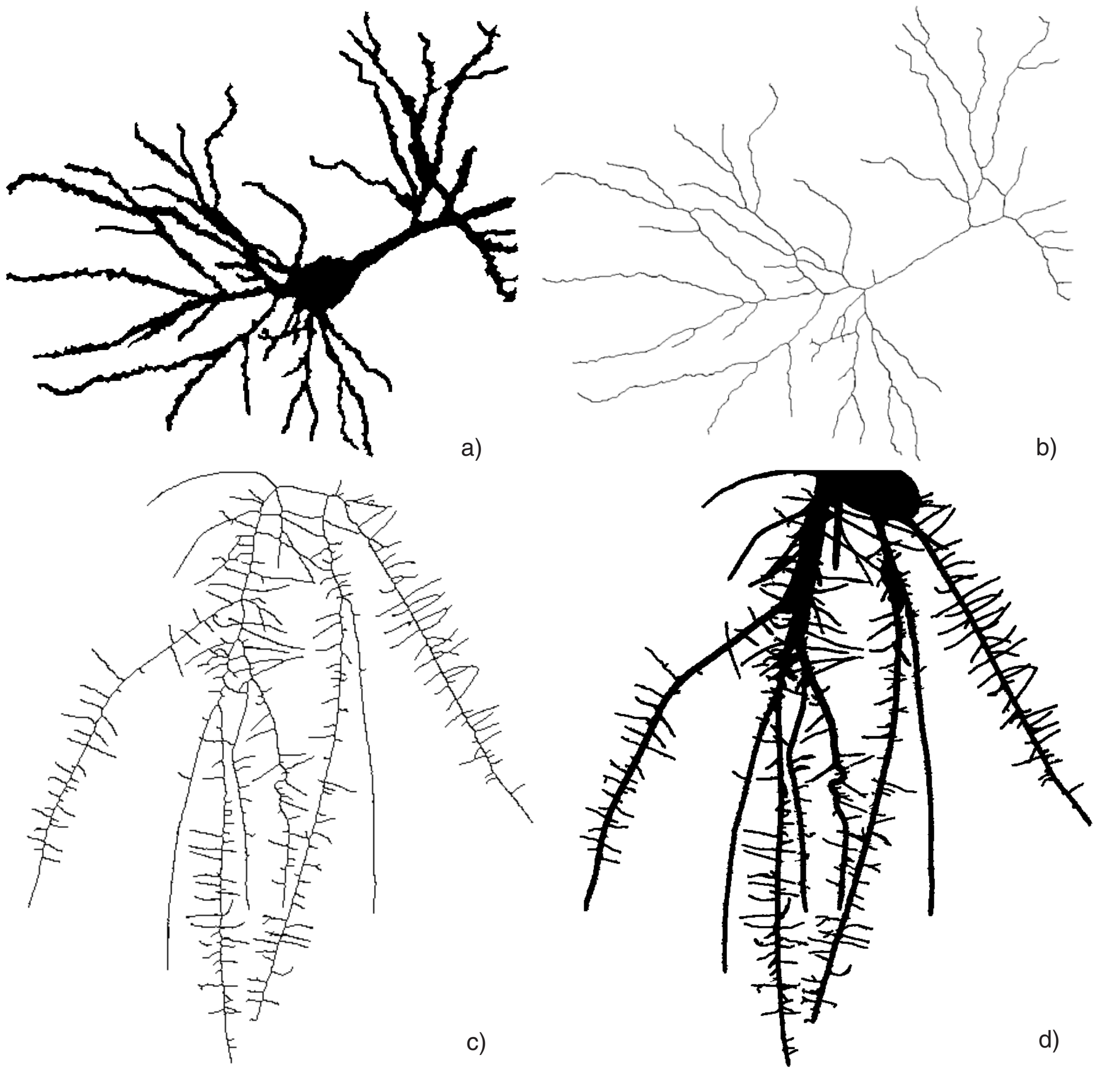

Figure 10: *Hardware-based skeletonization applications. Neuron images (a,b). Rice plant roots (c,d)*

tonization method based on level sets. *Proc. VisSym* (Dec. 2002), 151–158. 2, 3

[SBTZ99] SIDDIQI K., BOUIX S., TANNENBAUM A., ZUCKER S.: The hamilton-jacobi skeleton. *Proc. ICCV* (1999), 828–834. 2, 3

[Set99] SETHIAN J.: *Level Set Methods and Fast Marching Methods*. Cambridge Univ. Press, 1999. 7

[SM03] SUD A., MANOCHA D.: Difi: Fast distance field computation using graphics hardware. *Technical Report* `http://gamma.cs.uns.edu/DiFi` (July 2003). 2, 3, 7

[SP03] SIGG C., PEIKERT R.: Signed distance transform using graphics hardware. *Proc. Visualization* (2003), 83–90. 2, 3, 9

[TVW01] TELEA A., VAN WIJK J. J.: Visualization of generalized voronoi diagrams. *Proc. VisSym* (2001), 253–259. 1

[TVW02] TELEA A., VAN WIJK J. J.: An augmented fast marching method for computing skeletons and centerlines. *Proc. VisSym* (Dec. 2002), 251–260. 1, 2

[YW03] YANG R., WELCH G.: Fast image segmentation and smoothing using commodity graphics hardware. *Journal of Graphics Tools 7*, 4 (2003), 91–100. 3