Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization (2002)
D. Ebert, P. Brunet, I. Navazo (Editors)

An Augmented Fast Marching Method for Computing Skeletons and Centerlines

Alexandru Telea, Jarke J. van Wijk

Department of Mathematics and Computer Science
Eindhoven University of Technology, The Netherlands
alext|vanwijk@win.tue.nl

Abstract

We present a simple and robust method for computing skeletons for arbitrary planar objects and centerlines for 3D objects. We augment the Fast Marching Method (FMM) widely used in level set applications [11] by computing the paramterized boundary location every pixel came from during the boundary evolution. The resulting parameter field is then thresholded to produce the skeleton branches created by boundary features of a given size. The presented algorithm is straightforward to implement, has low memory costs and short execution times, and is robust with respect to the used threshold and initial shape noisiness. The produced skeletons are very similar to the ones delivered by more complex algorithms. Various 2D and 3D applications are presented.

1. Introduction

Skeletons and medial axes are of significant interest in many application areas such as object representation [9], flow visualization [8], path planning, medical visualization [16], computer vision [13, 3], and computer animation. Skeletons provide a simple and compact representation of 2D or 3D shapes that preserves many of the topological and size characteristics of the original. Skeletons can be defined in several ways. One of the first definitions was given by Blum [1, 2] as the locus of the centers of maximal disks contained in the original object. If the skeleton points are attributed with their distances to the original object's boundary, the skeleton can be used to exactly reconstruct the shape it originates from [9].

Skeletons and medial axes can be produced in three main ways. *Morphological thinning* methods iteratively peel off the boundary layer by layer, identifying points whose removal does not affect the object's topology [17]. However relatively straightforward, thinning methods need intricate heuristics to ensure the skeletal connectivity. Moreover, several thinning methods do not produce a true skeleton in the maximal disc sense mentioned above [1, 2]. *Geometric methods* compute the Voronoi diagram of a discrete polyline-like sampling of the boundary. The Voronoi diagram is the boundary's medial axis [9, 10]. Such methods produce an accurate connected skeleton, but are fairly complex to implement, require a robust boundary discretization, and are computationally expensive.

A third class of methods computes the *distance transform* (DT) of the object's boundary. Recent approaches for computing the DT use the robust and simple to implement Fast Marching Method (FMM) introduced by Sethian [11] for evolution of boundaries in normal direction with constant speed (see Fig. 1). The skeleton lies along the singularities (i.e. creases or curvature discontinuities) in the DT. These points coincide with the points where the moving boundary collapses onto itself, as the moving front coincides with the DT's isolines, or level sets [5].

The detection of the singularities of the DT (the evolving front shock points) is however difficult. Direct computation of the singularities is a numerically unstable and delicate process [7, 3, 13], which usually cannot guarantee connected and one-pixel-wide skeletons [17, 8]. Kimmel et al. present a DT-based method [5] which uses the observation that skeleton points are generated by compact boundary segments delimited by curvature maxima along the boundary. However, this method relies on accurate detection of curvature extrema along a possibly noisy boundary. Moreover, shapes with holes need to be treated in a rather complex manner. Siddiqi et al. [13] present another method which simulates the front evolution by tracking marker particles. The shock

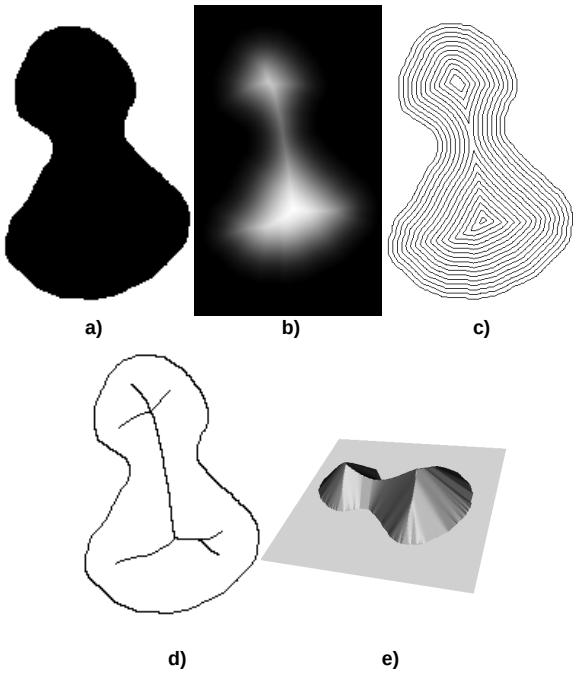


Figure 1: *Object (a), distance transform DT (b), boundary evolution (level sets) (c), skeleton (d), and DT elevation plot (e)*

points defining the skeleton are detected by finding locations where an energy conservation principle is violated. The method is claimed to be more numerically stable than detecting the DT singularities, but is considerably more difficult to implement and computationally more expensive: A fine coverage of the boundary with particles is needed, and particles must be inserted and removed to preserve a constant particle density.

Although attractive from a mathematical point of view, virtually all methods based on singularity detection fail to provide a detailed implementation, performance analysis, and discussion on how to set their various parameters to achieve the desired results. In most cases, such methods are complex to implement, slow, and sensitive to their parameters' choice (e.g. for numerical differentiation or skeleton pruning), thus not directly usable by non experts in the field.

In this paper, we present a new skeletonization algorithm which:

- is simple to implement (the detailed pseudocode is included in this paper);
- produces skeletons that are very similar to the ones produced by several of the methods cited above, according to our tests on the same datasets (see Sec. 4);
- delivers connected skeletons as well as distance data from which the original object can be reconstructed;
- runs in real time on large 2D datasets;
- behaves robustly with respect to noisy boundaries (the skeletons are not affected by spurious branches due to the noise);
- has a single threshold which is simple to set by an unexperienced end user.

Overall, our method can be readily used e.g. by visualization and computer graphics practitioners who need a robust, simple to code technique for producing skeletons, without the need to get involved in the technical details of the process. In Section 2, we present the Fast Marching Method (FMM) that is at the core of our algorithm. Section 3 introduces our augmentation to the FMM for computing skeletons. Section 4 presents several 2D and 3D applications. Finally, Section 6 concludes the paper.

2. Fast Marching Method

The key to our method is to integrate the observation that skeleton points are generated by the collapse of compact boundary segments [5] during the FMM algorithm's front evolution. In this section we describe the FMM algorithm. For a full mathematical description, see [11, 12]. The FMM algorithm calculates a scalar field T by solving the so called Eikonal equation:

$$|\nabla T| = 1 \tag{1}$$

with $T = 0$ on the object's boundary. The field T is a good approximation of the distance to the boundary. The FMM algorithm builds the solution T *outwards* from the smallest known T values. This is done by maintaining a so called *narrow band* of grid points around the evolving front and by marching this narrow band forward, freezing the computed T values of some points and bringing new ones into the narrow band. The FMM is easiest to explain algorithmically, as follows. For every 2D grid point or pixel with coordinates (i, j), we store its DT floating-point value T_{ij} and a flag f_{ij}. The flag may have three values:

- *BAND*: the point belongs to the current position of the moving front or narrow band. Its T value is undergoing update.
- *INSIDE*: the point is inside the moving front. Its T value is not yet known.
- *KNOWN*: the point is behind the moving front. Its T value is already known.

The FMM algorithm has an initialization and a propagation phase, as follows.

2.1. Initialization

T and f are initialized for all grid points by the code in Fig. 2. Here `MAX_VALUE` denotes a value larger than any practically reachable T, e.g. 10^6.

Here `NarrowBand` is a heap container holding all *BAND* points in ascending order of their T value. The C++ STL

```
for all points (i,j)
  if ((i,j) on initial boundary)
  {
    f(i,j)=BAND; T(i,j)=0;
    add (i,j) to NarrowBand;
  }
  else if ((i,j) inside boundary)
  {
    f(i,j)=INSIDE; T(i,j)=MAX_VALUE;
  }
  else /* (i,j) outside boundary */
  {
    f(i,j)=KNOWN; T(i,j)=0;
  }
```

Figure 2: *FMM initialization code*

`multimap` container [6] provides the desired implementation.

2.2. Propagation

After initialization, the FMM propagates the T and f information by iterating the code shown in Fig. 3. Step 1 extracts the first element from the heap, i.e. the point in the band with the smallest T. Step 2 marches the evolving front inwards by adding new points to it. Step 3 computes $T(k,l)$ by solving equation 1 in point (k,l). Finally, step 4 inserts point (k,l) with its recomputed T in `NarrowBand`. This brings new points in the narrow band and reorders existing narrow band points whose T values have been recomputed, to maintain the narrow band sorted on increasing T. Equation 1 is solved by finite difference discretization on a Cartesian grid. Following [11, 12], the discretization of Eqn. 1 yields

$$\max(D^{-x}T, -D^{+x}T, 0)^2 + \max(D^{-y}T, -D^{+y}T, 0)^2 = 1 \quad (2)$$

where $D^{-x}T(i,j) = T(i,j) - T(i-1,j)$ and $D^{+x}T(i,j) = T(i+1,j) - T(i,j)$ and similarly for the y direction. Equation 1 can be directly solved by iteratively solving Eqn 2 at every grid point, which leads to $O(N^2)$ computations for N grid points. We use the more efficient upwind scheme introduced by Sethian in [11]. In detail, for every neighbor (k,l) of the current point (i,j), we solve the restriction of Eqn 2 over each of the four 2D quadrants around (k,l) and retain the solution that produces the smallest T value at (k,l). Since implementing the above is not straightforward to be done from the mathematical description given in [11, 12], we provide the full implementation of the above in the function `solve()` in Fig. 3.

3. Augmented Fast Matching Method

The FMM presented in Sec. 2 computes the distance T from the object's boundary. In this section, we introduce our augmentation to the FMM that produces the skeleton. Figure 4 shows an overview of the whole process, applied to a jagged rectangle (see also color plate).

We exploit the observation that the skeleton points are always generated by compact boundary segments that 'collapse' as the front advances [5, 9, 10]. As pointed by several authors, the importance of a skeleton point is given by the length of the boundary segment that collapsed into that point. The idea behind our method is to determine, for every point in the advancing front, the boundary point it came from. To do this, we augment the FMM algorithm by one real value U per grid point. Initially, we set U to zero in an arbitrary boundary point. Starting from the $U = 0$ pixel, we assign a monothonically increasing U to every boundary pixel, equal to the distance, along the boundary, from that pixel to the $U = 0$ pixel. U is thus a boundary parameterization with the property that the distance between any two boundary points, measured along the boundary, is equal to the points' U difference. An exception to this are the points from which U starts being propagated on each connected boundary segment, e.g. the point with $U = 1$ in Fig. 5 b and the points $U = 1$ and $U = 33$ in Fig. 5 d. The treatment of these points is described in Sec. 3.2.

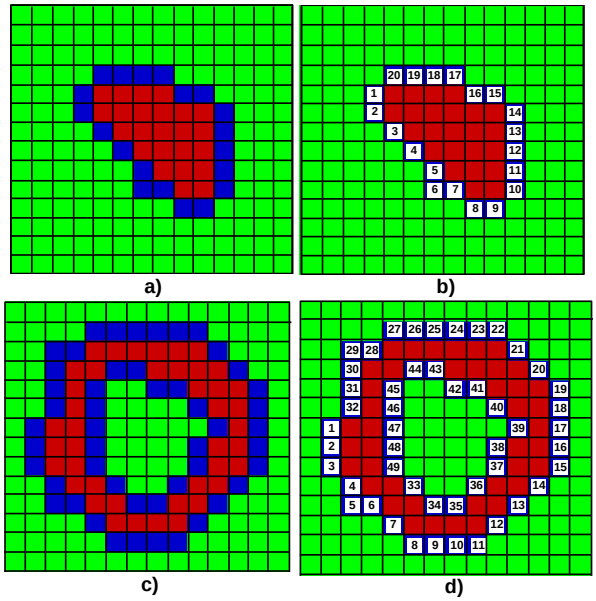


Figure 5: *Objects (a,c) and the order in which U is assigned to their boundaries (b,d)*

After initialization, U is propagated along with T (Fig. 4 c). To do this, we add a few statements to step 2 of the propagation code (Sec. 2.2), as shown in Fig. 6 a. The propagation of U marks every pixel inside the initial boundary with the U value of the boundary point that arrived at that location, due to the front evolution. U values are interpolated, via averaging, to account for boundary points located between the initial boundary pixels. Averaging takes place on concave boundary segments which expand (get longer)

```
while (NarrowBand not empty)
{
  P(i,j)=head(NarrowBand);                /* STEP 1 */
  remove P from NarrowBand;
  f(i,j)=KNOWN;
  for point (k,l) in {(i-1,j),(i,j-1),(i+1,j),(i,j+1)}
  if (f(k,l)!=KNOWN)
  {
    if (f(k,l)==INSIDE) f(k,l)=BAND;      /* STEP 2 */
    sol=MAX_VALUE;                        /* STEP 3 */
    solve(k-1,l,k,l-1,sol);
    solve(k+1,l,k,l-1,sol);
    solve(k-1,l,k,l+1,sol);
    solve(k+1,l,k,l+1,sol);
    T(k,l)=sol;
    insert (k,l) in NarrowBand;           /* STEP 4 */
  }
}

solve(int i1,int j1,int i2,int j2,float& sol)
{
 float r,s;
 if (f(i1,j1)==KNOWN)
   if (f(i2,j2)==KNOWN)
   {
     r = sqrt((2-(T(i1,j1)-T(i2,j2))*(T(i1,j1)-T(i2,j2)));
     s = (T(i1,j1)+T(i2,j2)-r)/2;
     if (s>=T(i1,j1) && s>=T(i2,j2)) sol=min(sol,s);
     else
     {
       s += r;
       if (s>=T(i1,j1) && s>=T(i2,j2)) sol=min(sol,s);
     }
   }
   else sol=min(sol,1+T(i1,j1));
 else if (f(i2,j2)==KNOWN) sol=min(sol,1+T(i1,j1));
}
```

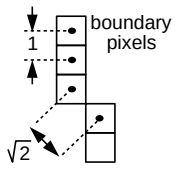
Figure 3: *FMM iteration code*

due to front evolution. However, if the U values around the current point differ by more than 2, this means the boundary points they came from were *not* neighbors, since the maximal distance between two boundary neighbor points is $\sqrt{2}$, realized in case of diagonally connected pixels (Fig. 6 b). The above happens along convex boundary segments that shrink (collapse) during front evolution. In this case, we have detected a skeleton point, so no averaging of U is done – we simply propagate further the U of one of the current point's neighors. The augmented FMM is illustrated in Fig. 7 where we compute the skeleton of a rectangle. The U values on the first three fronts evolved from the boundary are shown in Fig. 7 b. Parallel to the rectangle's edges, where no skeleton branch exists, U behaves as on the boundary, i.e. the U difference between neighbor points is 1. However, along the skeleton's branches, this difference increases as we get further from the boundary, as the 3D plot of U in Fig. 7 c shows.

Once the U field is computed, we find the skeleton points by detecting its *sharp* discontinuities (Fig. 4 d,e). We retain all points where U differs from the neighboring U's by more than a given threshold. This is roughly analogous to computing a thresholded derivative of the U field. However, an essential advantage of our method as compared to most derivative-based methods is that we do *not* need a

```
... original FMM code ...
if (f(k,l)==INSIDE)                  /* STEP 2 */
{
  f(k,l)=BAND;
  a = average of U over KNOWN neighbors of (k,l);
  m = min of U over KNOWN neighbors of (k,l);
  M = max of U over KNOWN neighbors of (k,l);
  if (M-m<2) U(k,l) = a; else U(k,l) = U(i,j);
}
... original FMM code ...
```

a)



b)

Figure 6: *Augmented FMM code (a) Boundary neighbor distances (b)*

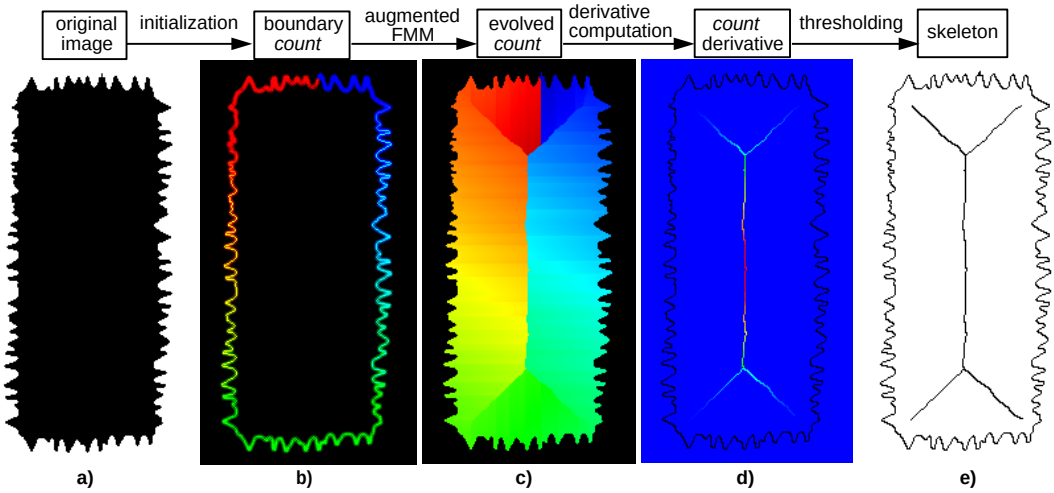


Figure 4: *Overview of the skeletionzation algorithm*

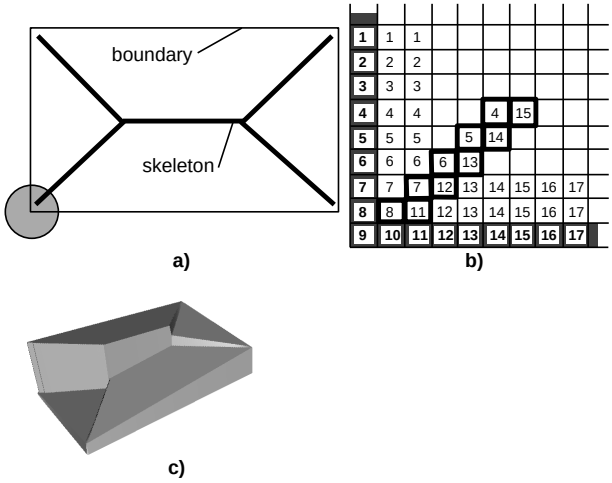


Figure 7: *Rectangle and its skeleton (a). Detail of the augmented FMM result around the rectangle's lower left corner (b). Complete U field (c)*

mathematically accurate differentiation. The discontinuities of U are strong enough so that the simple scheme outlined above is sufficient. Moreover, we obtain connected skeleton branches. Indeed, once the U values U_1 and U_2 of two neighbor points p_1 and p_2 differ by more than some threshold v, there will always be another neighbor q of p_1 or p_2 whose U difference with respect to its own neighbors will exceed v. Intuitively speaking, the reason for the above statement lies in the order in which the points are visited to compute the continuous function T which satisfies everywhere the condition $|\nabla T| = 1$.

3.1. Threshold Choice

As mentioned previously, the U difference between neighbor points increases as one goes further from the boundary. The points closer to the skeleton's main branches, and further from its tips, have larger U differences (see Fig. 4 d). To retain a given fraction of the skeleton's stem, we threshold the U difference field by a given fraction of its maximum. Alternatively, to prune all skeleton branches caused by boundary details shorter than t pixels, we set the threshold to t (Fig. 8). The pruning threshold has thus a precise geometrical meaning, which makes it simple to set and interpret by non experienced users. We have tested and validated the accuracy of the above threshold based pruning for figures where an analytic skeleton was known, such as rectangles. For all our test cases (see also Fig. 13), setting the threshold to some value between 20 and 40 pixels has given good results. For objects with noisy boundaries (Fig. 4 and 13 j,n), we further increased the threshold to around 100 pixels to prune the small skeleton branches created by the boundary jaggies.

3.2. Special Boundary Points

As mentioned in Section 3, the property that U increases monotonically along the boundary's compact segments does not hold for the points we start the parametrization from. For the object in Fig. 9 a, there are three such points, corresponding to its three disjoint boundary segments. Figure 9 b shows the initial U values, as well as the numbering order, using a rainbow colormap (blue denotes low values, red denotes high values, see color plate). The obtained U field has thus three false discontinuities corresponding to the three points mentioned (Fig. 9 c). The skeleton (Fig. 9 d) get thus three false branches, i.e. the vertical lines that connect it to the boundary.

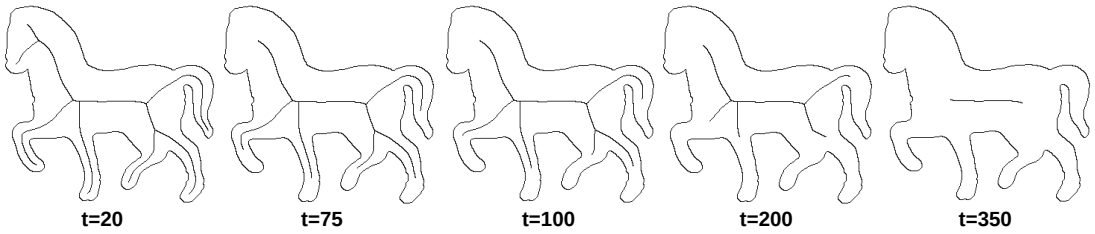


Figure 8: *Feature-based skeleton pruning for different thresholds t*

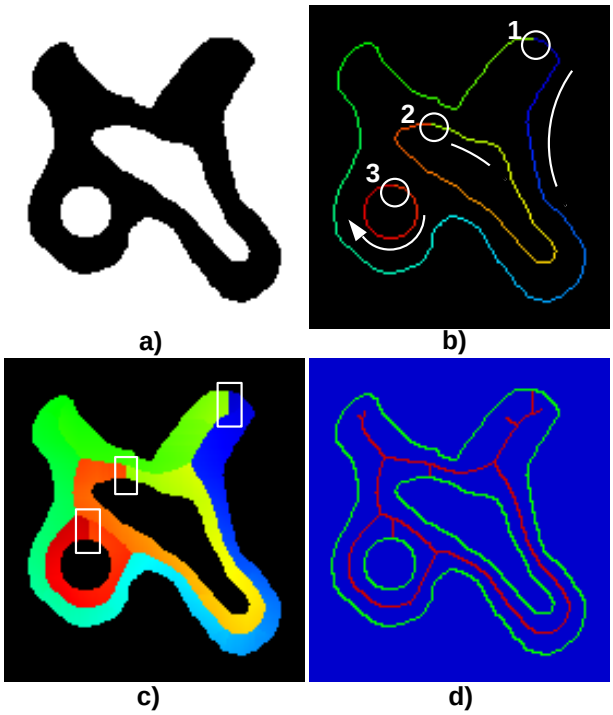


Figure 9: *Treatment of special boundary points: original object (a), initial U (b), computed U (c), and skeleton (d)*

Removing these false branches is easy. We execute the whole augmented FMM algorithm twice, starting the U initialization from different boundary locations, such as the boundary pixels with the smallest, respectively largest y coordinate. Next, we intersect the resulting skeletons to remove the false branches. This technique produced the correct skeletons in all the cases we have tested.

4. Applications

4.1. Skeletons

Figure 13 shows several applications of our method. For comparison purposes, the input shapes a-h and j-n are taken from [3, 13], respectively from [9]. Our skeletons are visually identical to the ones presented by the cited authors. The input images range between 250^2 and 600^2 pixels. The complexity of the FMM algorithm is roughly $O(NlogN)$ for N pixels. The skeletons are computed in subsecond time on a Pentium II 500 MHz machine.

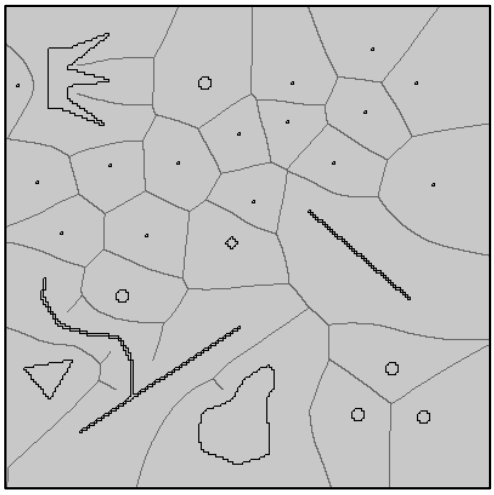


Figure 10: *Voronoi diagrams (grey) of arbitrary 2D shapes (black)*

4.2. Voronoi Diagrams

A second application of the presented method is the computation of Voronoi diagrams of arbitrary planar objects. This is done simply by computing the skeleton of the outer region defined by the objects' boundaries, which coincides with the Voronoi diagram for the boundaries (see example in Fig. 10). In contrast with other techniques for computing Voronoi diagrams [15, 4], our method handles, by definition, arbitrary 2D objects and is, again, simple to implement. In this respect, the technique presented here is more similar to the pixel-based Voronoi diagram computation method presented by us in [14]. However the current method handles arbitrary boundary shapes, we believe it would be difficult to extend it to handle other distance functions besides the Euclidean distance.

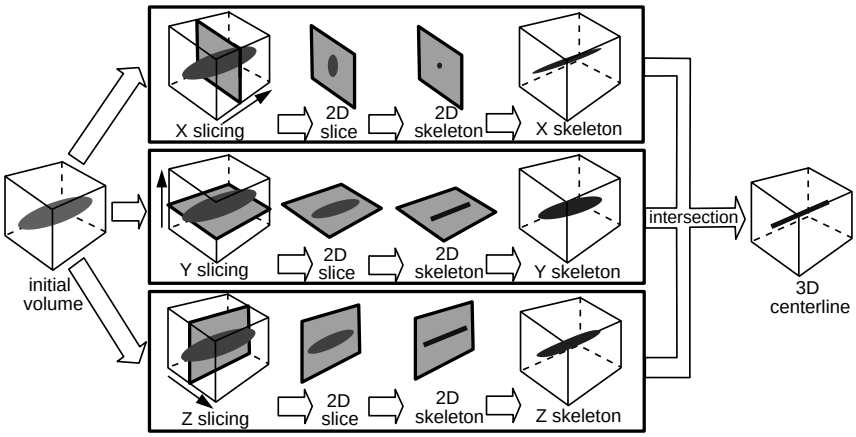


Figure 11: *Centerline computation pipeline*

4.3. 3D Centerlines

A much desired extension of the presented skeletonization method would be to generalize it to the 3D case. The generalization of the augmented FMM algorithm to 3D is indeed trivial. Unfortunately, our method relies on an initialization of the field U on the boundary such that the difference between the U values of two points is a measure for the distance between them on the boundary. The 2D boundary parametrization (Sec. 3) is, however, not straightforward to generalize to 3D. The main reason behind the above is the difference between the topological and ordering properties of 2D curves and 3D surfaces.

However, we have exploited the 2D skeletonization method for the computation of 3D *centerlines*. Loosely defined, centerlines are curves (and not surfaces as the true 3D skeletons). A point belongs to the centerline if there exists a sphere, centered in this point, that touches the boundary at opposite sides (as compared to any two points, in the case of skeletons). Applications of centerlines include 3D path planning and navigation [16], object recognition, and object simplification [8].

We compute 3D centerlines as follows (see Fig. 11). First, we extract the 2D skeletons of each axis-parallel 2D slice of the 3D volume dataset, using the already presented 2D skeletonization method. This produces three sequences of 2D skeletons, or three volumes, corresponding to the three slicing directions, called the X, Y, and Z skeletons. Next, we intersect these volumes, voxel by voxel, and obtain the 3D centerline of the initial object. The idea behind the above approach is that the voxels in the intersection belong, locally, to three axis-aligned 2D skeletons. These voxels are thus at maximal distance from the boundary, measured in the three orthogonal slice planes. This is a simplification of the general case where one would have to measure the distance to the boundary along *any* arbitrary slicing plane – we use three orthogonal circles instead of using a sphere. However, the above is often a sufficient approximation for the typical snake-like objects for which centerlines are usually computed. A drawback related to the above observation is that, whereas the 2D skeletons are connected, the 3D centerlines are not guaranteed to be so, especially if the initial volume dataset has a low resolution.

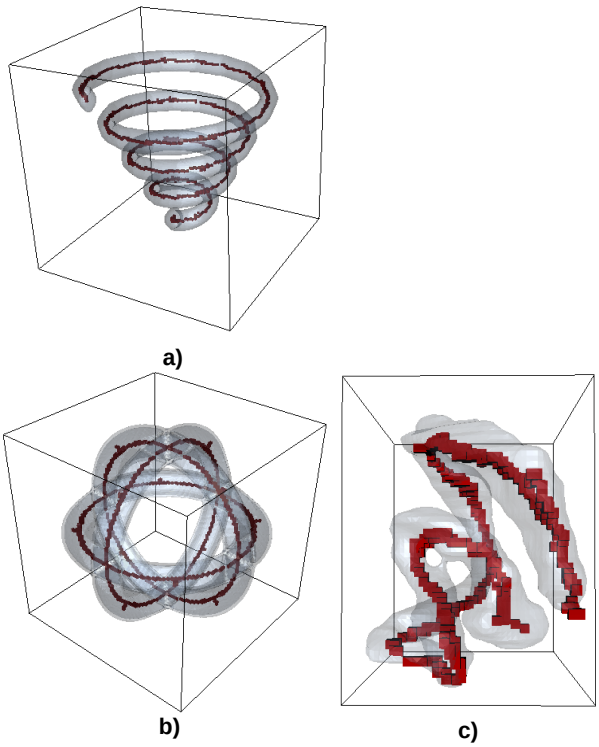


Figure 12: *Examples of 3D centerlines*

The 3D centerline computation is still an $O(NlogN)$ process for N voxels. Practically, processing of a 512^3 dataset takes about 10 seconds on a Pentium III 800 MHz machine. Figure 12 shows several examples of 3D centerlines. Figure 12 a shows the centerline of a 3D spiral object (512^3 voxels). Figure 12 b shows the centerline of a more complex 3D structure, showing that centerlines having a more complex topology (branching points) are correctly detected. Finally, Fig. 12 c shows the centerline of a frog duodenum, extracted from the VTK frog dataset [18] (resolution is about 60x70x90 voxels).

It is important to stress here that the centerline computation is not the main focus of the method presented here but only one of its possible applications. More specialized centerline construction methods exist, such as the one presented by Wan et al in [16]. Both Wan's and our methods are based on the same idea, namely performing a 'thinning' or inwards marching of the boundary and determinimg, for every dataset voxel, the boundary voxel it came from. However, whereas we employ Sethian's fast marching method for the boundary evolution, Wan et al employ a graph-based approach and graph-based distance definition. Since the two definitions of the distance from the boundary are different the computed centerlines may be different too. In terms of timing, our algorithm is $O(NlogN)$ for N voxels which is, we understand, the same as the complexity of the algorithm of Wan et al. The concrete timings are difficult to compare, since the timings of [16] exclude the computation of the distance transform. Finally, we believe that our implementation (presented in detail in this paper) is simpler as compared to the one of the algorithm of Wan et al.

5. Acknowledgements

For the original idea of using the FMM to compute skeletons, as well as for subsequent inspiring discussions on the various applications and properties of level set methods, we are endebted to prof. Martin Rumpf from the Department of Mathematics, Duisburg University, Germany.

6. Discussion and Future Work

The presented 2D skeletonization algorithm has several advantages as compared to other existing methods. First and foremost, it is simpler to implement than most similar methods we are aware of – our complete C++ implementation of about 500 lines of code is available at *http://www.win.tue.nl/∼alext/SKELETON*. Secondly, it runs quickly and reliably on large 2D datasets. Thirdly, there is a single threshold, expressing the length of boundary features creating skeleton branches, which is intuitive even for non experienced end users. Finally, the produced skeletons are very similar to the ones delivered by more complex methods.

The main next challenge is to find a generalization of the algorithm for the 3D case. For this, we would need a method to initialize the U field over the boundary, which would immediately allow the computation of 3D skeletons and 3D Voronoi diagrams of arbitrary objects. Separately, the 3D centerline method could be considerably accelerated by using an adaptive version of the FMM algorithm, as described in detail in [11].

References

1. H. BLUM, *A transformation for extracting new descriptors of shape*, In W. Walthen-Dunn, editor, *Models for the Perception of Speech and Visual Form*, MIT Press, 1967.

2. H. BLUM, R. N. NAGEL, *Shape description using weighted symmetric axis features*, Pattern Recognition, nr. 10, 1978, pp. 167-180.

3. S. BOUIX, K. SIDDIQI, *Divergence-based Medial Surfaces*, Proc. ECCV 2000, pp. 603-618, 2000.

4. K. HOFF, T. CULVER, J. KEYSER, M, LIN, D. MANOCHA, *Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware*, Proc. SIGGRAPH '99, ACM Press, 1999, pp. 277-286.

5. R. KIMMEL, D. SHAKED, N. KIRYATI, A. M. BRUCKSTEIN, *Skeletonization vis Distance Maps and Level Sets*, Computer Vision and Image Understanding, vol. 62, no. 3, pp. 382-391, 1995.

6. D. R. MUSSER, A. SAINI, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, Addison-Wesley Professional Computing Series, 1996.

7. C.W. NIBLACK, P.B. GIBBONS, D.W. CAPSON, *Generating skeletons and centerlines from the distance transform*, CVGIP: Graphical Models and Image Processing, nr. 54, 1992, pp. 420-437.

8. F. REINDERS, M. E. D. JACOBSON, F. H. POST, *Skeleton Graph Generation for Feature Shape Description*, Proc. IEEE VisSym 2000, Springer, 2000, pp. 73-82.

9. R. L. OGNIEWICZ, *Automatic Medial Axis Pruning by Mapping Characteristics of Boundaries Evolving under the Euclidean Geometric Heat Flow onto Voronoi Skeletons*, Harvard Robotics Laboratory, Technical Report 95-4, 1995.

10. R. L. OGNIEWICZ, O. KUBLER, *Hierarchic Voronoi Skeletons*, Pattern Recognition, nr. 28, 1995, pp. 343-359.

11. J. A. SETHIAN, *A Fast Marching Level Set Method for Monotonically Advancing Fronts*, Proc. Nat. Acad. Sci. vol. 93, nr. 4, pp. 1591-1595, 1996.

12. J. A. SETHIAN, *Level Set Methods and Fast Marching Methods*, Cambridge University Press, 2nd edition, 1999.

13. K. SIDDIQI, S. BOUIX, A. TANNENBAUM, S. W. ZUCKER, *The Hamilton-Jacobi Skeleton*, Intl. Conf. on Computer Vision ICCV '99, p. 828-834, 1999.

14. A. TELEA, J. J. VAN WIJK, *Visualization of Generalized Voronoi Diagrams*, Proc. IEEE VisSym '01, p. 165-174, Springer, 2001.

15. J. VLEUGELS, M. OVERMARS, *Approximating Generalized Voronoi Diagrams in Any Dimension*, Technical Report UU-CS-1995-14, Utrecht University, 1995.

16. M. WAN, F. DACHILLE, A. KAUFMAN, *Distance-Field Based Skeletons for Virtual Navigation*, Proc. IEEE Visualization '01, pp. 239–246, IEEE CS Press, 2001.

17. Y. ZHOU, A. W. TOGA, *Efficient Skeletonization of Volumetric Objects*, IEEE TVCG, vol. 5, no. 3, 1999, pp. 210-225.

18. *VTK Homepage*, `http://www/kitware.com`.