Accurate Computation of Geodesic Distance Fields for Polygonal Curves on Triangle Meshes

David Bommes, Leif Kobbelt

Computer Graphics Group, RWTH Aachen

Email: {bommes,kobbelt}@cs.rwth-aachen.de

Abstract

We present an algorithm for the efficient and accurate computation of geodesic distance fields on triangle meshes. We generalize the algorithm originally proposed by Surazhsky et al. [1]. While the original algorithm is able to compute geodesic distances to isolated points on the mesh only, our generalization can handle arbitrary, possibly open, polygons on the mesh to define the zero set of the distance field. Our extensions integrate naturally into the base algorithm and consequently maintain all its nice properties.

For most geometry processing algorithms, the exact geodesic distance information is sampled at the mesh vertices and the resulting piecewise linear interpolant is used as an approximation to the true distance field. The quality of this approximation strongly depends on the structure of the mesh and the location of the medial axis of the distance field. Hence our second contribution is a simple adaptive refinement scheme, which inserts new vertices at critical locations on the mesh such that the final piecewise linear interpolant is guaranteed to be a faithful approximation to the true geodesic distance field.

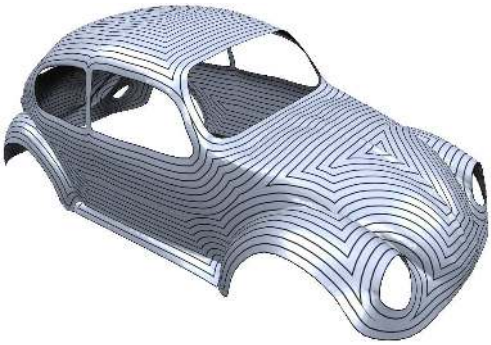


Figure 1: The isolines of the geodesic distance field with respect to the boundaries of the car model are visualized.

1 Introduction

The computation of geodesic distances on a triangle mesh has many applications in geometry processing, ranging from segmentation and low distortion parametrization to motion planning and tool path optimization. In most cases the true geodesic distance field is approximated by some fast marching method which leads to acceptable results on nicely structured meshes and away from singularities of the distance function. However, such simple propagation schemes tend to become numerically unstable on not-so-nice meshes as they often occur in practical applications. Moreover, since they use the same mesh as a representation for the input geometry as well as the distance field, the precision is limited by the mesh resolution. Surazhsky et al. [1] present a practical implementation of the geodesic distance algorithm of Mitchell et al. [2]. This was the first time that an exact geodesic distance computation has become applicable to arbitrary input meshes of practically relevant complexities. However, in this algorithm, the distance computation is initialized by one or more isolated points on the mesh and the distance is propagated from them (in Section 3 we present a summary of this algorithm). Unfortunately, for many practical applications this is too restricted. In general one would like to be able to compute the geodesic distance with respect to a curve on the surface, i.e., a polygon on the mesh since this allows us to take arbitrary boundary conditions into account. See Fig.1 for an example.

2 Previous work

In this paper we address two elementary mesh operations, geodesic distance computation and adaptive refinement.

Dijkstra's algorithm for computing shortest paths along edges can be used as an approximation for the geodesic field. Lanthier et al. [3] improved the initial poor results by adding many extra edges to the mesh.
Kimmel and Sethian [4] adapted the fast marching method to compute closer approximations of geodesic distances. On well-shaped input meshes this method performs very good, but in the case of obtuse angles or needle triangles even improved update rules and special handling as proposed before [5, 6, 7] can lead to large errors. Fast marching algorithms are able to approximate the geodesic distance field induced by polygonal sources, but the quality strongly depends on the mesh structure near the medial axis, which is typically not suited to represent the geodesic field, as we show in this paper.

The most famous exact algorithm was developed by Mitchell et al. [2] in 1987 and the first practical implementation was proposed eighteen years later by Surazhsky et al. [1]. They showed that the worst case complexity of $O(n^2 \log n)$ is quite pessimistic and in practice the average is close to $O(n^{1.5})$ which makes the algorithm practical for common model complexities. Details of this algorithm are presented in the next section. They also introduce a merging operation to design an approximation algorithm with guaranteed error bounds. Liu et al. [8] studied a robust implementation strategy which handles all degenerate cases that occur in real world data. In this paper we will generalize this exact and approximate algorithm from point sources to arbitrary source polygons.

In the context of adaptive mesh refinement two common techniques, namely red-green-triangulations [10] [11] and $\sqrt{3}$-subdivision, lead to regular structures and preserve the triangle quality. $\sqrt{3}$-subdivision is composed of one-to-three triangle splits and edge flips which changes not only the tessellation but also the geometry (and hence the geodesic distance field) of a triangle mesh and is thus not suited for our application.

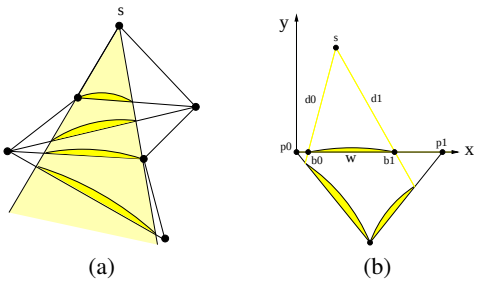


Figure 2: (a) Starting on the point source $\mathbf{s}$ a (shaded) pencil of rays is propagated through three unfolded triangles along of straight lines. Each window is highlighted by an arc which is always on the edge side pointing to the source. (b) An edge aligned two dimensional coordinate system is used to compute new windows which are induced from window w.

At the core of red-green-triangulations is the one-to-four triangle split. Red and green tags are used to preserve consistency. The refinement conserves the original geometry and could be integrated into our framework. However for our application a simpler and more local refinement is possible which we propose in section 6.

3 The exact geodesic algorithm

Since our algorithm is an extension of [1] we briefly explain the basic principles and the resulting base algorithm.

In the plane, the geodesic distance coincides with the Euclidean distance. Hence, with respect to an isolated point, it is the square root of a quadratic function. On a triangle mesh, i.e. on a piecewise planar surface, the geodesic distance with respect to a point turns out to be a piecewise function where in each segment the distance is given by the square root of a quadratic function plus an optional constant offset. This offset has to be introduced to properly handle saddle points on the surface.

The central idea of the algorithm [1] is to propagate exact distance information from one triangle to its neighbors with a Dijkstra-type algorithm. The key observation is that it is sufficient to store the piecewise distance function on the edges of the tri-

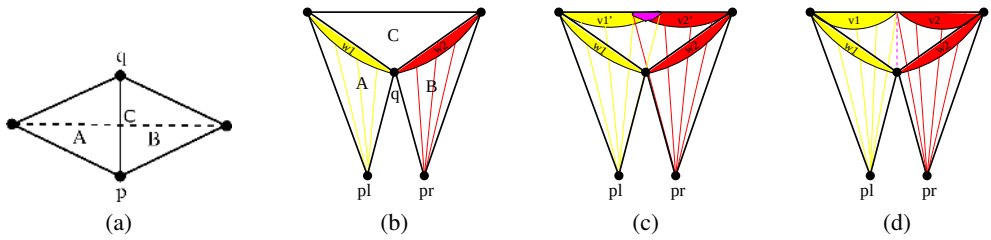


Figure 3: (a) The geodesic distance field w.r.t point $\mathbf{p}$ is computed on a cap consisting of triangles A,B, and C. (b) Cutting along the edge $\overline{pq}$ unfolds the cap isometrically and enables the distance propagation in the plane through windows w_1 and w_2. (c) The temporarily propagated windows v_1' and v_2' overlap in the middle region. (d) The final windows v_1 and v_2 are properly cut to represent the piecewise geodesic distance along the edge.

angle mesh since this is sufficient for the propagation and also for the exact evaluation of distances everywhere on the surface.

For each edge of the mesh the algorithm maintains a list of segments, so-called *windows*. Each window defines the geodesic distance field within a pencil of rays covering both neighboring triangles (see Figure 2). When distance information is propagated across a triangle, the (incoming) windows have to be mapped to the opposite side. The propagation includes the proper intersection of windows, because unlike the planar case on a surface propagated windows can overlap. Since the distance function is continuous, the intersection requires to find the point where the distance function values in both windows are identical.

We illustrate the procedure with a simple example. The cap of Figure 3 (a) consists of three isosceles triangles A,B and C. Now we want to compute the geodesic distance field w.r.t the point $\mathbf{p}$. Since $\mathbf{p}$ is coplanar with the triangles A and B they are covered with a pencil of rays emanating from $\mathbf{p}$ through single *windows* w_1 and w_2. To propagate the distance information through w_1 and w_2 we cut the cap along the edge connecting $\mathbf{p}$ and $\mathbf{q}$ and unfold the triangles isometrically into the plane, i.e. all edge lengths and angles of the triangles are preserved (see 3 (b)). In this setting $\mathbf{p}$ is doubled into $\mathbf{p}_l$ and $\mathbf{p}_r$. Now we are ready to propagate the pencil of rays defined by w_1 and w_2 across triangle C and create new temporarily overlapping windows v_1' and v_2' depicted in Figure 3 (c). Evaluating the distances induced by both pencils of rays the windows can be intersected and properly cut to final windows v_1 and v_2 (see Figure 3 (d)) which correctly represent the continuous piecewise geodesic distance function along the edge.

A nice feature of this *window* formulation is that all computations can be formulated in local two dimensional coordinates, i.e. only the mesh topology and scalar edge lengths are required. The necessary condition for this edge based algorithm is that geodesic paths can only pass through vertices with a total angle greater or equal than 2π, i.e. saddles and flat points. This result was proven by Mitchell et al. in [2]. Saddle points and concave boundary points act as pseudo sources which generate additional new windows covering the geometric shadow of the locally expanded surface.

3.1 Base algorithm

At first all source windows in the immediate vicinity of source points are created and pushed into a priority queue preferring shorter distances, because we want to compute the minimal geodesic distance. Notice that in general the result is independent of the propagation order but the priority queue ensures that windows are propagated as a wavefront which gives a strong speedup and makes the algorithm practical. Working off the queue the current window is always propagated into the next unfolded triangle, where new windows are created (see Figure 2). When the front reaches saddle or boundary vertices new source windows are added. All new windows can overlap with already existing windows and must be intersected accordingly. The algorithm terminates when all edges are parti-

tioned by the minimal geodesic distance windows, i.e. when the queue is empty. The pseudocode algorithm is presented below and all necessary computations are explained in more detail in the next sections.

Algorithm 1 Exact Geodesic Field

```
sourceWs = createSourceWindows()
PQueue.add( sourceWs )
repeat
  curW   = PQueue.popFront()
  newWs  = propagate( curW )
  newWs += saddleAndBoundaryWs( curW )
  newWs  = intersect( newWs, oldWs )
  PQueue.add( newWs )
until queue.empty()
```

3.2 Circular window propagation

In the next section we will define a second type of windows, so from now on windows originating from point sources are called *circular windows*. The starting point for a propagation is depicted in Figure 2 (b). Given a window w the corresponding edge $\overline{p_0 p_1}$ is aligned to the x-axis of the local coordinate system with the origin in $\mathbf{p}_0$. Each window is described by a six tuple $(b_0, b_1, d_0, d_1, \sigma, \tau)$ with σ representing the optional constant offset between a pseudo source and a real source. The binary flag τ determines on which side of the x-axis the unfolded pseudo source $\mathbf{s}$ lies (symbolized in pictures by the arc). The window extents are encoded in b_0 and b_1 which are in the range $[0..|\overline{p_0 p_1}|]$. Due to the fact that the distances d_0 and d_1 of the window endpoints from the pseudo source are known the unfolded position $\mathbf{s}$ can be reconstructed via circle intersection.

$$\begin{aligned} s_x &= \frac{1}{2}(b_0 + b_1 + \frac{d_0^2 - d_1^2}{b_1 - b_0}) \\ s_y &= -1^{\tau}\sqrt{d_0^2 - (c_x - b_0)^2} \end{aligned}$$

Using the local coordinates of $\mathbf{p}_3$ which are computed analogously to $\mathbf{s}$ the new windows are found by 2D ray intersection. There are different constellations which can lead to one, two or three (on saddle points) new windows.

3.3 Circular window intersection

If two windows overlap and one provides a smaller distance everywhere the other is simply clipped against it. If both are minimal in part of the overlapping interval, both ranges are clipped to the point where both distance functions are equal. Notice that clipped windows have to be reinserted into the queue because their priority can change. Using the unfolded pseudo source $\mathbf{s}$ from the previous section the distance function d_c of an arbitrary point $(p_x, 0)$ in the interval of a window can easily be formulated. Due to the fact that $\mathbf{s}$ is not necessarily a real source (e.g. saddles induce pseudo sources) the distance σ from all traversed pseudo sources to the real source must be added.

$$d_c(p_x) = \sqrt{(p_x - s_x)^2 + s_y^2} + \sigma \qquad (1)$$

Trying to find the intersection of two such distance functions, namely $d_{c1}(p_x) = d_{c2}(p_x)$ the computation ends up as the solution of a quadratic equation $Ap_x^2 + Bp_x + C = 0$. In this case there is exactly one solution in the overlapping interval and the coefficients of the polynomial are

$$\begin{aligned} A &= \alpha^2 - \beta^2 \\ B &= \gamma\alpha + 2s_{1x}\beta^2 \\ C &= \frac{1}{4}\gamma^2 - |s_1|^2\beta^2 \\ \alpha &= s_{1x} - s_{0x} \\ \beta &= \sigma_1 - \sigma_0 \\ \gamma &= |s_0|^2 - |s_1|^2 - \beta^2 \end{aligned}$$

4 Generalization to arbitrary sources

Our goal is to generalize the original geodesic distance computation algorithm from isolated points to polygonal curves on the surface. In a planar configuration the Euclidean distance function to a polygonal curve falls into several segments. In some segments the distance function is, again, the square root of a quadratic function. Those segments correspond to the vertices of the polygon. In other segments, the distance function is just linear. These segments correspond to the edges of the polygon. See Figure 1 for an example.

Going from the plane to a piecewise planar triangle mesh, we can still propagate the distance function from one triangle to its neighbors by storing

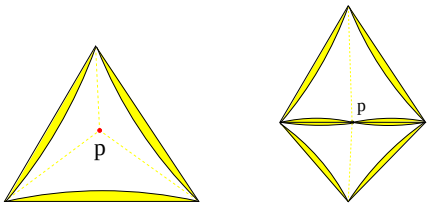


Figure 4: (a) An arbitrary point source p on the surface induces three windows in the corresponding triangle. (b) The six windows of a point source on an edge.

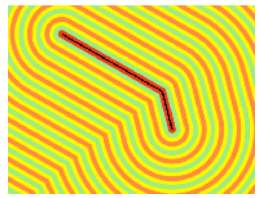


Figure 5: The geodesic distance field w.r.t the black polygon. Linear waves emanate orthogonal to line segments and circular waves emanate from each endpoint of a line segment.

windows of the piecewise distance function on each edge. The only difference regarding the last section is that now we need to handle two different types of windows: the ones where the distance function is of the form (1) and the ones where the distance function is linear. The Dijkstra-type propagation algorithm then has to handle all kinds of window intersections: circular-circular, circular-linear, and linear-linear. In the following we will give the explicit formulae for the corresponding intersection points where the two distance functions coincide. Additionally we need the ability to create circular source windows induced by arbitrary points on the surface which will be discussed first.

4.1 Arbitrary points

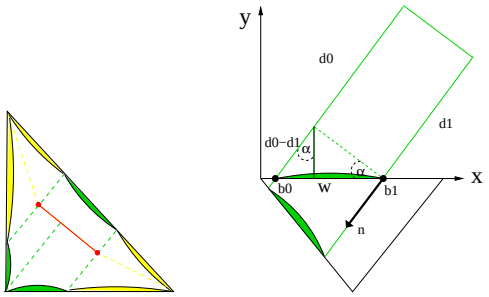
The original algorithm [1] was proposed to allow point sources only at vertex positions. However it is straightforward to overcome this limitation. Given an arbitrary point p on the surface the three edges of its containing triangle are initialized with windows emanating from this point as depicted in Figure 4. The new created windows are intersected with all other windows on an edge to handle multiple sources. Special care is needed for points lying exactly on an edge. In this case the edges of both triangles must be initialized.

4.2 Polygons on the mesh

As seen in Figure 5 straight line segments induce linear and circular waves from its endpoints. Consequently we create linear and circular windows for each segment of a piecewise linear polygon. Exploiting the window intersection algorithms already necessary for the window propagation the overall initialization becomes very simple, because overlaps are handled consistently.

As a preprocess we subdivide the piecewise linear input polygon such that every segment lies entirely in one triangle. This can easily be done by inserting vertices on all intersections between triangle and polygon edges. Using this decomposition it is possible to handle each polygon segment independently. We illustrate the procedure with one line segment in a single triangle as depicted in Figure 6 (a). At first we add linear windows (green) whose extents are computed by intersecting orthogonal rays starting from the endpoints of the line segment with all triangle edges. Additionally, both endpoints induce circular windows (yellow) which are computed as described in Section 4.1. All new windows are again intersected with windows already registered to an edge. Notice that due to the exact equal distance intersection the result is again independent of the order in which the windows are added.

To complete the algorithm we next describe the propagation and intersection of linear windows. Now each window is expressed as a seven tuple $(id, b_0, b_1, d_0, d_1, \sigma, \tau)$ in which the added type id is either *circular* or *linear*. In the case of a *circular* window we proceed exactly as described in Section 3. For *linear* windows the tuple components have analogous meanings. The key difference is that the emanating boundary rays of a window starting at $(b_i, 0)$ in local coordinates are computed in a different way. They do not intersect at a pseudosource center but are always parallel (see Figure 6 (b)). The distance function over a linear window is a simple linear function fully determined by b_i and d_i.



(a) (b)

Figure 6: (a) A line source within a triangle induces a set of linear (green) and circular (yellow) windows. (b) Computation of the propagation direction in local coordinates.

4.2.1 Linear window propagation

The starting point is depicted in Figure 6 (b). Similar to section 3 the window w covers the segment between b_0 and b_1 on the edge e. The x-axis is aligned to e and the y-axis lies in the plane of the triangle where the window should be propagated through. Using elementary geometric calculations the propagation direction $\mathbf{n} = (n_x, n_y)$ can be computed in terms of the local coordinate system. The differences $|d_1 - d_0|$ and $b_1 - b_0$ define the angle between the linear front and the mesh edge:

$$\sin\alpha = \frac{-n_x}{|d_0 - d_1|} = \frac{|d_0 - d_1|}{b_1 - b_0}$$

Solving the previous equation for n_x, n_y can be computed by the theorem of Pythagoras:

$$\begin{aligned} n_x &= -\frac{(d_0 - d_1)^2}{b_1 - b_0} \\ n_y &= -\sqrt{(d_0 - d_1)^2 - n_x^2} \end{aligned}$$

Using these ray direction instead of the ray directions induced by the unfolded pseudo source the remaining part of the window propagation is identical to Section 3. Here overlaps of propagated windows can happen as well. For this reason the next paragraph describes all possible cases, namely linear-linear and circular-linear window intersections. Both reduce to the solution of a quadratic equation.

4.2.2 Linear window intersection

Again there are two different cases for window intersections. The trivial one occurs when the distance function of one window is larger in the whole overlapping interval. In this case it is easily clipped against the other window.
The more interesting case happens when the minimal distance function in the overlapping interval is composed of both windows. In this case there must be a point $(p_x, 0)$ where both distance functions are equal.

The distance function of a linear window along an edge is a simple linear function (cp. Figure 6 (b)) which can be formulated in terms of $\mathbf{n}$ or directly using the window components. It fulfills the interpolation condition $d_l(b_i) = d_i$.

$$d_l(p_x) = p_x \underbrace{\frac{d_1 - d_0}{b_1 - b_0}}_{m} + \underbrace{\frac{b_1 d_0 - b_0 d_1}{b_1 - b_0} + \sigma}_{n}$$

Now we are ready to compute intersections of linear windows with linear and circular windows to find the separation point p_x on the corresponding edge:

1. linear-linear intersection

$$\begin{aligned} & d_{p1}(p_x) &&= d_{p2}(p_x) \\ \Leftrightarrow\ & p_x m_1 + n_1 &&= p_x m_2 + n_2 \\ \Leftrightarrow\ & p_x &&= \tfrac{n_2 - n_1}{m_1 - m_2} \end{aligned}$$

2. circular-linear intersection

$$\begin{aligned} & d_c(p_x) &&= d_l(p_x) \\ \Leftrightarrow\ & \sqrt{(p_x - s_x)^2 + s_y^2} + \sigma &&= p_x m + n \end{aligned}$$

Squaring the previous equation leads to a quadratic equation $Ap_x^2 + Bp_x + C = 0$ with coefficients

$$\begin{aligned} A &= 1 - m^2 \\ B &= -2(s_x + m(n - \sigma)) \\ C &= s_x^2 + s_y^2 - (n - \sigma)^2 \end{aligned}$$

Notice that unlike the previous intersections here exist possibly two valid solutions which can lead to a trisection of the overlapping interval. In this case the cut circular window lies in the middle of two disconnected parts of the linear window.

5 Approximation algorithm

The propagation of distance information across many triangles leads to an increasing number of windows per edge because windows split up at vertices. A large number of windows increases the time as well as the space complexity of the algorithm. So the idea for the ε-Approximation-Algorithm in [1] is to merge neighboring windows on an edge whenever the induced relative error is acceptable. Allowing for example a relative error of $\varepsilon = 0.1\%$ leads to visually indistinguishable results but enables the processing of huge models with several millions of faces which are far too complex for the exact algorithm. Again the proposed linear windows fit naturally in the original framework and share all properties necessary for window merging. Before we describe the merging of linear windows we shortly review the basic principles and the case of circular windows. For details see [1].

To guarantee consistency of the geodesic field some conditions must be checked before merging two neighboring windows.

1. **Directionality:** Both windows propagate into the same direction.
2. **Visibility:** The pencil of rays of the merged window must at least cover all rays of the original windows so that no gaps arise.
3. **Continuity:** The distance at the endpoints bounding the merged window must be preserved to conserve distance field continuity.
4. **Type:** Both windows must be of the same type, e.g. planar or circular.

Additionally the user can prescribe a relative error bound ε_U so that only those merges are perfomed where the relative difference between the distance function of the new window $d'(p_x)$ and the original piecewise distance function $d_{lr}(p_x) = d_l(p_x) \cup d_r(p_x)$ is smaller than ε_U, i.e.

$$\frac{|d_{lr}(p_x) - d'(p_x)|}{d_{lr}(p_x)} \leq \varepsilon_U$$

5.1 Merging of circular windows

Taking two neighboring circular windows

$$\begin{aligned} w_l &= (id, b_{0l}, b_{1l}, d_{0l}, d_{1l}, \sigma_l, \tau_l) \\ w_r &= (id, b_{0r}, b_{1r}, d_{0r}, d_{1r}, \sigma_r, \tau_r) \end{aligned}$$

which meet at the common point $(b_{1l}, 0) = (b_{0r}, 0)$ the merged window w' is already determined up to σ' due to the necessary conditions:

$$\begin{aligned} id' &= id \\ b'_0 &= b_{0l} \\ b'_1 &= b_{1r} \\ d'_0 &= d_{0l} + \sigma_l - \sigma' \\ d'_1 &= d_{1r} + \sigma_r - \sigma' \\ \tau' &= \tau_l = \tau_r \end{aligned}$$

The continuity constrain restricts w''s pseudosource $s' = (s'_x, s'_y)$ to lie on a conic curve $s'^2_y(s_x)$. Because of the positivity of the d'_i and the visibility constraint the valid domain of this conic curve is further restricted. If it is the empty set, the merge is disallowed and in all other cases the smallest possible σ' is chosen (see [1] for details and how to evaluate the approximation error).

5.2 Merging of linear windows

The distance values d_i of a linear window can always be transformed so that the corresponding pseudosource distance σ vanishes. So w.l.o.g. two neighboring linear windows

$$\begin{aligned} w_l &= (id, b_{0l}, b_{1l}, d_{0l}, d_{1l}, 0, \tau_l) \\ w_r &= (id, b_{0r}, b_{1r}, d_{0r}, d_{1r}, 0, \tau_r) \end{aligned}$$

which join at the common point $(b_{1l}, 0) = (b_{0r}, 0)$ can be merged into a linear window

$$w' = (id, b_{0l}, b_{1r}, d_{0l}, d_{1r}, 0, \tau_l = \tau_r)$$

which satisfies all necessary constraints and is fully determined by the original windows. Notice that the visibility constraint is always fulfilled because diverging linear windows can only occur in combination with an additional point source or a saddle. The maximum approximation error is obtained at the joining point and can be computed as

$$\varepsilon = |1 - \frac{d_{1r}(b_{1l} - b_{0l}) + d_{0l}(b_{1r} - b_{1l}))}{d_{1l}(b_{1r} - b_{0l})}|$$

6 Adaptive refinement

The algorithm presented in the last section is able to compute the exact geodesic distance field on a triangle mesh with respect to an arbitrary polygon

embedded on the mesh. However, the distance information is not given explicitly but rather through a set of windows defined on the edges of the mesh. For most geometry processing algorithms this implicit information has to be made explicit. The standard approach to do this is to simply sample the distance function at the mesh vertices and then use a linear interpolant on each face as an approximation of the original distance field. In order to have some guarantee about the approximation tolerance, we have to refine the mesh in regions where this tolerance is violated. Usually this happens in the vicinity of the geodesic medial axis. To decide where to refine we compare the exact geodesic distance on edges with the linear interpolant and check if a user-defined threshold is exceeded. In this case we split the edge and insert a new sample point.

The geodesic distance field is smooth with constant gradient magnitude everywhere except for the geodesic medial axis. By properly placing the newly inserted vertices on the medial axis (i.e. at the maximum distance value) we can avoid excessive local refinement. This feature sensitive placement leads to optimal convergence and is in the spirit of [9].

Since edge splits in arbitrary order lead to poor triangles we employ a strategy similar to adaptive red-green triangulations. An important feature is that our refinement does not change the underlying geometry and can be seen as a pure upsampling of the original geodesic distance field. Due to this fact no recomputation of the geodesic field is necessary. The geodesic distance has to be updated only for those edges that are newly inserted. The edge-based refinement and the evaluation procedure are described in more detail in the next sections.

6.1 Edge-based refinement

In each refinement step we evaluate for each edge the maximal deviation between the exact distance function given as a pieceweise function along the edge and the linear function interpolating the exact distance only at the edge endpoints. If this maximal deviation exceeds a user-defined threshold the edge is tagged for refinement and the corresponding point p_{max} is cached as the optimal splitting position. Simply splitting all tagged edges would result in poor triangle quality. We aim at applying a one-to-four split (see Figure 7) of triangles lying entirely in the refined region. The one-to-four split operator

Figure 7: Implementation of a one-to-four split of a triangle using only edge split and edge flip operators. (a) Each edge of the black triangle is first split in arbitrary order. (b) The green edge, characterized by two adjacent triangles with only one original edge segment, is flipped to complete the one-to-four refinement.

can be composed of edge split and edge flip operations. For one triangle this requires the splitting of all edges (in arbitrary order) and the flipping of one specific edge (see Figure 7). To increase the number of regular one-to-four splits we iteratively tag all edges which are adjacent to triangles with already two tagged edges. This edges will be splitted on their midpoint. Subsequently all tagged edges are split at their cached split positions and all necessary flips are done. Identifying which edges should be flipped is easy if we mark all new created edges as red during the splitting process. If both triangles of a red edge are bounded by exactly two (the edge itself plus one additional) red edges the edge must be flipped.

6.2 Evaluation of interpolation error

The Geodesic Distance Function along a mesh edge e is defined piecewisely and consists of linear and circular segments corresponding to linear and circular windows. To compute the maximum deviation between this exact function and the linear interpolant defined by the exact distances on the edge vertices it is possible to first evaluate the maximal deviation for each segment individually and then take the overall maximum.
In the case of a linear segment the evaluation is simple. The difference between two linear functions is again a linear function and so the maximum is always on the boundary of the corresponding linear window.
In the case of a circular window the maximum can be computed analytically. The difference of both distance functions along the edge

$$E(p_x) = \sqrt{(p_x - s_x)^2 + s_y^2} + \sigma - (ax + b)$$

has a single extremum at

$$q_x = s_x - a\sqrt{\frac{s_y^2}{a^2-1}}$$

If q_x is not in the valid interval $[b_0..b_1]$ of the window the maximal deviation is on the boundary of the window as in the linear case.
The optimal position for a new sample point is exactly the position p_{max} where the deviation is maximal. However allowing split points to lie arbitrarily close to the edge endpoints leads to degenerate triangles. In practice we clamp the splitting position to be in the range of $25 - 75\%$ of the edge length. Additionally if the optimal position lies between $12.5 - 25\%$ or $75 - 87.5\%$ we adjust the new vertex so that the optimal position lies exactly on the midpoint of the new created edge because this leads to better triangulations. Given the optimal sample position $t \in [0..1]$ the update is as follows:

$$t \mapsto \begin{cases} 0.25 & 0 \leq t < 0.125 \\ 2t & 0.125 \leq t < 0.25 \\ t & 0.25 \leq t \leq 0.75 \\ 2t-1 & 0.75 < t < 0.875 \\ 0.75 & 0.875 \leq t \leq 1 \end{cases}$$

7 Results

We demonstrate the results of our algorithm on models of different complexity. Table 1 shows the corresponding timings for the computation of the exact and approximated geodesic fields which were generated on an AMD 64 3500+ system with 2GB of RAM. Additionally the average number of windows per edge (WPE) is listed. On the David and the Fandisk model we computed the geodesic field w.r.t. the red polygonal curves on the surface (see figure 9). The visualization uses a 1D texture to transfer the linear interpolant of the geodesic field into a color. For the car model depicted in Figure 1 we computed the geodesic field for the boundary and applied the adaptive refinement to get an satisfactory visualization. The refined mesh is showed in Figure 9. Obviously most of the mesh refinement occurs in a thin local neighborhood near the medial axis of the geodesic field. The plane model in Figure 8 illustrates the quality gain of our adaptive refinement in more detail. The upper row shows the original mesh with the corresponding linear interpolant of the geodesic field. Even though the mesh structure looks nice the result is very noisy near the medial axis and shows large errors. Applying the presented adaptive refinement we gain a high quality explicit representation of the geodesic field showed in the lower row together with the generated mesh structure. The approximation error reduced by a factor of 100 while the number of faces increased by a factor of 4.

Table 1: Timings

Model	#Faces	Time exact	WPE exact	Time 0.1%	WPE 0.1%
Plane	422	4ms	2.40	2ms	1.2
Fandisk	12k	1.90 s	9.06	0.12s	1.6
Car	34k	3.03 s	7.06	0.91s	3.4
David	8M	-	-	165s	1.3

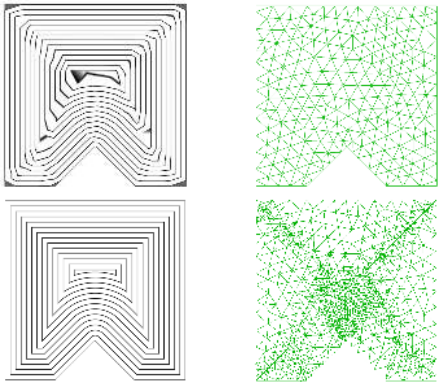


Figure 8: plane (422 faces)

8 Conclusion and future work

We have presented a generalization of the exact and approximate geodesic algorithm of [1] which allows to use arbitrary polygons as the boundary condition for the geodesic field. Our extensions integrate very naturally into the original algorithm and can easily be added to existing implementations. To increase the quality of the vertex based piecewise linear representation required for many applications using geodesics we included a well suited adaptive refinement technique which increases the quality of the piecewise linear representation to a user prescribed quality. The adaptive refinement strategy is very local and yields satisfactory mesh quality.

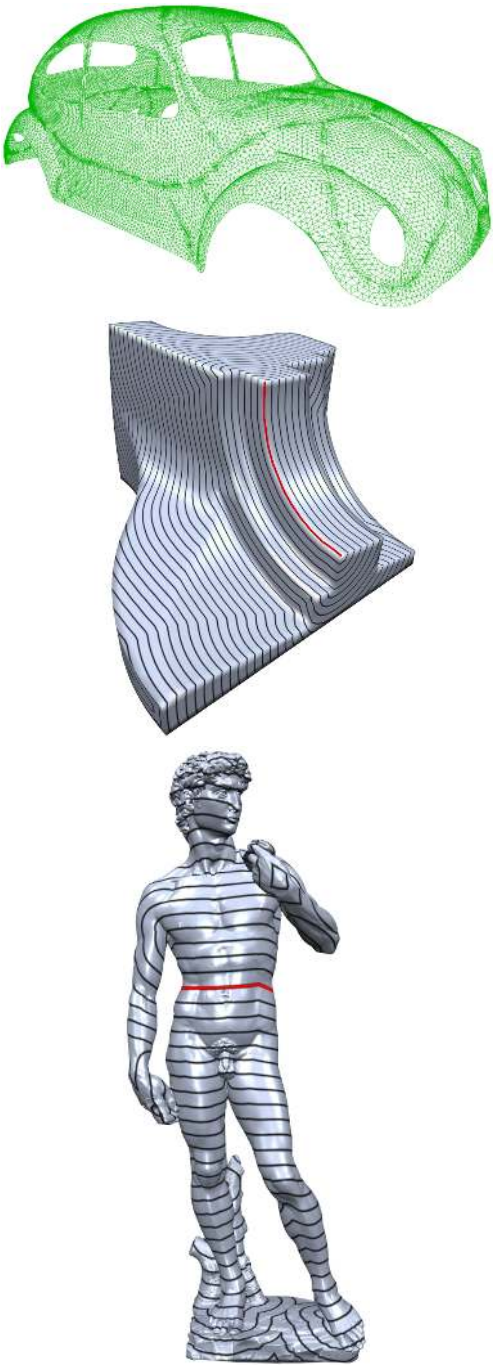


Figure 9: Car (34k faces), Fandisk (12k faces) and David (8M faces)

References

[1] V. Surazhsky, T. Surazhsky, D. Kirsanov, S. Gortler and H. Hoppe. Fast exact and approximate geodesics on meshes. Recognizing Surfaces using Three-Dimensional Textures. In *ACM SIGGRAPH Proc.,* 553–560, 2005.

[2] J.S.B. Mitchell, D.M. Mount and C.H. Papadimitriou. The discrete geodesic problem *SIAM Journal of Computing,* 16(4):647–668, 1987.

[3] M. Lanthier, A. Maheshwari, and J.-R. Sack. Approximating weighted shortest paths on polyhedral surfaces. In *Proc. 13th Annu. ACM Symp. Comput. Geom.*, 274–283, 1997.

[4] R. Kimmel, and J.A. Sethian. Minimal Discrete Curves and Surfaces. In *Proc. of National Academy of Sci.95(15)*, 8431–8435, 1998.

[5] M. Novotni and R. Klein. Computing geodesic distances on triangular meshes. In *Proc. of WSCG*, 341–347, 2002.

[6] D. Kirsanov. Minimal discrete curves and surfaces. PhD thesis, Applied Math., Harvard University.

[7] M. Reimers. Minimal discrete curves and surfaces. PhD thesis, Math., Univ. of Oslo.

[8] Y.-J. Liu, Q.-Y. Zhou and S-M. Hu. Handling Degenerate Cases in Exact Geodesic Computation on Triangle Meshes. Computer Graphics International, 2007, to appear.

[9] L. P. Kobbelt, M. Botsch, U. Schwanecke and H.-P. Seidel. Feature-Sensitive Surface Extraction From Volume Data. In *ACM SIGGRAPH Proc.,* 57–66, 2001.

[10] R.E. Bank, A. H. Sherman and A. Weiser. Refinement Algorithms and Data Structures for Regular Local Mesh Refinement. In *Sci. Computing*,IMACS/North Holland, Amsterdam, 3–17, 1983.

[11] M. Vasilescu and D. Terzopoulus. Irregular triangulation, discontinuities and hierarchical subdivision. In *Proc. of Computer Vision and Pattern Recognition Conference*, 829–832, 1992.

[12] L. Kobbelt. $\sqrt{3}$ Subdivision. In *Proc. of ACM SIGGRAPH* 103–112, 2000.