Published in Image Processing On Line on 2014–07–01.
Submitted on 2013–03–19, accepted on 2013–06–28.
ISSN 2105–1232 © 2014 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.2014.81

An Analysis and Implementation of a Parallel Ball Pivoting Algorithm

Julie Digne

Université Lyon 1, LIRIS-GeoMod, CNRS, France (julie.digne@liris.cnrs.fr)

Communicated by Thomas Lewiner *Demo edited by* Julie Digne and Miguel Colom

Abstract

The problem of surface reconstruction from a set of 3D points given by their coordinates and oriented normals is a difficult problem, which has been tackled with many different approaches. In 1999, Bernardini and colleagues introduced a very elegant and efficient reconstruction method that uses a ball pivoting around triangle edges and adds new triangles if the ball is incident to three points and contains no other points. This paper details an implementation and parallelization of this algorithm.

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 Ball Pivoting Algorithm is linked with 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; point cloud

1 Introduction

The Ball Pivoting Algorithm (BPA) is a powerful heuristic for reconstructing a triangular mesh surface from a set of scattered 3D points. It was introduced by Bernardini et al. [2] and provided a way to build an *interpolating mesh* in contrast with other research directions aiming at building an *approximating mesh* ([10], [9], [3]).

The method is strongly linked to the α-shapes theory and sampling conditions for the ball pivoting algorithm can be deduced from it ([6], [1]). The α-shape of a point set $\mathcal{P}$ is a filtering of its Delaunay triangulation keeping only Delaunay facets whose three vertices lie on the surface of an empty-interior $\frac{1}{\alpha}$-ball. Thus the ball pivoting algorithm computes a subset of the $\frac{1}{r}$-shape of the point set $\mathcal{P}$ and sampling conditions for BPA can be deduced from the α-shapes (see the paper by Bernardini et al. [2]

[1] https://doi.org/10.5201/ipol.2014.81

for details). An improvement of this algorithm was later proposed by Digne at al. [5], attempting to better preserve the details.

Let $\mathcal{P}$ be a sampling of a manifold M, the surface of a 3D object. This algorithm builds a watertight reconstruction of M from $\mathcal{P}$, provided $\mathcal{P}$ is dense enough. The water-tightness condition is that an r-ball cannot pass through $\mathcal{P}$ without touching 3 points. The algorithm starts by putting a ball in contact with three points (forming thus a seed triangle). The ball is then rotated around two of these three points: it remains in contact with the two points and rotates around the axis they form until it touches another point. Hence the creation of an expansion edge front, formed originally with the three edges of the seed triangle and then expanded.

The remaining of this paper is divided as follows: Section 2 details the data structures used in this implementation. Section 3 explains how spatial queries are handled, Section 4 explains the ball pivoting implementation itself, Section 5 explains the main limitations of the Ball Pivoting in its present state. Section 6 explains how the process is parallelized. Section 7 gives technical details about the code. Finally, Section 8 presents experimental results.

2 Data Structures

This method requires two important data structures. First, a point sorting and searching structure must be set up in order to access quickly range neighborhoods of a given position. Second, a surface mesh connectivity structure must be constructed. This structure is a *manifold with holes* type structure. The surface is guaranteed to be self-intersection free (no triangles will intersect each other except at an edge or vertex, and at most two triangles can be adjacent to an edge). But the structure will still be able to account for insufficient data by allowing for holes in the mesh: some edges will only be adjacent to a single triangle. Such an edge will be called *boundary edge*.

2.1 Search Structure

To access quickly range neighborhoods we use an octree as a search structure dividing the space into cells containing the points. The acceleration of the search structure is done through *locational codes* as explained by Frisken and Perry [8] (see Section 3.1). We give in Table 1 a short benchmark of computation times for finding range neighborhoods of all points in a point cloud containing 362269 points with diameter 0.256. See Section 3.1 for details on these locational codes.

radius (% size)	mean number of neighbors	computation time
0.0005 (0.20%)	5.24	$3s$
0.001 (0.39%)	22.00	$5s$
0.002 (0.78%)	88.29	$15s$
0.01 (3.9%)	2301.32	$8min19s$

Table 1: Typical neighbor search times for all points of a point cloud containing 362269 points with diameter 0.256. The computation time is roughly linear with respect to the number of neighbors. The point cloud used here is the Stanford Bunny (see Table 6 and Figure 10).

2.2 Mesh Structure

The chosen mesh structure describes a manifold with boundaries, we describe it for completeness. It consists in a set of triangular facets, edges and vertices. Each vertex stores its adjacent triangles

and edges. Each edge stores both of its end vertices and at most two adjacent facets.

We will use a specific vocabulary in the description below:

- *Sample:* a point of the input point set.
- *Seed facet:* an orphan facet built to serve as starting facet for the triangulation expansion.
- *Expansion front:* a set of edges from which the triangulation will be expanded. An edge of the expansion front will be called *front edge*; it has only one adjacent facet.
- *Inner edge:* an edge with two adjacent facets. This edge is *frozen*, its neighboring facets will not change anymore and it is not on the expansion front.
- *Boundary edge:* an edge with only one adjacent facet but which does not belong to the expansion front.
- *Front vertex:* a vertex which belongs to a front edge.
- *Inner vertex:* a vertex belonging to at least one facet and no front or boundary edge.

2.3 Guarantees for Triangles and Edges

The input of the algorithm is a set of oriented samples. Therefore, we must ensure that the created mesh remains consistent with this orientation throughout the reconstruction process. To do so, before adding a facet to the triangulation, we check that its normal is consistent with the normals of its three vertices: the normal of the triangle should have positive scalar product with each of the normals of the vertices. If it is not the case the facet is discarded.

By convention, a created facet will store its three vertices v_0, v_1, v_2 ordered so that the normal of the triangle has positive scalar product with $(v_1 - v_0) \otimes (v_2 - v_0)$.

3 Implementation Details: Spatial Queries

The neighborhood queries are done through an octree structure. Although these neighborhood queries are not per se part of the ball pivoting algorithm, we detail the implementation of spatial queries for completeness. First all samples are sorted into an octree, that is, a subdivision of the space into cells. Each cell has eight children cells obtained by splitting the cell into eight identical cubes. The splitting process stops when the desired octree depth is obtained. Note that even if the shape has a strong anisotropy, we will still use hypercubes and not hyperrectangles for simplicity reasons.

We first review briefly the locational codes used to speed up neighbor searches explained by Frisken and Perry [8]. Then we explain how these codes are used to get the neighbors of a given point.

3.1 Locational Codes

We summarize here the vocabulary used for the octree description

- *Octree*: a data structure in which each internal node has at most eight children. Each node corresponds to a subset of the space called cell. In our case, we will not create nodes if no sample lies inside their corresponding cells (unbalanced octree).

- *Cell*: a cubic subset of the space corresponding to a node of the octree. Each cell has one parent (except for the root, see below) and at most eight children. It contains pointers to its parent and children, its size, depth, origin point and its child index (relative position of the cell with respect to its parent midpoint). In addition a leaf cell maintains a list of the input samples it contains.

- *Root*: the first node of the octree. This node has no parent cell. Its cell corresponds to the whole bounding box of the point set.

- *Leaf*: the last nodes of the octree. These nodes have no children and will be the ones whose cells actually contain the samples, as a single list.

- *Child*: each child cell corresponds to a subset of its parent. The children are numbered as explained in Figure 1.

- *Octree Depth*: the number of node subdivisions. For example, an octree with depth 0 corresponds to an octree with a single cell (the root). An octree with depth 1 corresponds to the root and its eight children.

- *Depth of a cell*: depth of the corresponding node in the octree. The depth of the root is equal to the octree depth and the depth of the leaves is 0.

- *octree.origin*: corner of the bounding box. All samples s of the point set are such that $octree.origin.x < s.x \leq octree.origin.x + octree.size$ (similarly for $s.y, s.z$). A similar definition holds for *cell.origin*.

- *octree.size*: size of the bounding box. A similar definition holds for *cell.size*.

For simplicity, we will identify the nodes with their corresponding cells, in the remaining of this paper.

The search in the octree structure is accelerated using *locational codes* [8]. In a nutshell a locational code of a given position p with coordinates $p.x, p.y, p.z$ is the triplet:

$$(\lfloor \frac{p.x - octree.origin.x}{octree.size} \cdot \rfloor, \lfloor \frac{p.y - octree.origin.y}{octree.size} \cdot \rfloor, \lfloor \frac{p.z - octree.origin.z}{octree.size} \cdot \rfloor)$$

expressed in binary notation. Read from left to right, these codes yield the path in the octree starting from the root to the leaf actually containing p. They can be derived for trees in any dimension (binary tree, quadtree, octree, n-tree, ...). In a binary tree, for example, the code 1011 encodes the path starting from the root cell then right - left - right - right child.

In the following pseudo-code, we denote by *binsize* the binary notation of 2^{octree_depth}. We will also perform logical shifts operations: $>> l$ means shifting the binary code l bits to the right. For example, $1000 >> 3$ yields 1, suppressing the last three bits of the code. Similarly $<< l$ means shifting the binary code to the left. For example, $1 << 3$ gives 1000. & corresponds to the AND operator between two codes. For example, 10010&01011 gives 00010. The main operations using these locational codes are detailed below:

- Computing the code of a cell: the cell codes are computed when the tree is built. The codes are computed recursively as explained in Algorithm 1.

- Computing the code of a sample point: the code of a sample point p is computed as the integer part of $\frac{p.x - octree.origin.x}{octree.size} \cdot binsize$ expressed in binary notation (and similarly for $p.y$ and $p.z$).

- Translating the code into a path in the octree: the code gives a succession of child indices composing the path. For example $(01, 11, 10)$ means the child index 3 followed by 5 (see line 10 of Algorithm 1).

- Getting the code of the left neighboring cell: the code is given by $(cell.xloc - 1, cell.yloc - 1, cell.zloc - 1)$.

- Getting the code of the right neighboring cell at level l: the code is given by $(cell.xloc + 2^l, cell.yloc + 2^l, cell.zloc + 2^l)$. For leaves, it means computing $(cell.xloc + 1, cell.yloc + 1, cell.zloc + 1)$

Algorithm 1: *BuildOctree*$(\mathcal{P}, r)$. Octree construction

Input: A set of points $\mathcal{P}$, a loose bounding box given by a point o and a size s, a depth *depth*
Output: An octree

1 $binsize \leftarrow binary(2^{depth})$
2 *Initialize the octree*: build an empty root cell, set its size to s, its origin to o, and its level to *depth*
3 **for** $p \in \mathcal{P}$ **do**
4 Compute the codes $xloc, yloc, zloc$ of p as $xloc = binary(floor(\frac{p.x - octree.origin.x}{octree.size} \cdot binsize))$ (similarly for *yloc* and*zloc*)
5 $cell \leftarrow$ root of the octree
6 $l \leftarrow depth - 1$
7 **while** $cell\_depth > 0$ **do**
8 $childbranchbit = 1 << l$
9 $x = xloc\&childbranchbit >> l$ (and similarly for y, z)
10 $childindex = (x << 2) + (y << 1) + z$
11 **if** *cell.child(childindex) does not exist* **then**
12 create cell.child
13 $child.size \leftarrow \frac{1}{2} cell.size$
14 $child.origin.x \leftarrow cell.origin.x + x \cdot child.size$ (similarly for y, z)
15 $child.depth \leftarrow cell.depth - 1$
16 $child.xloc \leftarrow cell.xloc + (x << child.depth)$ (and similarly for *yloc* and *zloc*)
17 $cell = cell.child(childindex)$
18 $l \leftarrow l - 1$
19 Add p to *cell.points*

Algorithm 1 sums up the octree construction method. We give below some details:

- Line 9: childbranchbit is a binary integer that in base 2 has a single 1 at position $l+1$ starting from the right. For example 1000 for level 3. Then $(xloc\&childbranchbit) >> l$ gives either 1 or 0: the next direction to take in the path to the leaf.

- Line 10: the child index gives the index (between 0 and 7) corresponding to the child on the path at depth l (see Figure 1 for the numbering of the children). It is obtained by concatenating the bit at position l in the three codes $xloc, yloc, zloc$.

- Line 14: the origin of the newly created cell is easily deduced from the locational code.

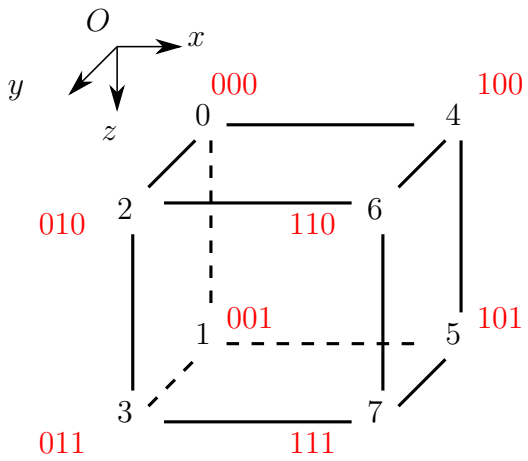


Figure 1: Numbering of the children of a cell.

- Line 16: to get the code of the child cell, one simply has to add bit x to the code of the parent cell $cell.xloc$ (and similarly for y and z).

We give in Figure 2 an example of the codes involved in the building of a binary tree. The process is identical to the one of an octree (except that three separate codes will be required for an octree, one per dimension).

3.2 Neighborhood Queries

In the whole paper we will denote by $\mathcal{P}$ the point cloud and by $\mathcal{N}_r(p)$ the range neighborhood of radius r centered at p, i.e. $\mathcal{N}_r(p) = \{q \in \mathcal{P} |\ \|q - p\| \leq r\}$.

Algorithm 2: $\mathcal{N}_r(p) = getNeighbors(p, \mathcal{P}, r)$. Finding the neighbors with fixed radius of a given point (see the text for details)

Input: An octree containing a point cloud $\mathcal{P}$, a query point p, a radius r
Output: A set of neighbors $\mathcal{N}_r(p)$

```
1 l ← depth corresponding to radius r
2 C_l ← cell containing p at level l
3 Cells ← cells of level l adjacent to C_l and intersecting B(p, r)
4 for C ∈ Cells do
5     for q ∈ C do
6         if ||p − q|| ≤ r then
7             q is added to N_r(p)
```

The spatial query method is explained in Algorithm 2 and details of each important step are given in the text below.

- Line 1: the input radius r corresponds to a depth l in the octree. l is chosen to be the smallest depth such that the cells of depth l have a size superior to $2r$.

- Line 2: to find the cell C_l of level l that contains p, one has to convert the coordinates of p into locational codes and follow the path given by those codes starting from the root. Notice that C_l is not necessarily a leaf depending of the chosen radius and octree depth.

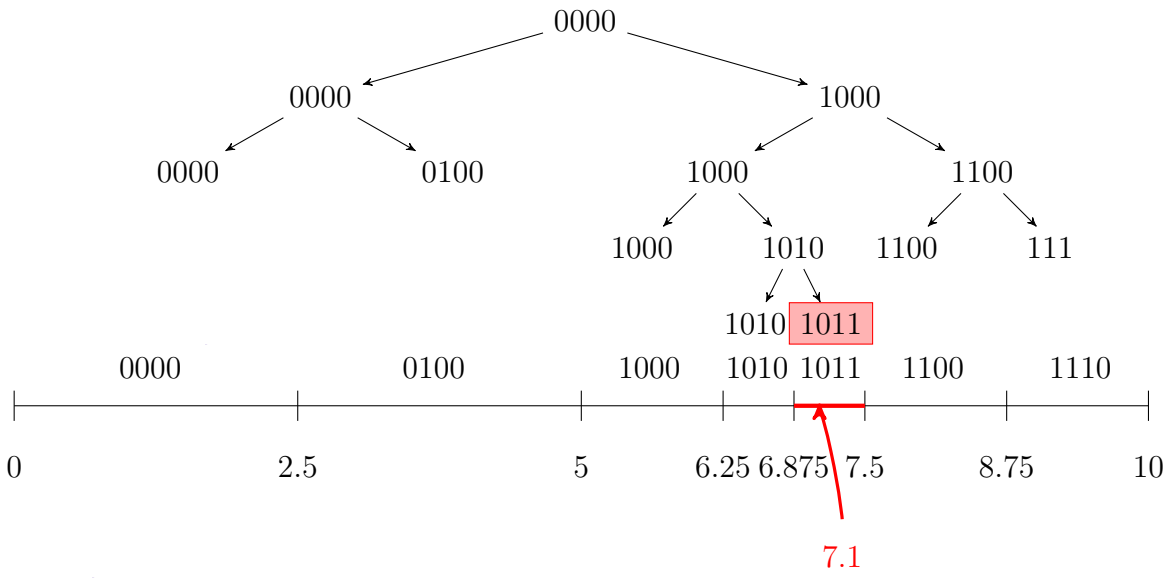


Figure 2: Example of an unbalanced binary tree of depth 5 representing segment $[0, 10]$ and the locational codes of the cells (top). Corresponding partition of the 1D-segment (bottom). 7.1 has binary code $binary(floor(0.71 \cdot 2^4)) = binary(11) = 1011$ and the cell it belongs to is depicted in red in both tree and segment representations. Other examples can be found in the paper by Frisken and Perry [8].

- Line 3: since r is less than half the size of the cells of level l, the number of neighboring cells intersecting the ball of radius r centered at p is at most 8 (including C_l). In each direction x, y, z, the ball can only intersect other cells on one side of C_l. To get those cells, one computes the corresponding locational codes, as explained above, and gets the cells accordingly.

4 Implementation Details: the Ball Pivoting Algorithm

Short Preliminary: Elementary Relation in Triangles

This small section only details how the circumcenter of a sphere passing through three points and with given radius r is found (if it exists). First, a sphere passing through all points is necessarily located on a line passing through the circumcenter of the triangle and orthogonal to the triangle plane. Let us consider triangle ABC, and denote by a (resp. b, c) the size of the triangle side opposite to vertex A (resp. B, C). Then the circumcenter H of ABC has barycentric coordinates:

$$\big(a^2(b^2+c^2-a^2), b^2(a^2+c^2-b^2), c^2(a^2+b^2-c^2)\big)$$

and the square circumradius is

$$r_c^2 = \frac{a^2 \cdot b^2 \cdot c^2}{(a+b+c)\cdot(b+c-a)\cdot(c+a-b)\cdot(a+b-c)}.$$

Thus, such a sphere exists only if $r^2 - r_c^2 \geq 0$. Let us denote by $\mathbf{n}$ the normal to the triangle plane, oriented such that is has a nonnegative scalar product with each of the normals of the vertices. Provided $r^2 - r_c^2 \geq 0$ the sphere exists and the center O of the sphere is:

$$O = H + \sqrt{r^2 - r_c^2} \cdot \mathbf{n}.$$

By convention, we only consider spheres with circumcenter *above* the triangle plane.

4.1 Overview

The Ball Pivoting algorithm consists in two steps:

- *Finding a seed triangle*: the algorithm looks for three orphan vertices that lie on the surface of an empty-interior sphere of given radius r.
- *Expanding the triangulation*: triangles are added to the triangulation by pivoting a ball around front edges until no more front edge remains.

If the point cloud is not r-connected, then the expansion step will stop before all points are triangulated. Therefore another seed must be found as shown in Algorithm 3, and both steps must be alternated. In addition to this iterative process, we add a small post-processing step in the end, to fill the remaining triangular holes, that might occur due to numerical instabilities for example (see Section 7.3).

Algorithm 3: *Reconstruct*$(\mathcal{P}, r)$. Ball Pivoting Reconstruction

Input: An oriented point cloud $\mathcal{P}$, a radius r
Output: A surface mesh $\mathcal{T}$

```
1 while A new seed can be found do
2     T = FindSeedTriangle(P, r) (Algorithm 4)
3     T = ExpandTriangulation(T, P, r) (Algorithm 5)
```

Both steps will be explained in more detail in the remainder of this section.

4.2 Finding a Seed Triangle

The search for a seed triangle is the first step of this algorithm: it starts with a single point; and tests for each set of two neighbors if a triangle can be built such that its circumsphere with radius r has an empty interior. In practice more than one triangle can fit the description. The steps are detailed in Algorithm 4 below.

The key steps of Algorithm 4 to find a triangulation seed are explained below:

- Line 2: if a seed triangle incident to p exists, then all its vertices lie in $\mathcal{N}_{2r}(p)$. Therefore it is enough to reduce the search to this neighborhood.
- Line 4: three vertices are compatible if the normal to the triangle they form has positive scalar product with each of the normals of the vertices.
- Line 6: the r-ball passing through p, q, s is entirely included in the ball passing through p with radius $2r$. Therefore to check that the points are in empty ball configuration, one only needs to compute the center c of the r-ball (Section 4), and traverse $\mathcal{N}_{2r}$ to check that no point is in the interior of the r-ball centered at c.

Algorithm 4: $T = FindSeedTriangle(\mathcal{P}, r)$. Finding a seed triangle in the point cloud (see text for details)

Input: An oriented point cloud $\mathcal{P}$ sorted in an octree; a radius r
Output: A seed triangle T

```
for p ∈ P do
    Look for all points in a ball neighborhood N_2r(p) sorted by increasing distance
    for (q, s) ∈ N_2r(p) do
        if p, q, s are not compatible then
            Continue
        if p, q, s are in empty ball configuration then
            return a facet with vertices (p, q, s)
if No Facet was found then
    exit
```

Once a seed triangle is found, its three edges are added to the set of expansion front edges. As explained in Section 2.3, facets will store their vertices in a particular order. Throughout the triangulation process, when an edge e is created, it has a single adjacent facet formed by the end-vertices of e and an opposite point v. Edge e will be oriented coherently with the facet orientation: $(e.target - e.source) \otimes (v - e.source)$ must have positive scalar product with the normal to the facet. If it is not the case the source and target of the edge are switched. This edge orientation is only valid as long as the edge has only one adjacent facet (front or boundary edge). If the edge has two adjacent facets, the orientation is not valid with respect to the last-added facet, but it has no consequence since this edge will no longer intervene in the triangulation (it is *frozen*). Thus, edges in the edge front are always oriented coherently (the target vertex of one edge is the source vertex of the next).

4.3 Expanding the Triangulation

Expanding the triangulation is done by considering edges of the edge front. Let us denote by e an edge of the expansion front. It has only one adjacent facet f (by construction there are no orphan edges). We proceed by *rotating the ball* of radius r from its initial position as r-circumsphere of f around edge e until it meets another point v. We then test if the sphere is empty or not. If the sphere is empty we can create a new facet from e and v. This ball rotation is illustrated in Figure 3.

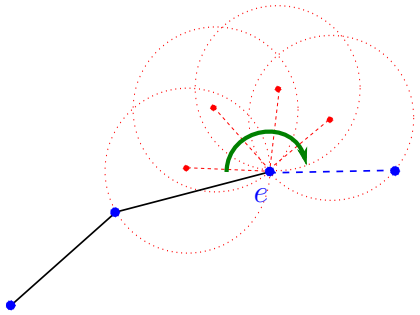
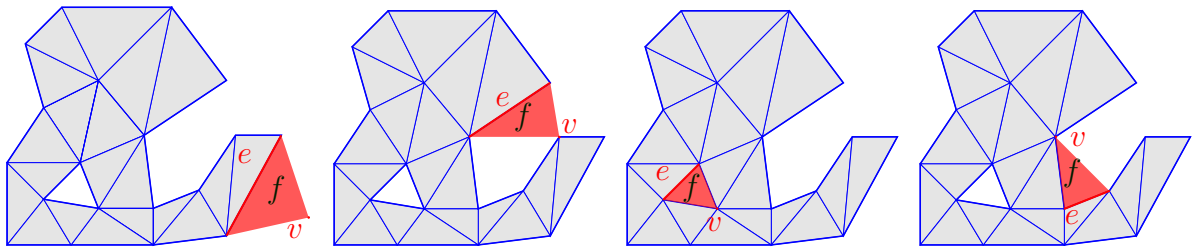


Figure 3: A rotating ball

Several cases (illustrated in Figure 4) are possible:

- *Expansion case*: v is an orphan vertex. A new facet is created, e is removed from the expansion front, and the two newly created edges are added to the front. The front size increases by 1.
- *Gluing case*: v is not an orphan vertex (it belongs to a front edge), but is not linked to the endvertices of e. In this case, e is removed from the front. A new facet is created and the two newly created edges are added to the front. The front size increases by 1.
- *Hole filling case*: v is already linked by edges to the two endvertices of e. A new facet is created and e is removed from the expansion front along with the edges linking v to e. The front size decreases by 3.
- *Ear filling case*: v is linked to only one of e's endvertices by a front edge e'. A new facet is created from e and e' which are both removed from the front. Only one edge is created and added to the front. The front size decreases by 1.



(a) Expansion case (b) Gluing case (c) Hole filling case (d) Ear filling case

Figure 4: Several cases for rotating the ball around an edge. The existing facets are depicted in light gray. The edge e around which the ball is pivoting is depicted using a bold red segment, the candidate vertex v is depicted using a red dot. The resulting facet f is depicted in red. Active edges are drawn with bold lines.

We ensure that each vertex will be *manifold* (i.e. it will not be a junction between two surface sheets) by checking that, in the gluing case, the candidate vertex is not an inner vertex. We ensure that each edge is manifold by checking that when the facet to be created involves an existing edge (hole filling and ear filling cases), this edge is a front edge.

The process is summed up in Algorithm 5. Some of the steps of this algorithm call for an explanation:

- Line 3: the pop() command removes and returns the first element in the list container, effectively reducing the list size by one.
- Line 5: an edge in the edge front can still have an *inner* or *boundary* tag. For example, in the ear filling and hole filling cases (see Figure 4), the facet creation implies turning more than one front edge into inner edges.
- Line 6: the *findCandidate* method is explained in Algorithm 6.
- Line 10: when a facet is created from an edge and a vertex, one first looks if the edges end vertices are already linked by an edge to the vertex. If it is not the case the edges are created. The facet is then created and stored.
- Lines 11 and 12: those edges were created or reused during the facet creation.

Algorithm 5: $\mathcal{T} = ExpandTriangulation(T, \mathcal{P}, r)$. Expanding the triangulation (see the text for details)

Input: An input seed triangle T, a point cloud $\mathcal{P}$ to triangulate, a radius r
Output: A surface mesh $\mathcal{T}$

```
1  Add the three edges of T to edge_front
2  while edge_front is not empty do
3      e ← edge_front.pop()
4      if e is tagged as boundary or inner edge then
5          Continue
6      v ← FindCandidate(e, P, r)
7      if v == NULL then
8          Tag e as boundary edge
9          Continue
10     Create facet f(e, v) and add it to T
11     e_s ← edge linking e.source and v
12     e_t ← edge linking e.target and v
13     if e_s has two adjacent facets then
14         Tag e_s as inner edge
15     else
16         add e_s to the edge front
17     if e_t has two adjacent facets then
18         Tag e_t as inner edge
19     else
20         add e_t to the edge front
```

- Lines 13 and 17: if the edges were created during facet creation, then they must have only one adjacent facet. Otherwise if they were re-used, they have two adjacent facets and are now inner edges and tagged as such.

Let us now detail the key-steps of Algorithm 6 (*findCandidate* method):

- Line 6: to look for all potential candidates, one can restrict the search to a neighborhood of radius $r' = \|m - c\| + r$ instead of $2r$ (as used in the search for a seed). This is simply due to the fact that $\|m - c\| = \|m - c_{new}\|$, where c_{new} is the center of the r-ball of the triangle formed by the edge and the candidate.

- Line 11: e and v are compatible if and only if $(e.source - e.target) \otimes (v - e.target)$ has positive scalar product with the normals of v, $e.source$ and $e.target$.

- Line 15: to compute the angle, we compute the dot product between $\frac{c_{new}-m}{\|c_{new}-m\|}$ and $\frac{c-m}{\|c-m\|}$, by taking the arc-cosine, one gets an angle θ. The sign ambiguity is removed by considering $((c - m) \otimes (c_{new} - m)) \cdot (e.target - e.source)$, if it is negative, replace θ by $2\pi - \theta$.

- Line 16: to check that the ball is empty, one traverses the neighborhood $\mathcal{N}_{r'}(m)$ and checks that no points is in the interior of the r-ball centered at c_{new}. It avoids querying for the neighbors of c_{new}.

Algorithm 6: $v = FindCandidate(e, \mathcal{P}, r)$. Finding a vertex v to create a facet with edge e (see the text for details)

Input: A front edge e, a point cloud $\mathcal{P}$, a radius r
Output: A vertex to be linked with e

$c \leftarrow$ center of the r-circumsphere of the facet adjacent to e
$m \leftarrow e.midpoint()$
$r' = \|m - c\| + r$
$\theta_{min} = 2\pi$
Find $\mathcal{N}_{r'}(m)$ the set of all points in $\mathcal{P}$ within radius r' of m (Algorithm 2)
$L \leftarrow \emptyset$
for $v \in \mathcal{N}_{2r}(m)$ **do**
    if *v is an inner vertex, or belongs to e* **then**
        Continue
    if *v is not compatible with e* **then**
        Continue
    Compute the r-circumsphere center c_{new} of the triangle (e, v)
    if *c_{new} does not exist* **then**
        Continue
    Compute the angle θ between $c - m$ and $c_{new} - m$
    if *e, v are in empty ball configuration and $\theta < \theta_{min}$* **then**
        $candidate = v$
        $\theta_{min} = \theta$
if *No vertex is found* **then**
    return NULL
else
    return *candidate*

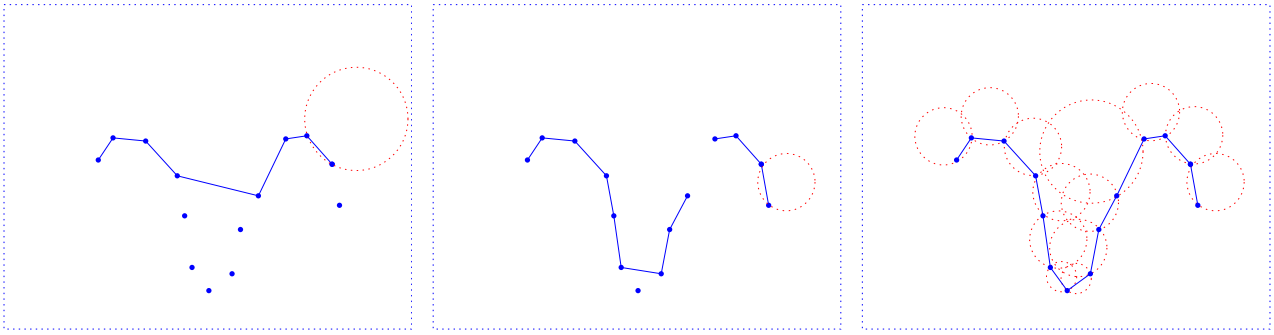
4.4 Post-Processing

At this point, there may remain holes in the triangulation, either due to a lack of points, (acquisition holes) or to some bad normal orientation that would cause the manifold test to fail. To avoid that, we added an extra post-processing step, that looks for triangular holes: three boundary edges forming a loop, and creates a triangle if the hole boundary orientation is coherent. More precisely, for a boundary edge e one looks for two neighboring boundary edges e_1 and e_2 such that $e.source = e_2.target$, $e_1.source = e.target$ and $e_1.target = e_2.source$ (in other words, the three edges form an oriented loop).

5 Known Limitations of the Method

The ball pivoting algorithm is known to be very sensitive to the chosen ball radius. A small radius will tend to generate more surface holes and possibly split the shape into several connected components. A larger radius can induce detail loss. Both those limitations are illustrated in Figure 5.

To deal with radius dependence, Bernardini et al. propose to use several radii instead of a single one. The user then provides a series of radii and the method is applied starting with the smallest radius. The set of boundary edges from this radius is then used as the front edge to rotate the ball



(a) Detail loss (b) Hole creation (c) A possible correction: using multiple radii

Figure 5: Limitations for the Ball Pivoting Algorithm: detail loss and hole creation due to a too large radius (left) and a too small one (middle). A possible solution is to use multiple radii (right): here three different radii are used successively from smallest radius to largest one, allowing for the recovery of details and the filling of the holes.

with increased radius. We give in Figure 6 the results of using a single radius 0.0003, two successive radii 0.0003 and 0.0005, and finally three radii $0.0003, 0.0005, 0.002$. Table 2 sums up the time spent (without any parallelization) and mesh quality obtained for the three reconstructions. As can be seen, using too small a radius yields a mesh with a lot of holes, using a sequence of radii allows for creating a watertight mesh. The choice of an adequate radius or series of radii is in itself a problem, for which we give a very naive heuristic below.

Figure 6: Reconstructing the Stanford Bunny point cloud, with a single radius (0.0003), two radii $(0.0003; 0.0005)$ and three radii $(0.0003; 0.0005; 0.002)$.

Radius choice The choice of the radius parameter is a difficult one. We propose a naive way to estimate this parameter given the size s of the bounding box and the number of samples N. If the shape was a perfect sphere, then the surface of the equivalent sphere would be πs^2, thus the number of points per unit surface would be $\frac{N}{\pi s^2}$. The experimentations done on several shapes led us to select a radius such that, in average, the number of points per range neighborhoods is around 20. Then an estimation of the radius is $r = \sqrt{\frac{20}{N}}s$ which is the one proposed in the IPOL demonstration of the algorithm. This radius is only a rough estimate and should only be considered as such. When building the octree a more adapted radius can be estimated by considering the octree statistics (the

Radius	Time(s)	vertices	facets	boundary edges
0.0003	$10s$	318032	391898	272832
$0.0003; 0.0005$	$21s$	356252	698963	22727
$0.0003; 0.0005; 0.002$	$29s$	361443	713892	7897

Table 2: Computation time and watertight quality of the reconstructed surface depending on the radius quality. When choosing a small radius, the created mesh contains numerous holes (corresponding to a high number of boundary edges) and a relatively low number of vertices: data points are missed by the reconstruction. On the contrary, when using multiple radii, the mesh has fewer holes (fewer boundary edges) and more vertices. The original point set contains 362269 points. The corresponding reconstructions are shown in Figure 6

average number of points per non-empty cell at each scale). Let l be the scale where there are roughly 20 points in average, then a better estimate can be obtained by taking the radius equal to half the size of the cells at scale l.

Algorithm modification for multiple radii Using multiple radii does not change radically the algorithm. Assuming we are given a series of radii $(r_i)_{i=0,\cdots,n}$, we start by performing the regular BPA reconstructions with radius r_0. When the reconstruction with radius r_i is over, we consider the set of boundary edges and test for each boundary edge if its adjacent facet is in empty ball configuration for radius r_{i+1}, if it is the case, the edge is removed from the set of boundary edges and added to the front. Then the front expansion is performed again. The process is iterated until no radius is available anymore. An important aspect is that no seed triangle search is performed for radii r_i with $i \geq 1$, only front expansions.

6 Parallelization

6.1 Principle

The code can be well parallelized since it is a front propagation method. We use a shared memory parallelization method: each thread accesses the shared memory and processes independently a part of the data. More practically, we use the octree data structure to sort cells into sets, each set containing cells that can be processed independently as shown in Figure 7.

Processing a cell consists in finding all front edges contained (even partly) in the cell and expanding the front from them while ensuring that all added vertices remain in the dilated cell (i.e. in the cell or in a narrow band around the cell as depicted in Figure 7). Once the expansion is done, the algorithm looks for a seed triangle, and, in case such a seed is found, expands the new front until no front edge remains. Since these dilated cells do not intersect, one can safely process them simultaneously. After all cells in the batch have been processed, the sets of created facets are merged, and the next batch of cells is processed (Figure 7: first red cells are processed simultaneously then yellow, green and blue cells).

There is no guarantee at all that the result of the parallelized version of the algorithm will provide the same result as the single-threaded version. Indeed, choosing N different seeds and propagating from these seeds is not equivalent to starting from a single seed and propagating from a single front. This parallelization strategy ensures that no additional triangulation stitching step is required in the single-radius case: once all cells are processed, the final triangulation is available. In the case of multiple radii, the parallelization is slightly more complicated. Indeed the seed facets are only

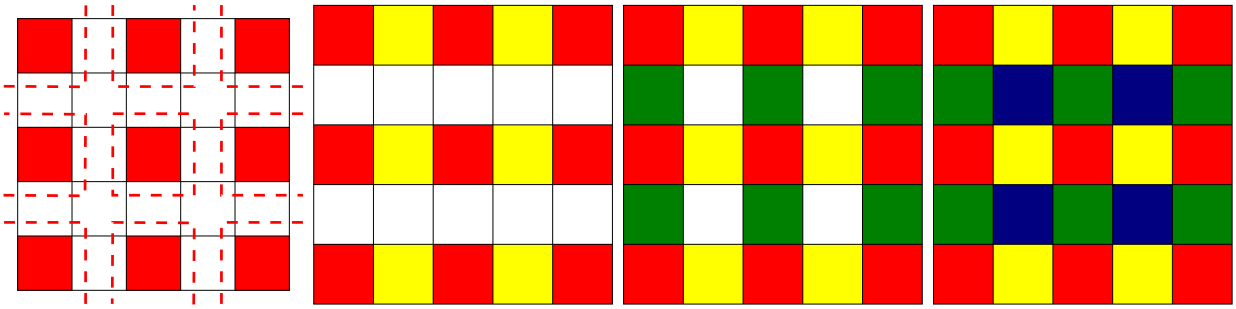


Figure 7: BPA parallelization in 2D. Cells that can be processed simultaneously are depicted in the same color. The first set of cells is depicted in red and the corresponding dilated cells are contoured in dashed red (left). The next set of processed cells is depicted in yellow, then in green and the last one in blue. Dilated cell contours are omitted on the last three figures for clarity.

looked for with the first radius: other radii will expand the front edge but will not create new seed facets. In that case, there is no guarantee that the front will not reenter an already processed cell: each set of cells can still be processed independently but after all batch of cells have been processed, there may remain some front edges. Therefore we add a small step that expands the triangulation globally from these edges. Fortunately, in practice, only a very limited amount of such edges remain so that this final expansion is not time-consuming.

Finally, the parallelization done here is not the same as the out-of-core extension proposed in the original article. Its goal is not to be able to handle larger data, but to accelerate the computation.

6.2 Parallelization Detail

Algorithm 7: Parallelization of the Ball Pivoting Algorithm (see text for details)

Input: An oriented input point cloud $\mathcal{P}$ sorted into an octree with a depth corresponding to the given radius r

Output: A surface mesh $\mathcal{T}$

1 $d \leftarrow 2 \cdot r$
2 $l \leftarrow$ smallest level at which cells have size greater than $2d$
3 **for** $i = 0 \cdots 8$ **do**
4 $cells \leftarrow$ cells of the octree at level l with child index i
5 **for** $C \in cells$ **do in parallel**
6 $C^d \leftarrow$ dilated cell C with bandwidth d
7 Find a seed inside C^d
8 Expand the triangulation (Algorithm 5) while ensuring that all vertices stay in C^d

The details are explained in Algorithm 7. Some important steps are explained hereafter:

- Line 1: the bandwidth to dilate the cell is chosen to ensure that once a dilated cell is processed, the triangulation front will not enter the cell again.
- Line 4: recall that each cell of the octree encodes its child index as explained in Figure 1.
- Line 8: the three vertices of the seed must lie inside the dilated containment cell C^d. When the ball is pivoted around an edge, if the candidate vertex does not belong to C^d, the edge is tagged as active, but still removed from the expansion set and no facet is created. On the

Point set	Number of Points	Radii	Computation Time	
			Single-threaded	Multi-threaded (4 cores)
Bunny	$360K$	$0.0003; 0.0005; 0.002$	$29s$	$12s$
Dragon	$1.5M$	$0.0003; 0.0005; 0.001$	$58s$	$41s$
Pyramid	$1M$	$0.3; 0.4; 0.6$	$99s$	$68s$

Table 3: Comparisons of single-threaded and multi-threaded versions of the code on several different point sets with different radii. The tests were done on a 4-core laptop, using the single-threaded version or the parallel version (*-p* option). The reconstructed shapes are shown in figures 10 and 11.

contrary, if no candidate at all is found (within C^d or outside of C^d), the edge is added to the set of *boundary_edges*.

The parallelization allows for much faster computation times as shown in Table 3. The computation time is strongly dependent on the sampling of the shape: if a single seed triangle search is needed, then it will be much faster than if this search has to be done a lot of times (since expanding the front is faster than looking for seeds).

7 Code

7.1 Dependencies

The code provided is a stand-alone C++ code, available at https://doi.org/10.5201/ipol.2014.81. 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 12.04, 12.10 and Debian 6.0.7 with g++ 4.4 or 4.7, 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 for OpenMP with the clang compiler, so that the parallelism is deactivated in that case.

7.2 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 standalone code, avoiding the need to link against such a heavy library as CGAL.

7.3 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, etc.). 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 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 the paper by Shewchuk [11]. Yet in our

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

case the consequences of not using any robust predicates is 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 disturbing global artifacts.

8 Experiments

8.1 An Artificial Shape

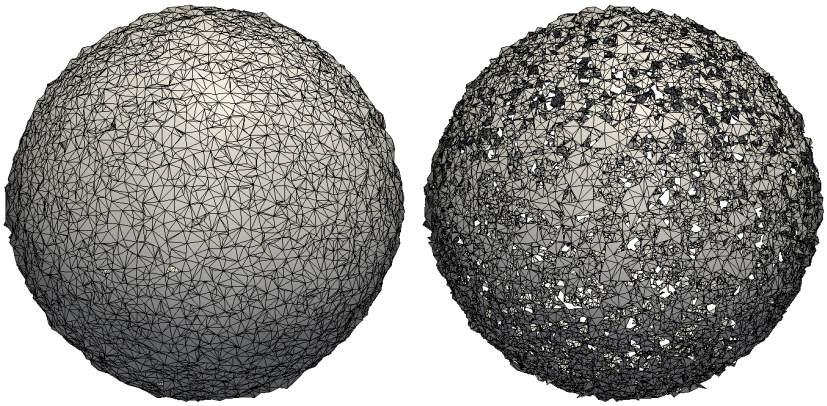
We first show the result of the Ball Pivoting on a perfect sphere of radius 2 containing 30000 points (Figure 8). We show in particular that increasing the radius on this simple toy example eventually allows for a watertight mesh. These experiments show that the ball pivoting is very dependent on the chosen input radius for the detail preservation. To show efficiently the effect of the ball pivoting algorithm when data are noisy, we show in Figure 9 the effect of applying the ball pivoting to a noisy sphere of 30000 samples. The algorithm creates a surface that is a kind of dilatation of the noisy shape (for the same reason as in Figure 5b). If a single radius is used (Figure 9a), it omits more than 2/3 of the input points, losing therefore its interpolating characteristic. The created mesh is rather watertight (10061 vertices, 19634 facets and 296 border edges). If several radii are used (Figure 9b), the mesh reconstructed interpolates more points but is not watertight at all (27568 vertices, 39544 facets and 73764 border edges).

(a) $r = 0.02$ (b) $r = 0.03$ (c) $r = 0.05$

Figure 8: Result of applying the ball pivoting to an artificial sphere with radius 2. Three experiments are run, each with a different radius: 0.02 (left), 0.03 (middle) and 0.05 (right). If the radius is too small, areas with lower density are not triangulated. But choosing too large a radius yields higher computation times.

8.2 Standard Shapes (Stanford Repository)

The Stanford Bunny contains 10 scans and 362000 oriented points. Bernardini et al. proposed to use a series of three radii (0.3, 0.5 and 2 mm, the shape being bounded in a box of size $150mm$). The pointset is obtained from the Stanford repository and a mean curvature motion is applied to filter the point positions as described in the paper by Digne et al. [5]. We also use a single scan containing 1568548 points from another standard shape: the Stanford Dragon. We show the result obtained by using multiple radii (0.3, 0.5 and 1mm). Both of these reconstructions are shown in Figure 10.



(a) $r = 0.05$ (b) $r = 0.02; 0.03; 0.05$

Figure 9: Applying the ball pivoting to a noisy sphere: $r = 0.05$ (left) and $r = 0.02; 0.03; 0.05$ (right). A single radius does not allow to interpolate the input data and applying multiple radii is not a solution in addition to being difficult to tune.

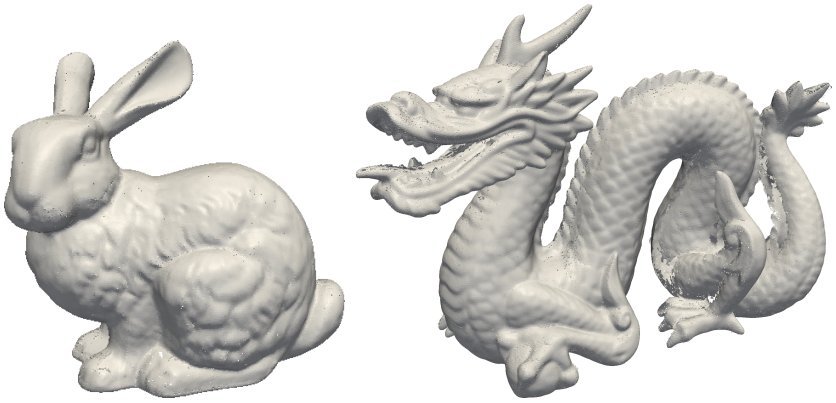


Figure 10: Bunny and Dragon reconstruction

8.3 The Farman 3D Point Sets [4]

The last displayed shapes come from the Farman dataset, a high resolution freely available dataset. The pyramid is a subsampled oriented version of the original dataset, containing only 1 million points (Figure 11).

9 Conclusion

This paper presented an implementation of the ball pivoting algorithm along with its parallelization, allowing for faster triangulation of oriented point clouds. Experiments show that the quality of the reconstructed surface heavily depends on the choice of the radius, or of an adequate series of increasing radii to fill up the holes. The question of the automatic choice of such series is open, although a naive strategy for an automatic radius was proposed here. It is the only hindrance to a fully automatic algorithm. A modification of the algorithm based on a *scale space* allowed for a better reconstruction using a single radius [5].

Figure 11: Reconstruction of the Pyramid using radii $0.3; 0.4; 0.6$mm (object height 80mm) and of the Tanagra using radius 0.1mm (object size: 220mm). Both shapes are from the Farman data set.

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").

Data Credits

The Stanford Bunny and Stanford Dragon (Figure 10) are from the Stanford 3D Scanning Repository[3]. The other shapes (Figure 11) are from the Farman Institute 3D Point Sets[4].

References

[1] F. Bernardini and C.L. Bajaj, *Sampling and reconstructing manifolds using alpha-shapes*, in Proceedings of the 9th Canadian Conference on Computational Geometry, 1997.

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

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

[3] 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), Springer, 2001, pp. 67–76. http://dx.doi.org/10.1145/383259.383266.

[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.

[5] J. Digne, J.M. Morel, C. 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.

[6] H. Edelsbrunner and E.P. Mücke, *Three-dimensional alpha shapes*, ACM Transactions on Graphics, 13 (1994), pp. 43–72. http://doi.acm.org/10.1145/174462.156635.

[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, ACM, 2009, pp. 538–539. http://doi.acm.org/10.1145/1653771.1653865.

[8] S.F. Frisken and R.N. Perry, *Simple and efficient traversal methods for quadtrees and octrees*, Journal of Graphics Tools, 7 (2002), pp. 1–11.

[9] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle, *Surface reconstruction from unorganized points*, in Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), ACM, 1992, pp. 71–78. http://doi.acm.org/10.1145/133994.134011.

[10] M. Kazhdan, M. Bolitho, and H. Hoppe, *Poisson surface reconstruction*, in Proceedings of the Fourth Eurographics Symposium on Geometry Processing, Eurographics Association, 2006, pp. 61–70. http://dl.acm.org/citation.cfm?id=1281957.1281965.

[11] J.R. Shewchuk, *Adaptive precision floating-point arithmetic and fast robust geometric predicates*, Discrete and Computational Geometry, 18 (1996), pp. 305–363.