Università di Pisa

Facoltà di Scienze
Matematiche Fisiche e Naturali

Corso di Laurea Magistrale in Informatica

Tesi di Laurea

Fast arbitrary geodesic computation on triangular meshes.

Candidato:

Rosario Aiello

Relatori:

Dott. Paolo Cignoni

Dott. Nico Pietroni

Controrelatore:

Prof. Francesco Romani

Anno Accademico 2013/14

Alla mia famiglia

Acknowledgements

First of all, I would like to thank Nico, Paolo and Francesco, who wisely guided me during the development of this project, always providing insightful advices and ideas. Indeed, I would like to thank all the people at the Visual Computing Laboratory for making me feel welcome from the very start. In particular, a big thank goes to Luigi and Andrea for having me in their office and for providing good company, which always makes the work a lot easier. A big thank goes to Marco Di B. for not firing me and to Federico, Giorgio and Marco P. for the always entertaining evening football matches.

I also want to thank Christian, Roberto, Lorenzo, Matteo and Luca, with whom I shared most of my time while following my studies, for being always available for a constructive discussion about any kind of problem. Thanks to Emanuele for the most needed coffee-breaks and chats.

Last but not least, I would like to thank my family, who always supported me and encouraged me; my friends, who I consider as part of my family: Antonio, Benigno and Davide, always present when it is time to party.

Thanks to Valentina, for supporting and "sopporting" me during the last weeks of work.

Abstract

We propose a method to accelerate the computation of geodesic over triangular meshes. The method is based on a precomputation step that allows to store arbitrary complex distance metrics and a query step where we employ a modified version of the bidirectional A* algorithm. We show how this method is significantly faster then the classical Dijkstra algorithm for the computation of point to point distance. Moreover, as we precompute the exact geodesic, it achieves better accuracy.

Contents

Chapter 1

Introduction

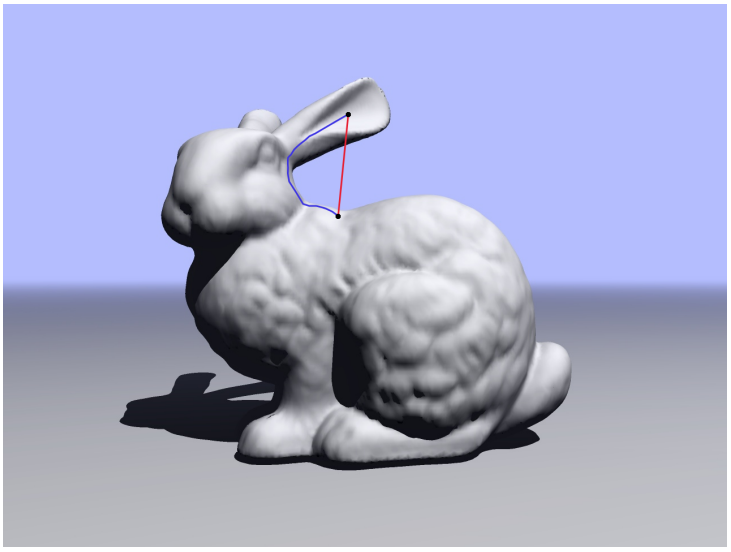


Figure 1.1: *A geodesic path between two points on the Bunny model is shown in blue. Euclidean distance would measure the norm of the vector connecting the points.*

1.1 What is a geodesic?

The term "*geodesic*" comes from *geodesy*, the science of measuring the size and shape of Earth; in the original sense, a geodesic was the shortest route between two points on the Earth's surface, namely, a segment of a great

circle. The term has been generalized to include measurements in much more general mathematical spaces; for example, in graph theory, one might consider a geodesic between two vertices/nodes of a graph. We can give a general definition of a geodesic to be a curve describing the locally shortest path (under a specific metric) between to points of a particular space. A *metric* on a set M is generally defined as a function $d : M \times M \to \mathbb{R}$, called distance function. For all x, y and z in M, d is required to follow these four conditions:

1. $d(x, y) \geq 0 \ \forall x, y \in M$ (non-negativity)
2. $d(x, y) = 0 \iff x = y$ (identity)
3. $d(x, y) = d(y, x)$ (symmetry)
4. $d(x, z) \leq d(x, y) + d(y, z)$ (triangular inequality)

These conditions express intuitive notions about the concept of distance: for example, that the distance between distinct points is positive and the distance from x to y is the same as the distance from y to x. The triangle inequality means that the distance from x to z via y is at least as great as from x to z directly. A *metric space* is an ordered pair (M, d).
In geometry processing we are interested in defining an *intrinsic* metric, that takes measurements by "walking only on the surface". Considering figure 1.1, we are not interested in the *extrinsic* measurement, i.e. the Euclidean distance given by the norm of the vector shown in red. Instead we are looking for the *intrinsic* measurement of the geodesic path (lying on the surface) connecting the two points, which is in blue. The distance between two points of a metric space relative to the intrinsic metric can be defined as the infimum of the length of all paths connecting them. We will give a more precise definition of an intrinsic metric in section 2.6. As we know, the shortest path between two points on a plane is a straight line. However, things get more complicated when we switch to e.g. manifold surfaces or triangular meshes.
For some applications, extrinsic distances may still yield an "*adequate approximation*" of the real intrinsic distance. For example, if we want to

compute distances in a very small neighborhood of a point, the surface could be considered to be planar in that neighborhood. Thus, depending on the degree of approximation required by the application, the above definition given for the planar case can be acceptable. If this is not the case, an algorithm to compute the exact (or approximate) geodesic distance can be mandatory, with the consequence of increasing the overall computation complexity.

1.2 Manifold meshes

We will assume our surface to be a *manifold triangular mesh*. Triangular meshes are widely used in Computer Graphics to model 3D objects. This is due to the fact that they are simpler to handle, both in terms of the data structures necessary to implement them and in terms of the graphic hardware, which is specifically thought to handle triangles. Moreover, there exist many different types of meshes like for example *quad*, *surface* or *volumetric* meshes. In general, a polygonal mesh consists of three kinds of elements: vertices, edges and faces. In the case of a triangle mesh, the faces consist of triangles. There are two kinds of information associated to mesh elements:

- Topology, which describes the incidence relations among mesh elements (e.g., adjacent vertices and edges of a face, etc).
- Geometry, which specifies the position and other geometric characteristics of a vertex

A mesh is *manifold* if these two properties hold:

1) Each edge is incident to only one or two faces
2) The faces incident to a vertex form an open or a closed fan (see figure 1.2).

The *orientation* of a face is a cyclic ordering of its incident vertices and we define the orientation of two adjacent faces to be "compatible" if the two vertices in the common edge are in opposite order. Therefore a manifold mesh is always *orientable*, meaning that any two adjacent faces have

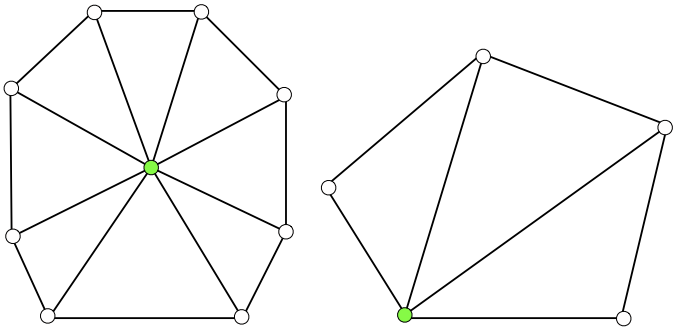


Figure 1.2: *Faces incident on the green vertex form a closed fan (left) and an open fan (right).*

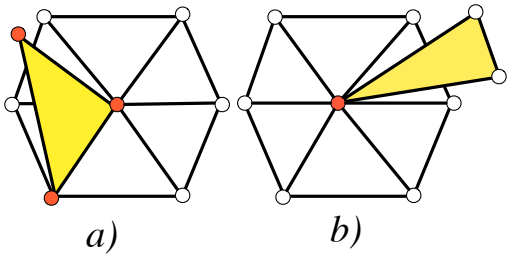


Figure 1.3: *a) Non-manifold edge, property 1) is violated; b) Non-manifold vertex, property 2) is violated.*

compatible orientation. Figure 1.3 shows two situations of non-manifold meshes.

1.3 Geodesic domain

The discretization of the underlying surface could be of great help in computing geodesic: after all, a mesh can be thought as a graph with its vertices and edges. Hence, a trivial approach to compute geodesic could be to employ one of the well-known shortest paths algorithms designed for graphs, such as Dijkstra's, weighting edges proportionally to their lengths. However this trivial solution, aside from producing results that are very triangulation dependent, is quickly limited by the underlying surface discretization. In fact, consider figure 1.4: we get different results for the same vertices depending on the meshing. Dijkstra will produce geodesic distance $d = \sqrt{2}$ for the gray pair (which is actually the exact distance) while for the yellow pair it will return $d = 2$, as the computed path must follow the edges defined by the triangulation. Moreover, in the specific case of Dijkstra's algorithm, there would be no information reuse. That is, if we applied Dijkstra's algorithm to computed the distance between a source point and any other vertex of the mesh, this information cannot be reused if we need to compute the distance between another pair of points.
As pointed out in [1] [2], the algorithms for computing geodesics on manifold surfaces can be divided into two major categories:

- The "computational geometry" approach, which is oriented on computing the exact geodesic distances with respect to the piecewise linear approximation of the surface.
- The Partial Differential Equation (PDE) [3] [4] [5] approaches, which are oriented on solving the the Eikonal equation. This equation states that every distance function d must satisfy the condition

  $$\|\nabla d\| = 1$$

  Where ∇ ("nabla") indicates the gradient of d. To see why this is a reasonable condition let us consider the example depicted in figure 1.5

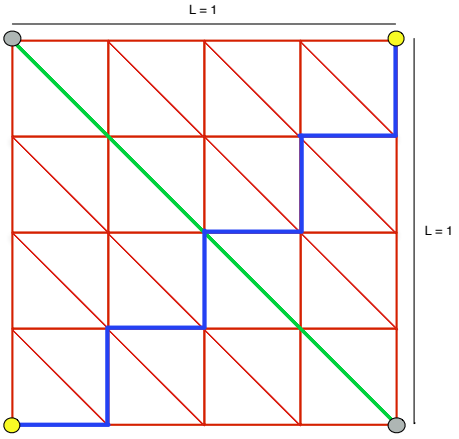


Figure 1.4: *Dijkstra's computed distance is triangulation dependent. The green path is actually exact while the blue one is forced to follow the triangulation.*

left, where we see a planar convex region Ω. The distance function in this case is trivially $d(x) = \|x - x_0\|$. If we compute its gradient we obtain $\nabla d = x - x_0/\|x - x_0\|$ and hence it holds that $\|\nabla d\| = 1$. For the concave shape on the right of figure 1.5 the Eikonal equation defines a local constraint on the gradient of the function that is used to compute the function globally on the domain Ω.

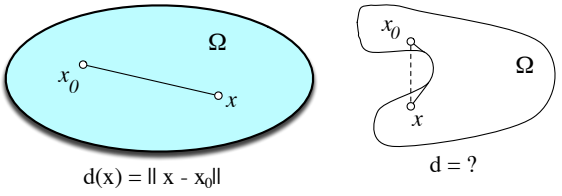


Figure 1.5: *(Left) Distance between two points inside a region Ω on a plane is a straight line. (Right) If the path is constrained to stay inside Ω, this is not true anymore.*

When we consider triangular meshes, we are handling a discretization of some 3D surface S. Since discrete surfaces cannot be explicitly differentiated, methods from differential geometry to compute geodesic paths and distances cannot be applied in this case. However, algorithms from differential geometry can be discretized and extended (see chapter 2). Moreover,

while the general problem of computing a shortest path between polyhedral obstacles in 3D has been shown to be NP-hard by [6], computing a geodesic path on a triangular surface is an easier problem and it is solvable in polynomial time.
As we will see in the next chapter, the "one-source/multiple, all-destinations" problem has been very widely studied whereas the results available for the "all-pairs" geodesic problem are much less. The discrete geodesic problem has attracted a great deal of attention since Mitchell, Mount and Papadimitriou [7] published their seminal paper in 1987. They presented an algorithm for computing single-source exact geodesic distance in $O(n^2 \log n)$ where n is the number of vertices in the input mesh. Later, practical implementations and performance improvements have been provided for the original algorithm by Chen and Han [8] and Surazhsky et al. [9]. Different approaches to the problem have been proposed: Campen and Kobbelt [10] focused on the geodesic problem applied to meshes containing defects like holes and gaps extending Sethian's Fast Marching Method [11] [4] to defected meshes. Crane [12] proposed a novel approach based on the heat method while Campen et al. generalized and extended well-known methods to the anisotropic case and proposed an ad hoc method called Short-Term Vector Dijkstra [13]. Ying et al. [1] recently proposed a novel approach called Saddle Vertex Graph, from which the idea for our work was first inspired. Finally, we must cite the work from Xin et al. [14] who proposed an approach similar to the one we developed.

1.4 Geodesic: applications

The computation of intrinsic geodesic distances and geodesic paths on surfaces is a fundamental low-level building block in countless Computer Graphics and Geometry Processing applications which require the query of geodesic distance between pairs of points on the mesh [2] [15]. An example of a geodesic Voronoi diagram computed on the FERTILITY model is shown in figure 1.6. Campen et al. [16] employ geodesic computation to implement a all-quadriteral patch layouts on manifold surfaces, guided by a field

of curvature directions (see figure 1.7). Moreover, parameterizing a mesh often involves cutting the mesh into one or more charts [17] [18], and the result generally has less distortion and better packing efficiency if the cuts are geodesic.

Campen and Kobbelt [10] employ their defect-tolerant geodesic computation algorithm to texture mapping on defected models (see figure 1.8). Mesh editing systems such as [19] use geodesics to delineate the extents of editing operations.

Geodesic paths are also used in segmenting a mesh into subparts, as done in [20] [21]. Moreover, since geodesic paths establish a surface distance metric, they are an essential building block for applications like skinning [22], mesh watermarking [23] and the definition of surface vector fields [24]. Parameterization metrics based on isomaps are also based on geodesics [25] [26]. Morse analysis of a geodesic distance field has been used in [27] for a shape classification algorithm.

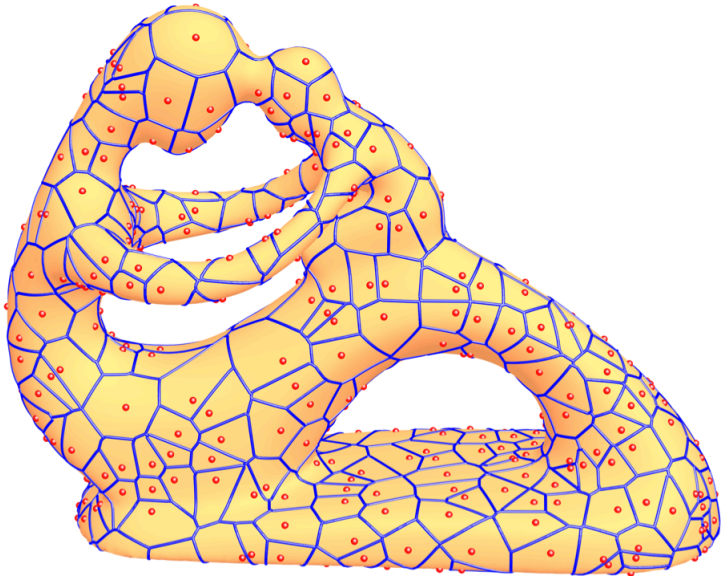


Figure 1.6: *Geodesic Voronoi diagram computed on the* FERTILITY *model.*

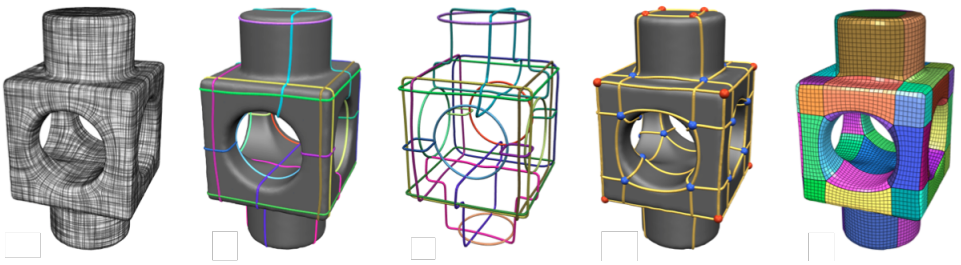


Figure 1.7: *Dual Loops Meshing approach that constructs coarse all-quadrilateral patch layouts with high geometric fidelity.*

Figure 1.8: *Texture mapping on a* FACE *model containing holes due to occlusion effects.*

1.5 Outline

In Chapter 2 we will review the current state of the art regarding geodesic computation. More in depth, we will analyze the various approaches starting from the algorithm proposed by Mitchell et al. in 1987 through the more recent ones. At the end of the chapter we will make some comparisons and analyze the advantages and flaws of the discussed methods.

In Chapter 3 we will describe the VoroGeo algorithm, a new approach for the all-pairs geodesic computation problem. We will show which problems were faced during the design of this approach and how they have been solved during implementation.

In Chapter 4 we will describe which tests have been made for parameters tweaking and we will show some results and statistics about our algorithm's speed and accuracy.

In Chapter 5 we will draw our final conclusions on our work. We will then briefly propose some possible future extension and improvements to our method.

Chapter 2

State of The Art

In this chapter, we will discuss the state of the art regarding methods for the geodesic computation. We will procede to describe:

- The exact geodesic algorithm proposed by Mitchell et al. [7], which has led to a deep interest in this field.
- Improvements and modifications to the original exact algorithm, proposed by Chen [8] and Surazhsky [9]. Bommes [28] proposed a generalization of Surazhsky's implementation to handle arbitrary, possibly open, polygons on the mesh to define the zero set of the distance field. Xin [29] improved Chen and Han (CH) [8] algorithm proposing a more efficient version know as ICH.
- Sethian and Kimmel [11] [4] Fast Marching Method (FMM) for computing distance fields by solving the Eikonal equation through numerical techniques for computing the position of propagating fronts.
- The method by Campen and Kobbelt [13], who focused on computing geodesics on defected models.
- The innovative adoption of the Heat Method (HM) proposed by Crane [12], who relates Varadhan's formula to distance computation.

- The Short Term Vector Dijkstra (STVD) proposed by Campen et al. [30] that is specifically thought to handle intrinsic anisotropic metrics.

- The Saddle Vertex Graph (SVG) [1] approach recently proposed by Ying, and The GTU method by Xin [14] which will be left for last to be thoroughly analyzed as they were the main source of inspiration for our work.

Finally, in section 2.9, we will briefly compare the presented methods and highlight their individual advantages and limitations.

2.1 Exact geodesic computation

Given a piecewise planar surface $\mathcal{S}$, Mitchell, Mount and Papadimitriou's (MMP) algorith [7] [9] computes an explicit representation of the *geodesic distance function* $\mathcal{D} : \mathcal{S} \rightarrow \mathcal{R}$. This function maps each point $p \in \mathcal{S}$ to the length of its geodesic path to the source v_s. The basic idea behind the MMP algorithm is to partition each mesh edge into a set of *windows* that encode all the shortest paths passing within it. The shortest path is governed by three basic properties:

- interior to a triangle, it must be a straight line;

- when crossing over an edge, a shortest path must correspond to a straight line when the two adjacent triangles are unfolded onto the same plane.

- finally, as proven in [7], the only vertices[1] a shortest path can pass through are boundary vertices, saddle vertices and parabolic vertices.

We call saddle vertex a vertex with a total angle greater than 2π, while a parabolic vertex (also known as Euclidean) is a vertex with an angle equal to 2π. A window w on and edge e is defined as a 6-tuple $<b_0, b_1, d_0, d_1, \sigma, \tau>$, where: $b_0, b_1 \in [0, \|e\|]$ encode the windows endpoints, d_0 and d_1 encode the

[1] Other from source and destination vertices.

distance of the source vertex from the endpoints (relative to the window in the planar unfolding), τ gives the side of the edge on which the source lies. As shown by 2.1a, the shortest paths are depicted as a *pencil of rays* emanating from the source s through the unfolded triangles. As it is shown, it is possible to express the position of the source s in terms of b_0, b_1, d_0, d_1 by intersecting two circles. This situation represents a shortest path that does not pass through any saddle vertex. To understand what is encoded by parameter σ of the window definition consider figure 2.1 b): s is a so called *pseudosource* (a saddle vertex), and all the paths passing within w also pass through s, hence w will encode the position of s with respect to the edge and $\sigma = \mathcal{D}(s)$, will hold the distance from the pseudosource s to the source v_s; b_0, b_1 and d_0, d_1 will still have the same meanings as before, just this time they are referring to the pseudosource s and not the original source v_s. This particularity in handling saddle vertices is due to the unfolding of the

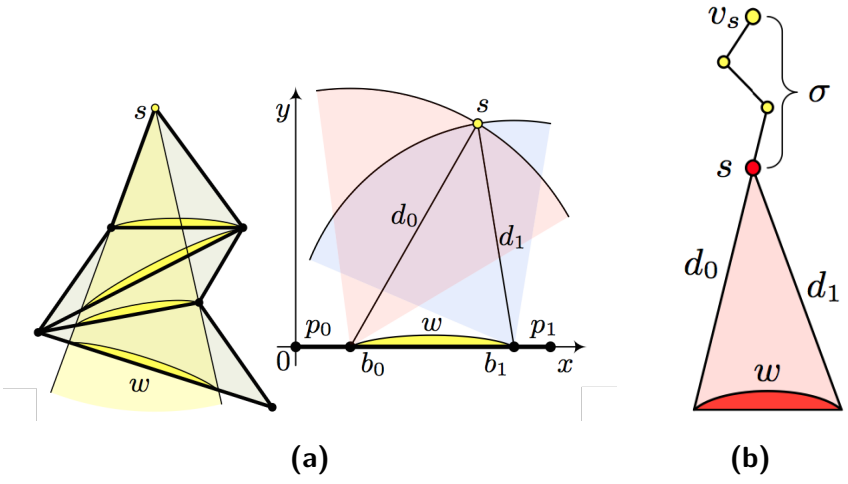


Figure 2.1: *(a) Pencil of rays emanating from the source (left) and parametrization of the source position (right). (b) A pseudosouorce s and its distance σ from the source.*

neighborhood of a saddle vertex: we have a red saddle vertex in figure 2.2. Unfolding the adjacent triangles into the plane of the upper triangle will result in two different "*images*" of v_s because the total angle is greater than 2π.

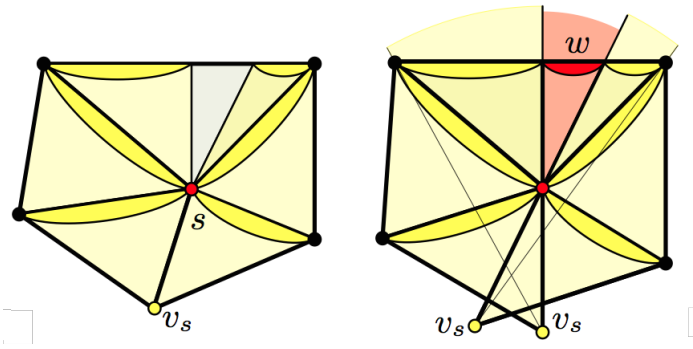


Figure 2.2: *Unfolding of a saddle vertex neighborhood. All shortest paths from v_s to the red window w pass through the saddle vertex s.*

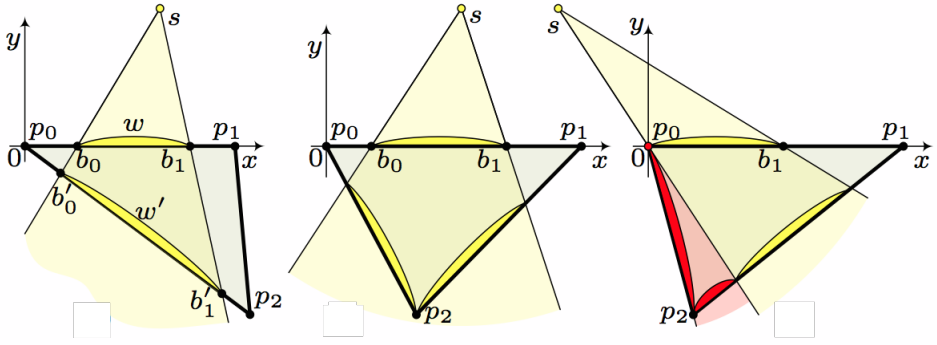


Figure 2.3: *Window propagation results in one new window (left). Window propagation results in two new windows (center). Window propagation results of one window plus two additional windows (right).*

The MMP algorithm propagates the distance field encoded into a window w across an adjacent face f by computing how the rays would extend on the opposite edges. However, the opposite edges could already contain previously propagated windows, so the information has to be merged in order to minimize the distance field. Three examples of window propagation are depicted in figure 2.3: considering the case on the right, we can notice that a ray passing through the saddle vertex p_0 will result into two additional windows, that cover the parts of the edges that lie to the left of the ray (s, p_0), and are not already "*illuminated*" by s through w. Therefore, p_0 will act as a new pseudosource for the two red windows with $\sigma = \mathcal{D}(p_0)$. When two

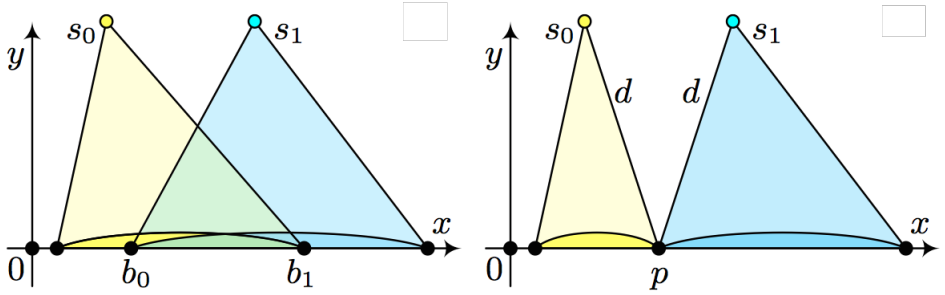


Figure 2.4: *Two overlapping windows with pseudosources s_0, s_1 and intersection $\delta = [b_0, b_1]$, case $\sigma_0 = \sigma_1$ is assumed (left). Resulting disjoint windows (right).*

windows w_0 and w_1 overlap on edge e two main situation could arise: in the simpler case one window defines a larger distance function everywhere on δ, so to resolve the conflict δ is simply cut away. A more interesting case is when one window (e.g. w_0) is minimal on part of δ while the other one is minimal on the remaining part of δ. Figure 2.4 shows the formation of two disjoint windows obtained finding the point $p \in \delta$ where the distance function defined by w_0 and w_1 are equal; i.e. $\|s_0 - p\| + \sigma_0 = \|s_1 - p\| + \sigma_1$. This issue can be reduced by solving a quadratic equation with a single solution if the planar coordinate system is defined to align e with the x axis, as shown in figure 2.4.

In the implementation of the MMP algorithm given by Surazhsky [9], a priority queue is used to propagate the windows through the whole mesh. The queue is initialized with a window for each edge adjacent to the source

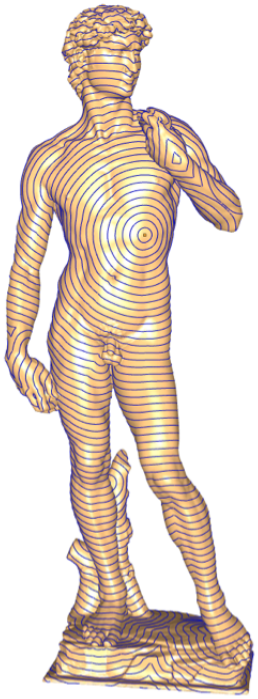


Figure 2.5: *Isolines computed on the 400k-triangles David model.*

v_s, the distance field defined by these initial windows is trivially given by the edge lengths. Then, windows are propagated as a wavefront by keeping the order of the queue according to its distance to the source vertex.
Theoretically, during propagation, each edge may have $O(n)$ windows. Therefore, in the worst case, the total number of windows can be $O(n^2)$. Hence, the theoretical worst case complexity of the MMP algorithm is $O(n^2)$ space and $O(n^2 \log n)$ time, where the $\log n$ factor is due to management of the priority queue and to the resolution of conflicts between overlapping windows. However, through a series of experiments on typical meshes, Surazhsky has shown in [9] that the average number of windows per edge is $O(\sqrt{n})$, lowering the algorithm complexity for practical cases. Moreover, he showed that the window complexity surprisingly decreases when the mesh surface has a rough texture. As explained in [9], this is intuitively due to the fact that bumpy features in a surface cause adjacent windows to overlap and annihilate each other.
Chen and Han (CH) [8] improved the time complexity to $O(n^2)$ which re-

mains the best-known complexity. However, extensive experiments have shown that this algorithm often runs much slower than the MMP algorithm's implementation given by Surazhsky. An improved version of the CH algorithm (known as ICH) was proposed by Xin and Wang which despite having still $O(n^2 \log n)$ time complexity outperforms both the MMP and CH algorithm in practice.
In figure reffig:david400k, we can see the isolines computed by the exact geodesic algorithm on a 400k-triangles David model. This result took 75 seconds as reported in [9].

2.2 Approximate algorithm

The method proposed by [9] works just like the exact algorithm, except for one key difference: before propagating a window, it tries to merge it with an adjacent window on the same edge. The algorithm computes an approximation $\mathcal{D}$ of the geodesic distance function D which is a lower bound, namely $\mathcal{D}(p) \leq D(p), \forall p \in \mathcal{S}$. The merging of two windows, w_0 and w_1, is only performed when some constraints are satisfied. These checks are on directionality, visibility, continuity, monotonicity and of course on bounding the error.

As reported in [9], the main bottleneck during the exact algorithm execution is the memory space required to store all the windows, providing strong motivation for the approximate algorithm. Experimental results have shown that with a 0.1% relative error bound, the algorithm runs significantly faster and uses less memory than the exact algorithm. On the David model shown in figure 2.5, the computation takes 11 seconds and the reported average relative error (i.e. $|\mathcal{D}(v) - D(v)|/D(v)$) is 0.05% of the object diameter. More comparisons are available in [9].

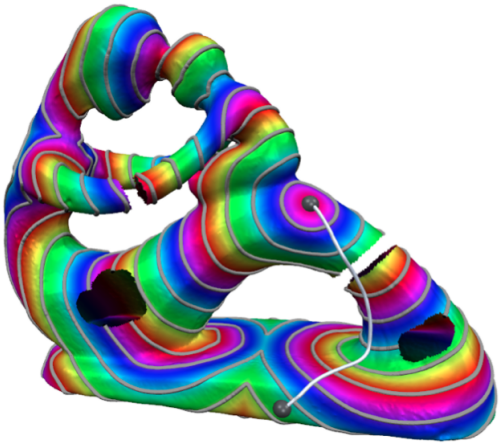


Figure 2.6: *An intrinsic distance field and a geodesic path computed on an defected mesh.*

2.3 Fast Marching Method

The Fast Marching Method (FMM) has been proposed by Sethian [11] for regular grids, and then extended by Kimmel and Sethian [4] to compute approximate geodesic distances on triangular surfaces. Further extensions and modifications to the original algorithm have been proposed also in [5] [31] and [32]. The FMM is a special case of the Level Set Method [33] for solv-

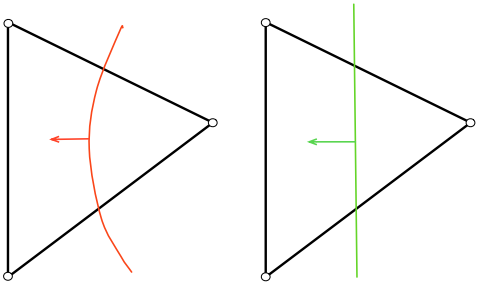


Figure 2.7: *Front approximation.*

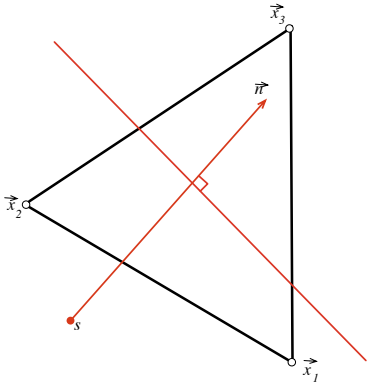


Figure 2.8: *Advancing front passed vertices x_1 and x_2 moving towards x_3.*

ing the so called *boundary value problems* of the Eikonal equation. Level set methods are numerical techniques for computing the position of propagating fronts. The FMM is strongly reminiscent of Dijkstra's algorithm being based on a marching front that is controlled through a heap structure. However as we discussed in section 1.3 the big problem of Dijkstra's algorithm is that it is unable to "cut through triangles" as the path must follow the mesh triangulation. The observation behind the FMM regards

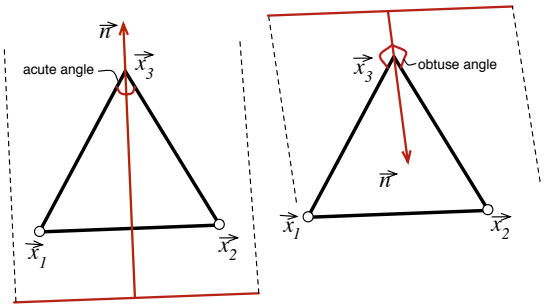


Figure 2.9: *Front normal vector making an acute (left) and obtuse (right) angle passing through* x_3.

what happens to the information wave-front while it moves away from the originating source point: we can think of this front as a circle getting bigger and bigger stepping away from the source.

As we know the curvature of a circle of radius r is defined as $1/r$, hence as the radius gets bigger the curvature decreases. At a local scale, the approximation made by the FMM is the one depicted in figure 2.7 where the circle-like front is approximated with a straight line.

Now let's consider the advancing front depicted in figure 2.8: assuming we know the distances d_1, d_2 of x_1 and x_2, we want to compute the distance d_3 relative to x_3, by exploiting the front approximation idea, working with planar coordinates (for simplicity, x_3 is assumed to be at the origin). In this coordinate system, since the wave-font is a straight line, we can always take a unit[1] normal vector $\vec{n}$ to this line and express $d_1 = \vec{n}^\top \vec{x_1} + p$ and $d_2 = \vec{n}^\top \vec{x_2} + p$, that is projecting $\vec{x_1}$ (resp. $\vec{x_2}$) over $\vec{n}$ and measuring how far away from the source those points are. Moreover since we put x_3 to be on the origin we can write $d_3 = \vec{n}^\top \vec{x_3} + p = p$. The factor p is relative to the fact that the position of the source in the planar unfolding is unknown. We can combine the two equations for d_1 and d_2 into $\vec{d} = \vec{n}^\top X + pI_{2\times 1}$. Solving

[1] We want the normal vector to be of unit norm because intuitively we are interested in measuring distance d and not $\|n\|d$. This can be related to what is stated by the Eikonal equation: the Eikonal equation states that if ϕ is the distance function, then the magnitude of its gradient $\nabla\phi$ should be equal to 1 everywhere; hence, $|\nabla\phi| = 1$ can be thought as of saying that "*distance changes at one meter per meter*"[12].

this matrix equation in terms of $\vec{n}$ results in

$$\vec{n} = X^{-1}(\vec{d} - pI_{2\times 1})$$

exploiting the fact that $\vec{n}^\top \vec{n} = 1$ we get a quadratic equation where the only unknown variable is p:

$$1 = p^2 I_{2\times 1}^\top Q I_{2\times 1} - 2pI_{2\times 1}^\top Q\vec{d} + \vec{d}^\top Q\vec{d} \tag{2.1}$$

where $Q = (X^\top X)^{-1}$ and $I_{2\times 1}$ is the 2×1 identity matrix. Equation 2.1 admits two solutions, one relative to the case of the normal vector making an acute angle passing through x_3 and the other one relative to the case of the normal vector making an obtuse angle at x_3. Which of the two solutions should be taken? Since we are trying to extend the already known distances d_1 and d_2 to compute d_3, this cannot be smaller than any of the other two distances. Hence we need to choose the solution that gives us a value larger than d_1 and d_2. As reported in [4] [12] [1] and [30] the update step of this method is not entirely stable since it requires an underlying non-obtuse triangulation which is a very strong assumption in most real cases. In figure 2.10 we depicted one of the issues that could arise in the update step, for more details the reader can refer to [11] [4]. The situation illustrates the issue of the front advancing towards an obtuse triangle, where the "causality" property is violated: this property assures that the computed distance for a triangle vertex can only be extended from the other two vertices know distances.

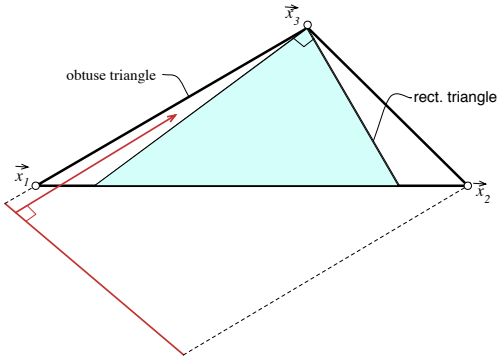


Figure 2.10: *In the case of an obtuse triangle, the front could reach x_3 before x_2*

2.4 Defect tolerant algorithm

Campen and Kobbelt proposed in [10] a method for computing intrinsic geodesic distances and geodesic paths on raw meshes in contrast with most of the available methods, which usually make some implicit or explicit assumption on the underlying structure of the mesh. Unfortunately some requirements are not always met in practice: real-world meshes often exhibit several kinds of defects depending on their origin holes, gaps (see figure 2.6), non-manifold configurations with singular edges and vertices, or they might even be just a soup of polygons, completely lacking any connectivity information.

A defect-tolerant method is convenient considering the fact that some of the mesh-repairing techniques are ill-posed and often exhibit various geometrical and topological ambiguities if no additional prior knowledge is available. In some cases, the application at hand does not actually require the mesh to be repaired anyway, spending these efforts solely in order to facilitate the requisite geodesic distance computations seems to be immoderate. As exposed in [10], the basic idea is to abstract from the mesh structure (and all its potential defects) and to perform all computations discretely in a crust volume tightly restricted to the spatial regions occupied by elements of the input. Due to the abstraction from the input, applicability of this method is not limited to polygon meshes; other representations such as point sets,

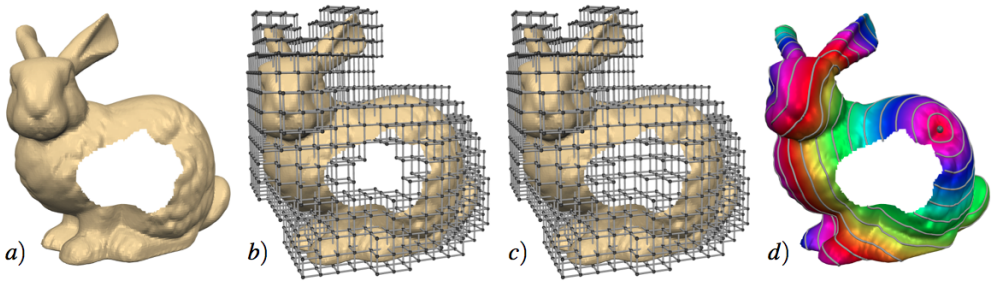


Figure 2.11: *a) Input mesh with defects. b) Initial cubical complex. c) Complex after applying topology-sensitive morphological operators; the hole is now bridged. d) Visualization of a geodesic distance field (with isolines) emanating from a point source, computed on the complex, and mapped to the input mesh by interpolation.*

implicit functions, or NURBS patches can be handled as well.
As depicted in figure 2.11, the first step of this method is to abstract the mesh to a cubical complex representation ("voxelization"), that is a cut-out of a three-dimensional Cartesian grid such that all elements of the mesh are contained in the union of its cells. This is obtained employing an octree $\mathcal{O}$ mapping its root to the bounding box of the mesh. The elements of the mesh are then "inserted" into $\mathcal{O}$ and intersected cells are refined up to a user specified maximum level l. However, performing distance computations on the resulting dilated complex would result in significantly lowered accuracy. Hence, topology-sensitive dilation and erosion operators are applied to the cubical representation to fill holes up to a specified size (see figure 2.12) obtaining the final cubical representation $\mathcal{C}$. To perform approximate geodesic

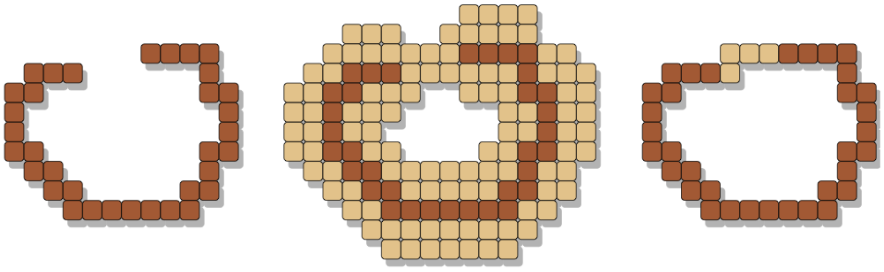


Figure 2.12: *2D schematic example of the employed morphological operators dilation (middle) and topology-preserving erosion (right), filling holes up to a specified size.*

distance computation, the Fast Marching Method (FMM) [11] [4] is applied. The FMM performs a front propagation starting from a set of sources all over the nodes of $\mathcal{C}$, hence the set $C_s \in \mathcal{C}$ of nodes containing the source points must be initialized. This means initializing the information about d for all nodes n in C_s $(N(C_s))$ by setting $d(n) = min_{s \in S} d(n, s)$. This distance can be approximated with the Euclidean distance or, if an average normal vector is available at n, accuracy can be improved by calculating $d(n, s)$ as the distance between n and the orthogonal projection of s onto the tangent plane T_n at n. Initialization for FM method is completed by setting $d(n) = \infty \forall n \notin N(C_s)$.

As stated in [10], once rasterization, dilation, and erosion have been performed the obtained cubical complex (resp. octree) can be used for the quick computation of multiple distance fields; it does not have to be rebuilt each time.

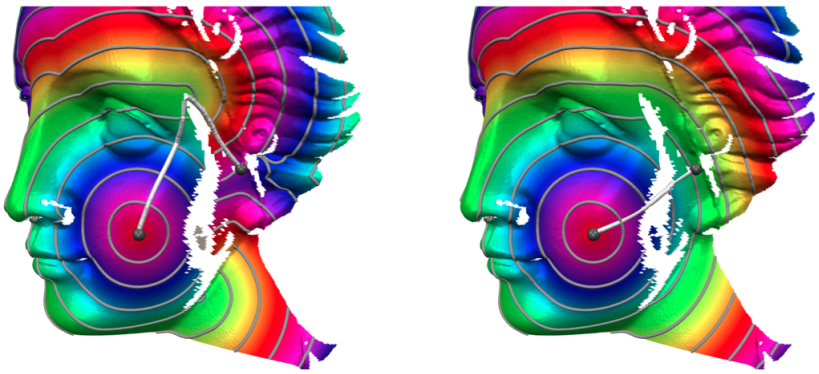


Figure 2.13: *Raw scanned model of a face containing holes due to occlusion effects. (left) Geodesic computation without morphological operations. (right) With morphological operations for hole bridging.*

Naturally, this method has some limitations too. Due to the automatic nature and generality of the method, it can not resolve ambiguities that are inherent in the input due to large missing parts. Hence, computed distance fields might be inconsistent with those of the object that is actually meant to be represented by input and additional knowledge about the object would be required to handle hole bridging more consistently in such cases.

In figure 2.13, we can see the results of a computation on a face model pre-

senting some defects due to occlusion effects happened during the scan. On the right side, we can see that by choosing dilation distance such that these holes are bridged the computed intrinsic distance approximations tolerate these defects.
In conclusion, we can say that the distances computed by this method usually deviate from actual intrinsic distances (on consistent models) to some degree due to the finite resolution and the FM approach. At high resolutions, the total runtime is dominated by the morphological operations since these cause a large number of cell neighbor queries, cell splits, and collapses [10].

2.5 Geodesic in Heat

Crane [12] proposed a totally new approach for computing geodesics, called *Heat Method*. The main idea is to exploit the relationship between the heat kernel function $k_{t,x}(y)$ and distance function. As depicted by Crane, the intuition behind this method is imagining to touch a point x on the mesh surface with a scorching hot needle. Heat diffusion can be modeled as a large collection of hot particles taking random walks starting at x, hence any particle that reaches a distant point y after a small time t has had little time to deviate from the shortest possible path.
The heat kernel function $k_{t,x}(y)$ measures the heat transferred from a source x to a destination y after time t. Varadhan's formula relates the heat kernel with distance saying that the geodesic distance ϕ between any pair of points (x, y) on a Riemannian manifold can be recovered via a simple pointwise transformation of the heat kernel: $\phi(x, y) = \lim_{t\to 0} \sqrt{-4t \log k_{t,x}(y)}$. Crane hypothesized that the reason why this kind of approach had not been considered so far is because it would require a precise reconstruction of the heat kernel, which is difficult to obtain. Thus Crane's intuition was that of working with a broader class of function, namely all those that have gradient parallel to geodesics. If we take an approximation u_t of the heat flow for a fixed time t. Unless u_t exhibits the same rate of decay, Varadhan's transformation will yield very poor results because it is very sensitive to

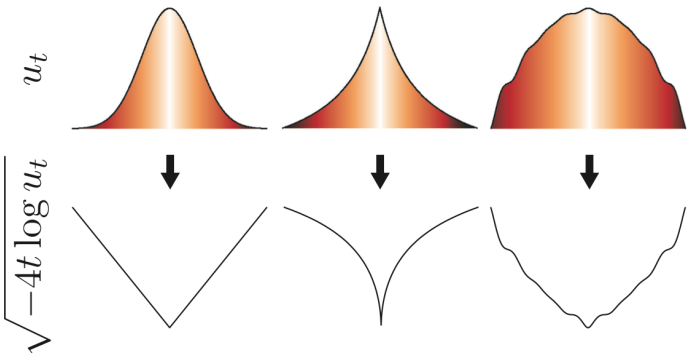


Figure 2.14: *Geodesic distance (bottom left) recovered from an exact reconstruction of the heat kernel (top left). In presence of numerical error, results may be very far from acceptable (middle, right).*

errors in magnitude (see figure 2.14). The heat method only requires that ∇u_t points in the right direction, that is, parallel to $\nabla \phi$. The heat method

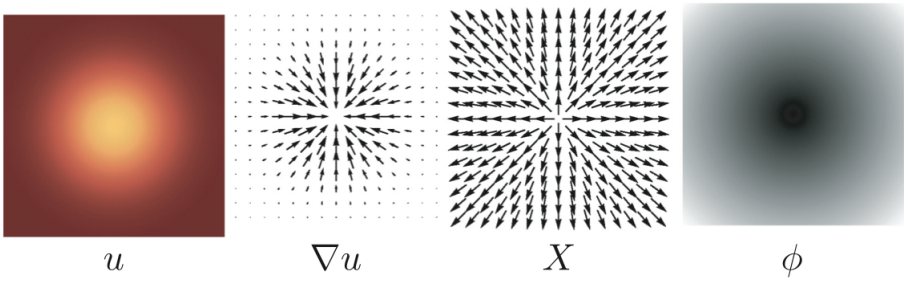


Figure 2.15: *Outline of heat method steps: 1) Heat u is allowed to diffuse for brief period. 2) Temperature gradient is normalized and negated to obtain a unit vector field X pointing along geodesics. 3) A function ϕ whose gradient follows X recovers the right distance.*

can be summed up in these three steps (see also figure 2.15):

1) Integrate the heat flow $\dot{u} = \Delta u$ for some fixed time t;

2) Evaluate the vector field $X = -\nabla u / |\nabla u|$;

3) Solve the Poisson equation $\Delta \phi = \nabla \cdot X$.

Function ϕ approximates geodesic distance, approaching true distance as t goes to zero. In step 3), the method finds the closest scalar potential ϕ by

minimizing $\int_M |\nabla\phi - X|^2$ which is equivalent to solving the Euler-Lagrange equations $\Delta\phi = \nabla\cdot X$. In step 2) and 3) the gradient magnitude can be safely ignored thanks to the Eikonal equation (see section 1.3). Several methods for computing distances are based on solving such equation, by imposing the condition $\phi|_\gamma = 0$ on some subset γ of the domain and solving the Eikonal equation everywhere else. However, this formulation of the problem has some issues related to the fact that it is *nonlinear* and difficult to solve requiring some specialized solver. What the heat method does is a change of variables moving from a nonlinear/hyperbolic problem to a linear/elliptic one.

Moreover, this formulation of the problem does not depend on the choice of spatial discretization, that is the HM can be applied to triangle meshes as well as to point clouds or grids, as long as a Laplacian, a gradient and a divergence can be evaluated. On the other hand, accuracy of the HM relies in part on the choice of the time step $t = mh^2$ where m is a constant and h is the mean space in between nodes (e.g. average edge length). As shown in [12], $m = 1$ yields very high accuracy on a wide variety of triangulated meshes. Looking at the comparisons reported in [12] (see also figure 2.16), maximum absolute error and mean error are relative to the mesh diameter. On the Ramses model (1.6M-triangles), the precomputation step takes 63.4 seconds, plus 1.45 seconds to compute distances. This gives a speedup of 68x the time needed by Surazhsky's algorithm [9] with respectively 0.49% max error and 0.24% mean error as opposed to the 0.29% max error and 0.35% obtained by Surazhsky's algorithm. Hence, HM performs better on average while still having a larger maximum error. Although the HM works quite well for smooth surfaces, the accuracy of the approximated distance becomes low for models with rich details.

Figure 2.16: *Distances from a single point source on the Bimba (149k-faces), Aphrodite (205k-faces), Lion (353k-faces) and Ramses (1.6M-faces).*

2.6 Short Term Vector Dijkstra

Most of the available methods to compute intrinsic geodesic distances are designed according to the standard Riemannian metric induced by the surface's embedding in Euclidean space. Campen, Heistermann and Kobbelt [30] investigate the possibilities for a generalization of well known algorithms (i.e. the ones we take under consideration in this section) to anisotropic metrics. They proposed a novel algorithm, called Short-Term Vector Dijkstra (STVD) which despite its simplicity provides practical accuracy at higher speed than the generalized versions of existing methods.
Generally, *anisotropy* is referred as the condition of holding a property which is dependent on the directions in which it is observed. Consider a 2-manifold M equipped with smoothly varying norms $\|\cdot\|_{g_x}$ on the tangent spaces T_xM. The total length of a continuously differentiable curve $\zeta : [0,1] \to M$ can be defined as $\ell(\zeta) = \int_0^1 \|\zeta'(t)\|_{g_{\zeta(t)}} dt$. Therefore we can define the *intrinsic metric* g measuring geodesic distances between two points p, q as the infimum over the lengths of all curves ζ connecting p and q, that is

$$g(p,q) = \inf_{\zeta} \{\ell(\zeta) : \zeta(0) = p, \zeta(1) = q\}$$

In this way we obtain the length metric space (M, g). In the case of the Riemannian metric, also called *standard metric*, we have $\|v\|_{g_x} = \sqrt{\langle v, v\rangle_x}$. In the standard metric it holds $r(v,x) = \|v\|_{g_x}/\|v\|_x \equiv r(x)$ i.e. the quotient is independent of v and the metric g is *isotropic* (have no directional dependency). In the case of an anisotropic metric we have that $r(v,x) \not\equiv r(x)$ and there is directional dependency. On a triangle mesh M, an anisotropic norm $\|\cdot\|_g$ is specified in a sampled manner. The samples can be given per vertex ($\|\cdot\|_{g_v}$), or face ($\|\cdot\|_{g_f}$) or edge ($\|\cdot\|_{g_e}$). In figure. 2.17 an anisotropic metric is visualized through curvature-related tensor ellipses: the resulting intrinsic Delaunay triangulation (right) based on the isotropic input mesh (middle) is depicted. To introduce the STVD idea, we briefly go back to Dijkstra's original algorithm as applied to the computation of the geodesic distance between two points on a triangular mesh. Due to the graph-nature of the algorithm, it will not compute the *real* path between the two points but rather it will consider the lengths of the edge paths that meander over

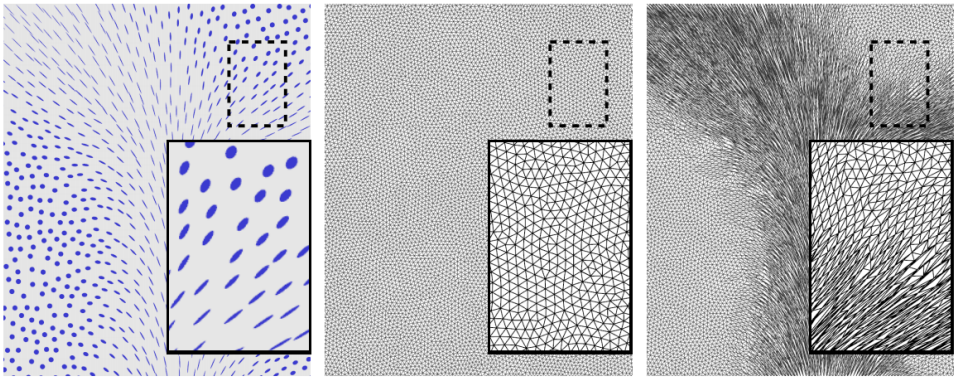


Figure 2.17: *Inverse tensor ellipses are used to visualize an anisotropic metric. The input (isotropic) mesh is shown in the middle while on the right it is shown the corresponding intrinsic Delaunay triangulation.*

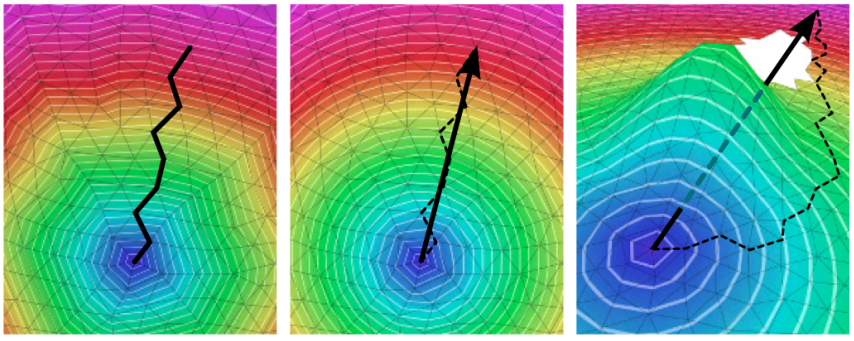


Figure 2.18: *(left) A geodesic path computed by Dijkstra's algorithm is very triangulation dependent. (middle) Computation using a vector-valued Dijkstra variant. (right) Shortcomings of the vector-valued variant: oblivious to holes, obstacles and geometric features.*

the surface (see figure 2.18 left). Thus the computed distances will be highly inaccurate and very triangulation dependent. In the middle and on the right of figure 2.18 we see the so called *vector-valued Dijkstra* algorithm employed by Schmidt et al. [34] to obtain geodesic distances for the purpose of local surface parameterization. This method is based on the idea of vectorially summing the edges first and measuring the lengths afterwards, as opposed to the original idea of measuring the edge lengths and then scalarly summing them. The situation depicted in the middle is the one favorable to the vector-valued Dijkstra: in the planar case the distance computed is actually exact. On the right we can clearly see the limitations and shortcomings of this method when it is applied to 2-manifold meshes: it is oblivious to holes, obstacles and geometric variations in the surface.

The idea behind STVD is to form an hybrid-method combining the positive

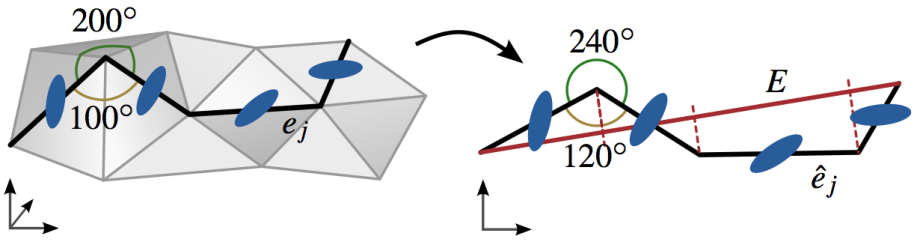


Figure 2.19: *Unfolding of edge chain to the plane. Edge lengths and 1-ring angles are preserved. The sampled norms $\|\cdot\|_{g_e}$ are visualized as blue tensor ellipses.*

aspects of the classical scalar-valued version and the vector-valued variant. This is done by equipping the scalar-valued version with a short-term vector-valued memory. In this way the meanders of the edge paths through the triangulation can locally be smoothed without globally disregarding surface geometry. The short-term vector-valued memory consists of a window of k preceding edge vectors, and the STVD algorithm is obtained by changing just the *distance update function* in the original scalar-valued version. In the modified version of the distance update function $update\_dist(v, w)$ the window of k preceding vectors is exploited by computing

$$dist \leftarrow \min_{i=1}^{k} w.pred^i.dist + \ell_g(\sum_{j=1}^{i}(w.pred^{j-1}, w.pred^j))$$

where $w.pred^{i+1} = w.pred^i.pred$ and $w.pred^0 = w$. As we explained earlier in this section the particularity of this method resides in the fact that the edges $e_j = (w.pred^{j-1}, w.pred^j)$ are vectorially summed and their length ℓ_g is computed with respect to g: in figure 2.19 we can see how the unfolded edge chain is vectorially summed to obtain vector E. The length ℓ_g is computed exploiting the edge-sampled norm $\|\cdot\|_{g_e}$ (visualized through blue tensor ellipses) as $\sum_j \hat{e}_j$ where the $\hat{e}_j$ are the unfolded edges. In practice, the edges' signed orthogonal projections (see figure 2.19) are used to obtain

$$\ell_g(\sum_j e_j) = \sum_j \hat{e}_j^T \bar{E} \|\bar{E}\|_{g_{\hat{e}_j}}$$

where $\bar{E} = E/\|E\|$.

Focusing on performance and accuracy, Campen et al. empirically observed

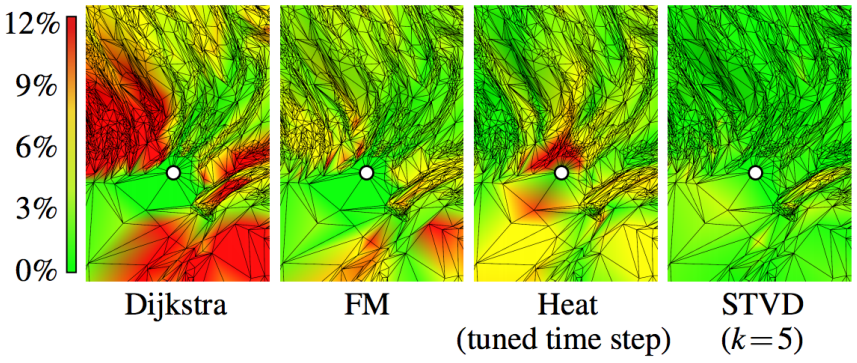


Figure 2.20: *Comparing different methods in the case of a mesh containing bad shaped elements. STVD obtains good accuracy w.r.t. exact [9] distances.*

that while the depth k of the vector-valued memory needs to be increased so as to increase the angular resolution of the distance propagation, the lengths of the used vector sums need to be decreased so as to reduce the approximation errors of the unfolding-based measurement. They achieved both this seemingly contradicting goals by mesh refinement: using 1-to-4 splits the edge lengths are reduced by a factor of 2. The value of k need to be increased by a factor <2. The results achieved by the STVD algorithm are very interesting in the case of high anisotropy where other methods tend to produce bad results. Moreover, also when dealing with meshes containing

bad shaped elements the STVD is able to yield considerably better results (see figure 2.20).

2.7 SVG algorithm

The Saddle Vertex Graph (SVG) approach proposed by Ying et al. [1] consists of a sparse undirected graph that encodes complete geodesic information so that every shortest path on the mesh corresponds to a shortest path on the SVG. Most of the available algorithms for geodesic computation employ a '*global*' approach by propagating distance information in a wavefront order. The SVG method solves the problem from a *local* perspective, breaking down the problem into smaller sub-problems allowing to reuse information. This is made possible by the local structure exhibited by the SVG which gets stronger moving from smooth surfaces to more complicated models with richer geometry details.
As already mentioned in section 2.1, a vertex v is called *saddle* if the total

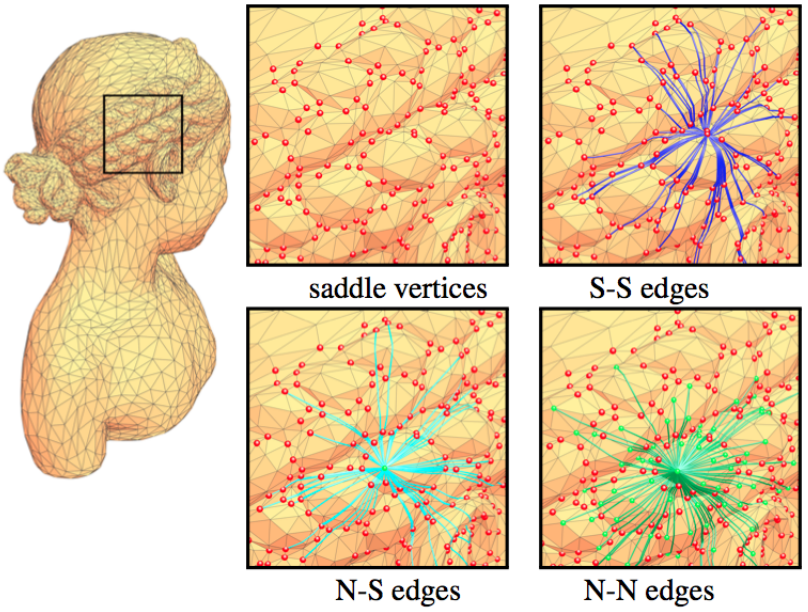


Figure 2.21: *SVG on a 9K-face Bimba model. S-S, N-S and N-N edges are displayed only for a vertex.*

vertex angle is greater than 2π. A globally shortest geodesic path $\gamma(p, q)$ is called *direct* if it does not pass through any saddle vertices, *indirect* otherwise. An indirect path can be partitioned into segments, each of which is a direct path.
For each vertex v, two sets of neighbors are defined: $\mathcal{S}(v)$ is the set of saddle vertices which can be reached from v via direct geodesic paths; $\mathcal{N}(v)$ is the set of non-saddle vertices which can be reached from v via direct geodesic paths. Let V be the set of vertices of the mesh; this can be split in two disjoint subsets V_S (saddle vertices) and V_N (non-saddle vertices). Therefore, we can define three *disjoint* sets of edges (see figure 2.21):

$$E_{SS} = \{\ \gamma(p,q) \mid p, q \in \mathcal{S}(v) \text{ and } \gamma(p,q) \text{ is direct }\},$$
$$E_{NS} = \{\ \gamma(p,q) \mid p \in \mathcal{N}(v), q \in \mathcal{S}(v) \text{ and } \gamma(p,q) \text{ is direct }\},$$
$$E_{NN} = \{\ \gamma(p,q) \mid p, q \in \mathcal{N}(v) \text{ and } \gamma(p,q) \text{ is direct }\}.$$

This allows us to define the SVG as a composition of three tiers S_1, S_2, S_3:

$S_1 = (V_S, E_{SS})$ is the core network consisting of all the S-S edges.
$S_2 = (V_S \bigcup V_N, E_{NS})$ connects non-saddle to saddle vertices.
$S_3 = (V_N, E_{NN})$ contains the N-N edges connecting non-saddle vertices.

To explain the idea behind the SVG, we can directly quote [1]:

> *If the mesh is viewed as a planet, the vertices are cities, and the geodesic path between two vertices is a flight route. So the SVG is indeed a flight route map, which covers every city on the planet. The saddle vertices are the hub cities and the non.saddle vertices are the small cities. The S-S edges are the major route connecting a small city to the nearest hub. Each N-N edge is a local flight route between two nearby small cities.*

The complexity (and density) of the SVG obviously depends on the number of saddle vertices and their distribution. In the extreme case of a convex polyhedron, there are no saddle vertices, so we have $E_{NN} = \binom{n}{2}$ with a dense S_3 and a dense SVG. Ying explains in [1] that after testing the saddle vertex ratio $r = \frac{|V_S|}{|V|}$ on common models of various resolution the typical r value ranged from 40% to 60% remaining fairly stable with respect to the

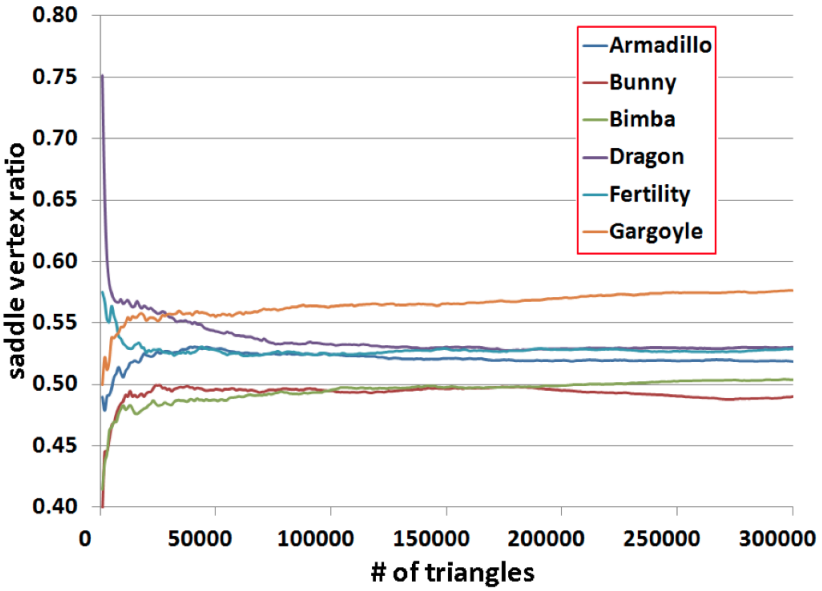


Figure 2.22: *Experimental results show that for bigger models, the saddle vertices ratio remains fairly stable.*

mesh resolution and tessellation (see figure 2.22). Moreover, the SVG exhibits a strong local structure that - as empirical results have proven - gets stronger when applied to more detailed models that have more complicated geometry.

To build the SVG, the direct geodesic paths must be computed for each vertex v. To do this, the approach described in section 2.1 is employed, with the difference that being interested only in direct paths, there will be no pseudosources and this means that the algorithm can stop when all "*direct windows*" have been computed.

Computing all the direct geodesic paths is clearly the bottleneck of the SVG construction algorithm, to have a faster computation without major loss in accuracy, a user provided parameter K is employed. This parameter indicates the maximal number of mesh vertices covered by a geodesic disk. Given a vertex v and a geodesic disk $\odot(v, R)$ centered in v and with radius R that contains no more than K vertices, all direct geodesic paths within $\odot(v, R)$ are taken as SVG edges. The geodesic disks and the direct geodesic paths for each vertex v are computed in parallel on the GPU, then the SVG

is reassembled. Clearly, computing the *complete* SVG graph would require much more time and memory space. Using the parameter K, the theoretical complexity of computing direct geodesic paths with Chen and Han [8] or MMP [7] algorithm is $O(nK^2 \log K/N)$ time where n is the number of vertices and N is the number of parallel threads used. The empirical complexity is sub-quadratic (as exposed by [29]) and is $O(nK^{1.5} \log K/N)$. As pointed out by Ying, a very important feature of the SVG approach is that the computed distances form a *metric*, i.e. they maintain the symmetry condition and triangle inequality. Hence, given an undirected connected graph G, the set V of vertices of G forms a metric space by defining $d(x, y)$ to be the length of the shortest path connecting the vertices x and y. Since the geodesic distance is computed using the shortest path distance on the SVG, the resulting distance is guaranteed to be a metric. These conditions do not hold for all of other approximate algorithms presented so far such as the one by Surazhsky (sec. 2.2), the Heat Method (sec. 2.5) and the GTU method which will be presented in the next section (see figure 2.1). To

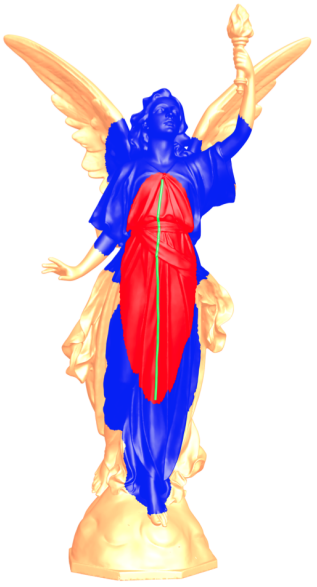


Figure 2.23: *A geodesic path is highlighted on the 263K-vertices Lucy model. The vertices visited by Dijkstra's algorithm are shown in blue, those visited by A* are in red.*

compute a single-source single-destination geodesic distance (SSSD) $d(s,t)$, firstly a Bidirectional Dijkstra search is run *directly on the mesh* to compute an upper bound U_{st}. Then, the $A*$ search is run *on the SVG* exploiting the upper bound U_{st} to prune the search by allowing only the point p satisfying the condition $d_s(p)+\|pt\| \leq U_{st}$. In figure 2.23, we can see the region visited by Bidirectional Dijkstra in blue while the region visited by $A*$ is shown in red. If the SVG is exact, that is if all direct geodesic paths have been precomputed; $A*$ has to be run only on tier 1 of the SVG (the core network). Otherwise the search applies to $S_1 \bigcup S_2 \bigcup S_3$. The single-source/multiple-source all destinations geodesic distances can be computed running Dijkstra on the SVG and updating the distances for all non-saddle vertices q with the shortest distance from q's saddle neighbors. If the exact SVG is available, the overall time complexity is $O(|E_{SS}| \log |V_S| + |E_{NS}|)$, where the $|E_{NS}|$ factor is due to updating non-saddle vertices' distances. If the SVG is approximate, Dijkstra has to be run on the entire SVG, hence the complexity becomes $O((|E_{SS}| + |E_{NS}| + |E_{NN}|) \log |V|)$. To retrieve the shortest path, a backtrace procedure based on triangles unfolding is employed.

As we have already mentioned before, the SVG approach does not fit well in case of convex polyhedrons where the SVG becomes very dense. Moreover, the computation of the exact geodesic paths is very time expensive so its implementation was written in CUDA 5.0 to be executed on an Nvidia Tesla K20 GPU with 2496 CUDA cores and 5GB of memory. As reported in [1], the "CPU-version" of the SVG construction algorithm is from 10 to 40 times slower than the GPU version. In figure 2.24 we can see how the SVG method is independent from mesh resolution and tessellation as the SVG was computed with fixed $K = 50$ on a Buddah model from a low-resolution mesh with many obtuse triangles to a high-resolution mesh with regular triangulation producing consistently high quality results. In the same way, we can see from figure 2.25 that on the Bimba model a value of 50 for parameter K already produces results that are hardly distinguishable from the exact.

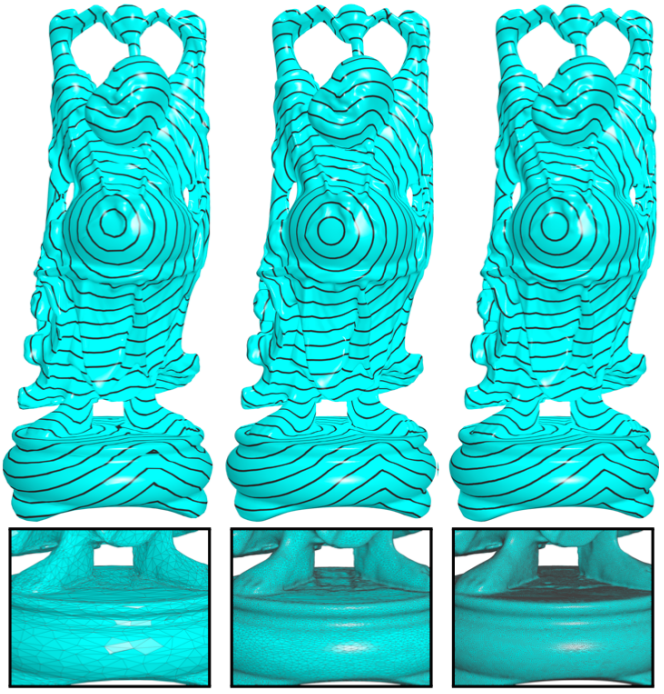


Figure 2.24: *SVG applied with fixed parameter K=50 to a 40K-face (left), 300K-face (middle) and 600K-face Buddah model. Consistently high quality results are produced.*

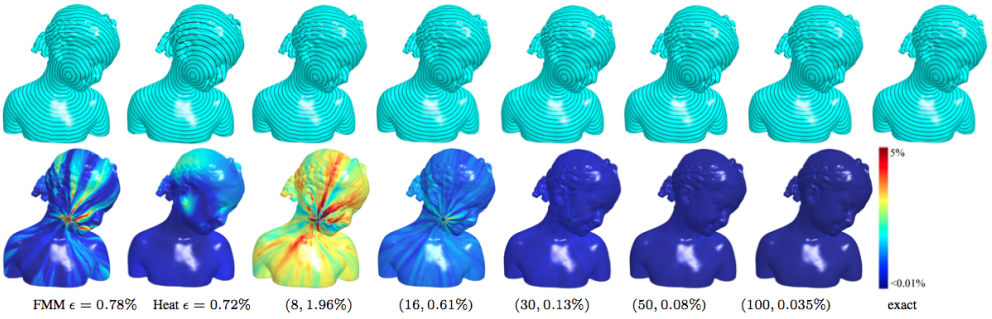


Figure 2.25: *FMM, HM and SVG applied to the Bimba model. From the third column the SVG results are show reporting the tuple (K, ϵ) where ϵ is the mean relative error.*

2.8 GTU method

The Geodesic Triangles Unfolding (GTU) method was proposed by Xin et al. [14]. It is based on a precomputation of geodesic distances in order to achieve fast queries for *all-pairs* geodesics. In the preprocessing step, a Delaunay triangulation of the mesh based on m uniformly distributed samples is computed.

This triangulation is induced by the geodesic fields that have to be computed starting from the random samples (figure 2.26). The surface is divided into *geodesic triangles*, i.e. triangles obtained replacing each Delaunay edge with a geodesic (see figure 2.27). The algorithm then computes for all geodesic triangles the geodesic distance from any inside vertex to the three corners. Distance between any pair of the m sample points is also computed. The preprocessing step has $O(mn^2 \log n)$ time and $O(m^2 + n)$ space complexity; it takes $O(mn^2 \log n)$ time to run the exact geodesic algorithm with each sample as a source and $O(n)$ time to replace each Delaunay edge with a geodesic. Since there are $O(m)$ edges, this has a $O(mn)$ time and overall prepropcessing time complexity is $O(mn^2 \log n)$. In the query step,

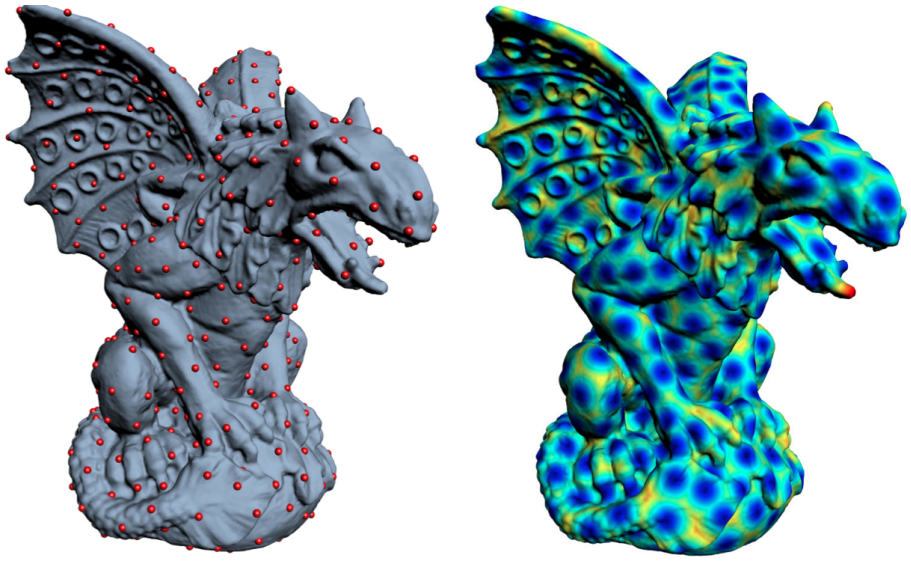


Figure 2.26: *Geodesic fields computed taking the m randomly sampled points as sources.*

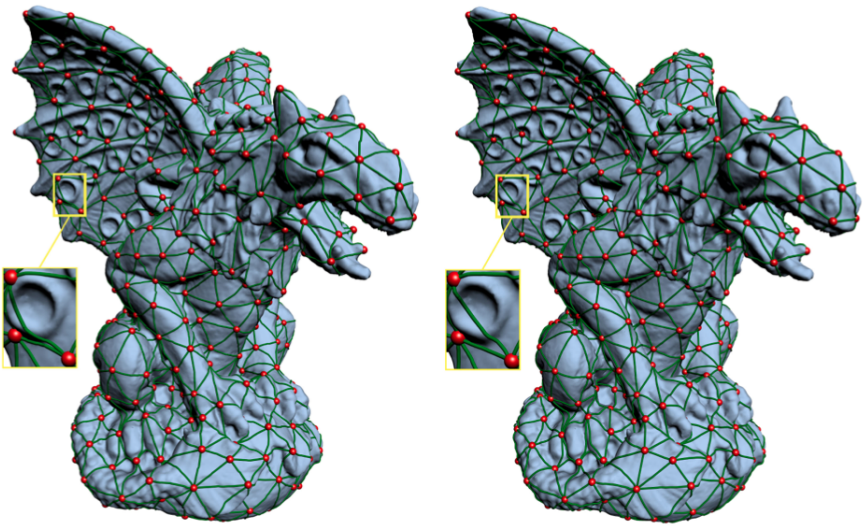


Figure 2.27: *On the left side, approximate Delaunay triangulation computed from m randomly placed samples. On the right side, each Delaunay edge is replaced with a geodesic, obtaining the so called "geodesic triangles".*

given two points q_1 and q_2 on the surface, the GTU method unfolds the corresponding geodesic triangles containing q_1 and q_2 to $\mathbb{R}^2$ and then uses Euclidean distance between their 2D images to approximate the geodesic distance on the mesh. The query points q_1 and q_2 are also mapped to $\mathbb{R}^2$. This unfolding process is completely local and has constant time complexity.

Consider figure 2.28. Given two geodesic triangles $\triangle psr$ and $\triangle qrs$, they can be unfolded to $\mathbb{R}^2$ (with a minimum distortion) obtaining two triangles $\triangle p's'r'$ and $\triangle q'r's'$ where the corresponding Euclidean edge lengths are equal to those of the geodesic edges. We denote this unfolding operation with respect to edge rs as $u(p, q|rs)$. As shown in figure 2.28, if p and q are on the same side of rs a "*one-side unfolding*" takes place, otherwise we have a "*two-side unfolding*". It has to hold that $d(r, s) = \|r's'\|$, $d(p, r) = \|p'r'\|$, $d(p, s) = \|p's'\|$, $d(q, r) = \|q'r'\|$ and $d(q, s) = \|q's'\|$. All possible cases that could arise during the query step are described thoroughly in [14], here for the sake of a concise exposition we will analyze only the most important ones. First let us consider the situation depicted in figure 2.29 a). Let us

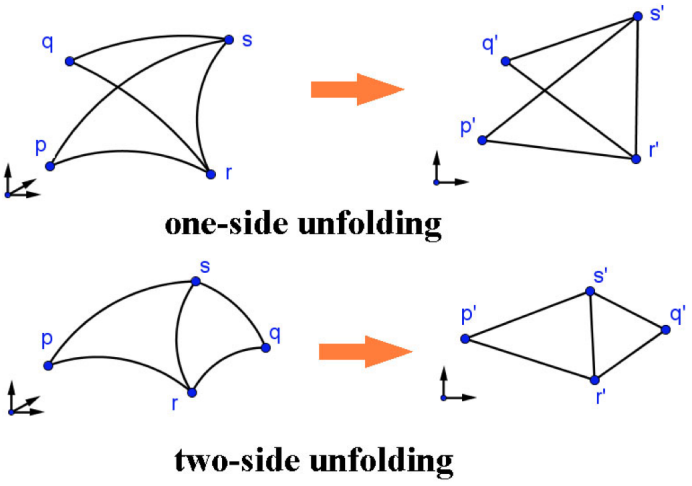
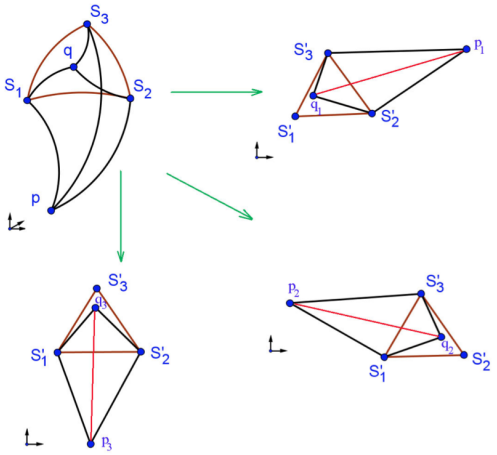


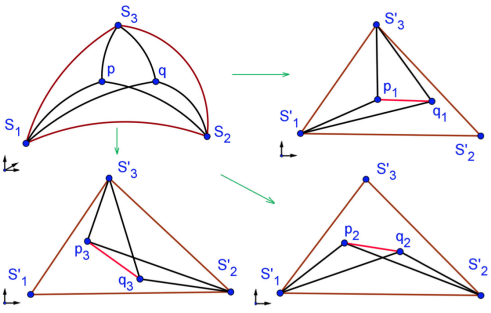
Figure 2.28: *Geodesic triangles unfolding.*

assume that given two query point p and q, p is a sample point while q is a vertex and they are in different geodesic triangles (let $\triangle s_1, s_2, s_3$ be the one containing q). Geodesic distances $d(q, s_i) i = 1, 2, 3$ are known as they were precomputed. Then, a two-side unfolding operation $u(p, q|s_i, s_{[i]+1})$ is applied to $\triangle s_1, s_2, s_3$. Finally, the approximate geodesic distance $\tilde{d}(p, q)$ is obtained as $\tilde{d}(p, q) = \min_{1 \leq i \leq 3} \|p_i q_i\|$ where p_i and q_i are the unfoldings of p and q. For the case depicted in figure 2.29 b) let us assume that $p, q \in \triangle s_1 s_2 s_3$. In this case, a one-sided unfolding with respect to each geodesic edge is applied and $\tilde{d}(p, q) = \min_{1 \leq i \leq 3} \|p_i q_i\|$. An upper bound for the approximation error is given in [14]. This is true for a query regarding two points in different triangles and states that $|d(q_1, q_2) - \tilde{d}(q_1, q_2)| \leq 2L + 2l$ where L (resp. l) is the maximum edge length of the geodesic (resp. mesh) triangles containing q_1 and q_2.

The maximum geodesic triangles edge lengths are closely related to the number m of samples. Therefore, the more samples are picked, the more accurate the geodesic distances will be obtained, but in turn as the samples number gets bigger the preprocessing time needed rises hence a good tradeoff must be found (see figure 2.30). In figure 2.31, we can see geodesic distance field computed on the Lucy model (263K vertices) for increasing values of m compared to the exact geodesic result shown on the right. In



(a) *p is a sample point, q is a vertex.*



(b) *Both p and q are vertices in the same geodesic triangle.*

Figure 2.29: *a) Two-sided unfolding of geodesic triangle $\triangle s_1, s_2, s_3$ with respect to edge $s_i s_{i+1}$ $i = 1, 2, 3$. b) One-sided unfolding.*

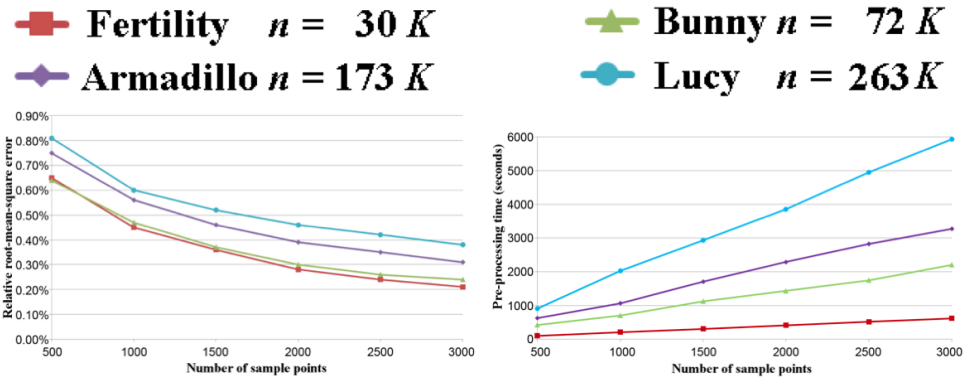


Figure 2.30: *On the left side, relative root-mean-square error VS number of sample points. On the right side, preprocessing time (seconds) VS number of sample points.*

figure 2.32, we can see the results on a smaller model (Fertility, 30K vertices). The error is given as a percentage of the model diameter. We can see how for small values of m the method has some clearly visible issues in recovering the right distance field and also for higher values of m this issues are still distinguishable. This method presents some limitations: as we can

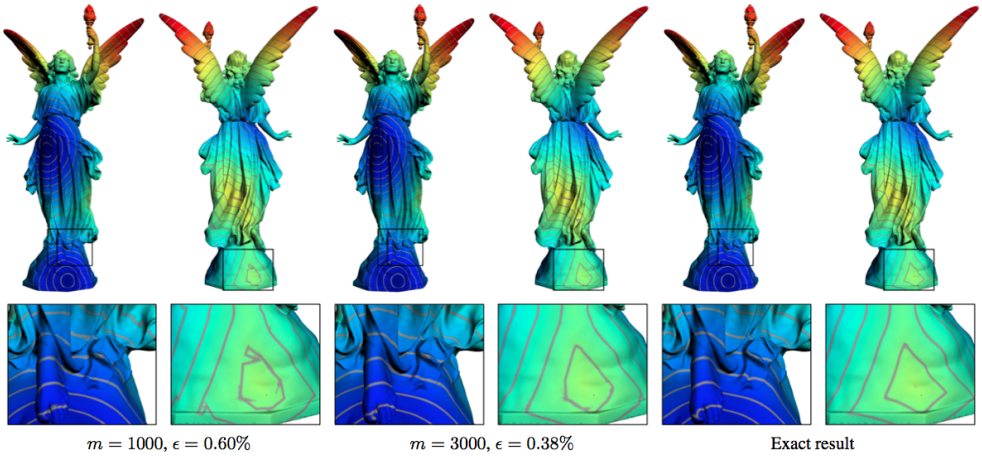


Figure 2.31: *Distance fields computed on the Lucy model (263K vertices).*

see from figure 2.33 (left) if two query points are close enough to be in the same geodesic triangle, the approximation error could be higher. Moreover, high approximation error could result from highly curved features contained into a geodesic triangle. As proposed by Xin, a solution to these problems (figure 2.33 (right)) could be to employ a finer geodesic triangulation; that is splitting $\triangle s_1, s_2, s_3$ such that the two features are in two separate geodesic triangles.

Another limitation of this method arises from the memory space requirement which is $O(m^2 + n)$: this requirement does not scale well with model dimension as m is inherently dependent on the model resolution and geometry.

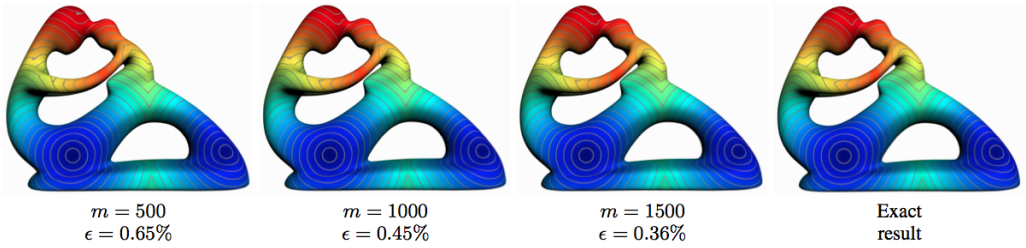


Figure 2.32: *Distance fields computed on the Fertility model (30K vertices).*

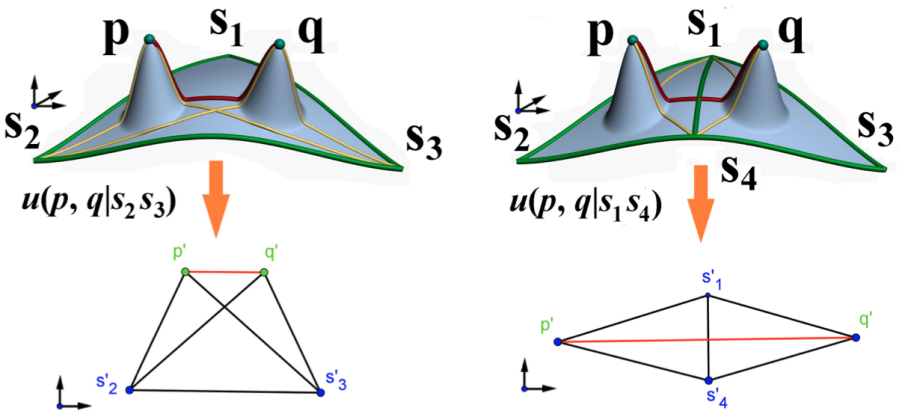


Figure 2.33: *(left) Query points located in the same geodesic triangle, unfolding could lead to high approximation error. (right) Finer geodesic triangulation could be employed to lower the approximation error.*

2.9 Comparisons

Method	Domain	MSAD	Info. reuse	Error bound	Metric
AMMP	meshes	$O(n^{1.5} \log n)$	no	yes	no
FMM	meshes & grids	$O(n \log n)$	no	no	no
HM	meshes & points	solve $\Delta\phi = \nabla \cdot X$	yes	no	no
SVG	meshes	$O(Dn \log n)$	yes	yes	yes
GTU	meshes	$O(n)$	yes	yes	no

Table 2.1: *Comparing different approximate algorithms: n is the number of vertices; D is the SVG maximum degree;*

Exact polyhedral distance:

- The seminal paper presented by Mount et al. [7] in 1987 gave birth to all the further research on geodesic computation. Different implementations of the original MMP algorithm exist, such as the one from Surazhsky [9] or Chen and Han [8] which differentiate themselves for the data structure used to manage the windows propagation. The MMP algorithm computes the exact distance with respect to the piecewise linear surface M, which in a geometry processing scenario its already an approximation of a smooth manifold. As noted in [30] the expense of employing a method like the MMP which has high time complexity becomes even more excessive when facing non standard metrics that are specified approximately (e.g. discretely per mesh element).

- Surazhsky use a priority queue while Chen employs a tree hierarchical structure. Their time performances are comparable in practice, while Chen's implementation tend to have e lower space complexity.

Approximated distance:

- Surazhsky et al. observed that the worst-case running time of the MMP algorithm is overly pessimistic and in practice the algorithm runs in sub-quadratic time. They proposed an effective way to reduce the window complexity by merging the adjacent windows without reducing the visibility region. This idea is employed in an approximated version of the MMP algorithm that applies window merging under certain conditions to improve time and space complexity.

- Chen and Han [8] (CH) adopt a different strategy, which organizes the windows in a hierarchical structure and stores only the branch nodes, leading to linear space complexity $O(n)$. Furthermore, the time complexity of their algorithm is $O(n^2)$. Later, through extensive experiments Xin and Wang [29] discovered that 99% of the propagated windows in CH algorithm do not contribute to shortest distance, and proposed an improved version of the CH algorithm (ICH).

- Campen and Kobbelt [10] proposed an extension of the FMM [11] able to deal with defected meshes obtaining some good results while still having some issues regarding undesired hole bridging that could change the model topology and result in high approximation error. The FMM is a numerical algorithm for solving the Eikonal equation on triangular meshes [4], it is widely used in the graphics community due to its simplicity and optimal time complexity. However, it requires non-obtuse triangulation to preserve the monotonicity of the wavefront propagation; otherwise, a complicated unfolding procedure must be applied.

- The Heat Method [12] represented a novel approach to the geodesic problem, using Varadhan's formula which reveals the relationship between the geodesic distance and the heat kernel. It certainly has some positive aspects, taking advantage of the well-established discrete Laplacian easily adaptable to a variety of geometric domains. Moreover, it is very easy to implement and with the pre-factored Laplacian matrix the distance can be solved in near-linear time, which

significantly outperforms the FMM in terms of speed. Visual checking of the results obtained from the HM show that it tends to smooth the sharp cusps of the isolines. This issue comes from the smooth nature of the heat kernel, however, the geodesic distance is not smooth due to the existence of the ridge points[1]. Both the HM and SVG algorithm require some precomputation: the HM pre-factors the Laplacian matrix into a lower triangular matrix and a upper triangular matrix, making it possible to solve the Laplacian and Poisson equations by backward substitution.

- The SVG algorithm precomputes a graph encoding approximate geodesic information. As exposed in [1], HM and SVG are comparable in speed when the value of the SVG's parameter K is between 100 and 500 while the HM outperforms the SVG algorithm for $K > 500$. However, for those values of K, the SVG achieves higher accuracy. Moreover, another positive aspect of the SVG method is that it resolves distance queries using Dijkstra's algorithm which does not involve any numerical computation. Finally, the HM works better on smooth surfaces since it tends to produce smooth distances, while it shows some limitations when dealing with models having richer geometric details. The SVG has the opposite limitation, producing better results on complex models.

- Compared to the GTU method, the SVG still has some advantages due to the poor-scalability of the GTU method and the lack of parallel implementations for the geodesic Delaunay triangulation on surfaces while the SVG can exploit the parallel and high computational power offered by the GPU.

- With regard to the possibilities of an extension of the presented algorithms to the anisotropic case, we have that more or less every approach has its shortcomings. Methods based on a straight Dijkstra algorithm implementation could rely on appropriate edge weights;

[1] A ridge point is a point p on the surface for which there exist at least two equal-length geodesic paths from the source s to p.

Fast Marching Methods rely on an acute triangulation of M, and this is hardly the case for practical models. An intrinsic Delaunay re-triangulation of the mesh could be applied, which is based on an intrinsic discrete metric. However this procedure and has some drawbacks such as numerical inaccuracies and a worst case quadratic time complexity. The Heat Method can be adapted to non-standard metrics by formulating it in terms of the discrete metric, i.e. based on the intrinsic edge lengths. This amounts to calculating the cotangent weights, element areas, and divergence values involved in the Laplacian and the Poisson system accordingly. The low intrinsic element roundness, however, does negatively affect robustness. The resulting distance fields then often show distortions and degeneracies like local minima.

Chapter 3

VoroGeo

In this chapter we will present our method for computing fast approximate all-pairs geodesic distances called VoroGeo. Our method is composed of two steps:

- A preprocessing step, outlined in figure 3.1
- A query step, where precomputed information is exploited to achieve fast and highly accurate reply to distance queries.

We will start our description by explaining which was the inspiration and the idea behind our work and then we will dive more into design and implementation issues that we faced during development.

3.1 Idea

As we have seen in the previous chapter, after Mitchell, Mount and Papadimitrou first exposed their algorithm in 1987, the geodesic computation problem has been tackled from various points of view and numerous approaches have been proposed over the years. After Chen and Han proved the empirical complexity of the exact solution to be $O(n^{1.5} \log n)$, a lot of the research interest has moved onto finding a good tradeoff for an approximate solution capable of achieving good time performance while producing highly accurate results. For what concerns accuracy, we have already pointed out

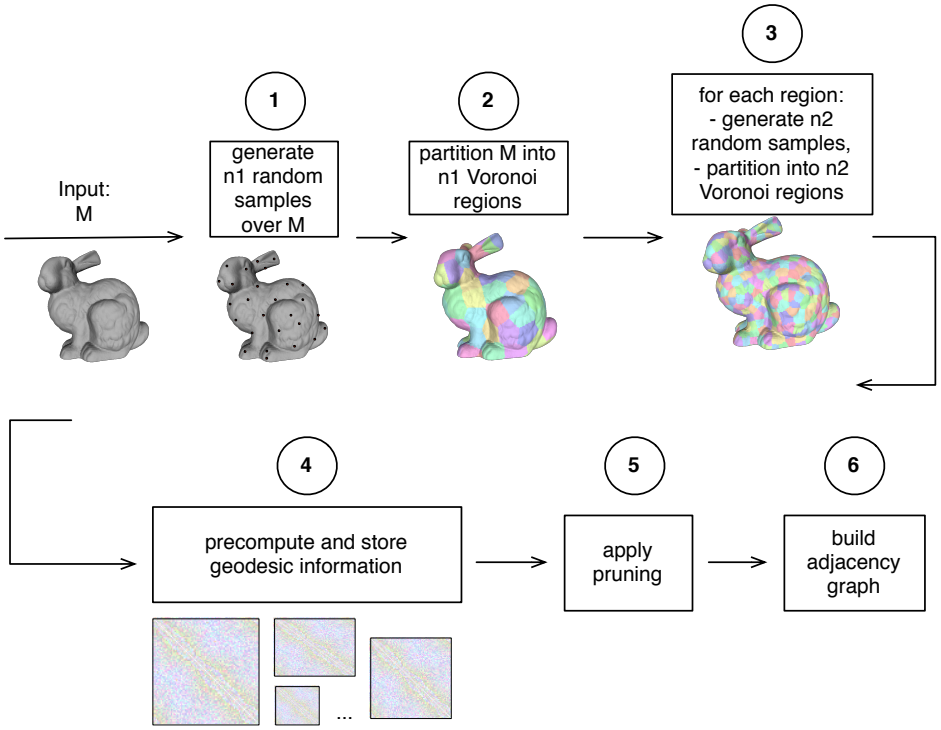


Figure 3.1: *Outline of our method's preprocessing step.*

in 2.9 that the Heat Method [12] has a limitation related to the fact that no bound is available for the approximation error while for the GTU method the upper bound for error closely relates to number m of samples chosen by the user to build the geodesic Delaunay triangulation. On the other hand, the SVG approach has a lot of advantages over the last two cited methods, but requires increasing the value of its parameter K to achieve high accuracy, which leads to a much higher precomputation time. Moreover, the SVG is not well fitted for developable surfaces or convex polyhedra as in those cases the SVG would become a complete graph.
The VoroGeo approach takes its inspiration from the same idea employed in the SVG: relating a geodesic path on the mesh to a shortest path on a graph allow us to solve the problem using a simple modification of the A* algorithm. Our goal was to design an approach for the all-pairs geodesics that could include the positive aspects of the SVG while overcoming some of its issues. Just like in the SVG or GTU methods, some precomputation is needed to be able to achieve fast reply to Single-Source-Single-Destination (SSSD) distance queries. Our approach consists of two steps: the preprocessing step and the query step:

- In the preprocessing step (see figure 3.1) we subdivide the initial problem of finding a global shortest geodesic path into smaller local subproblems. To achieve this, we employ a *double layer* partitioning of the mesh M into Voronoi regions computed from an initial set of point samples (step 1, 2 and 3).
  On each region, geodesic information is precomputed and stored for future use (step 4). We then build a pruned version of the complete graph which encodes approximate geodesic information (step 5 and 6). The pruning procedure is based on a threshold that bounds the relative average error introduced by node-pruning.

- In the query step, we reply to distance queries for any pair of vertices (p, q) by executing a modified version of the Bidirectional-A* algorithm which exploits the hierarchical partitioning applied to the mesh. The Bidirectional-A* expands two distance wave-fronts: one starting p toward q and one in the opposite direction. Intuitively, the

two fronts will meet "in the middle" of the shortest path connecting p and q.

The preprocessing step pseudocode is available in Algorithm 1 and 2. In the next sections we will dive more into the details behind those procedures.

Algorithm 1: Preprocessing step

Input: M, n_1, n_2, δ
Output: $G = (G_{fl} \cup G_{sl})$
Generate n_1 randomly placed samples on M
Compute first level partitioning: split M into n_1 patches
Parallel $\leftarrow$ **forall the** *f.l. patch* p_f **do**
    Compute Bv×Bv geodesics
    Build local Bv×Bv graph fg_{bb}
Compute second level partitioning: **forall the** *f.l. patch* p_f **do**
    Generate n_2 randomly placed samples on p_f
    Partition p_f into n_2 patches
Parallel $\leftarrow$ **forall the** *s.l. patch* p_s **do**
    Compute local Bv×Bv geodesics
    Build local Bv×Bv graph sg_{bb}
    Compute local Bv×Iv and Iv×Bv geodesics
    Build local Bv×Iv and Iv×Bv graph sg_{ib}
    Compute local Iv×Iv geodesics
    Build local Iv×Iv graph sg_{ii}
Prune(G_{fl}, δ)
$\delta_{sl} := \delta/\gamma$ **Prune**($\cup sg_{bb}$, δ_{sl})

$$G_{fl} = \bigcup_{i=0}^{n_1} fg_{bb} \qquad G_{sl} = \bigcup_{i=0}^{n_2} (sg_{bb} \cup sg_{ib} \cup sg_{ii})$$

Algorithm 2: Pruning procedure

Input: $G = (E, V)$, δ

forall the $(v, w) \in E$ *where* $\|\mathcal{P}(v)\| < 3$ *and* $\|\mathcal{P}(w)\| < 3$ *and* $\forall p \in \mathcal{P}(v) : p \in \mathcal{P}(w)$ **do**

    $e \leftarrow$ ComputeAverageError(v, w);

    $heap \leftarrow (v, w, e)$;

while !*heap.empty()* **do**

    $(i, j, c) \leftarrow heap$.back();

    $heap$.pop_back();

    if $c < \delta$ **then**

        Merge(i, j);

        $heap \leftarrow$ NewPossiblePairs();

3.2 Voronoi diagrams

Since we divide the initial problem into smaller sub-problems employing a Voronoi partitioning of the input mesh M into patches, we introduce here the basic concept of subdividing an initial domain into Voronoi regions [35]. To compute this subdivision, a set of points called *seeds* is provided as input. For each seed s there will be a corresponding region containing all the points that are closer to s than any other seed. So it is needed that the initial domain is endowed with a distance function d. In figure 3.2 we can see an example of a Voronoi diagram computed on an Euclidean plane where the density of the seeds is bigger toward the center. This produces long and skinny regions where the seed density is higher. The dual of the Voronoi diagram, the Delaunay triangulation, is the unique triangulation so that the circumsphere of every triangle contains no sites in its interior. Voronoi diagrams and Delaunay triangulations have been rediscovered or applied in many areas of mathematics and the natural sciences they are central topics in computational geometry with hundreds of papers discussing algorithms and extensions [36].

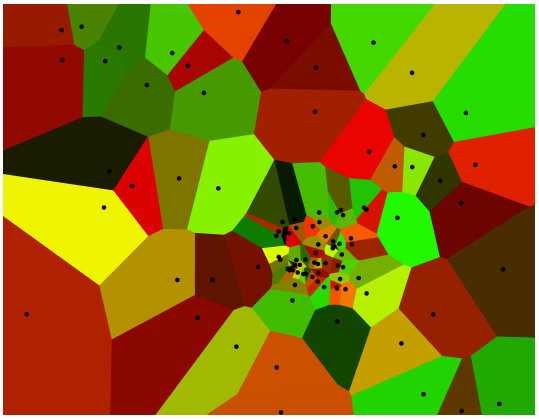


Figure 3.2: *An example of a Voronoi diagram computed on the Euclidean plane.*

3.3 Patch subdivision

From now on, we will refer to a Voronoi region as computed in the preprocessing step with the term *patch*, specifying when needed if the patch belongs to the first or second layer partitioning. On each patch we divide the vertices into two categories: *border* vertices (Bv) and *internal* vertices (Iv). In figure 3.3 is reported an example of a patch: boder vertices are colored in gray while internal vertices are in yellow. Border edges are highlighted in red.
For all patches (first and second layer), our algorithm needs to be able to establish an ordering of the so-called border vertices and internal vertices (see figure 3.3). Moreover, we shall not consider in any of these two categories those vertices that are adjacent to a hole in the mesh (see figure 3.4). We will refer to these vertices as "hole-adjacent vertices". All the hole-adjacent vertices are labeled at the start of the preprocessing step, so that they will be ignored by further computations. We classify border and internal vertices iterating over the vertices of a patch and as we find a border vertex, we label all border vertices by "walking on the border" of the patch. This *border walk* (i.e., walking on the red the border edges of figure 3.3) is very easy to implement using the *Pos* mechanism provided by the VCG library. Therefore, it is fundamental for our algorithm that each patch has a unique border ring. For example, a situation like the one shown in figure 3.5 is not

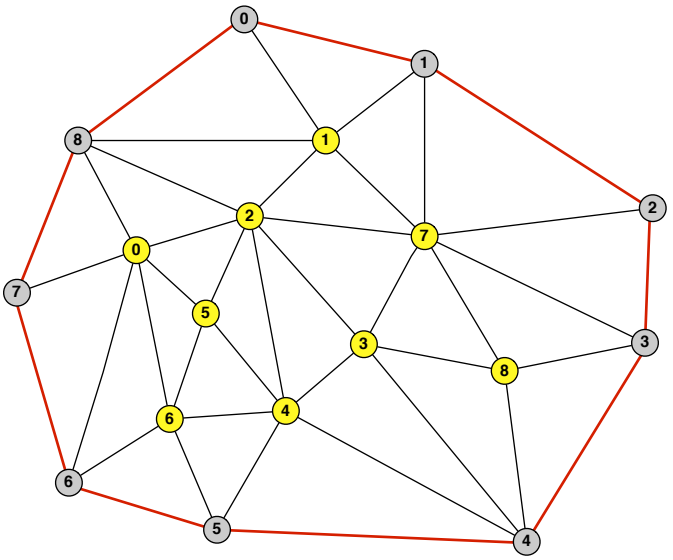


Figure 3.3: *An example of a triangulated patch. Boder vertices (Bv) are in gray, internal vertices (Iv) are in yellow.*

acceptable: we see a possible patching of the right ear of the ARMADILLO model where the yellow patch is adjacent to two other patches, colored in light green and pink. The yellow patch (shown in particular on the right of figure 3.5), will have two border rings. In general, this situation can happen when the model presents some sharp features (i.e. the ear of figure 3.5 or a tail which is also present in the ARMADILLO model). To overcome the issue, after subdividing the initial mesh into n_1 patches we check them to verify that the number n_1 of samples is enough to yield a set of patches having one border ring. If a patch with more than one border ring is found, we generate two more random samples on the faulty patch, then we split it and check if the issue is solved. Now we can easily define an ordering between the border and internal vertices (see figure 3.3).

The user-defined parameter n_1 is used to set the number of samples that will be randomly placed on the mesh to compute the first layer of Voronoi regions. Then, for each first layer patch p_f, we use another user-provided parameter n_2 to randomly place samples and partition each first layer patch p_f into smaller Voronoi regions and obtain the second layer of patches p_s.

An example of a first layer partitioning ($n_1 = 4000$) of the NEPTUNE model ($n = 268K$) is shown in figure 3.8, where first layer border vertices are visualized in green.

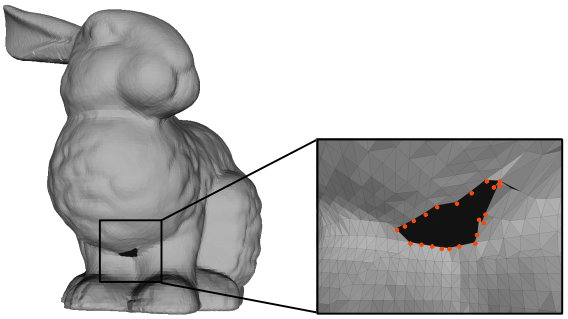


Figure 3.4: *A hole in the* BUNNY *model: vertices adjacent to the hole are shown in red.*

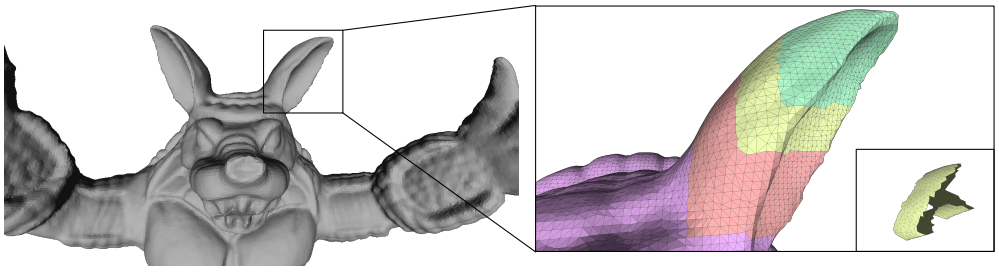


Figure 3.5: *A possible patching of the right ear of the* ARMADILLO *model. This partitioning creates a patch with two border rings.*

3.4 Geodesic precomputation

To precompute geodesic distances we employ Surazhsky [9] implementation of the MMP [7] algorithm, but in theory *any other* of the algorithms described in chapter 2 could be used in this step. We decided to use Surazhsky's algorithm because of two important reasons:

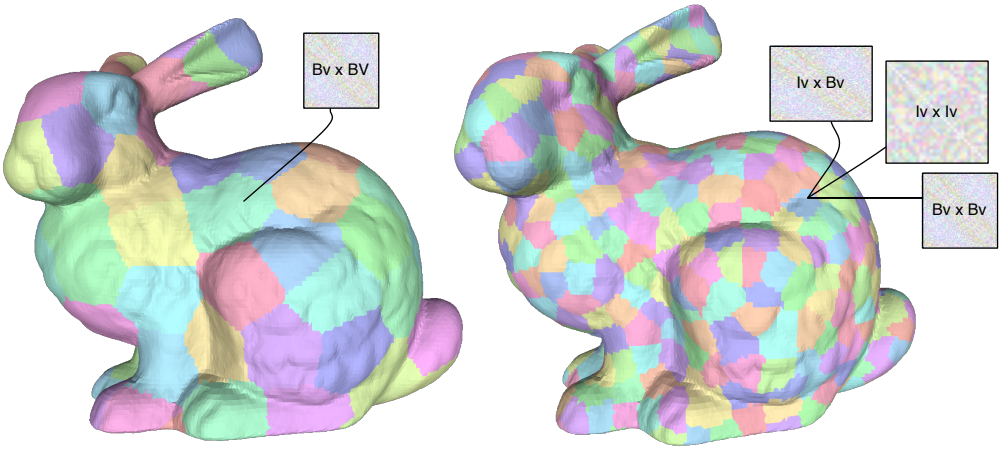


Figure 3.6: *First (left) and second (right) layer patches on the* BUNNY *model (70K-faces) computed for* $n_1 = 70$ *and* $n_2 = 5$. *Precomputed distances are stored as images.*

- The code was publicly available and easily integrable in our project[1].
- It provided us a strong comparison to evaluate the accuracy achieved by our algorithm since it computes the *exact* polyhedral distance.

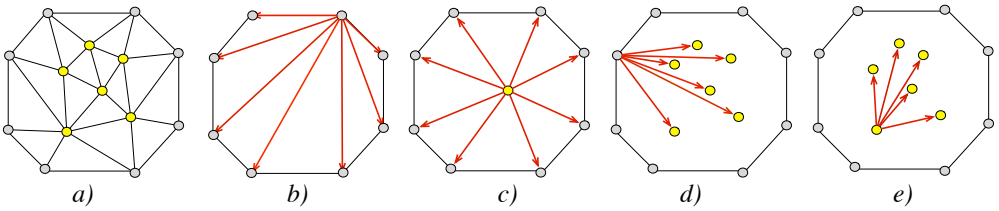


Figure 3.7: *a) A patch. b)* $Bv \times Bv$ *edges for a vertex are highlighted. c)* $Iv \times Bv$ *edges for a vertex are highlighted. d)* $Bv \times Iv$ *edges for a vertex are highlighted. e)* $Iv \times Iv$ *edges for a vertex are highlighted.*

For each first layer patch we precompute the distance from every border vertex to any other border vertex in the same patch. These edges are

[1]This cannot be said for other very interesting methods like the Heat Method, the SVG method and the GTU method. In fact, as of the time this thesis is being written, the code for those algorithms is still not publicly available

denoted as $Bv \times Bv$ (see figure 3.7 b)). Moreover, for each second layer patch we also precompute the distance from each internal vertex toward any border vertex of the same patch (hence, also the reverse path distance is obtained) and the distance from any internal vertex to any other internal vertex in the same patch. These edges are denoted as $Iv \times Bv$ (figure 3.7 c)), $Bv \times Iv$ edges (figure 3.7 d)) and $Iv \times Iv$ edges (figure 3.7 e)). As shown in figure 3.6, this globally results in:

- One $k \times k$ matrix for each first layer patch p_f (where $k = |Bv|$ for patch p_f),
- Three matrices for each second layer patch p_s:
  a) The $h \times h$ matrix containing $Bv \times Bv$ distances (where $h = |Bv|$ for patch p_s),
  b) The $h \times l$ matrix containing $Bv \times Iv$ distances (where $l = |Iv|$ for patch p_s) and
  c) The $l \times l$ matrix containing $Iv \times Iv$ distances

Of course k, h and l vary from patch to patch, depending both on which values we choose for n_1 and n_2 and how these samples are distributed over the mesh. These matrices implicitly encode the complete shortest path graph for the initial mesh M. We decided to store each of these matrix as an "*.png*" image, so that they can be easily saved to disk. Each pixel (i, j) of a certain image encodes the length of the shortest path between vertex i and j, therefore the images will be symmetric (see figure 3.6). We use *floats* to maintain distance values and the `RGBA8888` format to encode each float value into a pixel. Intuitively, assuming 32bit (4 bytes) floats, we map each of the 4 bytes of the float value into each channel of the `RGBA` image format. As we said before, the index of the row/column associated to each vertex is given by its position in the ordering we previously established. As pointed out in algorithm 1, we employ parallel computations to speed up the preprocessing step. Several threads are launched to fully exploit the parallelism degree of the underlying machine. However, referring to what stated in [1], also our algorithm would benefit from a GPU implementation

of the preprocessing step since this would produce an even higher speedup. The precomputation process has some accuracy limitations due to the non-convex nature that some patches may assume. In the case of a non-convex patch the "locally computed shortest path"[1] between two border vertices could be different from the real one. That is, it is possible that the "real shortest path"[2] could meander outside of the patch. Thus, some inaccuracy could be introduced. However, our experiments (see chapter 4) show that this is negligible compared to the cost of running the MMP algorithm over the entire mesh. In fact, this would imply in turn both an "explosion" concerning the space requirement (having a separate copy of the initial mesh for each thread) and the time complexity of the preprocessing step.

3.5 Graph pruning

The pruning procedure (see algorithm 2) is employed to lower the density of the complete graph, that is, the graph composed by all the adjacency matrices computed as described in the previous section. While theoretically we could use the complete version of the graph in the query step, this would lead to slower reply time due to the huge number of edges contained in the complete graph.

The pruning procedure is outlined in figure 3.9: we see an example of a patch[3] and the two nodes highlighted in red have been selected for pruning. In figure 3.9 a) their $Bv \times Bv$ edges are colored in blue and green respectively. The output of the pruning (figure 3.9 b)) of two nodes is a dummy node that will have the same $Bv \times Bv$ edges.

This pruning procedure is based on the idea that if two vertices are geometrically close, their $Bv \times Bv$ edge weights will be "similar" (that is, the

[1] With the expression "locally computed shortest path" we refer to the computation of the shortest path between the two points made considering only the current patch vertices.

[2] With the term "real shortest path" we refer to the computation of the shortest path between the two vertices made considering the whole mesh.

[3] Since only border vertices are involved in pruning, just the border vertices/edges of the patch are depicted.

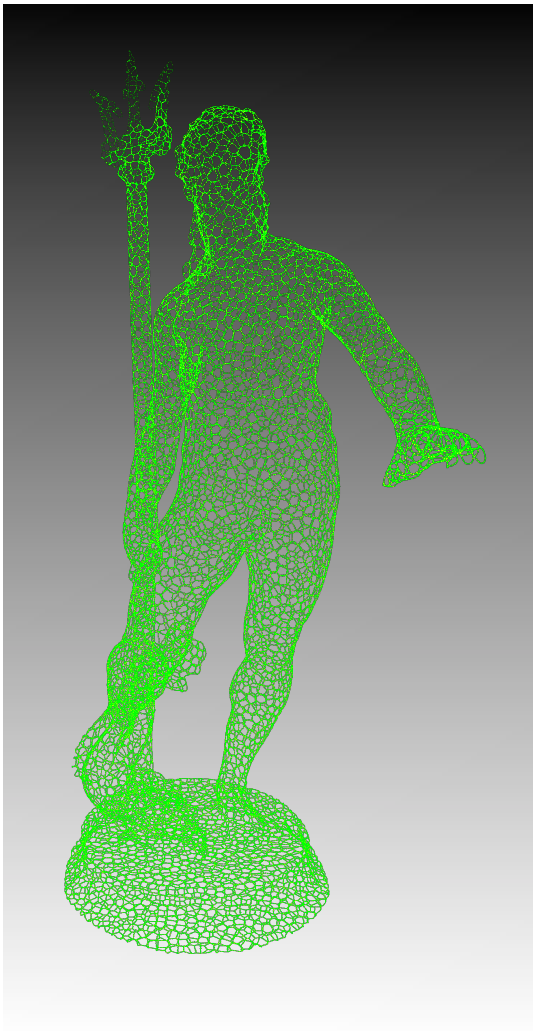


Figure 3.8: *First level partitioning on the* NEPTUNE *model (4M-faces). First level patches border vertices are depicted in green.*

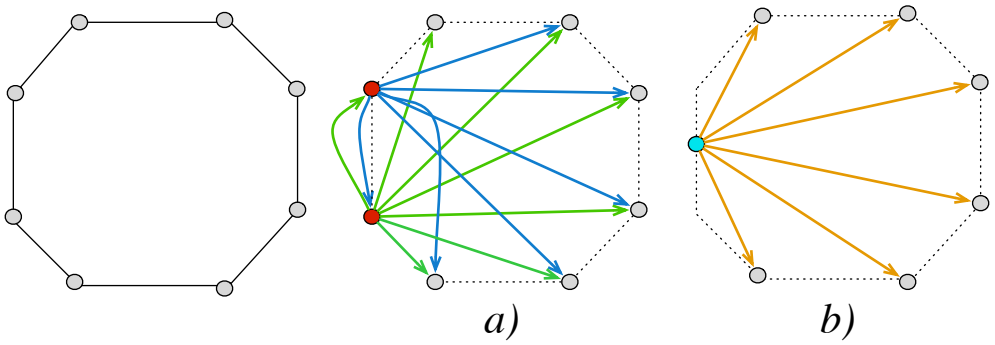


Figure 3.9: *Pruning of two nodes: a) Two red vertices are selected for pruning. b) Pruning produces a new dummy node.*

absolute value of their difference will be near to zero). Therefore it makes sense to simplify the complexity of the graph introducing a new dummy node that acts as a placeholder for the pruned nodes and has as weights the averages of the pruned nodes' weights.
Results have shown that thanks to the *ad hoc* version of the bidirectional A* search algorithm we designed (see section 3.7), the average query time is still quite low (see results in chapter 4) for small meshes (70K-200K faces) even if pruning is not applied. However, for bigger models, the pruning procedure is mandatory to obtain fast query replies.

3.6 Graph Pruning: details

We apply pruning to the first and second layer $Bv \times Bv$ graphs: this is done by

- Pruning the graph composed by all the local first layer $Bv \times Bv$ graphs (we will refer to this graph as the "global" $Bv \times Bv$ graph),
- Then the same pruning procedure is applied to the graph composed by all the local second layer $Bv \times Bv$ graphs.

As we said before, the pruning procedure is driven by a threshold value that we will indicate with δ. For readability reasons, we will describe the pruning

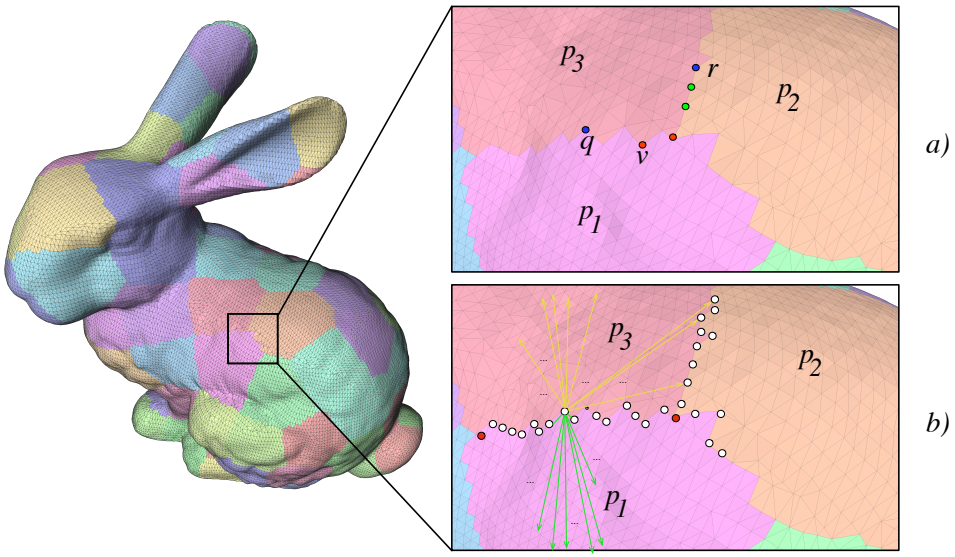


Figure 3.10: *a) Example of a pair of nodes eligible for pruning (green) and two pairs not eligible (blue and red). b) Example of a border vertex and its $Bv \times Bv$ edges in both the patches it is contained. For visibility reasons, edges are only sketched.*

procedure as applied to the first layer global $Bv \times Bv$ graph, but the same reasoning can be applied to the case of pruning the global $Bv \times Bv$ second layer graph. The only difference being that the threshold value δ_{sl} used, is obtained as

$$\delta_{sl} = \delta/\gamma \tag{3.1}$$

where γ is the average number of second layer patches created for each first layer patch. Moreover, in this section we will use the term 'vertex' and 'node' in an interchangeable way, referring in the first case to a patch border vertex: clearly, both terms refer to the same entity since there is a precise correspondence between a mesh vertex and a graph node. The pruning procedure uses a global heap filled with tuples of the form $\langle u, v, e \rangle$, where u and v are two different nodes and e encodes the mean average error we would introduce by *merging* those two nodes together into a new one that we will refer to as "*average node*". To compute e for a pair of nodes (u, v), we iterate over their outgoing edges and we evaluate the mean average error e as

$$e = \frac{\sum_{i=0}^{k} |w_{u,i} - w_{v,i}|}{edges\_num} \tag{3.2}$$

where $w_{u,i}$ (resp. $w_{v,i}$) is the weight associated to edge i at node u (resp. v). We indicate with k the total number of edges in the outgoing *star* $\mathcal{S}(u)$ or $\mathcal{S}(v)$[1], while *edges_num* indicates the number of edges that are still "active". To better explain how the pruning procedure works, let us consider figure 3.10 a): in this figure we see depicted the case of two nodes that are eligible for pruning. In fact, nodes u and v both belong to patches p_1 and p_2, and they both belong to only two patches. This condition is expressed in algorithm 2 as

$$\|\mathcal{P}(v)\| < 3 \quad and \quad \|\mathcal{P}(w)\| < 3 \quad and \quad \forall p \in \mathcal{P}(v): \quad p \in \mathcal{P}(w)$$

where $\mathcal{P}$ indicates the set of patches that contain that vertex. Thanks to the Voronoi partitioning we employ, a certain vertex can only be contained in the border of a maximum of three patches. Moreover, we exploit the previously established ordering between the nodes of a patch to define an ordering between the edges in the outgoing star of a node. To initialize the heap with all (initial) possible pairs of nodes that could be simplified, we iterate over all the first layer patches considering only pairs of eligible neighbors nodes (that is, border vertices that are connected by an edge on the patch). As shown in figure 3.10 b), when we evaluate the mean average error, we consider also the $Bv \times Bv$ edges that are not from the currently considered patch. For example, if we are adding to the heap all possible initial pairs from patch p_1, both yellow and green edges will be considered. In figure 3.10 a) we also picked two cases of node pairs that are not eligible for pruning: the pair highlighted in red is composed by node v and a so-called "intersection node", that is, a node that is belongs to the border of three different patches (p_1, p_2, and p_3); the pair highlighted in blue is made

[1]As will be clear later in this section, the size of the outgoing star is always equal for every pair of nodes added to the heap. That is, it always holds that $|\mathcal{S}(u)| = |\mathcal{S}(v)|$ $\forall \langle u, v, e \rangle$ added to the heap.

of two nodes that are contained in different patches: in fact we have

$$\mathcal{P}(q) = \{p_1, p_3\} \qquad \mathcal{P}(r) = \{p_3, p_2\}$$

Therefore it would have no sense to try and evaluate the error using equation 3.2. When a tuple $\langle u, v, e \rangle$ extracted from the heap is such that the average mean error $e < t$, we generate a *dummy* vertex u_1 that will replace u in the ordering of all patches where u is present, then we initialize its edges. This is implemented by overwriting u's row in the relative image with the new values. We also update v's row assigning $+\infty$ to each edge, marking those edges as "non active". Finally, we update distances *toward* u and v in order to maintain symmetry in the image which will in turn assure that the metric symmetry property (that is, $d(x, y) = d(y, x)$) holds. Edge weights for the average node u_1 are set using equation 3.3a while weights for the remaining nodes are updated iterating on the relative image row and applying equation 3.3b:

$$w_{u_1 k} = \begin{cases} 0 & \text{if } k = i \text{ or } k = j \\ +\infty & \text{if } w_{u_k} = +\infty \text{ or } w_{v_k} = +\infty \\ \frac{w_{u_k} + w_{v_k}}{2} & \text{otherwise} \end{cases} \tag{3.3a}$$

$$w_{x_l} = \begin{cases} w_{u_1 i} & \text{if } l = i \\ +\infty & \text{if } l = j \\ w_{x_l} & \text{otherwise} \end{cases} \tag{3.3b}$$

where i (resp. j) is the index of vertex u (resp v) in the vertex ordering. When an average node u_1 is generated and after all edge weights have been updated, we add to the heap two new possible pairs $\langle q, u_1, e' \rangle$ and $\langle u_1, p, e'' \rangle$, where p and q where respectively u's and v's neighbors. This is the reason why we use the *edges_num* counter in equation 3.2: we need not to consider edges having weight $w = +\infty$: those edges have been marked as *non active* from a previous pruning and they should not be considered anymore.
In practice, we express the threshold δ used during pruning as a percentage of the mesh bounding box diagonal. Results have shown that on small

meshes (such as the 70K-faces BUNNY model), very small values of δ such as $\delta = 0,01\%$, while producing very low edge pruning (around the 2% of edges are pruned), are already enough to obtain fast and accurate result in the query step. For high resolution meshes, higher values of δ are necessary to achieve faster query reply. We refer the reader to chapter 4 for a more ample discussion on how varying the value for threshold δ affects our algorithm's time performance and accuracy.

3.7 Query step

In the query step we exploit all the geodesic information gathered during the preprocessing step to perform Single-Source-Single-Destination (SSSD) or Multiple-Source-All-Destinations (MSAD) fast distance computations that can be used (as exposed in section 3.8) for example to compute a geodesic Voronoi partitioning of the initial mesh.

- The SSSD geodesic computation is based on a modified version of the Bidirectional-A* search algorithm, outlined in algorithm 3.
- The MSAD geodesic computation is based on a generalization of algorithm 3 that computes distance values, from a set of source vertices, for all other mesh vertices.

3.7.1 SSSD distance computation

The crucial point of algorithm 3 resides in how we expand the information front from a certain vertex extracted from the heap. The main goal is to try to examine the minimum number of edges necessary to be able to find the shortest path. To do this, we exploit the hierarchical structure we defined in the preprocessing step. Given two random vertices v_1 and v_2, we retrieve the first and second layer patches in which they are contained. If the two points are in the same second layer patch, the distance is returned with one access to the relative image pixel. Otherwise, we start bidirectional searching, that is from v_1 toward v_2 and from v_2 toward v_1. We use two priority queues to implement the fronts. The key point of the A* algorithm

is that it uses a knowledge-plus-heuristic cost function of node v (denoted with $f(v)$) to determine the order in which the search visits nodes in the graph. The cost function $f(v)$ is a sum of two functions:

- the *past* path-cost function, which is the known distance from the starting node to the current node v (denoted with $g(v)$)
- a *future* path-cost function, which is an admissible "heuristic estimate" of the distance from v to the goal (denoted with $h(v)$).

The $h(v)$ part of the $f(v)$ function must be an admissible heuristic; that is, it must not overestimate the distance to the goal. In our case, it is the Euclidean distance to the destination vertex. Therefore we obtain an heuristic that is monotone (i.e. satisfies $h(v) \leq d(v,u) + h(u)$ for every edge (v,y)) and moreover it is a lower bound. Indeed, the length of the shortest path between two points cannot be shorter than the norm of the vector connecting them. This allow us to implement the algorithm in a more efficient way since no node has to be considered more than once. As exposed in [37] [38] A* algorithm achieves better time performances when using admissible monotone heuristics. To describe our implementation, let us consider two arbitrary vertices v_1 and v_2. For the non-trivial cases we will consider the front expansion only in one direction, that is from v_1 toward v_2, the same reasoning can be applied to the front expanding in the opposite direction.

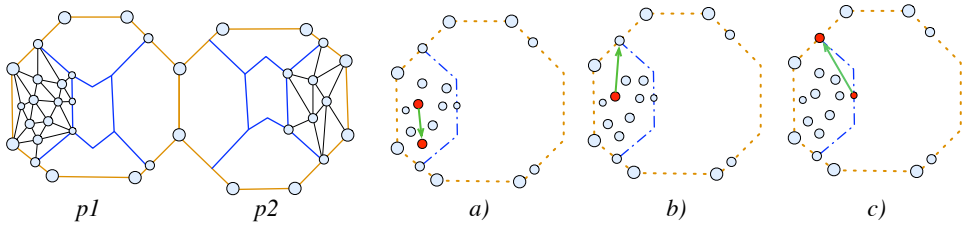


Figure 3.11: *Vertices in the same second layer patch.*

- If v_1 and v_2 are in the same second layer patch p_{sl}, we immediately return their distance (only one access to a distance matrix is needed). In fact, this is contained in

  a) the $Iv \times Iv$ matrix, if both v_1 and v_2 are internal vertices for p_{sl} (figure 3.11 a),

b) in the $Iv \times Bv$ matrix, if one of them is a border vertex for p_{sl} (figure 3.11 b),

c) in the $Bv \times Bv$ matrix, if if both of them are border vertices for p_{sl} (figure 3.11 c).

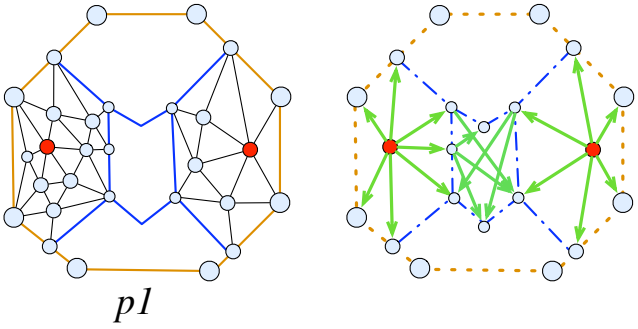


Figure 3.12: *Vertices in the same first layer patch.*

- If v_1 and v_2 are in the same first layer patch, but in different second layer patches (figure 3.12):

  a) if v_1 is an internal vertex for its second layer patch p_{sl_1} we first expand the front using $Iv \times Bv$ edges "to get out" of p_{sl_1}.

  b) otherwise we expand the front from v_1's second layer patch p_{sl_1} considering only second layer $Bv \times Bv$ edges until we reach v_2's second layer patch p_{sl_2}. If v_2 is a border vertex for p_{sl_2} then we will reach v_2 using $Bv \times Bv$ edges, otherwise when p_{sl_2} is reached we will start "going down" into p_{sl_2} considering only $Bv \times Iv$ edges for p_{sl_2}.

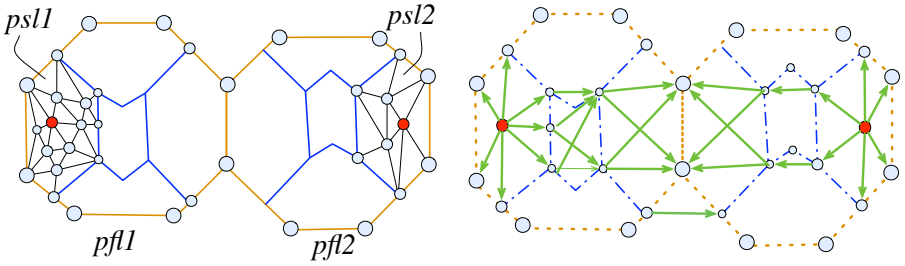


Figure 3.13: *Vertices in different first layer patches.*

- The most general case is represented in figure 3.13: v_1 and v_2 belong to different first level patches p_{fl_1} and p_{fl_2} (and of course different second layer patches p_{sl_1} and p_{sl_2}): the key idea is always to expand the front on edges "going up" to exit p_{sl_1}, then move using second layer $Bv \times Bv$ edges to reach the border of p_{fl_1}. Then we speed up the search by considering first layer $Bv \times Bv$ edges until we reach p_{fl_2}. Intuitively, first layer edges allow to perform "bigger jumps" on the graph, speeding up the search. We can then start "going down" using second layer $Bv \times Bv$ edges until we get to p_{sl_2} where we will reach v_2 directly by a $Bv \times Bv$ edge of by p_{sl_2}'s $Bv \times Iv$ edges.

The two parallel fronts expanding from both the source and the destination will meet "in the middle" of their path, but this event does not give us the certainty that we have found the shortest path. Let us consider the situation where we are expanding the front originated at the source[1] and we are examining a vertex u that has already been visited by the opposite moving front: we can update the *estimated length* of path $v_1 \to v_2$ as

$$est := d_{fw}(v_1, u) + d_{bw}(u, v_2)$$

where $d_{fw}(v_1, u)$ (resp. $d_{bw}(u, v_1)$) is the current length of the shortest path for u as computed by the expanding front moving forward (resp. backward) from the source (resp. destination). In this way est will keep track of the length of the currently best-known path from v_1 to v_2. Since we defined an admissible lower bound monotone heuristic $\hat{d}(x, y)$, we can exploit the halting condition expressed in algorithm 3: that is, we stop searching for a shorter path when the nodes x and y at the top of the priority queues implementing the fronts have current distance $d(v_1, x) + d(y, v_2) \geq est$. Finally, the current value of est is returned.

3.7.2 MSAD distance computation

The Multiple-Source-All-Destinations distance computation algorithm is a generalization of the approach described in the previous section. Naturally,

[1] Without losing generality, the same reasoning can be symmetrically applied to the front expanding from the destination.

Algorithm 3: Ad-hoc Bidirectional A*

```
Input: v1, v2
Output: d(v1, v2)
p_start := v1.SLPatch();      q_start := v1.FLPatch();
p_end   := v2.SLPatch();      q_end   := v2.FLPatch();
if (p_start = p_end) then
    return p_start.Distance(v1, v2);
else
    estimate := +∞;
    heap_fw.Insert(v1,0);
    heap_bw.Insert(v2,0);
    while (!front1.empty() and !front2.empty()) do
        if (front1.top() + front2.top() ≥ estimate) then
            break;
        curr := min(heap_fw.top(), heap_bw.top());
        Expand distance front from curr
        if (better estimate found) then
            estimate := new_estimate;
    return estimate;
```

it tries to take advantage of the precomputed information to speed up the computation. Given a set $\mathcal{S}$ of sources, we retrieve their second layer patch $p_{sl_s}, s \in \mathcal{S}$. Without loss of generality, let us assume that all p_{sl_s} are different. For each $s \in \mathcal{S}$ we compute distances for all the internal vertices of all p_{sl_s} using

- $Bv \times Iv$ edges if s is a border vertex for p_{sl_s},
- otherwise we exploit the $Iv \times Iv$ edges.

For implementing the front expansion we use a priority queue. The front is initialized with all the border vertices of all the patches p_{sl_s} (for which distance can be trivially computed from the relative source s by using either $Iv \times Bv$ edges or $Bv \times Bv$ edges). The rest of the algorithm consists of expanding the front by considering first and second layer $Bv \times Bv$ edges. All the internal vertices of all the second layer patches are reached using $Bv \times Iv$ edges and are not added to the heap for further expansion. Examples of a MSAD distance computation are reported in chapter 4.

3.8 Enhanced Voronoi partitioning

In this section we propose an application of our method to speedup the geodesic Voronoi partitioning of a mesh. The method is driven by a parameter k provided by the user, setting the number of seeds to be placed on the mesh. Our method is very easy to implement and employs the generation of a sphere centered in each seed. Then we try to assign each point to a specific seed. We iterate increasing the sphere radius until each vertex has been assigned to its seed. We use a Poisson distribution to generate random points over the mesh. This distribution guarantees that for each seed s no other seed is present inside a certain radius r of a disk centered in s. The Poisson distribution implemented in the VCG library produces as output a set of points on the mesh surface. For each point, we pick the vertex that is closer to that point. Initially we set the radius of the spheres to $r/2$. Moreover, we mark all vertices as "not-assigned". We generate the spheres and loop over the unassigned vertices keeping track, for each vertex v, of the list

of "candidates" (that is, seeds) to which it may belong. Three situations could take place:

i) v is included in only one sphere: there is only one candidate c, we compute $d(v, c)$ using our SSSD distance algorithm. Remove v from the "not-assigned" list.

ii) v is included in more than one sphere: there is a list $\mathcal{C} = c_0, c_1, \ldots c_l$ of candidates. We check $d_i(v, c_i)$ for $i = 0 \ldots l$ using our SSSD distance search. For candidate c_0 we use the SSSD distance search normally. When we have to compute $d_{i+1}(v, c_{i+1})$ we use d_i as a maximum distance threshold for the expansion of the A* front. Hence, we stop the search as soon as we find that $d_{i+1} > d_i$, without completing the computation. Remove v from the "not-assigned" list.

iii) v is not included in any sphere: v remains in the "not-assigned" list.

If the list of "not-assigned" vertices is not empty, we increase the the sphere radius by a factor which can be set by the user (during our tests, we used 0.25 as value for the increment) and proceed to check again iterating on the vertices that were left unassigned from the past attempt.

Chapter 4

Results

In this chapter we will present some statistics about time performance and accuracy of our method. First we will describe which parameters can be provided by the user and how these can affect our algorithm's behavior and performances.
We developed our algorithm in C++ [39] using the Qt [40] framework and exploiting the functionalities exposed by the Visualization and Computer Graphics (VCG) library. The VCG library is a open source portable C++ templated library for manipulation, processing and displaying of triangle and tetrahedral meshes. It is the base of most of the software tools of the Visual Computing Lab of the Italian National Research Council Institute (ISTI). All the tests where performed on a machine equipped with a 2.6GHz i7 8cores processor.

4.1 Parameters tuning

We made various experiments trying to investigate how the performances of our algorithm change with respect to variations in the method's input parameters. Our method does not have a lot of parameters, but a proper tweaking of n_1, n_2 and δ is very important to get the best results. Hence, we first tried tweaking n_1 and n_2 which are the parameters that regulate the

number of regions created. Then, after we found values for n_1 and n_2 that yield good reply time and accuracy, we made some tests oriented on tweaking the pruning threshold parameter δ. We found that for small models (in the range of 50K to 100K-faces) $\delta \in [0.0001, 0.0005]$ is enough to obtain fast query reply. For bigger models (i.e. with more than 500K faces) an higher pruning threshold needs to be used to produce an high percentage of nodes pruning while still obtaining fairly accurate results.

4.1.1 Tweaking n_1 and n_2

Some statistics relative to the tests we performed for tweaking the n_1 and n_2 parameters values for the ARMADILLO ($350K$-faces), BUSTO (500K-faces) and RAMESSES (1.6M-faces) models are reported in table 4.1, 4.2 and 4.3. Different values for n_1 and n_2 result in a different number of regions with different sizes and shapes. Thus, other than the time complexity of the preprocessing step, n_1 and n_2 also influence our pruning procedure capacity to simplify the graph. Analyzing the results we can see that, on average:

- For models like the ARMADILLO, which presents rich geometrical features, an higher density of first layer samples is required. Faster and more accurate queries are achieved setting $n_1 = 100$, $n_2 = 5$.
- For models that do not present sharp features (i.e. BUSTO and RAMESSES models), a lower density of first layer samples yields faster and more accurate results: our tests proved that the best trade-off between speed and accuracy is achieved by setting $n_1 = 150, n_2 = 5$ and $n_1 = 1000$, $n_2 = 5$ respectively.

Moreover, in table 4.4 and 4.5, we reported the preprocessing statistics relative to the previous tests performed for different values of n_1 and n_2. We can notice that, thanks to our parallel implementation of all the geodesics precomputations made on each patch, the preprocessing time decreases as we increase the total number of first and second layer regions. On the other hand, as we have seen, this does not imply a consequent decrease in

	n_1		
n_2	100	200	300
5	0.01126s	0.01128s	0.01145s
10	0.01134s	0.01137s	0.01176s
15	0.01166s	0.01164s	0.01215s

	n_1		
n_2	100	200	300
5	0.00089	0.00099	0.00187
10	0.00105	0.00169	0.00209
15	0.00136	0.00204	0.00251

Table 4.1: *Average query time (left) and mean average error (right) statistics for different n_1 and n_2 values on the* ARMADILLO *model* $(n = 173K)$.

	n_1		
n_2	100	150	200
5	0.01989s	0.01979s	0.01978s
10	0.01983s	0.01983s	0.01986s
15	0.01986s	0.01990s	0.01991s

	n_1		
n_2	100	150	200
5	0.00807	0.00143	0.00275
10	0.00773	0.00160	0.00293
15	0.007359	0.00177	0.00313

Table 4.2: *Average query time (left) and mean average error (right) statistics for different n_1 and n_2 values on the* BUSTO *model* $(n = 255K)$.

query time. Notice that, as proven by our tests, the increase in first layer regions forces the A* search algorithm (see section 3.7.1) to spend more time expanding the front through second layer patches before propagating across first layer edges.

4.1.2 Tweaking δ

In table 4.6, 4.7 and 4.8 we report statistics relative to different tests where we set the values for n_1 and n_2 (choosing those that lead to the most satisfying results, as explained in the previous section) and tried varying the pruning threshold δ in the range $[0, 0.002]$. We express δ as a percentage of the mesh bounding box diagonal. We report both the average query time T (in seconds) and the mean average error ϵ. Moreover, we also report the percentage of first and second layer nodes pruned, denoted by σ_{fl} and σ_{sl}. For testing our algorithm time performance and accuracy, we picked $n = 10$K random pairs of vertices (u, v) on each model and proceeded to

	n_1		
n_2	950	1000	1200
5	0.06295s	0.06088s	0.06112s
10	0.06335s	0.06027s	0.06202s
15	0.65986s	0.06077s	0.06243s

	n_1		
n_2	950	1000	1200
5	0.00763	0.00760	0.00829
10	0.00774	0.00786	0.00843
15	0.007989	0.00781	0.00856

Table 4.3: *Average query time (left) and mean average error (right) statistics for different n_1 and n_2 values on the* RAMESSES *model* ($n = 826K$).

	n_1		
n_2	100	200	300
5	333.4s	218.7s	182.2s
10	287.3s	197.5s	202.1s
15	264.2s	191.1s	171.5s

Table 4.4: *Preprocessing time for different values of n_1 and n_2 on the* ARMADILLO *model.*

estimate the mean average error as:

$$\epsilon = (\sum_{i=0}^{n} \frac{|\hat{d}(u,v) - d(u,v)|}{\hat{d}(u,v)})/n$$

Where $\hat{d}(u,v)$ is the exact polyhedral distance as computed by the MMP algorithm.

As shown in figure 4.1, the average query time quickly decreases as we increase the value of δ. Intuitively, as δ increases, more nodes will be selected for pruning by our procedure, therefore also the number of edges that our Bidirectional-A* has to consider while expanding the fronts decreases. Moreover, in figure 4.2, we report the growth of the mean average error ϵ with respect to an increasing value for δ. We can see that, while for $\delta \leq 0.0005$ it remains quite stable and low around 10^{-3}, for $\delta > 0.0005$ it grows with exponential speed. This is also due to the approximation introduced by merging two nodes and producing "average edges" as exposed in section 3.6.

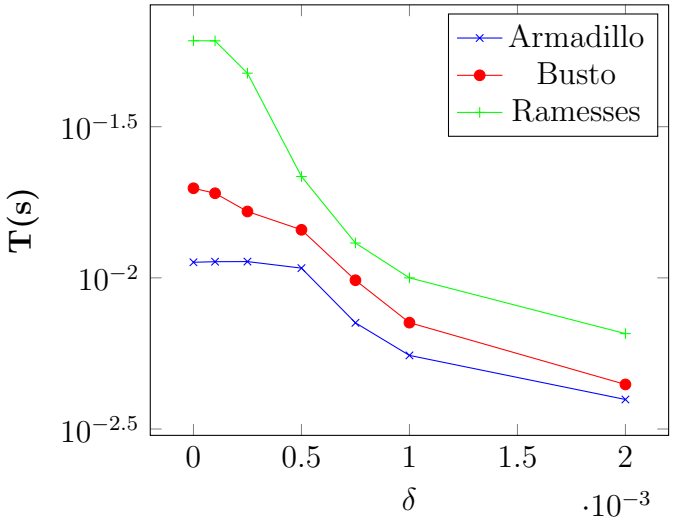


Figure 4.1: *Average query time $T(s)$ plotted for variable pruning threshold δ.*

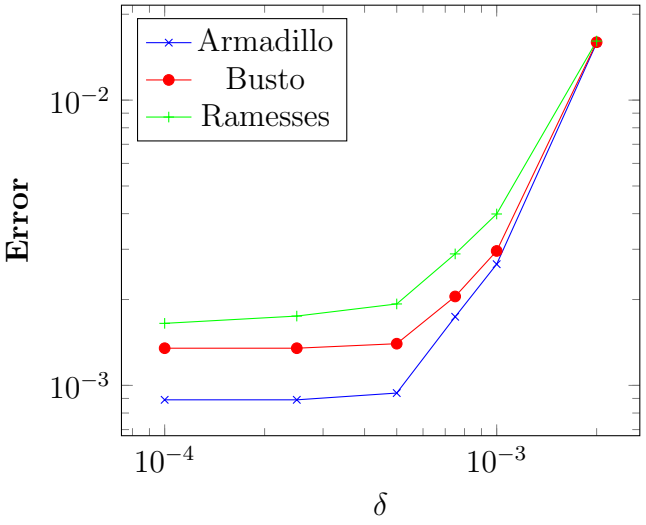


Figure 4.2: *Average mean error ϵ plotted for variable pruning threshold δ.*

	n_1		
n_2	100	150	300
5	1062.2s	794.3s	500.5s
10	855.1s	693.7s	484.1s
15	794.3s	602.5s	470.5s

	n_1		
n_2	950	1000	1200
5	1512s	1568s	1419s
10	1356s	1536s	1343s
15	1316s	1456s	1551s

Table 4.5: *Preprocessing time for different values of n_1 and n_2 on the* BUSTO *(left) and* RAMESSES *(right) models.*

δ	$T(s)$	ϵ	σ_{fl}	σ_{sl}
0.0%	0.01126s	0.00089	0.0%	0.0%
0.0001%	0.01131s	0.00089	0.18%	0.005%
0.00025%	0.01132s	0.00089	0.18%	0.005%
0.0005%	0.01077s	0.00094	8.04%	0.005%
0.00075%	0.00710s	0.00174	53.14%	0.005%
0.001%	0.00554s	0.00266	69.3%	0.01%
0.002%	0.00396s	0.01584	84.38%	0.03%

Table 4.6: *Average time $T(s)$ and mean relative error ϵ for different pruning threshold values on the* ARMADILLO *model ($n = 173K$).*

4.2 Speedup and accuracy

We now present some results of the computation of MSAD distance on different models (distance isolines are visualized). Our results are compared to those obtained by applying the MMP algorithm: in the top row of figure 4.3, 4.4 and 4.5 we reported the distances computed by our method. Results from MMP algorithm are in the bottom row. We can see that our method produces smooth distances. Moreover, checking the isolines, no artifacts are visible. The accuracy achieved makes our results hardly distinguishable from those computed by the MMP algorithm.
In table 4.9, we report the effective speedup achieved by our algorithm with respect to the geodesic algorithm employed in the VCG-Library. This algorithm implements an enhanced version of a basic Dijkstra's search, over the mesh. Despite its simplicity, our tests have proven that it achieves decent accuracy, providing a strong comparison for our experiments.
Due to the huge speedup provided by our method, we tried applying the

δ	$T(s)$	ϵ	σ_{fl}	σ_{sl}
0.0%	0.01979s	0.00143	0.0%	0.0%
0.0001%	0.01905s	0.00145	8.13%	0.54%
0.00025%	0.01658s	0.00145	21.33%	1.31%
0.0005%	0.01442s	0.00150	36.85%	2.59%
0.00075%	0.00982s	0.00225	60.79%	3.92%
0.001%	0.00711s	0.00326	74.05%	5.31%
0.002%	0.00444	0.01336	85.96%	10.86%

Table 4.7: *Average time $T(s)$ and mean relative error ϵ for different pruning threshold values on the* BUSTO *model* ($n = 255K$).

δ	$T(s)$	ϵ	σ_{fl}	σ_{sl}
0.0%	0.06088s	0.00760	0.0%	0.0%
0.0001%	0.06082s	0.00760	0.37%	0.0%
0.00025%	0.04754s	0.00773	27.13%	0.007%
0.0005%	0.02164s	0.00913	69.65%	0.01%
0.00075%	0.01304s	0.01209	80.79%	0.05%
0.001%	0.01s	0.01579	84.37%	0.13%
0.002%	0.00654s	0.02381	88.13%	4.55%

Table 4.8: *Average time $T(s)$ and mean relative error ϵ for different pruning threshold values on the* RAMESSES *model* ($n = 826K$).

same tests to a decimated versions of the models, to check the amount of mesh simplification needed for the VCG geodesic algorithm to be as fast as our method. Through several experiments we have seen that, applying the VCG geodesic algorithm to a decimated[1] version of the mesh, time performances became comparable. However, there is an exponential decrease in accuracy, which makes those results useless in practice.

[1]The number of faces of the mesh is reduced by a factor proportional to the original speedup.

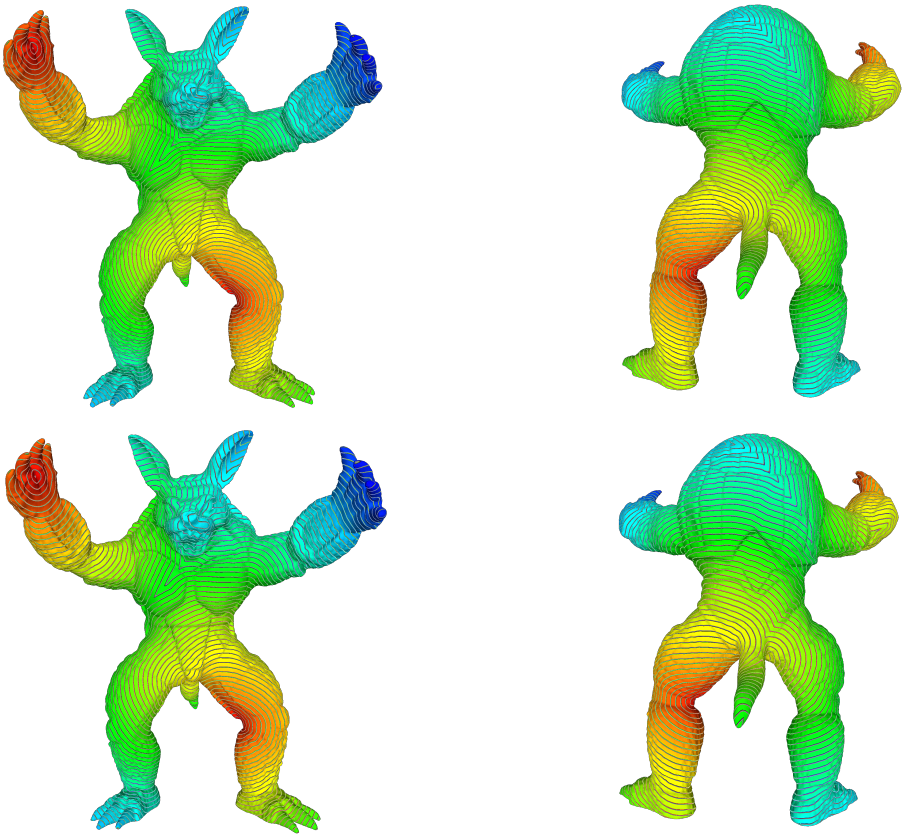


Figure 4.3: *All-destinations geodesic computation on the* ARMADILLO *model (350K-faces) from two point sources. Top row: computed by our algorithm; Bottom row: MMP algorithm.*

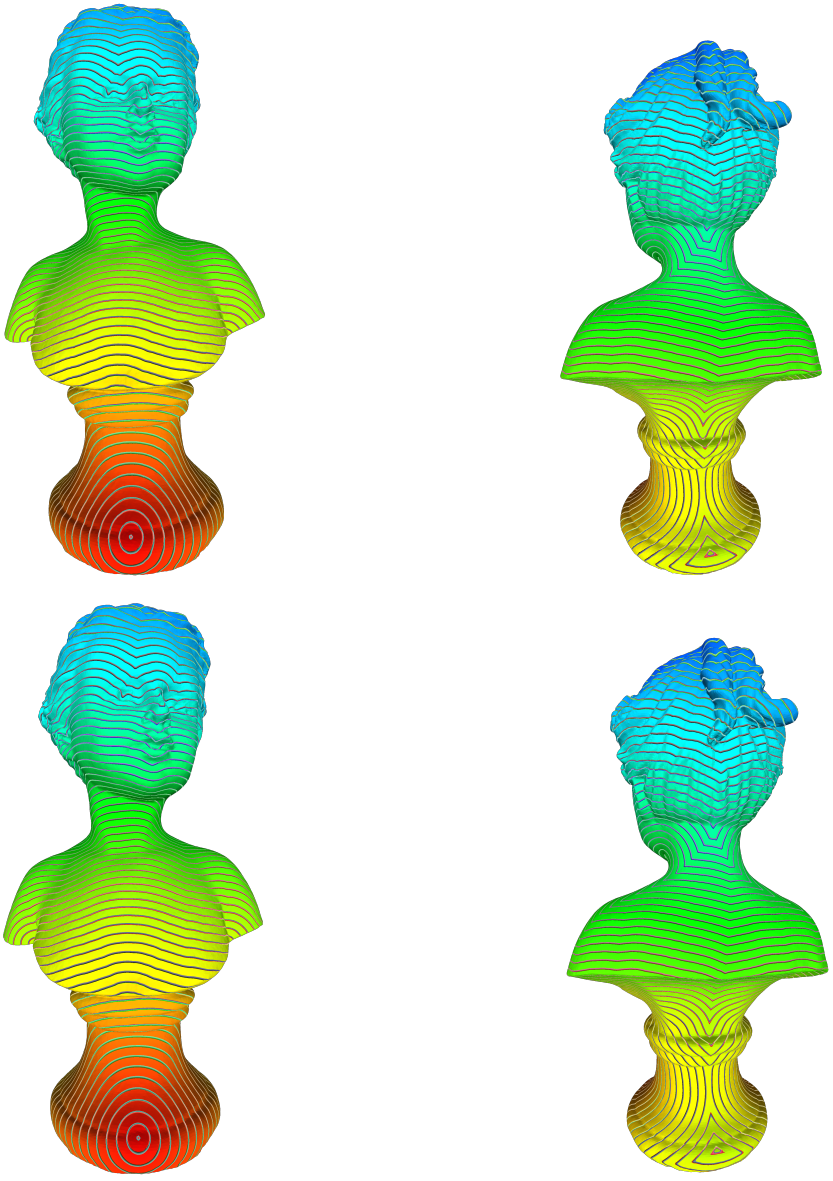


Figure 4.4: *All-destinations geodesic computation on the* BUSTO *model (500K-faces). Top row: computed by our algorithm; Bottom row: MMP algorithm.*

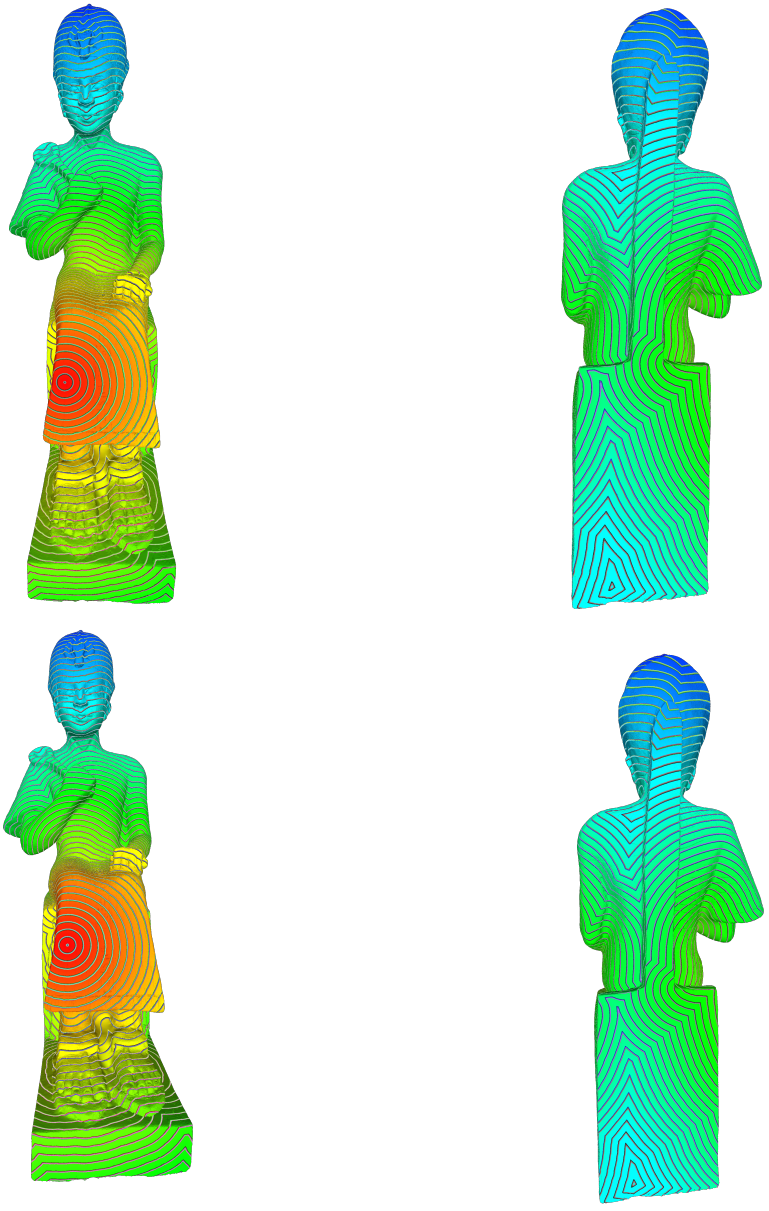


Figure 4.5: *All-destinations geodesic computation on the* RAMESSES *model (1.6M-faces). Top row: computed by our algorithm; Bottom row: MMP algorithm.*

Model	MPP	VCG	VoroGeo	Speedup
Armadillo	1.3s	0.57s	0.011s	51x
Busto	8.1s	0.82s	0.019s	44x
Ramesses	47.3s	2.73s	0.061s	45x

Table 4.9: *Speedup with respect to the algorithm for computing geodesic included in the VCG-Library.*

Chapter 5

Conclusions

We have presented a new approach for fast arbitrary geodesic computation on manifold triangular meshes based on a precomputation step and a query step. In the first step we compute and store geodesic information enconding it into a graph. This graph is then used in the successive query step to achieve fast and accurate replies to all-pairs geodesic distance queries.
Our method exposes three main parameters for user control: n_1, n_2 and δ, where n_1 and n_2 regulate the number of samples that are used to produce first and second layer Voronoi regions, while δ encodes an error threshold employed in the graph pruning procedure. In the precompuation step, our method is not limited to store a particular kind of geodesic, since any available method (i.e. any of those presented in chapter 2) could be "plugged in". We decided to use Surazhsky [9] implementation of the MMP [7] algorithm for precomputing geodesics because it computes exact polyhedral distances. This is very important to have a strong comparison for the distances computed by our algorithm and evaluate the accuracy achieved. Moreover, we designed a parallel implementation for the precomputation of geodesic information, to reduce the time complexity of this step.
Finally, we reported some statistics regarding the tuning of the input parameters, where we have shown that for most models a low density of first layer samples leads to faster and more accurate queries. An higher density of first layer samples is required for models presenting rich geometric details.

Through several experimental results and statistics we have shown that our algorithm is time efficient and accurate, achieving a considerable speedup against the geodesic algorithm employed in the VCG-Library, which consists of an enhanced version of the classical Dijkstra's search algorithm. Moreover, since we adopt an ad-hoc version of the Bidirectional-A* algorithm, our method guarantees that the computed distance is a metric.

5.1 Future work

Our method would surely benefit from several extensions:

- A GPU implementation of the preprocessing step (as exposed by [1]) would exploit the computational power provided by the GPU and speedup the precomputation of geodesic information by a reasonable amount.

- Employing a faster implementation of the MMP [7] algorithm, like the one proposed by Chen [8], would also improve both the time and the space complexity of the preprocessing step.

- Another possible improvement to our algorithm would be that of modifying the technique used for sampling. Introducing a curvature-driven sampling the Voronoi partitioning algorithm could produce more "well-posed" regions over which the geodesic precomputations would introduce less inaccuracy.

- Due to the large diffusion of triangle meshes, our algorithm was specifically designed to fit for manifold triangular meshes. Anyway, it could be easily adapted to quad-meshes, which are also very common.

- Our pruning technique compares the mean average error that would be introduced by pruning two nodes with a user-specified threshold, expressed as a percentage of the mesh bounding box diagonal. We decided to take this approach to have a measure that is indipendent from mesh resolution and tessellation. However, different pruning criteria

could be considered: for example, adapt the threshold for different part of the mesh presenting different curvature.

Bibliography

[1] Xiang Ying, Xiaoning Wang, and Ying He. Saddle vertex graph (svg): A novel solution to the discrete geodesic problem. *ACM Trans. Graph.*, 32(6):170:1–170:12, November 2013.

[2] A. Maheshwari and S. Wuhrer. Geodesic Paths On 3D Surfaces: Survey and Open Problems. *ArXiv e-prints*, April 2009.

[3] G.B. Folland. *Introduction to Partial Differential Equations*. Princeton University Press, 1995.

[4] R. Kimmel and J. A. Sethian. Computing geodesic paths on manifolds. In *Proc. Natl. Acad. Sci. USA*, pages 8431–8435, 1998.

[5] Danil Kirsanov. Minimal discrete curves and surfaces a thesis, 2004.

[6] John Canny and John Reif. New lower bound techniques for robot motion planning problems. In *Foundations of Computer Science, 1987., 28th Annual Symposium on*, pages 49–60, Oct 1987.

[7] Joseph S. B. Mitchell, David M. Mount, and Christos H. Papadimitriou. The discrete geodesic problem. *SIAM J. Comput.*, 16(4):647–668, August 1987.

[8] Jindong Chen and Yijie Han. Shortest paths on a polyhedron. In *Proceedings of the Sixth Annual Symposium on Computational Geometry*, SCG '90, pages 360–369, New York, NY, USA, 1990. ACM.

[9] Vitaly Surazhsky, Tatiana Surazhsky, Danil Kirsanov, Steven J. Gortler, and Hugues Hoppe. Fast exact and approximate geodesics on meshes. *ACM Trans. Graph.*, 24(3):553–560, July 2005.

[10] Marcel Campen and Leif Kobbelt. Walking on broken mesh: Defect-tolerant geodesic distances and parameterizations. *Computer Graphics Forum*, 30(2):623–632, 2011.

[11] J. A. Sethian. A fast marching level set method for monotonically advancing fronts. In *Proc. Nat. Acad. Sci*, pages 1591–1595, 1995.

[12] Keenan Crane, Clarisse Weischedel, and Max Wardetzky. Geodesics in heat: A new approach to computing distance based on heat flow. *ACM Trans. Graph.*, 32(5):152:1–152:11, October 2013.

[13] Marcel Campen, Martin Heistermann, and Leif Kobbelt. Practical anisotropic geodesy. In *Proceedings of the Eleventh Eurographics/ACMSIGGRAPH Symposium on Geometry Processing*, SGP '13, pages 63–71, Aire-la-Ville, Switzerland, Switzerland, 2013. Eurographics Association.

[14] Shi-Qing Xin, Xiang Ying, and Ying He. Constant-time all-pairs geodesic distance query on triangle meshes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 31–38, New York, NY, USA, 2012. ACM.

[15] Joseph S.B. Mitchell. Geometric shortest paths and network optimization. In *Handbook of Computational Geometry*, pages 633–701. Elsevier Science Publishers B.V. North-Holland, 1998.

[16] Marcel Campen, David Bommes, and Leif Kobbelt. Dual loops meshing: Quality quad layouts on manifolds. *ACM Trans. Graph.*, 31(4):110:1–110:11, July 2012.

[17] Venkat Krishnamurthy and Marc Levoy. Fitting smooth surfaces to dense polygon meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 313–324, New York, NY, USA, 1996. ACM.

[18] P. V. Sander, Z. J. Wood, S. J. Gortler, J. Snyder, and H. Hoppe. Multi-chart geometry images. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, SGP '03, pages 146–155, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[19] Leif Kobbelt, Swen Campagna, Jens Vorsatz, and Hans-Peter Seidel. Interactive multi-resolution modeling on arbitrary meshes. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 105–114, New York, NY, USA, 1998. ACM.

[20] Sagi Katz and Ayellet Tal. Hierarchical mesh decomposition using fuzzy clustering and cuts. *ACM Trans. Graph.*, 22(3):954–961, July 2003.

[21] Thomas Funkhouser, Michael Kazhdan, Philip Shilane, Patrick Min, William Kiefer, Ayellet Tal, Szymon Rusinkiewicz, and David Dobkin. Modeling by example. *ACM Trans. Graph.*, 23(3):652–663, August 2004.

[22] Peter-Pike J. Sloan, Charles F. Rose, III, and Michael F. Cohen. Shape by example. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, I3D '01, pages 135–143, New York, NY, USA, 2001. ACM.

[23] Emil Praun, Hugues Hoppe, and Adam Finkelstein. Robust mesh watermarking. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 49–56, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[24] Emil Praun, Adam Finkelstein, and Hugues Hoppe. Lapped textures. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 465–470, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[25] G. Zigelman, R. Kimmel, and N. Kiryati. Texture mapping using sur-

face flattening via multidimensional scaling. *IEEE Transactions on Visualization and Computer Graphics*, 8(2):198–207, April 2002.

[26] Kun Zhou, John Synder, Baining Guo, and Heung-Yeung Shum. Isocharts: Stretch-driven mesh parameterization using spectral analysis. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, SGP '04, pages 45–54, New York, NY, USA, 2004. ACM.

[27] Masaki Hilaga, Yoshihisa Shinagawa, Taku Kohmura, and Tosiyasu L. Kunii. Topology matching for fully automatic similarity estimation of 3d shapes. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 203–212, New York, NY, USA, 2001. ACM.

[28] David Bommes and Leif Kobbelt. Accurate computation of geodesic distance fields for polygonal curves on triangle meshes. In *VMV'07*, pages 151–160, 2007.

[29] Shi-Qing Xin and Guo-Jin Wang. Improving chen and han's algorithm on the discrete geodesic problem. *ACM Trans. Graph.*, 28(4):104:1–104:8, September 2009.

[30] Marcel Campen, Martin Heistermann, and Leif Kobbelt. Practical anisotropic geodesy. *Computer Graphics Forum*, 32(5):63–71, 2013.

[31] Dimas Martínez, Luiz Velho, and Paulo C. Carvalho. Computing geodesics on triangular meshes. *Comput. Graph.*, 29(5):667–675, October 2005.

[32] L. Bertelli, B. Sumengen, and B.S. Manjunath. Redundancy in all pairs fast marching method. In *Image Processing, 2006 IEEE International Conference on*, pages 3033–3036, Oct 2006.

[33] Stanley Osher and James A. Sethian. Fronts propagating with curvature dependent speed: Algorithms based on hamilton-jacobi formulations. *JOURNAL OF COMPUTATIONAL PHYSICS*, 79(1):12–49, 1988.

[34] Ryan Schmidt, Cindy Grimm, and Brian Wyvill. Interactive decal compositing with discrete exponential maps. *ACM Trans. Graph.*, 25(3):605–613, July 2006.

[35] Franz Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, September 1991.

[36] Steven Fortune. Handbook of discrete and computational geometry. chapter Voronoi Diagrams and Delaunay Triangulations, pages 377–388. CRC Press, Inc., Boca Raton, FL, USA, 1997.

[37] W. Zeng and R. L. Church. Finding shortest paths on real road networks: The case for a*. *Int. J. Geogr. Inf. Sci.*, 23(4):531–543, April 2009.

[38] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of a*. *J. ACM*, 32(3):505–536, July 1985.

[39] B. Stroustrup. *The C++ Programming Language*. Always learning. Addison-Wesley, 2013.

[40] Jasmin Blanchette and Mark Summerfield. *C++ GUI Programming with Qt 4*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.