

# A minimalistic approach for fast computation of geodesic distances on triangular meshes

Luciano A. Romero Calla<sup>a,b,c</sup>, Lizeth J. Fuentes Perez<sup>a,b,c</sup>, Anselmo A. Montenegro<sup>b</sup>

<sup>a</sup>*Visualization and MultiMedia Lab (VMM), Department of Informatics, University of Zurich, Switzerland*

<sup>b</sup>*Institute of Computing, Federal Fluminense University, Niteroi, Brazil*

<sup>c</sup>*IPRODAM3D research group, La Salle University, Arequipa, Peru*

## Abstract

The computation of geodesic distances is an important research topic in Geometry Processing and 3D Shape Analysis as it is a basic component of many methods used in these areas. In this work, we present a minimalistic parallel algorithm based on front propagation to compute approximate geodesic distances on meshes. Our method is practical and simple to implement, and does not require any heavy pre-processing. The convergence of our algorithm depends on the number of discrete level sets around the source points from which distance information propagates. To appropriately implement our method on GPUs taking into account memory coalescence problems, we take advantage of a graph representation based on a breadth-first search traversal that works harmoniously with our parallel front propagation approach. We report experiments that show how our method scales with the size of the problem. We compare the mean error and processing time obtained by our method with such measures computed using other methods. Our method produces results in competitive times with almost the same accuracy, especially for large meshes. We also demonstrate its use for solving two classical geometry processing problems: the regular sampling problem and the Voronoi tessellation on meshes.

**Keywords:** Geodesic distance, Fast marching, Triangular meshes, Parallel programming, Breadth-first search

## 1. Introduction

Computing geodesic distances on meshes is important to many problems in Geometry Processing and 3D Shape Analysis problems, such as parameterization [1], shape retrieval [2], isometry-invariant shape classification [3], mesh watermarking [4], object recognition [5], texture mapping [6], skinning [7], just to mention a few. Because finding geodesic distances is a basic step in many geometrical algorithms, the efficiency of its computation is an important issue.

Meshes are typically described as graphs. Consequently, one natural approach to solve the problem of geodesic computation on meshes is to generalize the ideas of distances on graphs to compute distance maps on surface representations. This path was pursued by Mitchell et al. [8], who proposed a continuous Dijkstra method that is able to yield exact results. Later, Surazhsky et al. [9] refined that method and proposed a more efficient approximate version. Other researchers approached the problem via physical phenomena analogy. Two of the most important methods in this class are based on models for wave propagation and heat diffusion. The Fast Marching approach belongs to the first category of methods and aims to solve the so-called Eikonal Equation [10], [11]. The Geodesics on Heat [12] belongs to the second category and explores the relationship between the heat kernel computation and distances on surfaces.

Here, we propose a parallel algorithm for the computation of distance maps on meshes that produces results with competitive accuracy and is simple to implement. Like the Fast Marching method, our method is also inspired by the grassfire propagation, but our approach does not require the maintenance of any priority queue. Instead, we propagate distances simultaneously around the frontier of propagation. This is the key to our parallelization strategy.

We provide an estimation of the complexity analysis (see section 6) that justifies the number of required iterations. Our experiments confirm that the estimation is not too tight, and show that far less than  $c\sqrt{n}$  iterations are necessary for the convergence of the method, where  $c$  is a small constant between  $1 \leq c \leq 2$  and  $n$  is the number of vertices of the mesh. We also discuss and show that our method is especially appropriate for solving the multi-source distance map problem (see 6.1.5).

### 1.1. Contribution

We propose a minimalist parallel method for computing geodesic distances on meshes with the following properties: it does not require heavy pre-processing steps; it produces quite good accuracy results even without pre-processing obtuse triangles; it is able to solve the problem for very large meshes; it can be used with single precision configurations if necessary; and finally, it produces speedup results comparable or better, in some cases, than the state-of-art methods.

We introduce an iterative parallel algorithm called *Parallel Toplesets Propagation* (PTP) to compute distance maps from a set of multiple sources on triangular meshes. Our method is in-

Email addresses: romero@ifi.uzh.ch (Luciano A. Romero Calla), fuentes@ifi.uzh.ch (Lizeth J. Fuentes Perez), anselmo@ic.uff.br (Anselmo A. Montenegro)

spired by the Fast Marching algorithm and is based on the propagation of distance information from the inner to outer groups of vertices that are equidistant to the source vertices. We call these groups *topological level sets* (*toplesets*, for short). The vertices in each *topleset* have their distances updated in parallel, independently, enabling us to explore the powerful parallel architectures present in the GPUs. To appropriately deal with coalescence problems when accessing the vertices in the GPU memory we devised a breadth-first search based graph representation similar to the one by Zhu et al. [13].

Our method is particularly appropriate for distance computation from multiple sources when the sources are predominantly distributed in a uniform way. This makes our method remarkably adequate when used as part of the implementation of the Farthest Point Sampling algorithm [14] for mesh resampling.

The proposed method is applied directly to the meshes and produces good accuracy results for large and irregular meshes. Additionally, the speedup values are competitive against methods where the preprocessing step dominates the overall complexity even without taking into account the preprocessing time.

## 1.2. Outline

This paper is organized as follows. In Section 2, we describe some of the most important works related to distance computation on graphs and minimal geodesic computation on meshes. Next, in Section 3, we describe and define some concepts and results that were used to develop our algorithm. The proposed method is presented in Section 4. In Section 5, we present the results produced by our algorithm. We first show two applications of our method: a parallel solution to the Farthest Point Sampling Problem and a parallel solution to the problem of computing the Voronoi Diagram on meshes. Next, we present the speedup of our parallel method for different meshes and measure the distance error for each experiment. We also compare our results with the exact method of [8], the Fast Marching Algorithm of [11] and the Geodesics on Heat method [12]. Finally in Section 7 we present our conclusions.

## 2. Related Work

The exact computation of geodesic distances on surfaces was proposed by Mitchell et al. [8]. Their MMP algorithm is based on a continuous Dijkstra algorithm and its computational complexity is  $O(n^2 \log n)$ , where  $n$  is the number of vertices.

Due to this high computational cost, approximate methods were developed. These methods present a better performance and maintain a comparable level of accuracy. A fast implementation of the MMP algorithm was presented by Surazhsky et al. [9]; that algorithm requires less memory and has a computational complexity of  $O(n \log n)$ .

Over the last decade, several approximate methods were proposed to compute distances by solving the Eikonal equation:

$$\|\nabla \phi\| = 1 \quad (1)$$

where  $\phi$  is a distance function. We can divide these approaches into two families: the Fast Marching and the Fast Sweeping.

The Fast Marching method (FM) was introduced by Sethian to solve distance computation on regular grids [10] and later extended to triangular meshes by Kimmel and Sethian [11]. The algorithm preserves the spirit of the Dijkstra algorithm as it uses a priority queue. It is a single-source to all-vertices algorithm, whose main advantage is the fast calculation of distances of vertices that are close to the source vertices. However, due to the sequential requirement of the priority queue, it is not possible to parallelize this algorithm without critical modifications.

The Fast Sweeping approach [15] has  $O(n)$  linear computational complexity. However, it requires a lot of sweeps to converge, particularly when the grids are unstructured.

A method based on Sethian's Fast Marching method and Polthier's straightest geodesics theory [16] is proposed by Martinez et al. [17]. In this work, they propose an iterative method to improve the discrete geodesic distances on triangulated meshes using inexact algorithms. It starts computing an initial approximate discrete geodesic curve  $\gamma_0$  using the Fast Marching Algorithm. In the sequel, it applies a sequence of correction operation that computes a sequence of discrete geodesic curves  $\gamma_i$  by applying local correction operations. The local correction operations aim at improving the positions of the discrete curve vertices and are inspired by the definitions of *straightest geodesics* and *shortest geodesics* introduced by Polthier and Schmies. The authors claim that their method can be used to improve the accuracy of any inexact method that computes discrete geodesics.

A parallel version of the Fast Marching Algorithm on parametric surfaces was proposed by Weber et al. [18]. In this method, the surface must be divided into several regular grids. Then, the distance map is computed for each grid, using a parallelization strategy based on the Raster Scan algorithm [19]. Finally, the reconstruction is done by joining the distance maps associated with each grid via the Dijkstra algorithm. Up to now, this method is the fastest and highly parallelizable for computing geodesic distances on triangular meshes. However, the error of the computed distance map depends on the distortion of the parameterization.

A method that does not use a priority queue is the Fast Iterative method [20], which is an algorithmic framework to solve the Eikonal equation. The main difference with our proposed method is that they use an unordered list of vertices which are removed and added constantly until the distances of the vertices converge, whereas our method keeps a structure of *toplesets* where the *toplesets* that are removed cannot be added again to the set of vertices to be updated.

There exists approaches that do not solve the Eikonal equation, such as the Saddle Vertex Graph (SVG) [21], which encodes the geodesic information in a sparse undirected graph, and the method proposed by Wang [22] which outperformed the SVG by using a divide and conquer technique.

Recently there was introduced a new family of methods that require the solution of Poisson-like systems. The Heat method was introduced by Crane [12, 23] and requires the solution of the Poisson equation. This method can be used with different kinds of representations because it is possible to compute the Laplacian operator for many different models, including trian-

gular meshes, point clouds and polygonal meshes. However, the accuracy of the distance map computation is sensitive to the choice of a parameter. A parallel and scalable version of the Heat method was proposed by Tao et al. [24]. Finally, Litman and Bronstein [25] proposed a method that also works on the spectral domain, called the Spectrometer. Both methods require as preprocessing the computation of the Laplacian matrix. Moreover, the Spectrometer requires the computation of the Heat kernel, which involves computing the eigendecomposition and is computationally expensive. In contrast, our method avoids heavy pre-processing steps.

### 3. Background

In the next subsections, we present some of the basic concepts and results on which our method is based.

#### 3.1. Discrete distance maps

Many graphical objects are described by 2-d manifolds, embedded in a three-dimensional space, that is, *embedded surfaces*. Surfaces are usually represented by piecewise linear representations known as *triangle meshes*  $T = (V, E, F)$ , where  $V$  is the set of vertices,  $E$  the set of edges and  $F$  the set of faces. We now state the problem of computing distance maps on a triangulated mesh  $T$ .

Let  $T$  be a triangulated mesh. Given a subset  $S \subset V$ , called source set, compute a map  $d_S(v) : V \rightarrow \mathbb{R}$  that associates to each  $v \in V$  its geodesic distance to  $S$ . For the sake of simplicity, we denote  $d_S$  as  $d$ .

Informally, we define *geodesic distance* as the length of the minimum path connecting two vertices of the mesh  $T$  possibly passing through the faces of the mesh. In contrast, we use the term *topological distance* as the number of edges in the minimum path connecting two vertices passing only through the meshes' edges.

#### 3.2. Fast Marching algorithm

The Fast Marching algorithm (Algorithm 3.1), simulates the propagation of the distance information in a discrete set. It is possible to make an analogy with the propagation of fire in a grassland or simply *fire propagation*. In the beginning, each source vertex  $s \in S$  has its distance fixed to zero ( $d(s) = 0, s \in S$ ) and is inserted in a priority queue  $R$  (red vertices), whose priority is defined in terms of the smallest distance. All other vertices  $v \notin S$  are labeled with distance equal to infinity ( $d(v) = \infty$ ). At each step, one vertex  $v$  is selected from the priority queue and inserted in the list of processed vertices  $B$  (black vertices); for all neighbor vertices  $v_0 \in \mathcal{N}(v)$  of  $v$ , all triangles incident to it  $F(v_0)$  are updated using Algorithm 3.2, which was proposed by Kimmel and Sethian in [11]. The new vertices on the updated triangles are inserted in the priority queue  $R$  and the algorithm proceeds until all vertices are processed, that is,  $B = V$ .

The update step (Algorithm 3.2) is one of the distinct features of the Fast Marching method. It yields a linear local approximation to the continuous distance and guarantees that the solution

obeys both the *consistence condition* and the *monotonicity condition* ( $QX^T n < 0$ ) of the signal propagation. When considered together, they ensure that, for a given triangle, defined by three vertices  $v_0, v_1, v_2$ , if  $v_1$  and  $v_2$  are closer to the source set, then  $v_0$  cannot be reached by the signal before  $v_1$  and  $v_2$ , producing a correct solution for the Eikonal Equation. In geometrical terms, this means that the triangles in the mesh cannot be obtuse [26]. One solution for dealing with meshes that have such triangles is to subdivide them or to locally unfold the mesh [11]. For more details about the Fast Marching method, we refer to the reader to the following references [11] and [26].

---

#### Algorithm 3.1 Fast Marching (FM) [18, 26]

---

**Require:** Triangular mesh  $(V, F)$ , source vertices  $S \subset V$

**Ensure:** Distances map  $d : V \rightarrow \mathbb{R}$

```

1:  $\forall v \in V : d(v) \Leftarrow \infty, \forall s \in S : d(s) \Leftarrow 0$ 
2:  $R \Leftarrow S$ 
3:  $G \Leftarrow V \setminus R$ 
4:  $B \Leftarrow \{\}$ 
5: while  $B \neq V$  do
6:    $v \Leftarrow \arg \min_{v \in R} d(v)$ 
7:    $R \Leftarrow R \setminus \{v\}$ 
8:    $B \Leftarrow B \cup \{v\}$ 
9:   for all  $v_0 \in \mathcal{N}(v)$  do
10:     $G \Leftarrow G \setminus \{v_0\}$ 
11:     $R \Leftarrow R \cup \{v_0\}$ 
12:    for all  $(v_0, v_1, v_2) \in F(v_0)$  do
13:       $d(v_0) \Leftarrow \min\{d(v_0), \text{update\_step}(d_{k-1}, v_0, v_1, v_2)\}$ 
14:    end for
15:  end for
16: end while
17: return  $d$ 

```

---



---

#### Algorithm 3.2 update\_step [18, 26]

---

**Require:** A distance map  $d : V \rightarrow \mathbb{R}$ , triangular face  $(v_0, v_1, v_2) \in F$ .

**Ensure:** Solution of the Eikonal equation for the triangle  $(v_0, v_1, v_2)$  and the vertex  $v_0$ .

```

1:  $x_1 \Leftarrow v_1 - v_0$ 
2:  $x_2 \Leftarrow v_2 - v_0$ 
3:  $X \Leftarrow [x_1 \quad x_2]$ 
4:  $t \Leftarrow [d(v_1) \quad d(v_2)]^T$ 
5:  $1^T = [1 \quad 1]$ 
6:  $p \Leftarrow \frac{1^T Q t + \sqrt{(1^T Q t)^2 - 1^T Q 1 \cdot (t^T Q 1 - 1)}}{1^T Q 1}$ 
7:  $n \Leftarrow X Q (t - 1)$ 
8: if  $Q X^T n < 0$  then
9:   return  $p$ 
10: else
11:   return  $\min\{d_{k-1}(v_1) + \|x_1\|, d_{k-1}(v_2) + \|x_2\|\}$ 
12: end if

```

---

Both exact and Fast Marching based methods rely, to a lesser or greater extent, on the use of priority queues. This makes it

hard to parallelize them. Thus, we decided to completely abandon the use of priority queues in our proposed method. Instead of fixing the final distance for the closest vertex, at each iteration, we update the distances on subsets of vertices that are good candidates for the propagation of distance information. More precisely we take advantage of the discrete topological structure of the mesh which can be decomposed in topological distance level sets around the source vertices to propagate the information simultaneously and independently in multiple phases until the distances converge.

#### 4. Proposed Method

The method proposed here works by simulating the distance information through an ordered set of vertices using sequential iterations that refine the estimated distances. One of the main features of our method is to exploit the natural ordering induced by the topology of the graph associated with the mesh. Vertices with the same topological distance in the unweighted graph induced by the mesh are grouped in sets defined here as *topological level sets (toplesets)*. The *toplesets* and their topological distances to the source are used to guide the propagation and also mark off its scope. Our algorithm is an iterative algorithm that uses the preliminary ordering defined by the topological distances as an initial estimate of the real continuous distance. At each iteration, the distances of a subset of all topological sets, defining a band (the *update band*), are updated in parallel using a relaxation scheme, which is the update function (Algorithm 3.2) of the Fast Marching method. This relaxation process is applied for a number of iterations until all distances converge.

In this section, we define more precisely the concept of *topological level sets (toplesets)* and describe the notion of *topological level sets propagation*, which is the kernel of our algorithm. Next, we describe the proposed algorithm and its parallel implementation on GPU. Finally, we explain how to leverage the properties of our method to solve the multi-source version of the problem.

##### 4.1. Topological level sets propagation

The *topological level sets propagation* relies on the concept of *topological level set (topleset)*.

**Definition 1** (*topleset  $V_r$* ). *Let  $G$  be the unweighted graph induced by the combinatorial topological structure of a triangle mesh  $T = (V, E, F)$  and  $S \subset V$  the set of source vertices. A *topleset*  $V_r$  in  $G$  is the set of all vertices  $v \in G$ , such that the length of the shortest path from  $v$  to the source vertices  $S$  in  $G$  is equal to a constant  $r \in \mathbb{N}$ ,  $r > 0$ . A *topleset*  $V_r$  is defined by the recurrence relation*

$$V_r = \left\{ v \in N(V_{r-1}) : v \notin \bigcup_{r'=0}^{r-1} V_{r'} \right\}$$

where  $V_0 = S$  and  $N(V_{r-1})$  is the set of all the vertices that are neighbors of the vertices  $v \in V_{r-1}$ .

The properties below can be proved by inspection using the definition.

1.  $\bigcup_{r=0}^{\rho-1} V_r = V$
2.  $\forall r, r' \in [0, \rho - 1], r \neq r' : V_r \cap V_{r'} = \emptyset$
3.  $\sum_{r=0}^{\rho-1} |V_r| = |V|$

where  $\rho$  is the number of *toplesets* from  $S \subset V$  in a triangular mesh. Note that the *toplesets* are 0-indexed because we start counting the *topleset*  $V_0 = S$ .

A *topological level set propagation* is defined as the propagation of the distance information through a sequence of *toplesets* which defines a partial ordering on the vertices of an unweighted graph  $G$  induced by the mesh's combinatorial structure.

##### 4.2. An algorithm based on *topleset distance propagation*

The proposed algorithm propagates and relaxes distance information through the meshes' *toplesets*. Algorithm 4.1 performs the distance propagation, updating at each  $k$ -th iteration, the current distance map  $d_k$ , as a function of the previous distance map  $d_{k-1}$  at iteration  $k - 1$ , for all vertices in the *update band*  $B_{i_k}^{j_k}$ .

We define the *update band*

$$B_{i_k}^{j_k} = \bigcup_{r=i_k}^{j_k} V_r$$

for each iteration  $0 < k \leq K$ , where  $K$  is the maximum number of iterations. The *update band*  $B_{i_k}^{j_k}$  defines the set of vertices that will be updated in the current iteration  $k$ , and is composed of a set of consecutive *toplesets*  $V_r$ ,  $i_k \leq r \leq j_k$ . The whole set of consecutive *toplesets* are computed with a breadth-first traversal in the induced graph.

We define the *update band's* boundaries  $i_k$  and  $j_k$  as sequences of values at each iteration  $k$  as follows:

$$i_k = \begin{cases} i_{k-1} + 1 & \text{if } \frac{|d_k(v) - d_{k-1}(v)|}{d_{k-1}(v)} < \epsilon, \forall v \in V_{i_{k-1}} \\ i_{k-1} & \text{otherwise} \end{cases} \quad (2)$$

$$j_k = \begin{cases} k & \text{if } k < \rho \\ \rho - 1 & \text{otherwise} \end{cases} \quad (3)$$

where  $\epsilon$  is a threshold for the relative change condition,  $\rho$  is the *toplesets* number, and  $d_k$  is the distance map at iteration  $k$ . This means that the lower boundary index is increased, thus (shortening the band) if the relative change of the vertices in the corresponding *topleset* is smaller than an  $\epsilon$  from one iteration to the next. In the experiments, we set  $\epsilon = 0.001$ . Algorithm 4.1 (lines 12 to 18) computes at each iteration the size of the *update band*. A graphical example of this explanation is depicted in Figure 1.

One of the most important operations in the topological level sets propagation algorithm is the update step (Algorithm 3.2),

---

**Algorithm 4.1** Parallel Tuplesets Propagation (PTP).

---

**Require:** Triangular mesh  $(V, F)$ , source vertices  $S \subset V$   
**Ensure:** Distances map  $d : V \rightarrow \mathbb{R}$

- 1:  $d_k$  current distance map at iteration  $k$
- 2:  $\forall v \in V : d_0(v) \leftarrow \infty$
- 3:  $\forall s \in S : d_0(s) \leftarrow 0$
- 4:  $i \leftarrow 1, j \leftarrow 1, k \leftarrow 1$
- 5: **while**  $i \leq j$  **do**
- 6:   **for all**  $v_0 \in \bigcup_{r=i}^j V_r$  (**in parallel**) **do**
- 7:      $d_k(v_0) \leftarrow d_{k-1}(v_0)$
- 8:     **for all**  $(v_0, v_1, v_2) \in F(v_0)$  **do**
- 9:        $d_k(v_0) \leftarrow \min\{d_k(v_0), \text{update\_step}(d_{k-1}, v_0, v_1, v_2)\}$
- 10:     **end for**
- 11:   **end for**
- 12:   **if**  $\frac{|d_k(v) - d_{k-1}(v)|}{d_{k-1}(v)} < \epsilon, \forall v \in V_i$  **then**
- 13:      $i \leftarrow i + 1$
- 14:   **end if**
- 15:    $k \leftarrow k + 1$
- 16:   **if**  $k < \rho$  **then**
- 17:      $j \leftarrow k$
- 18:   **end if**
- 19: **end while**
- 20: **return**  $d \leftarrow d_k$

---

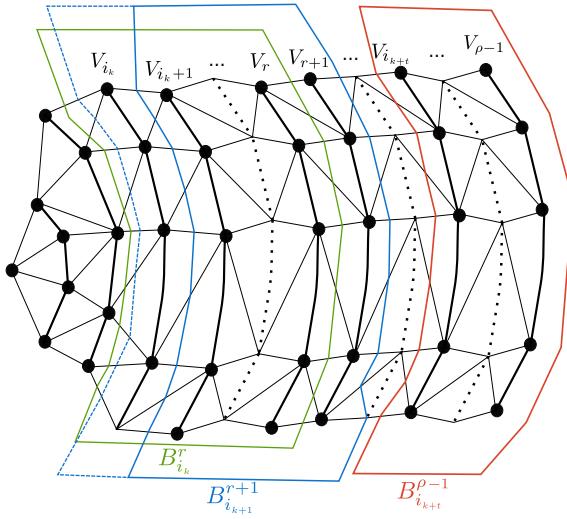


Figure 1: A graphical example of the *update band* for the iterations  $k, k+1$  and some iteration  $k+t \geq \rho$ .

which is based on the Fast Marching method update step. It is possible to update all vertices independently (line 6, Algorithm 4.1), since each vertex  $v$  updates its new distance  $d_k(v)$  according to the distance map  $d_{k-1}$  computed in the preceding iteration  $k-1$ .

One distinct ingredient of the Parallel Tuplesets Propagation (PTP) is that the *update band* deals with the fact that, as the frontier travels forward, there will be vertices whose distances have already converged; vertices which are located in the first

*tuplesets* of the propagation. Such vertices are discarded by using the relative change based approach (lines 12 to 18, Algorithm 4.1) as mentioned before.

### 4.3. Parallelization and Implementation on GPU

The Parallel Tuplesets Propagation algorithm completely eliminates the dependency on the priority queue which is necessary for the classical Fast Marching Algorithm. This permits us to reuse the calculations of the previous distance maps to generate a new estimate. Furthermore, as the calculation of distances is independent for each vertex  $v_0$ , the loop (see Line 6 in Algorithm 4.1) is highly parallelizable on SIMD and GPU processors.

Since meshes are irregular graphs, they induce random accesses to the global memory. Thus, the non-coalescence problem needs to be handled carefully to avoid an inefficient implementation on GPU. In this section, we explain how we deal with this problem. We were inspired by the method presented in [13] which addressed the problem of iterative traversing-based graph processing, for instance, Breadth First Search (BFS). Our method is quite similar to a BFS approach, with the difference that traversing tasks are being performed at the same time. Thus, we adapted some ideas from the WolfPath algorithm [13] to deal with meshes. To solve the problem of non-coalescing memory access, they convert the graph into a layered tree representation using a breadth-first traversal, where the vertices that are in the same layer are duplicated in the next level. The data structure is a layered edge list composed of two arrays, a source, and a destination array. Our method performs a similar step when it is used on a GPU. We compute the *tuplesets* and we change the order of the data array that represents a mesh as a Compact Half Edge data structure (CHE) [27]. In this way, coalesced memory access is guaranteed.

## 5. Experimental evaluation

We have implemented two versions of the proposed algorithm using C++ with OpenMP and CUDA to execute the experiments on CPU and GPU, respectively. We have made available the source code for the PTP implementation as part of our framework *gproshan* (*geometry processing and shape analysis framework*) in <https://github.com/larc/gproshan>.

The experiments were performed in a Docker container hosted in a machine with this configuration: an Intel Xeon E5-2698 v4 2.2 GH with 40 cores and 256 GB of RAM, a Tesla P100 with 16GB of RAM memory and 3584 cuda cores, and Ubuntu 16.04.3. The Docker container was set with the GCC/G++ 7.3 compiler, CUDA 9.0, and the required libraries to run the algorithms.

We have implemented the Fast Marching algorithm that uses a priority queue, to compare its performance with the performance of our algorithm. To validate our FM implementation, we compared its performance with Gabriel Peyré's implementation used in the FM Toolbox Graph [28]. Table 1 shows our test values. The time of our FM is better than Peyré's FM for large meshes. For meshes, where the number of vertices are in the range of 25000 and 100000, we have competitive time.

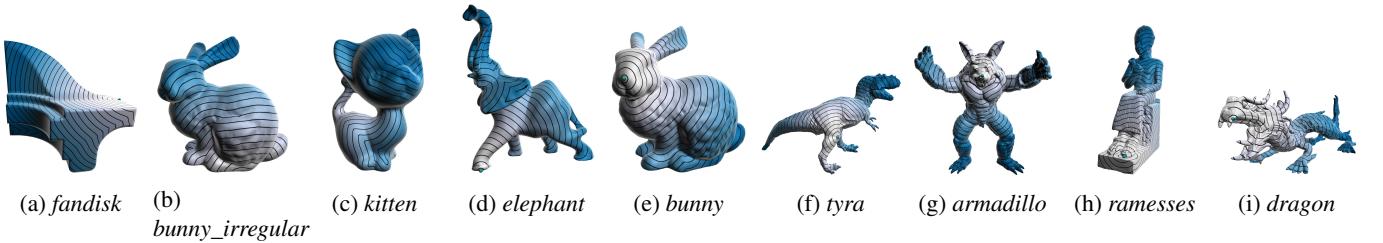


Figure 2: Geodesic distances map for each mesh in Table 2, from  $m = 1$  sources.

Table 1: FM from Toolbox Graph compiled with -O3 flag vs our FM implementation. Comparison time and mean absolute percent error (double precision). The experiment was performed on a Intel(R) Core(TM) i7-6700 CPU, 3.40GHz.

FILENAME	VERTICES	FM toolbox		FM ours	
		TIME	ERROR	TIME	ERROR
fandisk	6475	<b>0.053s</b>	<b>0.95%</b>	0.105s	1.07%
bunny_irregular	7500	<b>0.078s</b>	<b>1.49%</b>	0.129s	2.14%
kitten	14472	<b>0.138s</b>	<b>1.13%</b>	0.232s	1.13%
elephant	24955	<b>0.323s</b>	<b>0.88%</b>	0.403s	0.90%
bunny	34835	<b>0.497s</b>	<b>0.91%</b>	0.558s	0.96%
tyra	100002	1.991s	0.77%	<b>1.665s</b>	0.93%
armadillo	172974	4.395s	0.62%	<b>2.838s</b>	0.73%
ramesses	826266	24.769s	0.54%	<b>13.838s</b>	0.64%
dragon	3609455	733.167s	0.37%	<b>56.988s</b>	0.39%

Since our FM implementation is faster for large meshes, the speedup values obtained with the PTP method are lower than what we would obtain with Peyré’s implementation, so the time comparison is reliable. We can observe that our FM’s error is quite similar but not better than Peyré’s FM. In our implementations (our FM’s code and PTP), we did not deal with the presence of obtuse triangles by using unfolding or triangle subdivision. Nevertheless, as it can be seen in the results, PTP produced quite good accuracy results even without pre-processing obtuse triangles.

To compare the accuracy of the algorithms, we used the MMP algorithm proposed by Mitchell et al. [8], to compute exact geodesics, which is included in the MeshLP package [29, 30]. We also compare our method with the Heat method proposed in [12, 23], which requires a heavy pre-processing step. However, after the computation of the pre-processing step, the time complexity to calculate distance queries is quite fast because the linear system is sparse. We chose this method because it is one of the fastest in literature and its source code is available [31].

We performed experiments on CPU and GPU, in single and double precision. The main advantage of *single precision* over *double precision* is memory consumption. The *double precision* requires twice the number of bits than *single precision* requires to represent a decimal number. The specific details of the implementation as well as the performance and accuracy results obtained from the experiments will be given in the following sub-sections. The performance and accuracy are evaluated over the meshes shown in Figure 2. Tables 2 and 3 summarize the experiments of distance computation from  $m = 1$  source.

### 5.1. Performance

We evaluated the performance in single and double precision separately. The PTP method in CPU was accelerated using the OpenMP library; the number of threads was 40. For the GPU implementation, we used a block size of 64.

Table 2 shows the execution times and speedup for the PTP algorithm implemented on CPU and GPU with single precision. The speedup is considered over the FM algorithm.

The experiments with double precision are summarized in Table 3. We included the Heat method experiments only in this part because the solution of Cholesky factorization requires double precision.

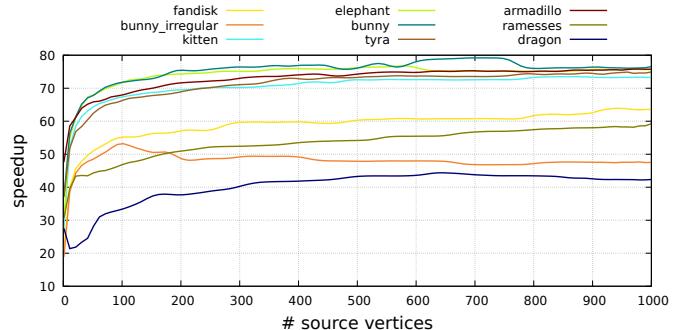


Figure 3: PTP speedup with  $m \in [1 : 1000]$ .

In Table 3 we observe that the implementation of the Geodesics in Heat method using the Cholmod library is fast and outperforms our method for small and medium size instances using a hybrid GPU/CPU solver (using the Cholmod library). For larger instances, the Geodesic in Heat method could not produce the correct results whereas the PTP computed the correct distance maps with an approximate speedup of 55x for the *dragon* mesh. Moreover, the implementation of the Geodesics in Heat method using the Cholmod library requires the use of double precision while the PTP can be run using both double and single precision. In the experiments, we noticed that the Cholmod did not activate the GPU component using only the CPU cores. We speculate that the reason for this is that the fill ratio of the Laplacian matrices of the meshes is too small. The meshes we used in our experiments have a fill ratio between 5 and 8; we compute these values according to the definition of fill ratio presented in [32]. We can also claim that the PTP yields better accuracy results than the Geodesic on Heat for irregular meshes like *bunny\_irregular* and *tyra*.

Table 2: Single precision: comparison times, mean absolute percent error and speedup.

single precision		FAST MARCHING		PTP (OpenMP)		PTP (cuda)	
FILENAME	VERTICES	TIME	ERROR	TIME	ERROR	TIME	ERROR
bunny_irregular	fandisk	6475	0.158s <b>1.07%</b>	<b>0.006s</b> ( <b>25.3x</b> )	1.07%	0.008s ( <b>19.2x</b> )	1.07%
	bunny_irregular	7500	0.195s 2.14%	<b>0.008s</b> ( <b>23.0x</b> )	<b>2.12%</b>	0.010s ( <b>18.9x</b> )	2.12%
	kitten	14472	0.357s <b>1.13%</b>	<b>0.009s</b> ( <b>38.9x</b> )	1.14%	0.012s ( <b>30.4x</b> )	1.14%
	elephant	24955	0.624s <b>0.90%</b>	0.018s ( <b>34.3x</b> )	0.91%	<b>0.015s</b> ( <b>40.7x</b> )	0.91%
	bunny	34835	0.864s <b>0.96%</b>	0.024s ( <b>36.7x</b> )	0.96%	<b>0.015s</b> ( <b>57.0x</b> )	0.96%
	tyra	100002	2.571s <b>0.93%</b>	0.137s ( <b>18.7x</b> )	1.00%	<b>0.032s</b> ( <b>79.5x</b> )	1.01%
	armadillo	172974	4.391s <b>0.72%</b>	0.121s ( <b>36.2x</b> )	0.89%	<b>0.035s</b> ( <b>126.7x</b> )	0.90%
	ramesses	826266	21.438s <b>0.51%</b>	1.978s ( <b>10.8x</b> )	1.28%	<b>0.263s</b> ( <b>81.6x</b> )	1.35%
	dragon	3609455	89.149s 0.28%	10.946s ( <b>8.1x</b> )	<b>0.27%</b>	<b>1.732s</b> ( <b>51.5x</b> )	0.31%

Table 3: Double precision: comparison times, mean absolute percent error and speedup.

double precision		FAST MARCHING		PTP (OpenMP / cuda)		HEAT METHOD (Cholmod / cusolverSp)		
FILENAME	VERTICES	TIME	ERROR	TIME	ERROR	PRECOMP	SOLVE	ERROR
fandisk	6475	0.159s	1.07%	0.005s ( <b>29.4x</b> )	1.07%	OpenMP	<b>0.068s</b> ( <b>002s</b> ) ( <b>101.1x</b> )	0.84%
				0.009s ( <b>18.6x</b> )	1.07%	Cuda	<b>0.855s</b> (0.646s) ( <b>0.2x</b> )	<b>0.84%</b>
bunny_irregular	7500	0.195s	2.14%	0.008s ( <b>24.6x</b> )	<b>2.12%</b>	OpenMP	<b>0.085s</b> ( <b>002s</b> ) ( <b>117.1x</b> )	14.28%
				0.011s ( <b>18.3x</b> )	2.12%	Cuda	<b>0.862s</b> (1.225s) ( <b>0.2x</b> )	14.28%
kitten	14472	0.357s	<b>1.13%</b>	0.009s ( <b>41.1x</b> )	1.14%	OpenMP	<b>0.189s</b> ( <b>004s</b> ) ( <b>96.9x</b> )	1.29%
				0.012s ( <b>29.6x</b> )	1.14%	Cuda	<b>0.888s</b> (1.235s) ( <b>0.3x</b> )	1.29%
elephant	24955	0.623s	<b>0.90%</b>	0.017s ( <b>36.0x</b> )	0.91%	OpenMP	<b>0.307s</b> ( <b>006s</b> ) ( <b>102.3x</b> )	0.99%
				0.016s ( <b>39.4x</b> )	0.91%	Cuda	<b>0.902s</b> (1.733s) ( <b>0.4x</b> )	0.99%
bunny	34835	0.863s	0.96%	0.023s ( <b>37.3x</b> )	0.97%	OpenMP	<b>0.444s</b> ( <b>009s</b> ) ( <b>95.7x</b> )	<b>0.93%</b>
				0.016s ( <b>52.8x</b> )	0.97%	Cuda	<b>0.953s</b> (20.694s) ( <b>0.0x</b> )	0.93%
tyra	100002	2.572s	<b>0.93%</b>	0.139s ( <b>18.5x</b> )	1.02%	OpenMP	<b>1.340s</b> ( <b>029s</b> ) ( <b>89.4x</b> )	1.16%
				0.036s ( <b>71.2x</b> )	1.02%	Cuda	infs (infs) ( <b>0.0x</b> )	inf%
armadillo	172974	4.390s	0.72%	0.122s ( <b>36.0x</b> )	0.90%	OpenMP	<b>2.384s</b> (0.066s) ( <b>66.8x</b> )	<b>0.59%</b>
				<b>0.038s</b> ( <b>114.3x</b> )	0.90%	Cuda	infs (infs) ( <b>0.0x</b> )	inf%
ramesses	826266	21.449s	<b>0.64%</b>	1.678s ( <b>12.8x</b> )	1.89%	OpenMP	<b>12.067s</b> (0.347s) ( <b>61.7x</b> )	nan%
				<b>0.268s</b> ( <b>80.2x</b> )	1.89%	Cuda	infs (infs) ( <b>0.0x</b> )	inf%
dragon	3609455	89.054s	<b>0.39%</b>	9.311s ( <b>9.6x</b> )	0.68%	OpenMP	<b>60.709s</b> (2.628s) ( <b>33.9x</b> )	nan%
				<b>1.596s</b> ( <b>55.8x</b> )	0.68%	Cuda	infs (infs) ( <b>0.0x</b> )	inf%

We can observe that in the single precision experiments the PTP method in CUDA outperforms the PTP method on CPU. However, for small meshes like *fandisk* and *bunny irregular*, both methods have similar speedup values. This behavior is expected since the acceleration does not compensate for the GPU overhead for small meshes.

The Heat method requires the solution of a linear system. To this end, the usual implementation computes the Cholesky factorization, which takes a considerable time to compute. However, once we calculate this matrix, the time for the queries is quite fast. To accelerate the Heat method on the CPU, we used Cholmod from SuiteSparse library [32]. However, we could not find a method that works with sparse matrices and solves the problem entirely on the GPU device. We found the CUDA library *cusparse*, which is the fastest; the problem is that the conditions are incomplete and therefore the results do not lead to a proper distance map. Another CUDA library we found is the *cusolverSp*, which uses a hybrid method as parts of the computation are done on CPU and others on GPU; the use of *cusolverSp* produced the expected results, but it is still slow which can be explained by a possible overhead due to multiple memory copies from the device to host and host to the device. Clearly, this is a limitation of current CUDA libraries to implement the Heat method. It is difficult to get benefits from powerful GPU devices when we use those implementations of

the Heat method as is shown in Table 3. The speedup values obtained from the Heat method with *cusolverSp* are quite low. Additionally, the solution to the Cholesky factorization is not guaranteed, even with double precision. In our experiments, it was not possible to get a proper solution for the largest meshes *tyra*, *armadillo*, *ramesses* and *dragon*. In the case of CPU, the Cholmod library is more stable and we got a good result for the *armadillo* mesh. Recently, Tao et al. [24] proposed a scalable version of the Heat Method, which has been tested only on CPU for large meshes. This method uses Gauss–Seidel iterations instead of solving the linear system using Cholesky decomposition. The method also improved the memory consumption of the previous implementation of the Heat method. However, it still depends on the choice of a good parameter, which for irregular meshes still presents a higher error on the distance computation.

In contrast, the PTP method successfully computes the distance map using single and double precision. Moreover, as expected, the speedup values obtained for the PTP method, using single precision are faster than the speedup values obtained using double precision.

In Section 6, we showed the importance of the inverse relationship between the complexity of the PTP algorithm and the number of sources and the way they are distributed. We also discussed how the distance can be computed independently

from each source at each iteration allowing the parallel implementation.

Figure 3 presents the speedup of the PTP algorithm for each mesh from  $m$  sources with approximate uniform distribution. For this experiment we did not perform the rearrangement of the vertices on the GPU memory, since there are multiple sources. Note that the speedup of meshes with a large number of vertices increases with the number of sources  $m$ ; this illustrates how our algorithm, performs well when computing distances from multiple sources and also reinforces the expected inverse relationship between performance and number of sources in the proposed PTP algorithm.

## 5.2. Accuracy

We tested the accuracy of the proposed algorithm, performing a comparison based on the Mean Absolute Percent Error (MAPE) between the FM algorithm and the PTP algorithm, summarized in Tables 2 and 3. We can observe that the PTP algorithm has error (MAPE) values similar to the FM algorithm, and even less for *fandisk* and *dragon* meshes. We can improve PTP's accuracy, modifying the  $\epsilon$  value for the relative error. This happens because we neither performed mesh unfolding nor subdivided obtuse triangles, which are necessary steps for the FM algorithm. We show experimentally that our algorithm corrects the Dijkstra's estimate in the case of obtuse triangles in subsequent iterations.

We evaluated the convergence of the PTP algorithm as a function of the number of iterations. The experimental results show that no more than  $c \sqrt{n}$  iterations are required for the PTP convergence, with  $c \approx 1.5$  for predominantly regular meshes. An analysis of such behavior is discussed in Section 6.

Figure 4 presents the empirical absolute mean errors as a function of the number of iterations; each chart shows values starting with an iteration  $k = \rho$  (number of *toplesets* of the mesh) and finishing within  $K$  (number of iterations) empirically estimated by our algorithm for each mesh. Note that the error decreases as the number of iterations are increased and it converges for some  $k$  between  $\rho$  and  $K$ . The exact number of iterations  $K$  depends on the topology of the triangular mesh and the geometric relation between the propagation of level sets and the geodesic distance map.

The relationship between the level sets propagation and the geodesic distance map is depicted in Figure 5. The topological level sets propagation and the isocurves defined by the distance map are quite similar on the disk. This means that the geodesic curves are almost perpendicular to the discrete curves defined by the *toplesets* and that the initial edge-distance is a good approximation to the final continuous distance. In this case, our algorithm only needs about  $\rho$  iterations to compute the correct distance map. For the bunny mesh, we observe a similar behavior. Nevertheless, as the *topleset* arrangement diverges more from the final isocurves of distance, we need more than  $\rho$  iterations, but no more than  $1.5\rho$  as expected for well-behaved meshes; see Figure 4.

### 5.2.1. Comparison with the Heat method

One of the main problems with the Heat method was the choice of the parameter. According to the paper [23], a good approximation is the square mean of the mesh edges' length. However in case of the *ramesses* mesh it was not possible to obtain a correct result. For this reason, in Table 3, the error is *nan*. Another limitation of the considered implementation of the Heat method is that the use of double precision is mandatory since this implementation of the Heat method requires to solve the Cholesky factorization. Furthermore, even with double precision, the CusolverSparse library does not always produce a solution when dealing with large meshes. In our experiments, it crashed for the *armadillo* and *ramesses* meshes. So, we got an *inf* error. This is another limitation of this implementation of Heat method because it depends on the Cholesky factorization for sparse matrices and this computation is still unstable, especially in libraries for GPU devices.

In contrast, our method works faster when dealing with single precision and the accuracy is little affected. These results are shown in Tables 2 and 3.

Finally, in Table 3, we note that for the *irregular bunny*, the Heat method presents a higher error than the proposed method. Thus, our method is robust when dealing with irregular meshes and obtuse triangles because it is able to correct the error through multiple iterations.

To summarize, we showed how our method scales, has competitive speedup values and yields better accuracy results.

### 5.2.2. PTP performance on meshes with obtuse triangles

In the next experiments, we show that the presence of obtuse triangles affects the accuracy of the PTP and FM methods. We compared the accuracy of the methods in a regular grid mesh, shown in Figures 6a (input grid), 6b and 6c, that shows the distance maps for FM and PTP methods, respectively. Both algorithms have the same error of 0.1229%. Though, under the presence of obtuse triangles, the precision of the methods is slightly different.

We created an artificial mesh which contains obtuse triangles, to measure the robustness of the proposed method against the FM method. We are aware that FM method requires a pre-processing step, called unfolding, to remove all the obtuse triangles of the mesh. However, we omitted this step in both the PTP and FM to see how they behave in such conditions. The initial obtuse mesh is depicted in Figure 6d. This is a mesh, which contains 1170 vertices. The distance map obtained with the FM is depicted in Figure 6e. The result produced by the PTP method is shown in Figure 6f. We observe that the distance map produced by the PTP method is smoother and presents less error (2.799%) than the FM method (3.754%). We believe that this outcome is due to the distance correction or improvement that our algorithm performs in each iteration. The greedy behavior of Fast Marching in these cases is not suitable because obtuse triangles introduce error, which increases as the propagation moves away from the origin. Thus, we believe, according to the experiments, that our method is more robust than the FM algorithm when dealing with obtuse triangles without unfolding or subdivision.

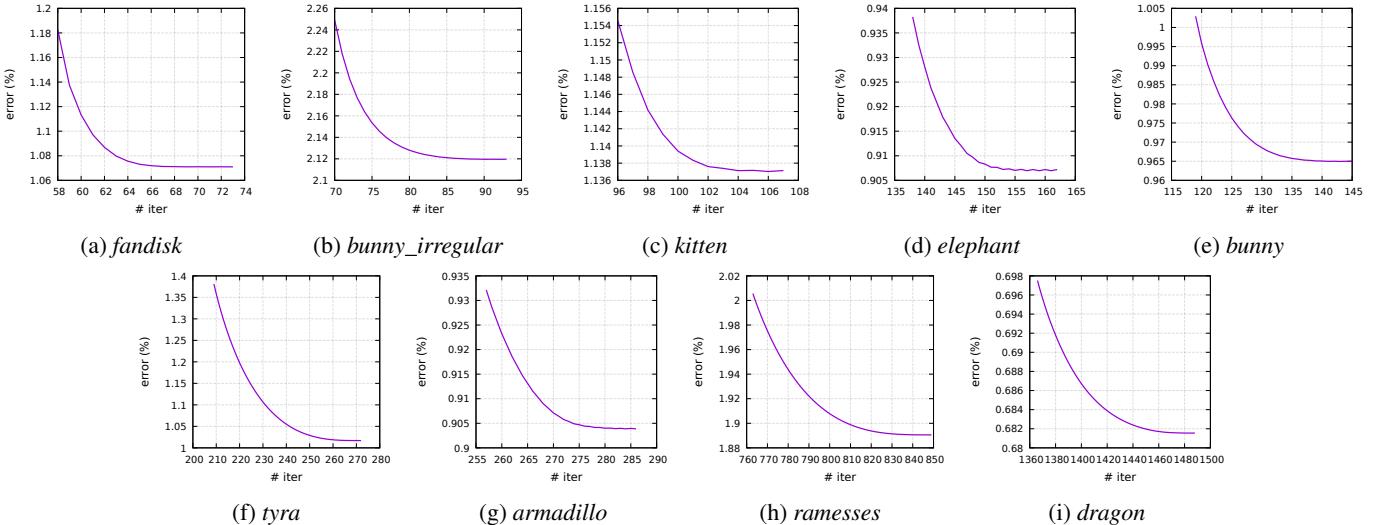


Figure 4: Mean absolute percent error per number of iterations.

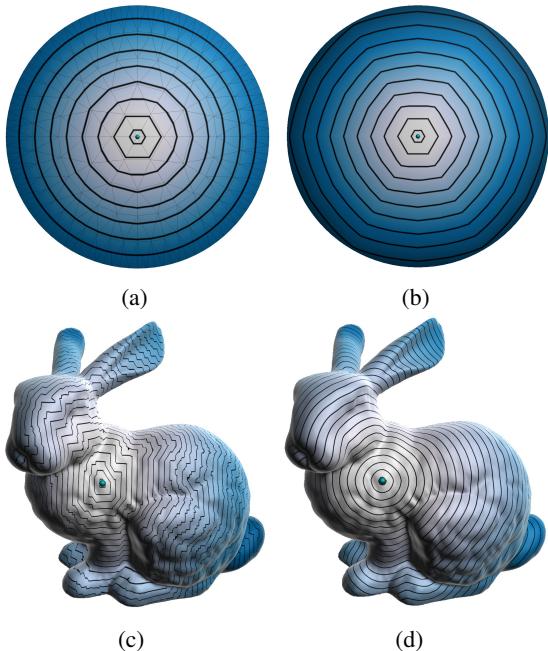


Figure 5: Level sets propagation in 5a and 5c, geodesics distances map in 5b and 5d.

### 5.3. Farthest Point Sampling (FPS) and Voronoi Diagrams

The Farthest Point Sampling (FPS) is a generic algorithm introduced by Eldar [14] that generates a regular sampling. At each iteration, the farthest vertex to the current set of samples  $S$  is computed and inserted into  $S$ .

The FPS algorithm exploits the fact that, in the PTP algorithm, the performance increases when the number of samples grows. This happens because the FPS algorithm in each iteration computes a new sample, which is added to the set of sources  $S$ . This case is very favorable to the PTP algorithm as the FPS tends to generate uniformly distributed samples.

Figure 7 presents the speedup of the computation of  $m$  sam-

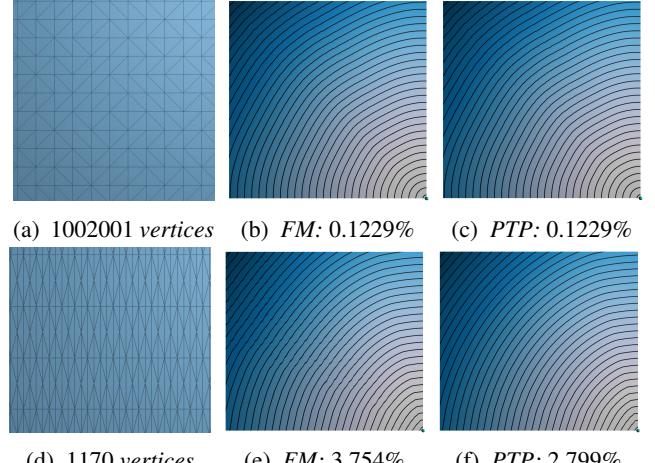


Figure 6: Distance maps and error comparison between a regular triangular mesh and an obtuse triangular mesh. Figure 6a shows a piece of a regular mesh, 6b shows the FM's distance map computed for 6a, and 6c shows PTP's distance map computed for 6a. Figure 6d shows a piece of an obtuse triangular mesh, 6e shows the FM's distance map computed for 6d, and 6f shows PTP's distance map computed for 6d.

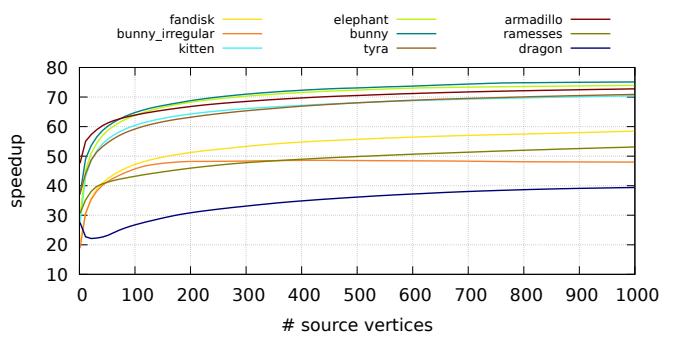


Figure 7: Farthest Point Sampling speedup with  $m \in [1 : 1000]$ .

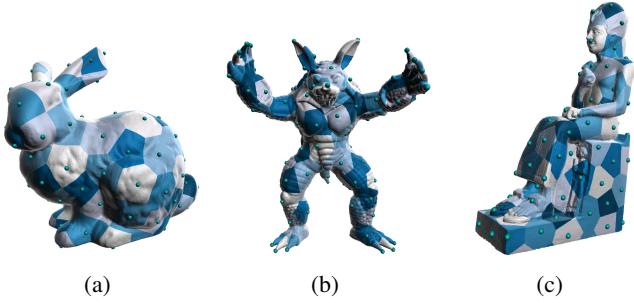


Figure 8: Computation of the Voronoi regions in 8a, 8b and 8c; from the calculation of  $m = 100$  samples with the FPS algorithm.

ples, using the FPS algorithm based on the GPU implementation of the PTP algorithm to compute the geodesic distances. Figure 8 presents some visual results for the computation of  $m$  samples using the FPS algorithm with the geodesics distances computed with the PTP algorithm; it also presents the Voronoi regions with  $m$  sources, which were computed with the FPS algorithm.

## 6. Estimative complexity analysis

In this section, we provide an estimative complexity analysis for our algorithm. To this end, instead of considering an update band based on the relative change, as in Algorithm 4.1, we use a fixed update band. This modification enables us to define an upper bound to our algorithm complexity. We can classify the *toplesets* in monotonically increasing, stationary and monotonically decreasing sequences. These *toplesets* sequences are present in any triangular mesh. Figure 9 shows the number of vertices per *topleset* for each mesh in the experiments. We can observe that for each mesh’s *toplesets*, the first ones shape a monotonically increasing sequence, the last *toplesets* shape a monotonically decreasing sequence, and the *toplesets* in the middle can alter among monotonically increasing, monotonically decreasing and stationary sequences.

A finite sequence of topological level sets  $(V_i)_{i=a}^b$ , is a set of contiguous topological level sets where  $a$  is the starting index of the sequence and  $b$  is the ending index. We classify a sequence  $(V_i)_{i=a}^b$  according to the behavior of the growth of the cardinality of each *topleset* that belongs to it, as follows:

1. *Monotonically increasing topleset sequence*: when  $|V_i| - |V_{i-1}| > 0, a \leq i \leq b$
2. *Stationary topleset sequence*: when  $|V_i| - |V_{i-1}| = 0, a \leq i \leq b$
3. *Monotonically decreasing topleset sequence*: when  $|V_i| - |V_{i-1}| < 0, a \leq i \leq b$

Now, let  $T$  be a triangle mesh with  $\rho$  *toplesets* and  $n = |V|$ .  $T$  is a *monotonically increasing mesh* when all of its *toplesets* define one monotonically increasing sequence  $(V_i)_{i=0}^\rho$ . Similarly,  $T$  is a *monotonically decreasing mesh* when all of its *toplesets* define one monotonically decreasing sequence  $(V_i)_{i=0}^\rho$ . When all its *toplesets* have the same cardinality, except for a small finite number of *toplesets*  $c \ll n$ , then the set of all *toplesets*

form a stationary sequence  $(V_i)_{i=c}^\rho$  and we say that  $T$  is a *stationary mesh*. In any other case we call  $T$  an *arbitrary mesh*.

The proposed algorithm propagates and relaxes distance information through the meshes’ *toplesets*.

At each iteration  $0 < k < K$ , where  $K$  is the maximum number of iterations, we define a band  $B(k)$ , called *update band*, that defines the vertices that will have their distances updated.  $B(k)$  is composed of a set of consecutive *toplesets*  $V_i$ ,  $s(B(k)) \leq i \leq k$ , where  $s(B(k))$  is the index of the first *topleset* in  $B(k)$ .  $B(k)$  and  $s(B(k))$  are computed using a breadth-first traversal through all consecutive *toplesets*  $V_i$  (see Section 6.1).

### 6.1. Complexity analysis sketch and comparison

We begin the complexity analysis of the Parallel Toplesets Propagation algorithm without taking the number of threads  $\mathcal{T}$  into consideration. In the final part of the analysis, we discuss the impact of  $\mathcal{T}$  which will have a strong effect in the throughput when  $\mathcal{T} \approx \sqrt{n}$ .

The first step in the algorithm is to define the order used to propagate the distances through the topological level sets. This can be done in linear time if the mesh is structured using a half-edge or any similar topological data structure. Now we discuss the definition of the size of the band  $B(k)$  at each iteration  $k$ . The size of the band depends on the combinatorial structure of the mesh but can be defined for each iteration  $k$  corresponding to each *topleset* in monotonically increasing, decreasing and stationary subsequences of the mesh. This step requires segmenting the mesh’s *toplesets* in such subsequences. To obtain such segmentation, it suffices to start with a given *topleset*  $V_0$  and analyze the behavior of the function  $2(H[|V_1| - |V_0|] - 1/2)$  where  $H[n]$  is the Heaviside step function. To classify the first sequence into increasing (+1), decreasing (-1) or stationary (0), we iterate through the *toplesets* using a breadth-first traversal until  $2(H[|V_1| - |V_0|] - 1/2)$  changes, signalizing the beginning of a new subsequence. The process continues until all *toplesets* are grouped into monotonically increasing, decreasing and stationary subsequences. We show later in the text that for monotonically increasing subsequences the size of the band is linear and equal to  $k/2$ , where  $k$  is the order of the iteration, and for monotonically decreasing and stationary subsequences the size is equal to one.

The next step that the Parallel Toplesets Propagation (Algorithm 4.1) executes is to relax the vertices’ distances using the update function for a total of  $K$  iterations. Consequently, the complexity of the PTP algorithm is given by the total number of update operations  $f(n)$ , in the main loop of the Algorithm 4.1, expressed as a function of the number of its vertices  $n = |V|$ :

$$f(n) = c \sum_{k=1}^K \sum_{r=s(B(k))}^k |V_r| \quad (4)$$

where  $K$  is the number of iterations,  $c$  is a constant, and  $s(B(k))$  is the index of the first *topleset* in  $B(k)$ .

Let  $V_r = \max_{r \in [0, \rho-1]} |V_r|$  be the topological level set with the largest number of vertices, and  $|V_r|$  the maximum number of vertices in any *topleset*. Thus, we claim that:

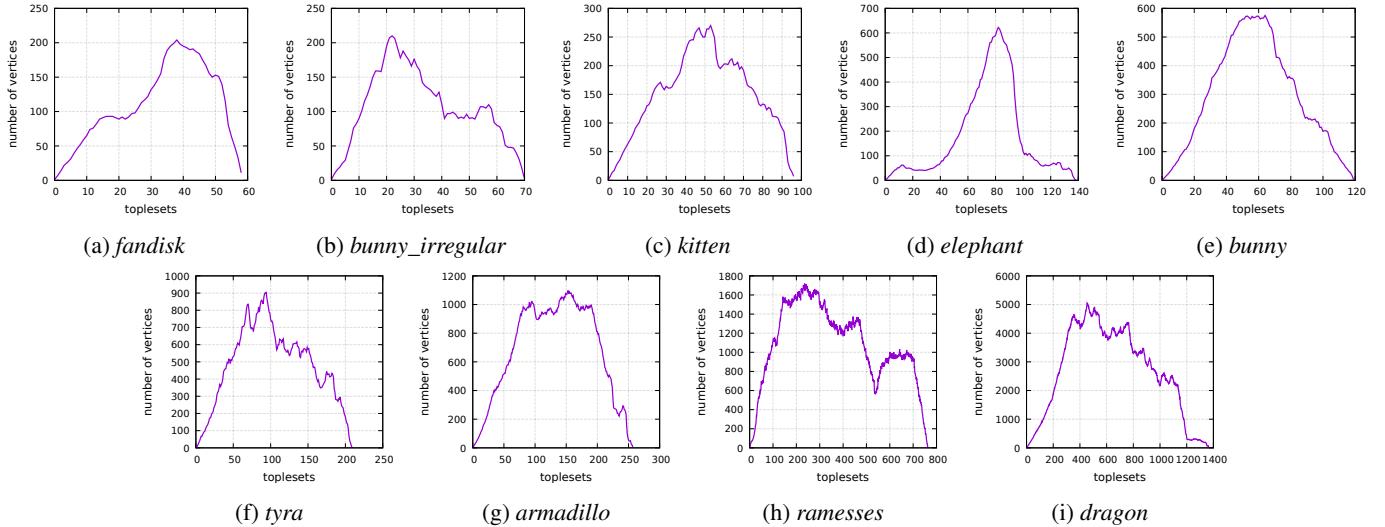


Figure 9: Number of vertices per *topleset* for each mesh in the experiments.

$$\begin{aligned} c \sum_{k=1}^K \sum_{r=s(B(k))}^k |V_r| &\leq c \sum_{k=1}^K \sum_{r=s(B(k))}^k |V_{r'}| \\ f(K) &\leq c|V_{r'}| \sum_{k=1}^K \sum_{r=s(B(k))}^k 1 \end{aligned}$$

Here,  $s(B(k))$  denotes the index of the starting *topleset* of the band  $B(k)$ . The number of iterations  $K$ , the number of vertices  $|V_r|$  in a *topleset*  $V_r$  and the depth of the bands  $|B(k)|$  all depend on the combinatorial topology of the mesh and are intricately coupled with each other. Nevertheless, an analysis can be done for meshes with specific *topleset* cardinality growth.

Let us now analyze the complexity of each one of the cases and posteriorly express the worst case complexity for an arbitrary mesh.

#### 6.1.1. Stationary mesh case

Let the complexity of a *stationary* mesh be defined as a function  $g(n)$ . For a stationary mesh, the number of *toplesets* multiplied by its constant cardinality  $|V_i| = |V_r|$  is  $\rho|V_r| = O(n)$ . To define the size of  $B(k)$  for a *stationary* mesh we use the result from the following lemma, which is proved as a special case of Lemma 5 in the Appendix B.2.

**Lemma 1.** [Number of iterations necessary to fix the distances of a *topleset* of a stationary mesh] Given a stationary mesh  $T$ , and a *topleset*  $V_k$ , the number of iterations necessary to fix the distances of all vertices  $v \in V_k$  is equal to 1.

Thus, the size of the band used in all iterations of a stationary mesh is equal to 1. As each *topleset* requires one iteration to fix all its distances and the total number of *toplesets* is  $\rho$ , then we claim that  $K = \rho$ . Finally, we state that the complexity of the proposed algorithm when applied to a stationary mesh is:

$$g(n) \leq c|V_{r'}| \sum_{k=1}^K (k - s(B(k))) = c|V_{r'}| \sum_{k=1}^K 1 = c|V_{r'}|K$$

$$g(n) = O(n)$$

#### 6.1.2. Monotonically increasing mesh case

Let the complexity of a *monotonically increasing* mesh be defined as a function  $h_1(n)$ . For a *monotonically increasing* mesh, Theorem 2 claims that  $\rho = O(\sqrt{n})$ .

**Theorem 2.** [Number of *toplesets*  $\rho$  of a monotonically increasing mesh] Let  $T = (V, E, F)$  be a triangular mesh and  $s \in V$  be a source vertex such that the cardinalities of its *toplesets*, ordered according to their distances to the sources, define a monotonically increasing sequence. Then the number of *toplesets*  $\rho$  from  $s$  satisfies  $\rho = O(\sqrt{n})$ , where  $n = |V|$ .

A proof to Theorem 2 is given in the Appendix B.1. In the same way, we can prove that for a triangular mesh  $T = (V, E, F)$  and a set of uniformly distributed source vertices  $S \subset V$ , the number of *toplesets* is  $\rho = O(\sqrt{\frac{n}{m}})$ , where  $m = |S|$ .

As the number of *toplesets* for a monotonically increasing mesh is  $\rho = O(\sqrt{n})$  and we are limited to  $n = |V|$ , the largest possible *topleset* has cardinality equal to  $|V_r| = O(\sqrt{n})$ .

To define the size of the band  $B(k)$  for a monotonically increasing mesh, we rely on the partial result presented in the proof in Appendix B.2. It defines the number of iterations  $K_r$  necessary to update all vertices in a *topleset*  $V_r$  as

$$K_r \leq \left\lceil \frac{(\Delta v - 3)}{2} \right\rceil (r - 1) + 1 \quad (5)$$

where  $\Delta v$  is the largest degree of the vertices  $v \in V_r$ . For a regular monotonically increasing mesh, whose valency is 6, we have  $K_r = 2r - 1$ ; we can observe that in the  $K_r$ -iteration the vertices in the *topleset*  $V_r$  with order  $r = \left\lfloor \frac{K_r+1}{2} \right\rfloor$  have their final distances computed; then we can define an updating band between all vertices included in the level sets  $r = \left[ \left\lfloor \frac{K_r+1}{2} \right\rfloor, K_r \right]$ .

Moreover, Theorem 3 states that for monotonically increasing meshes the maximum number of iterations is  $K = O(\sqrt{n})$ .

**Theorem 3.** [Maximum number of iterations  $K$  to compute the distances of a monotonically increasing mesh] Let  $T = (V, E, F)$  be a triangular mesh and  $s \in V$  be a source vertex such that the set of its toplesets  $V_r$  ordered by their distances to  $s$  define a monotonically increasing sequence. Then, the maximum number of iterations is  $K = O(\sqrt{n})$ , where  $n = |V|$ .

A sketch of the proof of Theorem 3 is given in the [Appendix B.2](#), which is supported by the experimental evaluation in Section 5. Based on the Theorem 3, we can compute an upper bound for the function  $h_1(n)$  which defines the complexity behavior of the PTP for monotonically increasing meshes.

$$\begin{aligned} h_1(n) &\leq c|V_{r'}| \sum_{k=1}^K (k - s(B(k))) \\ h_1(n) &= \frac{c}{2}|V_{r'}| \sum_{k=1}^K k = \frac{c}{4}|V_{r'}|K(K+1) \\ h_1(n) &= O(|V_{r'}|(K^2 + K)) = O(|V_{r'}|K^2) \\ h_1(n) &= O(|V_{r'}| \sqrt{n^2}) = O(n \sqrt{n}) \end{aligned}$$

#### 6.1.3. Monotonically decreasing mesh case

The complexity of a monotonically decreasing mesh is expressed here as a function  $h_2(n)$ . Similarly to the previous case, the upper bound to the number of toplesets for a monotonically decreasing mesh is  $\rho = O(\sqrt{n})$ . Although we do not present a formal proof for this case here, this property can be verified considering that this case is symmetrical to the case of *monotonically increasing* meshes.

Differently, though, the number of iterations necessary to fix all vertices in a topleset  $V_r$  in a monotonically decreasing mesh is similar to the stationary case and equal to  $K_r = 1$ . This occurs because the number of vertices from topleset  $V_{r-1}$  to  $V_r$  decreases and the edges in the chain associated to topleset  $V_{r-1}$  contains a sufficient number of vertices with fixed distances to set the correct distances of all vertices in  $V_r$ . Consequently,  $|B(k)| = 1$  and the number of iterations is  $K = \sqrt{n}$ . Moreover, in this case  $|V_r| = O(\sqrt{n})$

Hence, we claim that:

$$\begin{aligned} h_2(n) &\leq c|V_{r'}| \sum_{k=1}^K (k - s(B(k))) = c|V_{r'}| \sum_{k=1}^K 1 \\ h_2(n) &= c|V_{r'}|K = c' \sqrt{n} \sqrt{n} = O(n) \end{aligned}$$

#### 6.1.4. General case

An arbitrary mesh is composed of a combination of segments that can be *monotonically increasing*, *monotonically decreasing* and *stationary*. Let  $T$  be a mesh such that its toplesets  $V_i$ ,  $0 \leq i \leq \rho$  are grouped into contiguous *monotonically increasing*, *monotonically decreasing* and *stationary* sequences  $s_j = (V_i), k_1(j) \leq i \leq k_2(j)$  where  $0 \leq k_1(j), k_2(j) < \rho$ . Let us denote by  $|s_j| = \sum_{V_i \in s_j} |V_i|$  the amount of vertices in all toplesets of a sequence  $s_j$ . Let  $I$ ,  $D$  and  $C$  denote, respectively, the

sets of *monotonically increasing* ( $si$ ), *monotonically decreasing* ( $sd$ ) and *stationary* segments ( $ss$ ) of toplesets. The sum of all vertices in  $I \cup D \cup C$  is equal to  $n = |V|$ . The number of update operations in all segments is given by:

$$\begin{aligned} f(n) &= \sum_{si_i \in I} g(|si_i|) + \sum_{sd_i \in D} h_1(|sd_i|) + \sum_{ss_i \in C} h_2(|ss_i|) \\ f(n) &= \sum_{si_i \in I} |si_i| + \sum_{sd_i \in D} |sd_i| \sqrt{|sd_i|} + \sum_{ss_i \in C} |ss_i| \\ f(n) &\leq c_1 n + c_2 n \sqrt{n} + c_3 n \\ f(n) &= O(n \sqrt{n}) \end{aligned}$$

#### 6.1.5. Multi-source problem

The complexity of the proposed algorithm is impacted not only by the size of the mesh but also by the number  $m$  of source vertices  $s \in S$ . For special cases, in particular, when the source vertices are uniformly distributed, the number of toplesets can be shrunk by a factor of  $m$ .

For arbitrary meshes, the complexity is dominated by the set of monotonically increasing subsequences which depend on the number of vertices  $n$  and the diameter of the mesh  $\sqrt{n}$ . Hence when the number of toplesets shrinks to  $\sqrt{\frac{n}{m}}$  we have an overall complexity of:  $O\left(\frac{n \sqrt{n}}{\sqrt{m}}\right)$ .

This will have a great impact on the use of the PTP algorithm for implementing the *Farthest Point Sampling Algorithm*, an algorithm used to sample a mesh uniformly which will be described in 6.1.7.

#### 6.1.6. Complexity of the parallel implementation

For a parallel implementation of the PTP algorithm with a number  $\mathcal{T}$  of threads, applied to an input given by  $m = |S|$  sources and a mesh with  $n = |V|$  vertices and  $\rho$  toplesets, the final worst case complexity is  $O\left(\frac{n \sqrt{n}}{\mathcal{T} \sqrt{m}}\right)$ .

#### 6.1.7. Application: Farthest Point Sampling

The computational complexity analysis of FPS is  $O(mn \log n)$  using FM, whereas the parallel version of FPS using the proposed PTP has complexity of  $O(\sqrt{mn}^{3/2})$ . We present this complexity analysis in Section [Appendix A](#). The experimental results demonstrate that as the number of samples grows, there is an increase in the performance of the algorithm.

Table 4: Comparison of complexity algorithm analysis.

Algorithm	Complexity
Update Fast Marching	$O(1)$
Fast Marching	$O(n \log n)$
Farthest Point Sampling	$O(mn \log n)$
Update Fast Marching modified	$O(1)$
Parallel Toplesets Propagation	$O\left(\sqrt{\frac{n}{m}} \frac{n}{\mathcal{T}}\right)$
Farthest Point Sampling with PTP	$O\left(\sqrt{m} \frac{n^{3/2}}{\mathcal{T}}\right)$

Table 4 summarizes the analyzed algorithms. We include the term  $\mathcal{T}$  to the proposed algorithm because the parallelization is a feature in our algorithm, and also has impact in the FPS algorithm.

#### 6.1.8. Note

While the current complexity analysis, based on the sketch proof, states that the complexity of our algorithm has an upper bound of  $O(n\sqrt{n})$ , it is not tight enough if we consider the behavior of our algorithm for regular and non-regular meshes as it is shown in the experiments. The number of iterations  $K$  necessary for our algorithm to converge is bounded by the number of *toplesets*  $\rho$ ; it is  $K \leq c\rho$ . The experimental results depicted in Figure 4, show that  $c \approx 1.5$ .

Figure 10 shows the total number of vertices that are updated in each iteration of our algorithm and Figure 11 depicts the comparison of the number of *toplesets* per iteration between the fixed band (green curve) and the dynamic band (purple curve). The dynamic band is purely empirical, because it depends on the relative change of the previous iteration (*update band* in Algorithm 4.1). The fixed band set the number of *toplesets* to be updated per iteration according the analysis of the algorithm asymptotic complexity. We can observe that the green curve requires more iterations to converge and also it updates more *toplesets* per iteration. In contrast, the purple curve shows that the algorithm updates a smaller number of *toplesets* per iteration and also it requires fewer iterations to converge. Note that in the case of irregular meshes like *bunny\_irregular* and *tyra*, the empirical curve is closer to the fixed curve. We left as future work, finding a tighter upper bound for the algorithm asymptotic complexity.

## 7. Conclusion

We have presented a minimalistic parallel algorithm to compute approximate distance maps on triangular meshes. We also provide an implementation of our proposal on GPU.

The main advantages of our approach are:

- **Minimalistic:** We believe our method fits the class of minimalistic methods because it uses the fewest elements as possible to achieve the best results. The method itself takes into account the topological structure (*toplesets*) of the mesh and take advantage of it to correct the distances using only the necessary iterations. At the same time, the nature of our algorithm enables us to leverage powerful parallel architectures. Besides, our method avoids complicated pre-processing steps and dependence on parameters that are difficult to set up.
- **Scalable:** The distances are computed simultaneously through multiple iterations, instead of using a priority queue. We have demonstrated how our algorithm scales and leverages the use of GPU devices. Furthermore, the proposed method improves the speedup considerably when used to solve the multi-source problem.

- **Memory consumption management:** The proposed method supports single precision and does not require double precision to achieve a good accuracy result. Moreover, the speedup value is considerably increased when using single precision while the accuracy is maintained.

- **Robustness:** Our method achieves an accuracy similar to the classical Fast Marching method. Additionally, it performs better when applied to irregular meshes and large meshes.

From the experiments, we can conclude that our method, Parallel Toplesets Propagation, achieves competitive speedup values without any preprocessing time. For problems where multiple sources are required and when intensive distance queries will not be performed subsequently, such as the FPS algorithm, the speedup increases, as the number of sources increases.

A limitation of our method is that it requires triangular meshes. Also, our method requires more iterations when dealing with irregular meshes.

As future work, we aim at finding a tighter upper bound for the algorithm asymptotic complexity than the one presented in our complexity analysis. Also, we plan to give a formal analysis of the relation between  $\epsilon$  and the obtained distance error.

Finally, we also want to investigate the use of our method as part of a method to compute Centroidal Voronoi tessellations.

## Acknowledgments

The authors would like to express their gratitude to Cristian Lopez Del Alamo, Hueverton Souza, and Marcos Lage for their help and their valuable inputs. We also thank CAPES for funding this work. Models are courtesy of Stanford University (*armadillo*, *bunny*, *asian dragon*) and AIM@SHAPE shape repository.

## References

- [1] Kurtek, S, Klassen, E, Ding, Z, Avison, MJ, Srivastava, A. Parameterization-Invariant Shape Statistics and Probabilistic Classification of Anatomical Surfaces. Berlin, Heidelberg: Springer Berlin Heidelberg; 2011, p. 147–158.
- [2] Rabin, J, Peyré, G, Cohen, LD. Geodesic Shape Retrieval via Optimal Mass Transport. Berlin, Heidelberg: Springer Berlin Heidelberg; 2010, p. 771–784.
- [3] Bronstein, AM, Bronstein, MM, Kimmel, R. Generalized multidimensional scaling: a framework for isometry-invariant partial surface matching. Proc National Academy of Sciences (PNAS) 2006;103(5):1168–1172.
- [4] Bennour, J, Dugelay, J. Protection of 3d object visual representations. In: 2006 IEEE International Conference on Multimedia and Expo. 2006, p. 1113–1116.
- [5] Hamza, AB, Krim, H. Geodesic Object Representation and Recognition. Berlin, Heidelberg: Springer Berlin Heidelberg; 2003, p. 378–387.
- [6] Oliveira, GN, Torchelsen, RP, Comba, JLD, Walter, M, Bastos, R. Geodesic-driven visual effects over complex surfaces. Vis Comput 2011;27(10):917–928.
- [7] Sloan, PPJ, Rose III, CF, Cohen, MF. Shape by example. In: Proceedings of the 2001 Symposium on Interactive 3D Graphics. I3D ’01; New York, NY, USA: ACM; 2001, p. 135–143.
- [8] Mitchell, JSB, Mount, DM, Papadimitriou, CH. The discrete geodesic problem. SIAM J Comput 1987;16(4):647–668.

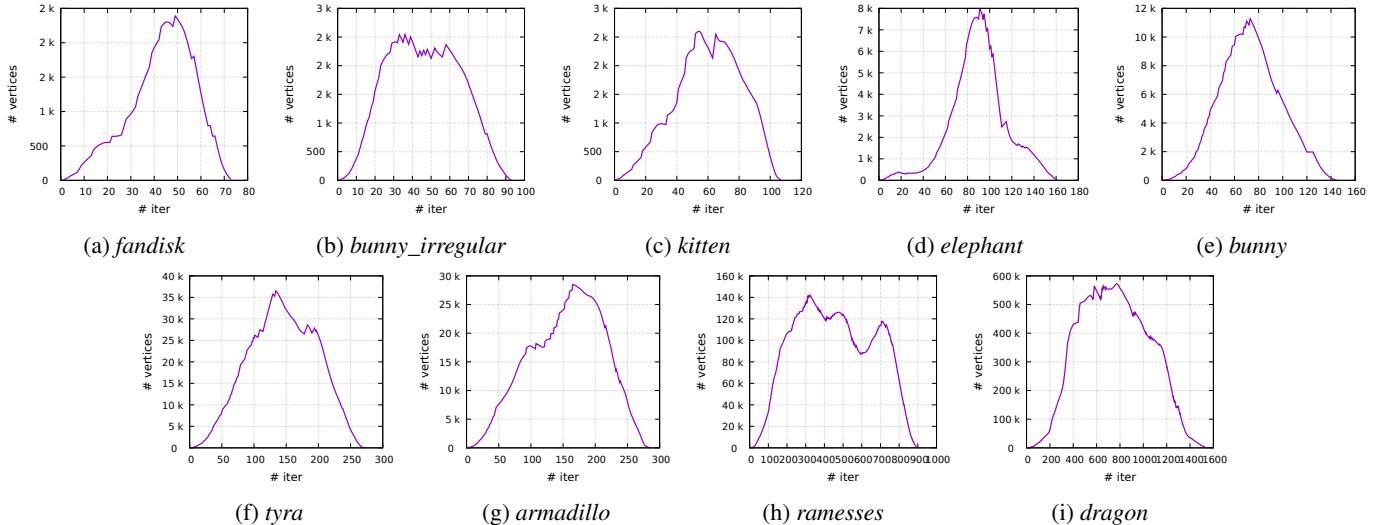


Figure 10: Number of updated vertices per iteration.

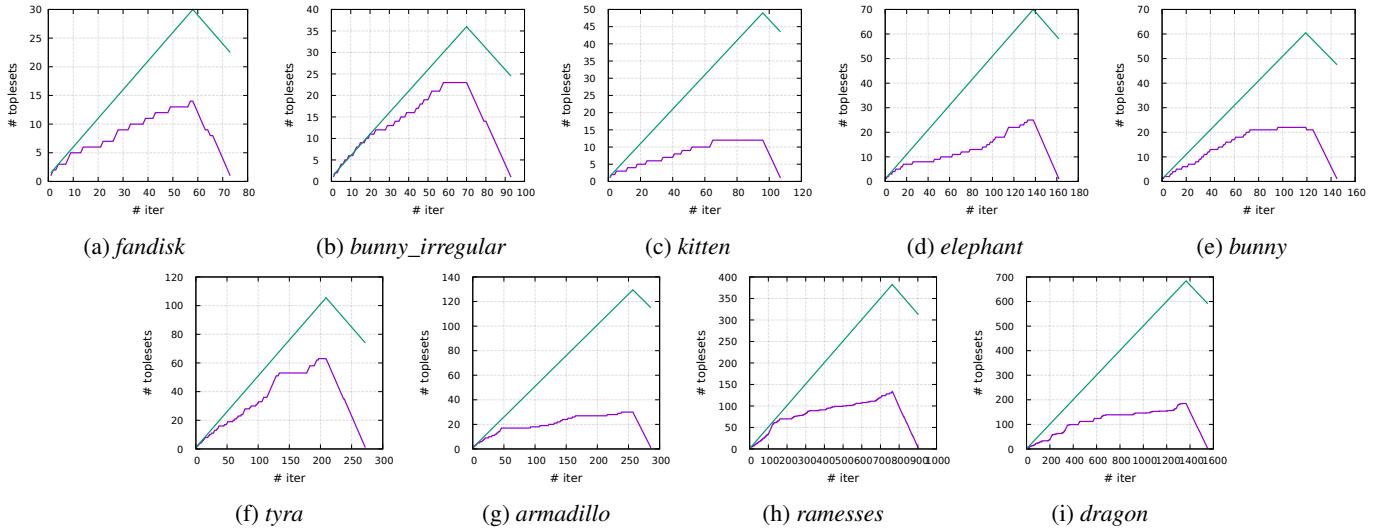


Figure 11: Comparing the number of *toplesets* per iteration between the dynamic band (purple curve), from empirical result for the Algorithm 4.1; and fixed band (green curve) with size equal to  $k/2$  *toplesets*, considered in the analysis and sketch proofs.

- [9] Surazhsky, V, Surazhsky, T, Kirsanov, D, Gortler, SJ, Hoppe, H. Fast exact and approximate geodesics on meshes. In: ACM SIGGRAPH 2005 Papers. SIGGRAPH '05; New York, NY, USA: ACM; 2005, p. 553–560.
- [10] Sethian, JA. Level set methods and fast marching methods : evolving interfaces in computational geometry, fluid mechanics, computer vision, and materials science. Cambridge monographs on applied and computational mathematics; Cambridge, GB: Cambridge University Press; 1999. Première édition parue sous le titre : Level set methods, 1996.
- [11] Kimmel, R, Sethian, JA. Computing geodesic paths on manifolds. In: Proc. Natl. Acad. Sci. USA. 1998, p. 8431–8435.
- [12] Crane, K, Weischedel, C, Wardetzky, M. Geodesics in heat: A new approach to computing distance based on heat flow. ACM Trans Graph 2013;32(5):152:1–152:11.
- [13] Zhu, H, He, L, Fu, S, Li, R, Han, X, Fu, Z, et al. Wolfpath: Accelerating iterative traversing-based graph processing algorithms on gpu. International Journal of Parallel Programming 2017;.
- [14] Eldar, Y, Lindenbaum, M, Porat, M, Zeevi, YY. The farthest point strategy for progressive image sampling. In: Pattern Recognition, 1994. Vol. 3 - Conference C: Signal Processing, Proceedings of the 12th IAPR International Conference on. 1994, p. 93–97 vol.3.
- [15] Qian, J, Zhang, Y, Zhao, H. Fast sweeping methods for eikonal equations on triangular meshes. SIAM Journal on Numerical Analysis 2007;45(1):83–107.
- [16] Polthier, K, Schmies, M. Straightest Geodesics on Polyhedral Surfaces. Berlin, Heidelberg: Springer Berlin Heidelberg; 1998, p. 135–150.
- [17] Martínez, D, Velho, L, Carvalho, PC. Computing geodesics on triangular meshes. Comput Graph 2005;29(5):667–675.
- [18] Weber, O, Devir, YS, Bronstein, AM, Bronstein, MM, Kimmel, R. Parallel algorithms for approximation of distance maps on parametric surfaces. ACM Trans Graph 2008;27(4):104:1–104:16.
- [19] Danielsson, PE. Euclidean distance mapping. Computer Graphics and Image Processing 1980;14(3):227 – 248.
- [20] Fu, Z, Jeong, WK, Pan, Y, Kirby, RM, Whitaker, RT. A fast iterative method for solving the eikonal equation on triangulated surfaces. SIAM J Sci Comput 2011;33(5):2468–2488.
- [21] Ying, X, Wang, X, He, Y. Saddle vertex graph (svg): A novel solution to the discrete geodesic problem. ACM Trans Graph 2013;32(6):170:1–170:12.
- [22] Wang, X, Fang, Z, Wu, J, Xin, SQ, He, Y. Discrete geodesic graph (dgg) for computing geodesic distances on polyhedral surfaces. Computer Aided Geometric Design 2017;52-53:262 – 284. Geometric Modeling and Processing 2017.

- [23] Crane, K, Weischedel, C, Wardetzky, M. The heat method for distance computation. *Commun ACM* 2017;60(11):90–99.
- [24] Tao, J, Zhang, J, Deng, B, Fang, Z, Peng, Y, He, Y. Parallel and Scalable Heat Methods for Geodesic Distance Computation. arXiv e-prints 2018;:arXiv:1812.06060.
- [25] Litman, R, Bronstein, AM. Spectrometer: Amortized sublinear spectral approximation of distance on graphs. CoRR 2016;.
- [26] Bronstein, A, Bronstein, M, Kimmel, R. Numerical Geometry of Non-Rigid Shapes. 1 ed.; Springer Publishing Company, Incorporated; 2008.
- [27] Lage, M, Lewiner, T, Velho, L. Che: A scalable topological data structure for triangular meshes. Tech. Rep.; 2005. URL: <http://citeseerv.ist.psu.edu/viewdoc/summary?doi=10.1.1.523.7580>.
- [28] Peyré, G. Fm toolbox matlab (fibonacci heap). 2014. URL: <https://github.com/gpeyre/numerical-tours>.
- [29] Sun, J. Meshlp package. 2008. URL: <https://github.com/areslp/matlab/tree/master/MeshLP/MeshLP>.
- [30] Sun, J. Multiple source/target exact geodesic (shortest path) algorithm for triangular mesh (triangulated 2d surface in 3d). 2008. URL: <http://code.google.com/p/geodesic/>.
- [31] Crane, K. Heat method: Cholmod implementation. 2013. URL: <https://github.com/dgdec/course>.
- [32] Rennich, SC, Stosic, D, Davis, TA. Accelerating sparse cholesky factorization on gpus. In: Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms. IA3 '14; Piscataway, NJ, USA: IEEE Press; 2014, p. 9–16.

## Appendix A. Complexity: Farthest Point Sampling

The FM algorithm has a complexity of  $O(n \log n)$  similar to the Dijkstra algorithm. The complexity of the FPS algorithm without taking into consideration the cost of calculating distances is  $O(mn)$ , where  $m$  is the number of samples in  $S$ . However, to compute a sub-sampling in a triangular mesh, we must compute the distance map with the FM algorithm in each iteration. Hence the FPS algorithm complexity is  $O(mn \log n)$ .

The number of operations  $f(n)$  of the FPS algorithm using the PTP algorithm is

$$f(n) = \sum_{i=1}^m \left( c_1 \frac{n \sqrt{n}}{\sqrt{i}} + c_2 n \right) \quad (\text{A.1})$$

where  $c_1$  and  $c_2$  are constants. The terms inside the sum in Equation A.1 represent the operations in each iteration: the PTP algorithm (Algorithm 4.1) which computes the geodesic distance map (first term) and the selection of the vertex with the maximum distance from the  $i$  sources (second term). Equation A.3 reduces Equation A.1:

$$\sum_{i=1}^m \left( c_1 \frac{n \sqrt{n}}{\sqrt{i}} + c_2 n \right) \leq \sum_{i=1}^m (c_1 + c_2) \left( \frac{n \sqrt{n}}{\sqrt{i}} \right) = c \sum_{i=1}^m \frac{n \sqrt{n}}{\sqrt{i}} \quad (\text{A.2})$$

where  $c = c_1 + c_2$ . We can conclude that:

$$\sum_{i=1}^m \left( c_1 \frac{n \sqrt{n}}{\sqrt{i}} + c_2 n \right) \leq c \sum_{i=1}^m \frac{n \sqrt{n}}{\sqrt{i}} = cn \sqrt{n} \sum_{i=1}^m \frac{1}{\sqrt{i}} \quad (\text{A.3})$$

It is possible to prove that:

$$\sum_{i=1}^m \frac{1}{\sqrt{i}} = O(\sqrt{m}) \quad (\text{A.4})$$

*Proof.* We can prove that:

$$\sum_{i=1}^m \frac{1}{\sqrt{i}} = O(\sqrt{m}) \quad (\text{A.5})$$

as a consequence of the following facts:

$$\sum_{i=1}^m \frac{1}{\sqrt{i}} \leq \int_1^m \frac{1}{\sqrt{x}} dx = 2\sqrt{m} - 2 \quad (\text{A.6})$$

hence, we can claim that:

$$\sum_{i=1}^m \frac{1}{\sqrt{i}} = O(\sqrt{m}). \quad (\text{A.7})$$

□

Thus, combining Equation A.3 with Equation A.6 we obtain

$$cn \sqrt{n} \sum_{i=1}^m \frac{1}{\sqrt{i}} = O(n \sqrt{n} \sqrt{m}) = O(\sqrt{mn^{3/2}}). \quad (\text{A.8})$$

## Appendix B. Sketch Proofs

In this section we analyse the behavior of the number of *toplesets* and the number of required iterations for certain classes of meshes. We show that the number of *toplesets* in a monotonically increasing mesh is  $\rho \leq \sqrt{n}$  and is  $\rho \leq n$  for a stationary mesh in the worst case, where  $n$  is the number of vertices; also we have established a relation between  $\rho$  and the maximum number of iterations  $K$  to compute the distances map for special classes of meshes. The definitions of monotonically increasing, decreasing and stationary meshes are established in the Section 6.

### Appendix B.1. Theorem 2

**Theorem 2.** [Number of *toplesets*  $\rho$  of a monotonically increasing mesh] Let  $T = (V, E, F)$  be a triangular mesh and  $s \in V$  be a source vertex such that the cardinalities of its *toplesets*, ordered according to their distances to the sources, define a monotonically increasing sequence. Then the number of *toplesets*  $\rho$  from  $s$  satisfies  $\rho = O(\sqrt{n})$ , where  $n = |V|$ .

*Proof.* Let a triangular mesh  $T = (V, E, F)$  and the *toplesets*  $V_r$ ,  $r \in [1 : \rho]$ . We can establish that:

$$|V_r| - |V_{r-1}| \geq c \quad (\text{B.1})$$

where  $c \geq 1$  and  $V_0 = \{s\}$  is the set containing the source vertex  $s$ . Without loss of generality we can choose a constant  $c = 1$ . Then we have

$$|V'_r| = |V'_{r-1}| + 1 \quad (\text{B.2})$$

where  $V'_r$  is a new increasing distribution of *toplesets* with the minimum difference such that  $|V'_{r-1}| \leq |V_r|$ ,  $r \in [1 : \rho' - 1]$ .

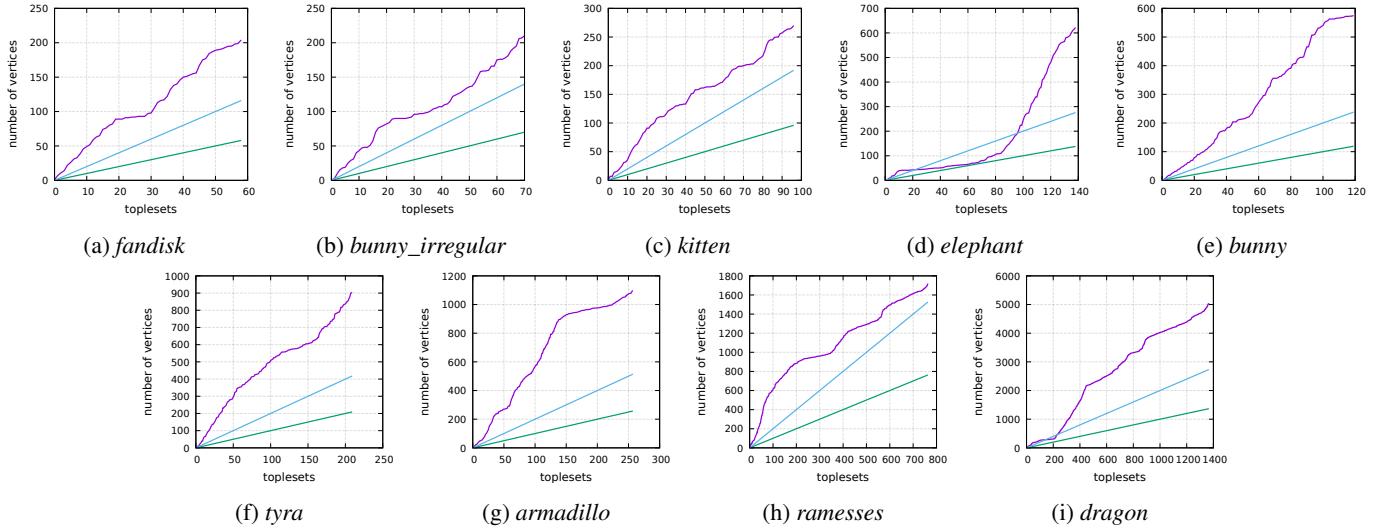


Figure B.12: Sorted *toplesets* meshes distribution.

Observe that  $\rho \leq \rho'$ ; then, we can solve the recurrence:  $|V'_r| = r$

for all  $r \in [1 : \rho' - 1]$ . Now, by  $\sum_{r=0}^{\rho'-1} |V'_r| = n$  we have:

$$1 + \sum_{r=1}^{\rho'-1} r = n$$

$$1 + \frac{\rho'(\rho' - 1)}{2} = n$$

$$2 + \rho'^2 - \rho' = 2n$$

$$\rho' = O(\sqrt{n})$$

because the fact that  $\rho \leq \rho'$ , we can claim that

$$\rho = O(\sqrt{n})$$

□

Figure B.12 shows the distribution of *toplesets* sorted by the number of vertices, and the functions  $y = x$  (green line) and  $y = 2x$  (blue line). We can observe that the curve increases over the minimum increase constant  $c = 1$ , which is considerate in the proofs. This plots confirm that the number of *toplesets*  $\rho \leq \sqrt{n}$ .

## Appendix B.2. Theorem 3

To prove Theorem 3, we need the next lemmas:

**Lemma 4.** Let  $T = (V, E, F)$  be a triangular mesh, and  $s \in V$  a source vertex; the number of *toplesets*  $\rho$  is  $\Omega(\log n)$ .

*Proof.* To count the number of vertices  $|V_r|$  at *topleset*  $r \in [1 : \rho]$ , we first consider the number of vertices in *topleset*  $V_r$  that must be connected to *topleset*  $V_{r-1}$ . We can claim that it is at least equal to degree  $\deg(v) - 3$ , for each  $v \in V_{r-1}$ . The number 3 in  $\deg(v) - 3$  accounts for the two mandatory vertices connecting neighbors in the same *topleset*  $V_{r-1}$  together with the neighbor

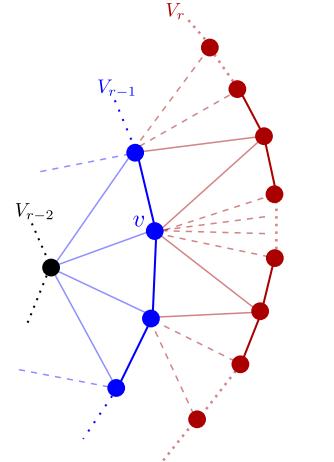


Figure B.13: Counting vertices at *topleset*  $V_r$ .

vertex at *topleset*  $V_{r-2}$ . This is the maximum number of vertices that  $V_r$  must have satisfying the constraints given by the degrees of the vertices  $v \in V_{r-1}$  (see Figure B.13). This is represented by Equation B.3:

$$|V_r| \leq \sum_{v \in V_{r-1}} (\deg(v) - 3) - |V_{r-1}|, \quad |V_0| = 1 \quad (\text{B.3})$$

We use the maximum degree  $\Delta_V = \max_{v \in V} \deg(v)$ , to limit the number of vertices at *topleset*  $r$ :

$$|V_r| \leq \sum_{v \in V_{r-1}} \deg(v) - 3|V_{r-1}| - |V_{r-1}| \leq \sum_{v \in V_{r-1}} \Delta_V - 4|V_{r-1}|$$

$$|V_r| \leq \sum_{v \in V_{r-1}} \Delta_V - 4|V_{r-1}|$$

$$|V_r| \leq \Delta_V |V_{r-1}| - 4|V_{r-1}|$$

$$|V_r| \leq (\Delta_V - 4)|V_{r-1}|. \quad (\text{B.4})$$

Let  $b = \Delta_v - 4$ . Solving recurrence (B.4) we obtain:

$$|V_r| \leq b^r, \quad |V_0| = 1$$

Now, knowing that  $\sum_{r=0}^{\rho-1} |V_r| = n$ , where  $n = |V|$  we have

$$\sum_{r=0}^{\rho-1} |V_r| \leq \sum_{r=0}^{\rho-1} b^r$$

$$n \leq \frac{b^\rho - 1}{b - 1} \leq b^\rho$$

$$\log_b n \leq \rho$$

therefore, as long as  $b$  is limited, we can conclude that the number of *toplesets*  $\rho$  is  $\Omega(\log n)$ .  $\square$

**Lemma 5.**  $K_r \leq \left\lceil \frac{\Delta_v - 3}{2} \right\rceil (r - 1) + 1$  iterations compute the final distances of all vertices  $v \in V_r$ .

*Proof.* Let  $K_r$  be the total number of iterations necessary to fix the final distance for all vertices  $v \in V_r$ . Similarly, let  $K_{r-1}$  be the total number of iterations necessary to fix the final distances of the vertices  $v' \in V_{r-1}$  in the previous *topleset*.

We claim that  $V_r$  requires at most the number of iterations of  $V_{r-1}$  plus  $i$  iterations, that is:

$$K_r \leq K_{r-1} + i \quad (\text{B.5})$$

solving this recurrence, we obtain:

$$K_r \leq i(r - 1) + 1. \quad (\text{B.6})$$

In order to compute a superior bound to  $i$ , we analyze the behavior (see Table B.5) of the number of remaining vertices  $|\bar{V}_r|$  in  $V_r$  to have its distances fixed at each iteration  $i'$  and the total number of fixed distances in  $V_{r-1} \cup V_r$  (first column of Table B.5).

Table B.5: Updated vertices in  $V_r \cup V_{r-1}$  at beginning of iteration  $i'$  and remaining vertices  $|\bar{V}_r|$  at end of iteration  $i'$ .

$i'$	$V_r \cup V_{r-1}$	$ \bar{V}_r $
0	$ V_{r-1} $	$ V_r $
1	$2 V_{r-1} $	$ V_r  -  V_{r-1} $
2	$4 V_{r-1} $	$ V_r  -  V_{r-1}  - 2 V_{r-1} $
3	$6 V_{r-1} $	$ V_r  -  V_{r-1}  - 2 V_{r-1}  - 2 V_{r-1} $
4	$8 V_{r-1} $	$ V_r  -  V_{r-1}  - 2 V_{r-1}  - 2 V_{r-1}  - 2 V_{r-1} $
$\vdots$	$\vdots$	$\vdots$
$i$	$2i V_{r-1} $	$ V_r  -  V_{r-1}  - 2(i - 1) V_{r-1} $

In Figure B.14, the circular icon corresponds to vertices in  $V_{r-1}$  with distances already fixed. The triangle icons indicate vertices with distances fixed at iteration  $i' = 1$ . At  $i' = 1$  each edge in  $V_{r-1}$  together with a vertex in  $v \in V_r$  (triangle icon) defines an updating triangle that is used to compute the final distance of  $v$ . Hence, there are  $|V_{r-1}|$  new fixed distances totaling  $2|V_{r-1}|$ . For  $i' > 1$ , we have the double of edges that can be

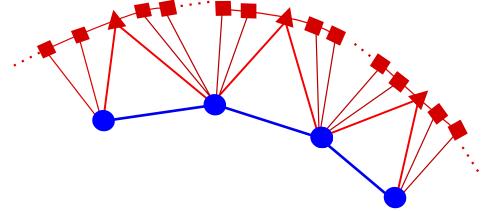


Figure B.14: Counting iterations to updated the *topleset*  $V_r$ .

used to define updating triangles. Hence,  $2|V_{r-1}|$  vertices have their distances fixed except for  $i' = i$  which may fix less than  $|V_{r-1}|$  vertices as  $|\bar{V}_r| \geq 0$  (the number of remaining distances to be fixed cannot be negative).

When all vertices  $v \in V_r$  have their distances fixed,  $|\bar{V}_r| = 0$ . Hence, we state that:

$$|V_r| - |V_{r-1}| - 2(i - 1)|V_{r-1}| = 0 \quad (\text{B.7})$$

$$i = \left\lceil \frac{|V_r| + |V_{r-1}|}{2|V_{r-1}|} \right\rceil \quad (\text{B.8})$$

Consequently, we also claim that:

$$K_r \leq \left\lceil \frac{|V_r| + |V_{r-1}|}{2|V_{r-1}|} \right\rceil (r - 1) + 1 \quad (\text{B.9})$$

Besides, according to Lemma 4, equation (B.4) (lower bound to the number of *toplesets* of a mesh), we know that:

$$|V_r| \leq (\Delta_v - 4)|V_{r-1}|$$

where  $\Delta_v$  is the maximum degree of  $v \in V_r$ .

Thus,

$$\begin{aligned} |V_r| + |V_{r-1}| &\leq (\Delta_v - 4)|V_{r-1}| + |V_{r-1}| \\ \frac{|V_r| + |V_{r-1}|}{2|V_{r-1}|} &\leq \frac{(\Delta_v - 4)|V_{r-1}| + |V_{r-1}|}{2|V_{r-1}|} \\ \frac{|V_r| + |V_{r-1}|}{2|V_{r-1}|} &\leq \frac{\Delta_v - 3}{2} \end{aligned}$$

finally:

$$\begin{aligned} K_r &\leq \left\lceil \frac{|V_r| + |V_{r-1}|}{2|V_{r-1}|} \right\rceil (r - 1) + 1 \\ K_r &\leq \left\lceil \frac{\Delta_v - 3}{2} \right\rceil (r - 1) + 1. \end{aligned} \quad (\text{B.10})$$

When the mesh is regular  $\Delta_v = 6$  and we have:

$$K_r \leq 2(r - 1) + 1$$

$$K_r \leq 2r - 1.$$

When the mesh has *toplesets* whose cardinality is stationary, that is  $|V_r| = |V_{r-1}|$  for all  $r$  we have:

$$\begin{aligned} K_r &\leq \left\lceil \frac{|V_r| + |V_{r-1}|}{2|V_{r-1}|} \right\rceil (r - 1) + 1 \\ K_r &\leq \left\lceil \frac{2|V_{r-1}|}{2|V_{r-1}|} \right\rceil (r - 1) + 1 \end{aligned}$$

then  $K_r \leq r$ , and one iteration is sufficient to fix all distances in  $V_r$  once  $V_{r-1}$  has its distances fixed.  $\square$

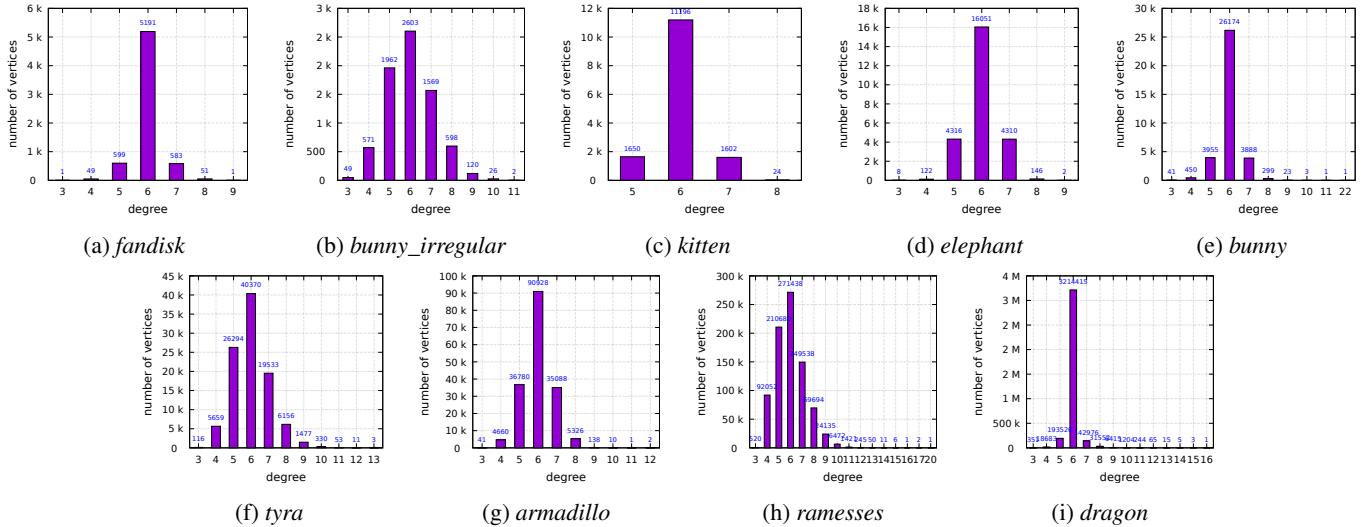


Figure B.15: Degree histogram for the meshes in the experiments.

Figure B.15 shows the degree histogram considering all the meshes used in the experiments. We can observe that the predominant degrees are 6, 4 and 5.

**Theorem 3.** [Maximum number of iterations  $K$  to compute the distances of a monotonically increasing mesh] Let  $T = (V, E, F)$  be a triangular mesh and  $s \in V$  be a source vertex such that the set of its toplesets  $V_r$  ordered by their distances to  $s$  define a monotonically increasing sequence. Then, the maximum number of iterations is  $K = O(\sqrt{n})$ , where  $n = |V|$ .

*Proof.* To proof Theorem 3 we use the Lemma 5. According to it, the number of iterations  $K$  to compute the final distance to the last topleset  $V_\rho$  is  $K = c(\rho - 1) + 1$ ,  $\rho \leq \sqrt{n}$ , and  $c = \lceil \frac{\Delta_v - 3}{2} \rceil$ . Thus, we can claim that  $K \leq c\sqrt{n}$  and  $K = O(\sqrt{n})$ . Observe that for regular meshes  $c = 2$ .  $\square$