IPOL IMAGE PROCESSING ON LINE

Published in Image Processing On Line on 2015–11–23.
Submitted on 2013–08–19, accepted on 2014–11–24.
ISSN 2105–1232 © 2015 IPOL & the authors CC–BY–NC–SA
This article is available online with supplementary materials,
software, datasets and online demo at
https://doi.org/10.5201/ipol.2015.102

An Implementation and Parallelization of the Scale-Space Meshing Algorithm

Julie Digne

LIRIS, CNRS UMR 5205,Université Lyon 1 (julie.digne@liris.cnrs.fr)

Communicated by Pascal Monasse *Demo edited by* Pascal Monasse

Abstract

Creating an interpolating mesh from an unorganized set of oriented points is a difficult problem which is often overlooked. Most methods focus indeed on building a watertight smoothed mesh by defining some function whose zero level set is the surface of the object. However in some cases it is crucial to build a mesh that interpolates the points and does not fill the acquisition holes: either because the data are sparse and trying to fill the holes would create spurious artifacts or because the goal is to explore visually the data exactly as they were acquired without any smoothing process. In this paper we detail a parallel implementation of the *Scale-Space Meshing* algorithm, which builds on the scale-space framework for reconstructing a high precision mesh from an input oriented point set. This algorithm first smoothes the point set, producing a singularity free shape. It then uses a standard mesh reconstruction technique, the Ball Pivoting Algorithm, to build a mesh from the smoothed point set. The final step consists in back-projecting the mesh built on the smoothed positions onto the original point set. The result of this process is an interpolating, hole-preserving surface mesh reconstruction.

Source Code

The ANSI C++ source code permitting to reproduce results from the on-line demo is available at the IPOL web page of this article[1]. The scale-space meshing algorithm uses the Ball Pivoting Algorithm which is linked to patent US6968299B1. It is made available for the exclusive aim of serving as a scientific tool to verify the soundness and completeness of the algorithm description.

Keywords: surface reconstruction; scale-space

[1] https://doi.org/10.5201/ipol.2015.102

1 Introduction

Surface mesh reconstruction methods can be divided into two categories: methods that build a closed surface out of the input point set (implicit surface methods) and methods that aim at finding a mesh whose vertices are the input point set. While the first approach is widely spread because it generates smooth, closed and economical meshes, usually by extracting the zero level set of some potential field ([8], [9], [10]), the second kind of approach is crucial in some cases, when the surface is intrinsically open, or when one wants to visualize exactly the outcome of the laser scanner. For example, a LIDAR device acquiring a town yields point sets that are impossible to reconstruct with implicit surface reconstruction methods. And even in the case of a laser-scanner acquired object surface, an implicit surface reconstruction method will lose the input surface accuracy: it will reconstruct a smooth surface. In this paper we are interested in the second kind of approach, which aims at building a mesh interpolating the initial point set.

Most mesh interpolating methods are based on building a Delaunay diagram of the input point set and filtering facets. Though efficient, these processes involve building a global structure that is not always desirable. To overcome this limitation, the Ball Pivoting Algorithm is a powerful heuristic for building a Delaunay-like triangulation of scattered 3D points without resorting to a global structure. It was introduced by [1] and provided a way to build an *interpolating mesh* in contrast with other research directions aiming at building an *approximating mesh* (e.g. [2], [8], [9], [10]). Although this method works well for noiseless data, or data with details at a scale coherent with the chosen radius of the ball, it fails dramatically when data contains small details or noise, as will be shown in Section 7. Yet if the data have to be smoothed before building the mesh, then the interpolating property of the method is lost. To overcome this limitation [6] proposed a method that, through the use of a scale-space, allows for a better interpolation of the original raw points even in the presence of noise and small details.

A scale-space is a representation of a shape at different geometric scales, i.e. at different degrees of smoothness. The principle behind the scale-space meshing method is that once the shape is smoothed, one can use a standard surface mesh reconstruction algorithm to interpolate the point set. The mesh for the original scale can then be deduced from the smoothed scale mesh. In short, the scale-space meshing is one of the applications of the scale-space framework, which can be used to infer a wide variety of information on an original point set by estimating it on a smoother version of it.

This paper describes a parallel implementation of the scale-space meshing algorithm. The mesh reconstruction part of the algorithm is based on the Ball Pivoting Algorithm for which we provide a parallel implementation slightly adapted from [3], where the reader will find all necessary details on the data structures. The next paragraph is a brief reminder of the data structures used in this implementation.

Data structures. The scale-space meshing algorithm needs two important data structures: a search structure that allows for fast queries of neighborhoods and a mesh structure that will be built incrementally. The search structure is an octree that will store the points and provide methods for fixed range neighborhood queries. The mesh structure we build is a *manifold with holes* structure: a set of triangular facets between vertices were each triangle edge can only be adjacent to two facets, and edges with only one adjacent triangle are authorized. We refer the reader to [3] for all the details on these two structures.

The remainder of this paper is organized as follows: Section 2 explains the scale-space and its particular implementation. Section 3 explains how the scale-space framework is used in the reconstruction setting. Section 4 describes the parallelization of the method. Section 5 deals with the choice of parameters for the method. Section 6 explains the dependencies of the code. Finally Section 7 shows several experiments using the Scale-Space Meshing algorithm and offers comparisons

with other existing methods.

2 A Scale-Space for Point Sets

2.1 Definitions

Let $\mathcal{M}$ be a smooth surface in $\mathbb{R}^3$ assumed to be at least $\mathcal{C}^2$. At each point x of the surface one can define a normal direction $\mathbf{n}(x)$, a vector perpendicular to the tangent plane. There are two possible orientations for this vector (pointing either inwards or outwards). In the continuous surface setting, the normal $\mathbf{n}(x)$ is always oriented towards the concavity of the shape. At each point x, one can pick a normal plane containing the normal and a chosen tangent direction (i.e. a vector in the tangent plane). The intersection of this plane and the surface is a planar curve whose curvature at x is the surface directional curvature corresponding to the chosen tangent direction. The principal curvatures $k_1(x)$ and $k_2(x)$ of the surface at x are defined as the minimum and maximum directional curvatures of the surface and the *mean curvature* of $\mathcal{M}$ at x is $H(x) = \frac{1}{2}(k_1(x) + k_2(x))$.

The scale-space for point sets is described in [5] and [6]. It consists in applying the mean curvature motion (MCM) to a set of points. The mean curvature is written:

$$\frac{\partial x}{\partial t} = H(x)\mathbf{n}(x).$$

In other words, all points move toward the concavity of the shape at a rate equal to the surface mean curvature.

2.2 Mean Curvature Motion Implementation

As shown in [6], the mean curvature motion can be approximated by the iterative process of projecting each point of the data set onto its local regression plane. The iterative projection process allows for the computation of robust geometric information (after several scale-space iterations) and this geometric information can be backtracked to the initial scale, for example by associating the curvature of an evolved point computed at a scale t with the initial position of the point at scale 0. The method for computing one step of the mean curvature motion is explained in Algorithm 1.

Numerically, the orientation of the normal is different from the continuous case: the normals are not oriented toward the concavity but *consistently* over the surface (i.e. all normals point either inwards or outwards). The projection algorithm making no use of the normal, the choice of the orientation is irrelevant for the mean curvature motion, but having this information is useful for the ball pivoting algorithm.
Three steps require some explanations:

- Line 3: If the point does not have enough neighbors, it is considered as an outlier and discarded.
- Line 4: $w(q)$ ensures stability of the approximation by giving more weight to neighboring points than to remote points:
$$w(q) = \exp -\frac{\|p-q\|^2}{2r^2}.$$
- Line 6: The regular mean curvature motion would stop after line 6, yet we use a slightly modified motion where the normals are smoothed jointly with the positions, to reflect the normal direction of the current shape (line 7). Smoothing normals is important in case of noisy normals input but also to improve the performances of the Ball Pivoting.

Algorithm 1: $MCM(p, \mathcal{P}, r)$: One step of the Mean Curvature Motion (MCM)

Input: A point set $\mathcal{P}$, a query point p, a radius r
Output: A point p', result of one discrete step of the MCM applied to p

1 Get the set of neighbors $\mathcal{N}_r(p)$ out of $\mathcal{O}$;
2 **if** $\#\mathcal{N}_r(p) < 5$ **then**
3 Remove point p
4 $\bar{p} \leftarrow \frac{\sum_{q\in\mathcal{N}_r(p)} w(q)q}{\sum_{q\in\mathcal{N}_r(p)} w(q)}$ and $C \leftarrow \sum_{q\in\mathcal{N}_r(p)} w(q)(q-\bar{p})\cdot(q-\bar{p})^T$;
5 $v_0 \leftarrow$ eigenvector corresponding to the least eigenvalue of C;
6 $p' \leftarrow p - \langle p-\bar{p}, v_0\rangle v_0$;
7 $p'.n \leftarrow \frac{p-p'}{\|p-p'\|} \cdot sign(\langle p-p', p.n\rangle)$;

The barycenter and covariance computation (Line 4 - function *performLocalPCA* in the code) is performed using a numerically stable online Algorithm [14]. The covariance matrix being a 3×3 real symmetric matrix, a simple ad hoc eigendecomposition algorithm is used [13]. Algorithm 1 is exactly one step of the projection on the Moving Least Squares Surface of order 1 (MLS1) induced by the point set. Moving Least Squares Surfaces are surfaces defined locally by performing an explicit surface regression (e.g. a polynomial surface defined over the local tangent plane) around a position.

2.3 Scale-Space Implementation

As previously stated, the discrete scale-space consists in applying Algorithm 1 to the whole point set. In practice, the iterations require building a new point set after each iteration. But there is no need to keep at each step all the results of all the iterations. In fact, we only need to store the original point set (for back-projection, which will be explained later), the result of the last iteration and a buffer for the point set being constructed at the current iteration.

Compared to [3], points are still stored in an octree, but the nodes of the octree do not contain a single set of points but three sets of points. The first set contains the points of the original cloud and the other two serve for storing intermediary scale-space iterations. More precisely, *Set* 0 will be preserved, it contains the original point positions. *Sets* 1 and 2 are alternatively the result of the previous and current scale-space iterations. The octree always stores the index of the current set and updates it after each iteration. Spatial queries are only slightly modified by this, the same octree is traversed using the same method with the only difference that the lists with the current index are considered at each iteration, while the others are ignored. In practice, when applying the mean curvature motion to either the initial point set or to the result of an even number of iterations, the result of the projection will be stored in *Set* 1. The result of an odd number of projection iterations will be stored in *Set* 2. This way the initial and final sets will always be available. In addition, each point keeps track of the point of $\mathcal{P}_0$ it originated from (Algorithm 2, lines 3 and 10). For completeness, we summarize the method in Algorithm 2.

In practice, the implementation uses a little trick to be faster. Instead of getting the neighbors of each point *from scratch*, it uses the fact that when looking for neighbors, one needs to get the cell containing the point at a given depth [3]. Yet for all points of a given cell this parent cell is the same. The process is then to traverse the octree in a depth-first manner: when a cell A at the right depth is reached, the cell is stored and the traversal continues, until a leaf descending from A is reached. Thus the points in this leaf will be processed faster, since there is no need to look for the right ancestor node for each point. This is why there are overloaded *applyScaleSpace* functions in the code. We refer the reader to [3] for a precise explanation of the neighbor search functions.

Algorithm 2: *scale_space*$(\mathcal{P}, N, r)$: applying the scale-space iterations to a point set $\mathcal{P}$

Input: A point set $\mathcal{P}$ a number of iterations N and a radius r
Output: A modified point set $\mathcal{P}_N$

Sort and store $\mathcal{P}$ in $\mathcal{P}_0$ endowed with an octree structure;
for $p \in \mathcal{P}_0$ **do**
    Set $p.origin \leftarrow p$;
$idx \leftarrow 0$;
for $i = 0, \cdots, N-1$ **do**
    $new\_idx \leftarrow mod(idx, 2) + 1$;
    for $p \in \mathcal{P}_{idx}$ **do**
        $p' \leftarrow MCM(p, \mathcal{P}_{idx}, r)$;
        Store p' in $\mathcal{P}_{new\_idx}$;
        $p'.origin \leftarrow p.origin$;
    if $idx > 0$ **then**
        Erase $\mathcal{P}_{idx}$
    $idx \leftarrow new\_idx$;

In the end one has the original point set, the result of the scale-space iteration and a correspondence between the points of both point sets which allows for the *back-projection* of the mesh obtained at a coarse scale to obtain the final fine scale mesh. Contrary to other existing scale-space structures (e.g. [11]), this scale-space does not iteratively subsample the shape: the same amount of points is preserved throughout the scale-space iterations. It is a geometric multi-resolution scheme, where the initial shape becomes increasingly smooth but the data size instead does not change. In practice however, some points might be lost if their neighborhoods do not contain enough points for estimating a regression plane, but this loss is minimal (less than 0.1% of the points in general). The result of the scale-space iterations is a denoised point set representing a smooth surface. This is precisely the kind of data that is very easily meshed by an interpolating method, such as the Ball Pivoting Algorithm [1]. The next section explains how to use this scale-space for meshing an input point set.

3 Using the Scale-Space for Surface Reconstruction

The scale-space meshing algorithm is summarized in Figure 1. It consists of three steps:

- *Scale-space iterations.* The scale-space is iterated on the point set as described in Section 2.
- *Meshing step.* A triangular mesh is built out of the smoothed point set resulting from the scale-space iterations.
- *Back-projection.* The resulting mesh is back-projected onto the original point set, creating an interpolating mesh.

The scale-space iteration step consists in applying Algorithm 2 to the set of points and does not require any further explanation. Notice that if the number of scale-space iterations is 0, then the scale-space meshing reduces to the traditional Ball Pivoting Algorithm. The next subsections detail the meshing and back projecting steps.

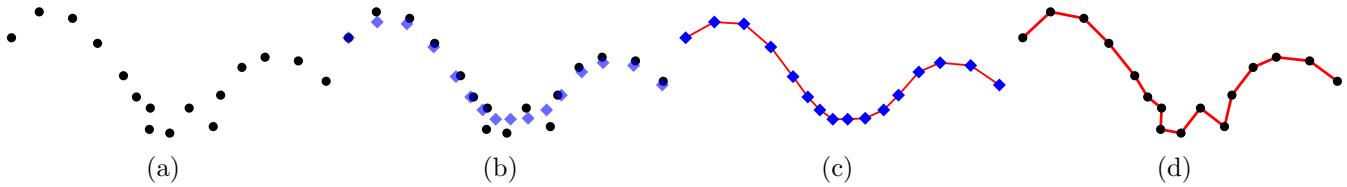


Figure 1: Graphical summary of the scale-space meshing algorithm. Initial points (black dots) (a), initial points with their corresponding filtered positions after scale-space iterations (transparent blue diamonds) (b), mesh of the filtered positions (c) and back-projection of the mesh onto the initial positions (d).

3.1 Input Data

The input data of the algorithm is a set of oriented points, an unorganized set of points *consistently oriented*. In other words, all normals should point either inwards or outwards. These normals are required by the Ball Pivoting Algorithm to ensure that the resulting surface mesh is orientable. It is important to notice that the scale-space iterations themselves do not require this knowledge: the MCM can be applied to points without normals since computing the regression plane does not use this information.

3.2 Meshing Step

The only constraint on the meshing method is that it should interpolate the points and not create any additional vertex. We chose to use the Ball Pivoting Algorithm [1], whose implementation is described thoroughly in [3]. We use this implementation to reconstruct an interpolating surface from the smoothed positions.

In a nutshell, the Ball Pivoting Algorithm builds incrementally a manifold mesh from an input point set by adding a triangle to the mesh if there is an empty-interior ball of fixed radius r passing through three data points. It then creates a triangle and the ball is pivoted around each of the edges until another point is met. The process being incremental, it allows for a fast parallelization. The resulting triangulated surface is a manifold mesh possibly with holes and multiple connected components. The mesh is efficiently constructed on the smoothed point set and is therefore not interpolating the original point set, which is why a *back-projection* is performed next. At the end of the Ball Pivoting Algorithm, similarly to [3], we apply an additional step to fill triangular holes that may remain. We refer the reader to [3] for details about this step.

3.3 Back-projection

Back-projecting the resulting mesh on the original point set is simple since every point of the last point set keeps a track of the point it originated from (see lines 3 and 10 of Algorithm 2). Therefore we simply have to transfer the connectivity from a point to its origin. The back-projection is summed up in Algorithm 3. In the proposed implementation it is done directly when saving the mesh.

The result of this back-projection is an interpolating mesh of the original point set. Yet, the back-projection process does not guarantee that the final mesh will be self-intersection free. By construction, the coarse scale mesh cannot self-intersect. Yet, moving each point to its original position might cause self-intersections of the mesh. For a reasonable radius and number of iterations, this phenomenon was not observed or at least not in a way that would hinder the visualization.

Algorithm 3: *back_project*(M_N) back projecting the final mesh

Input: A surface mesh M_N of the last smoothed point set $\mathcal{P}_N$
Output: A surface mesh M_0 of the initial point set $\mathcal{P}_0$
1 **for** *each triangle* $t \in M_N$ **do**
2 Let v_0, v_1, v_2 be the vertices of t;
3 Create a triangle t' with vertices $v_0.origin, v_1.origin, v_2.origin$;
4 Add t' to M_0;

The output of the algorithm is a set of vertices linked by triangular facets stored in the Stanford PLY format. In this format each facet is given by the three indices of its vertices. The indices are given in a clockwise order relatively to the oriented normal of the facet. This is done in this implementation in the *saveOrientedFacet* function (class FileIO).

4 Parallelization

The scale-space meshing algorithm consisting in very local computations the process parallelizes nicely provided some precautions are taken. The octree data structure is used to sort cells into sets, each set containing cells that can be processed independently as shown in Figure 2. This parallelization is done for both the scale-space iterations and the meshing step.

For the scale-space iterations, each thread processes a different cell (see Algorithm 4). Processing a cell consists in applying the scale-space to all the points in the cell and storing them in the corresponding set. The only precaution to take is to check that the projected points obtained in two different threads will not be stored in the same set of the same cell, since that would cause conflicts between the threads. Since the projection of a point p lies inside a ball with radius r centered at p, it is enough to ensure that for two cells processed simultaneously, their dilatations of radius r do not contain a common leaf cell. The processing depth is therefore set as the minimum depth such that the size of the cell is above $d = 2.1r$, and at least 1 (Algorithm 4, lines 1- 2). This is easily done by computing

$$level = \max(octree.depth - \lfloor log_2 \frac{octree.size}{d} \rfloor, 1), \tag{1}$$

where $octree.size$ is the length of the largest size of the bounding box. The only remaining possible conflict happens when two threads simultaneously try to add a point in a branch not yet created. Though this case is rare, it is handled by preventing the simultaneous creation of branches (critical section in method *addPoint* of class *Octree*).

The Ball Pivoting Algorithm is parallelized similarly and we refer the reader to [3] for more details. Computation times for the whole Scale-Space meshing algorithm with and without parallelization are given in Table 1 for a computation on a 4 cores laptop (4×2.9GHz).

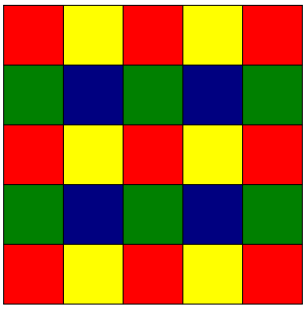


Figure 2: Parallelization principle in 2D. Cells that can be processed simultaneously are depicted in the same color.

Point set	Number of Points	Radius	Computation Time	
			Single-threaded	Multi-threaded (4 cores)
Bunny	$360K$	0.0005	$21s$	$10s$
Dragon	$1.5M$	0.0005	$328s$	$167s$
Pyramid	$1M$	0.4	$515s$	$205s$

Table 1: Computation time with and without parallelization for a 4 core laptop (4×2.9GHz). The number of iterations was set to 4 for all these point sets.

Algorithm 4: Parallelization of Scale-Space iterations.

Input: An oriented input point cloud $\mathcal{P}$ sorted into an octree $\mathcal{O}$ with given depth, a radius r.
Output: A denoised point set stored in the octree $\mathcal{O}$.

$d \leftarrow 2.1 \cdot r$;
$l \leftarrow \max(1,$ smallest level at which cells have size larger than $d)$ (cf Equation 1);
for $i = 0 \cdots 8$ **do**
  $cells \leftarrow$ cells of the octree at level l and with child index i;
  for $C \in cells$ **do in parallel**
    for $p \in C$ **do**
      $p' \leftarrow MCM(p, r, \mathcal{P})$;
      $C' \leftarrow$ octree cell containing p';
      Store p' in the point list of C' corresponding to the next index;

5 Parameters Choice

There are three parameters for the scale-space and one parameter for the Ball Pivoting Algorithm. Yet those parameters can be set by choosing two values: a radius r and a number of iterations N. Below, we explain how all the parameters of the scale-space and ball pivoting can be deduced from these two values.

- **Scale-space parameters.** Three parameters are required by Algorithm 2: the radius of the projection filter, the standard deviation σ for the Gaussian weight, and the number of iterations. The scale-space radius is set to $2r$, the standard deviation is set to $\sigma = 2r$ and the number of iterations is equal to N.

- **Ball Pivoting algorithm parameters.** The Ball Pivoting Algorithm needs a single parameter, the radius of the pivoting ball, which is taken equal to r.

One could choose unrelated radii for the projection filter and the ball radius but this setting is particularly well suited for the experiments. It is nevertheless very important that the radius of the projection filter is larger than the radius of the ball pivoting in order for the regression plane to be stable. The number of iterations is by default set to 4. If the shape is very noisy, one may set a higher number of iterations, but the details and sharp features are always contained in the first iterations, so that few iterations are necessary. A coarse heuristic for setting r consists in considering the number of points N_{points}, the size of the bounding box l and deduce the radius: $r = \sqrt{20/N_{points}} \cdot l$. Indeed, if the shape was a perfect sphere (enclosed in the same bounding box), its surface area would be $4\pi(\frac{l}{2})^2 = \pi l^2$, thus the number of points per unit surface would be $\frac{N}{\pi l^2}$. The area of the surface in a neighborhood of radius r is approximated to πr^2 (given $r << l$), therefore to obtain around 20 neighbors, one should set r such that $r^2 \frac{N}{l^2} = 20$, hence the formula.

The next section reviews some technical details of the provided code including dependencies.

6 Code

The dependencies and properties of this implementation are the same as for the Ball Pivoting Algorithm [3], and we copy the same paragraphs below for completeness. As a matter of fact, if the number of iterations is set to 0, the algorithm is exactly equivalent to the Ball Pivoting Algorithm with a single radius.

6.1 Dependencies

The code provided is a stand-alone C++ code, available at https://doi.org/10.5201/ipol.2015.102. It uses the C++ standard template library extensively. The user can choose between the single-threaded implementation and its parallel version. The single-threaded version does not rely on any external libraries. The parallelization is done through OpenMP[2], a standard API for shared memory multiprocessing programming. The code was tested successfully on Ubuntu 14.04 with g++ 4.8, and on MacOS 10.8 using g++4.8.0. The compilation is done through the CMake build system to be cross-platform. The code compiles with g++ and with clang, but there is no support yet for OpenMP with the clang compiler, so that the parallelism is deactivated in that case.

6.2 Integration in a Larger Project

The code is templated and the structures are kept as simple as possible in order for a better integration into different C++ projects. In particular, it should be easy to interface it with the CGAL library [7] and thus benefit from CGAL geometry kernels. Nevertheless, the goal here is to have a stand-alone code, avoiding the need to link against such a heavy library as CGAL.

6.3 Numerical Robustness

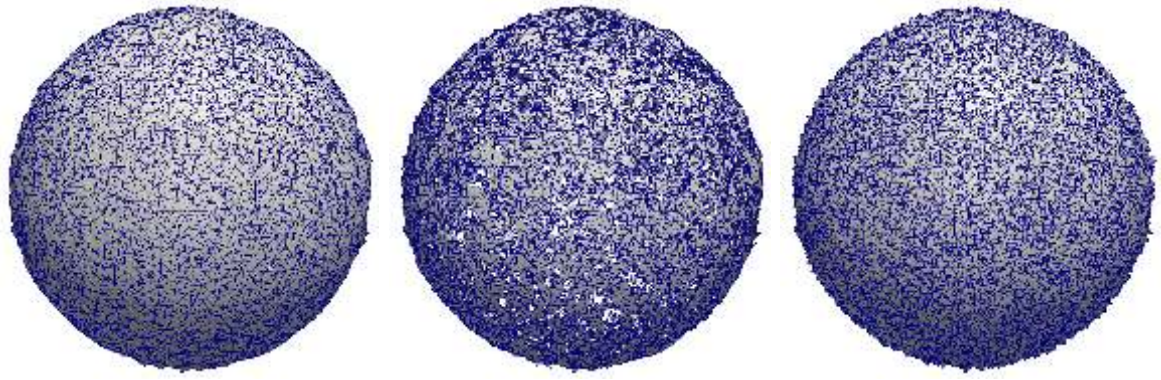
The current implementation relies heavily on geometric tests (e.g. to know whether a point lies within a sphere given by a point and a radius, to know if a point lies on the right side of a potential triangle...). These problems are known to generate numerical robustness problems potentially dramatic for global structures such as Delaunay. A solution to that problem is to use exact arithmetic, which

[2]The OpenMP API specification for parallel programming. http://openmp.org

is very time-consuming. An alternative is to use robust arithmetic through robust predicates (i.e. predicates that will give consistent results) as described in [12]. Yet in our case the consequences of not using any robust predicates are not so dramatic since the construction of the mesh is incremental and local. A numerical instability will only affect the choice of a particular triangle instead of another but will not create global artifacts.

7 Experiments

The first example (Figure 3) is precisely the one that was presented as a failure case for the Ball Pivoting Algorithm (BPA)[3]. The goal is indeed to interpolate a noisy point cloud and in that case neither the single-radius BPA nor the multiple radius version succeed in recovering a closed interpolating mesh. In comparison the Scale-Space Meshing Algorithm recovers a closed mesh interpolating exactly the input noisy point set, allowing for a visual quality assessment of the shape reconstruction. Figure 4 shows the result of the Scale-Space Meshing Algorithm on some well-known shapes from the Stanford repository, showing that the method works in standard cases.



(a) BPA with a single radius 0.05 (b) BPA with multiple radii $0.03, 0.04, 0.05$ (c) SSM with $N = 4$ scale-space iterations and $r = 0.05$

Figure 3: A noisy sphere reconstructed by the Ball Pivoting Algorithm (BPA) and Scale-Space Meshing Algorithm (SSM). Only the Scale-Space Meshing (right) allows for the reconstruction of a closed interpolating mesh (30000 vertices, 60000 facets).

One can also judge the interpolating quality of the mesh by comparing the number of vertices to the number of input points. For the pyramid point set (Figure 6) for example, the Scale-Space Meshing Algorithm builds a mesh with 99.24% of the input point set as vertices compared to only 79.36% for the Ball Pivoting Algorithm.

Figures 3 and 5 provide comparisons of the Scale-Space Meshing and the Ball Pivoting Algorithms with the exact same meshing radius. In particular, Figure 5 shows that the details are much better preserved with the Scale-Space meshing Algorithm. Figure 6 shows the shape evolution of a pyramid point set [4] with respect to the scale. The shape loses its details and becomes smoother: as a consequence, computations done at the low scale will be more robust than those done at the highest scales. The result of the processing thus strongly depends on the choice of the scale.

Finally, we show the performance of the Scale-Space Meshing using various input shapes from the Farman dataset [4]. Figure 7 shows the coarse and fine scale meshes built from cropped point clouds from the Farman dataset [4].

Figure 4: Results of the Scale-Space Meshing algorithm on some standard shapes: the Stanford Bunny (left, $r = 0.0005, N = 4$) and the Stanford Dragon (right, $r = 0.0005, N = 4$).

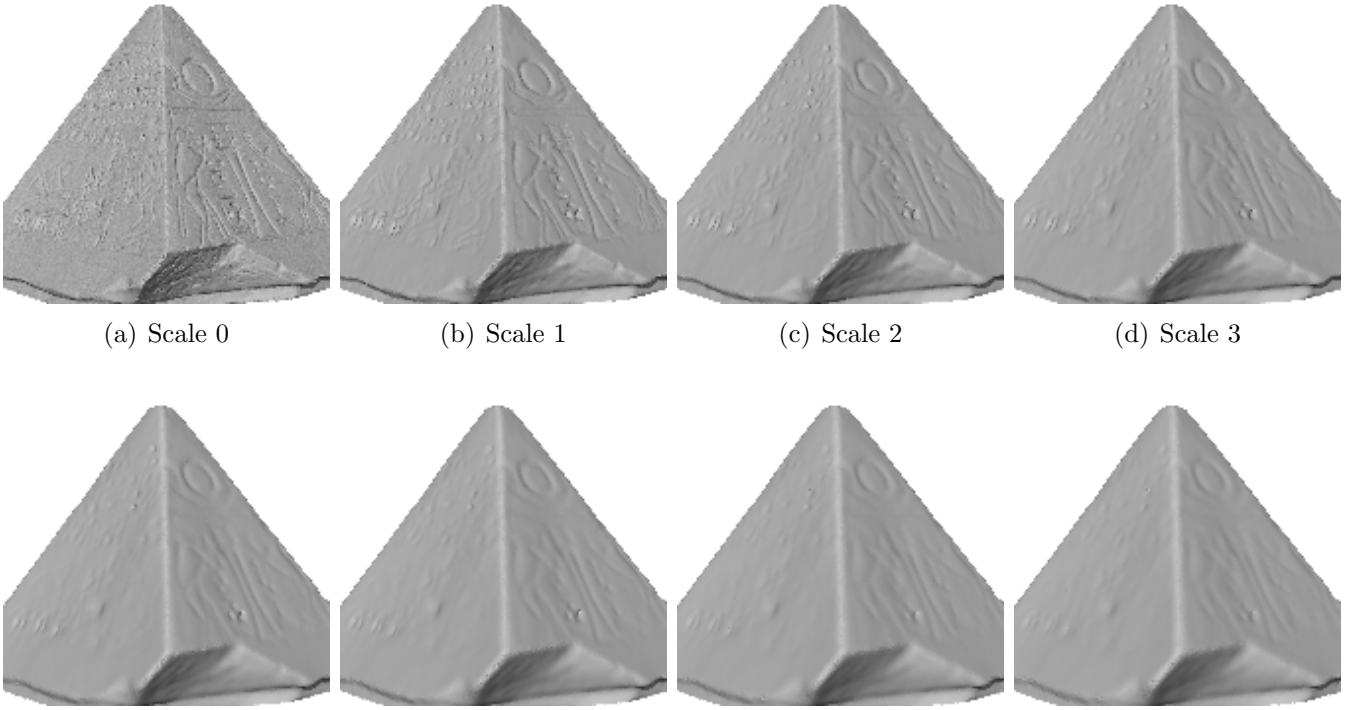
Figure 5: Comparison of the Scale-Space Meshing (left) with the standard Ball Pivoting Algorithm (middle) and Poisson Reconstruction (right)

8 Conclusion

This paper presented the parallel implementation of the scale-space meshing algorithm, a method to build an interpolating mesh for point sets containing possibly details and sharp features as well as noise. The code for this implementation is available for download and online tests (https://doi.org/10.5201/ipol.2015.102).

Acknowledgments

This work was partially funded by Direction Générale de l'Armement, Office of Naval Research (Grant N00014-97-1-0839) and the European Research Council (ERC Advanced Grant "Twelve Labours").



(a) Scale 0 (b) Scale 1 (c) Scale 2 (d) Scale 3

(e) Scale 4 (f) Scale 5 (g) Scale 6 (h) Scale 7

Figure 6: Evolution of a shape with the scale-space

Data Credits

The Stanford Bunny and Stanford Dragon (Figure 4) are from the Stanford 3D Scanning Repository[3]. The other shapes (Figures 5, 6 and 7) are from the Farman Institute 3D Point Sets[4].

References

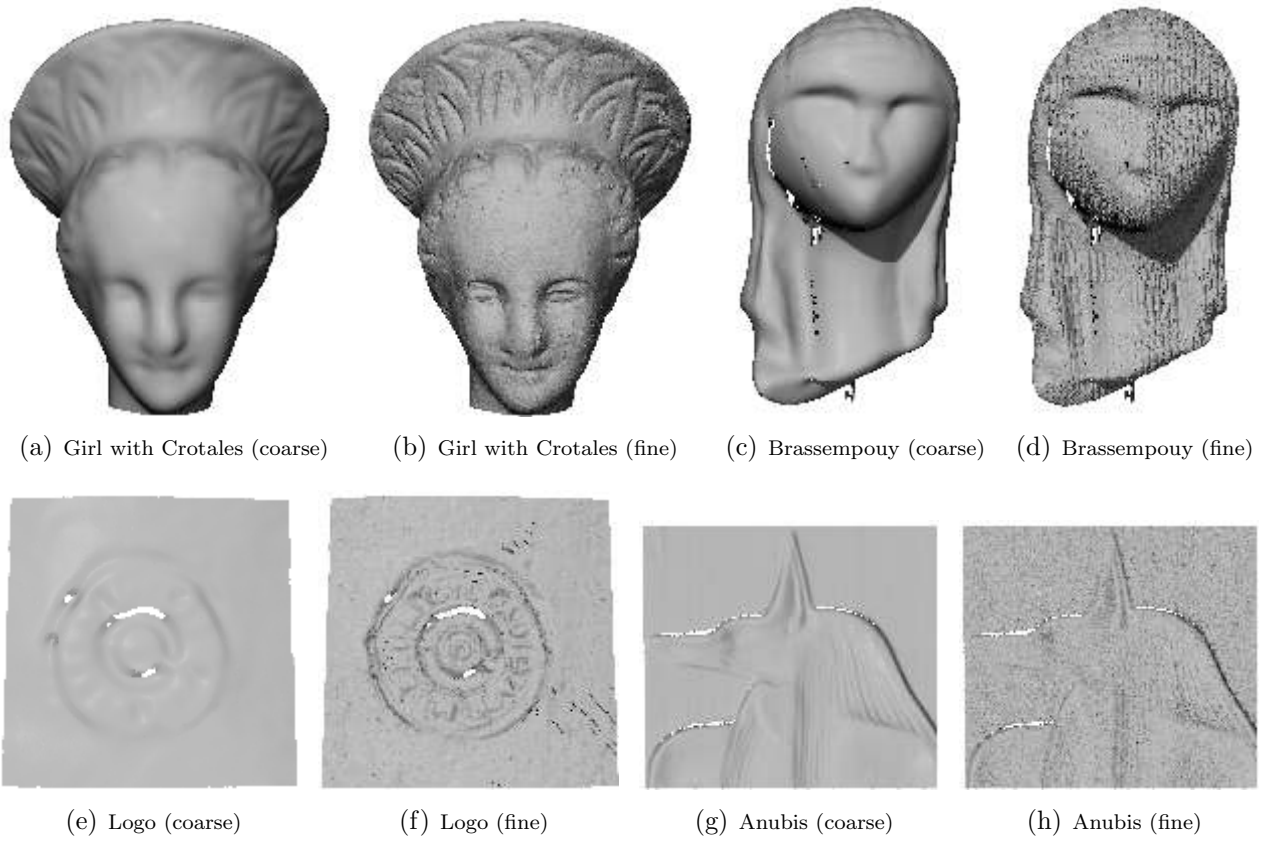
[1] F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva, and G. Taubin, *The ball-pivoting algorithm for surface reconstruction*, Visualization and Computer Graphics, IEEE Transactions on, 5 (1999), pp. 349–359. http://dx.doi.org/10.1109/2945.817351.

[2] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans, *Reconstruction and representation of 3D objects with radial basis functions*, in Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH, 2001, pp. 67–76. http://doi.acm.org/10.1145/383259.383266.

[3] J. Digne, *An Analysis and Implementation of a Parallel Ball Pivoting Algorithm*, Image Processing On Line, 4 (2014), pp. 149–168. http://dx.doi.org/10.5201/ipol.2014.81.

[4] J. Digne, N. Audfray, C. Lartigue, C. Mehdi-Souzani, and J-M. Morel, *Farman Institute 3D Point Sets - High Precision 3D Data Sets*, Image Processing On Line, 1 (2011). http://dx.doi.org/10.5201/ipol.2011.dalmm_ps.

[3] http://graphics.stanford.edu/data/3Dscanrep/
[4] http://www.ipol.im/pub/art/2011/dalmm_ps/



(a) Girl with Crotales (coarse) (b) Girl with Crotales (fine) (c) Brassempouy (coarse) (d) Brassempouy (fine)

(e) Logo (coarse) (f) Logo (fine) (g) Anubis (coarse) (h) Anubis (fine)

Figure 7: The coarse and fine scale meshes produced by the Scale-Space Meshing algorithm applied to datasets of the Farman data set, the parameters used are: $r = 0.2$ (Girl with Crotales), $r = 0.16$ (Brassempouy), $r = 0.2$ (logo), $r = 0.1$ (Anubis). In all the experiments, $N = 4$. NB: these pointsets are not subsampled or normalized as the ones proposed in the demo.

[5] J. Digne and J-M. Morel, *Numerical analysis of differential operators on raw point clouds*, Numerische Mathematik, 127 (2014), pp. 255–289. http://dx.doi.org/10.1007/s00211-013-0584-y.

[6] J. Digne, J-M. Morel, C-M. Souzani, and C. Lartigue, *Scale space meshing of raw data point sets*, Computer Graphics Forum, 30 (2011), pp. 1630–1642. http://dx.doi.org/10.1111/j.1467-8659.2011.01848.x.

[7] A. Fabri and S. Pion, *CGAL: The computational geometry algorithms library*, in Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, 2009, pp. 538–539. http://doi.acm.org/10.1145/1653771.1653865.

[8] G. Guennebaud and M. Gross, *Algebraic point set surfaces*, in ACM SIGGRAPH Papers, 2007. http://doi.acm.org/10.1145/1275808.1276406.

[9] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle, *Surface reconstruction from unorganized points*, SIGGRAPH Computer Graphics, 26 (1992), pp. 71–78. http://doi.acm.org/10.1145/142920.134011.

[10] Bolitho M. Kazhdan, M. and H. Hoppe, *Poisson surface reconstruction*, in Eurographics SGP, 2006, pp. 61–70. ISBN:3-905673-36-3.

[11] M. Pauly, R. Keiser, and M. Gross, *Multi-scale feature extraction on point-sampled surfaces*, Computer Graphics Forum, 22 (2003), pp. 281–289. http://dx.doi.org/10.1111/1467-8659.00675.

[12] J. R. Shewchuk, *Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates*, Discrete & Computational Geometry, 18 (1997), pp. 305–363.

[13] O.K. Smith, *Eigenvalues of a symmetric 3x3 matrix*, Communications of the ACM, 4 (1961), pp. 168–. http://doi.acm.org/10.1145/355578.366316.

[14] D.H.D. West, *Updating mean and variance estimates: An improved method*, Communications of the ACM, 22 (1979), pp. 532–535. http://doi.acm.org/10.1145/359146.359153.