The Ball-Pivoting Algorithm for Surface Reconstruction

Fausto Bernardini Joshua Mittleman Holly Rushmeier Cláudio Silva Gabriel Taubin

Abstract—**The Ball-Pivoting Algorithm (BPA) computes a triangle mesh interpolating a given point cloud. Typically the points are surface samples acquired with multiple range scans of an object. The principle of the BPA is very simple: Three points form a triangle if a ball of a user-specified radius ρ touches them without containing any other point. Starting with a seed triangle, the ball pivots around an edge (*i.e.* it revolves around the edge while keeping in contact with the edge's endpoints) until it touches another point, forming another triangle. The process continues until all reachable edges have been tried, and then starts from another seed triangle, until all points have been considered. We applied the BPA to datasets of millions of points representing actual scans of complex 3D objects. The relatively small amount of memory required by the BPA, its time efficiency, and the quality of the results obtained compare favorably with existing techniques.**

Keywords—**3D scanning, shape reconstruction, point cloud, range image.**

I. Introduction

Advances in 3D data-acquisition hardware have facilitated the more widespread use of scanning to document the geometry of physical objects for archival purposes or as a step in new product design. A typical 3D data acquisition pipeline consists of the following steps (adapted from [1]):

Scanning: Acquisition of surface samples with a measurement device, such as a laser range scanner or a stereographic system.

Data registration: Alignment of several scans into a single coordinate system.

Data integration: Interpolation of the measured samples, or points derived from the measured samples, with a surface representation, usually a triangle mesh.

Model conversion: Mesh decimation/optimization, fitting with higher-order representations etc.

This paper focuses on the data integration phase. We present a new method for finding a triangle mesh that interpolates an unorganized set of points. Figure 1 shows a closeup view of a 14M triangle mesh obtained by running our algorithm on hundreds of scans of Michelangelo's Florentine Pietà. The model took 30 minutes to compute on a Pentium II PC.

The method makes two mild assumptions about the samples that are valid for a wide range of acquisition techniques: that the samples are distributed over the entire surface with a spatial frequency greater than or equal to some application-specified minimum value, and that an estimate of the surface normal is available for each measured sample.

A. Main Contributions

- The method is conceptually simple. Starting with a seed triangle, it pivots a ball around each edge on the current mesh boundary until a new point is hit by the ball. The edge and point define a new triangle, which is added to the mesh, and the algorithm considers a new boundary edge for pivoting.
- The output mesh is a manifold subset of an alpha-shape [2] of the point set. Some of the nice properties of alpha-shapes can also be proved for our reconstruction.
- The Ball Pivoting Algorithm (BPA for short) is efficient in terms of execution time and storage requirements. It exhibited linear time performance on datasets consisting of millions of input samples. It has been implemented in a form that does not require all of the input data to be loaded into memory simultaneously. The resulting triangle mesh is incrementally saved to external storage during its computation, and does not use any additional memory.
- The BPA proved robust enough to handle the noise present in real scanned 3D data. It was tested on several large scanned datasets, and in particular was used to create models of Michelangelo's Florentine Pietà [3] from hundreds of scans acquired with a structured light sensor (Visual Interface's Virtuoso ShapeCamera).

The rest of the paper is structured as follows: In section II we define the problem and discuss related work. In section III we discuss the concepts underlying the Ball-Pivoting Algorithm, and in section IV describe the algorithm in detail. We present results in section V, and discuss open problems and future work in section VI.

IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598. email: {fausto, mittle, holly, taubin}@watson.ibm.com

Claudio Silva's current address: AT&T Labs-Research, Shannon Laboratory, 180 Park Avenue, Florham Park, NJ 07932-0971. csilva@research.att.com

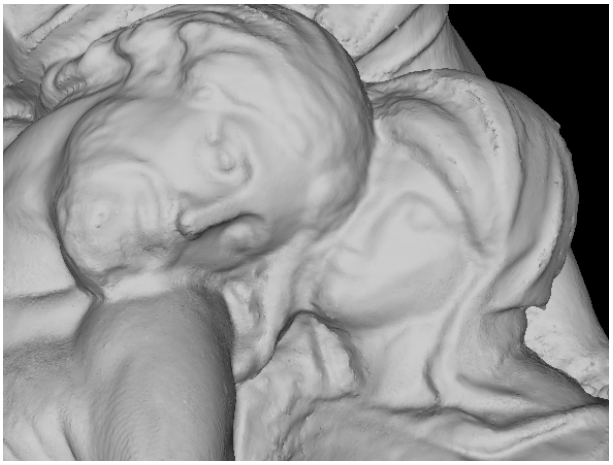


Fig. 1. Section of Michelangelo's Florentine Pietà. This 14M triangle mesh was generated from more than 700 scans using the ball pivoting algorithm.

II. Background

Recent years have seen a proliferation of scanning equipment and algorithms for synthesizing models from scanned data. We refer the reader to two recent reviews of research in the field [4],

[5]. In this section we focus on the role interpolating meshing schemes can play in scanning objects, and why they have not been used in practical scanning systems.

A. Interpolating Meshes in Scanning Systems

We define the scanning problem: Given an object, find a continuous representation of the object surface that captures features of a length scale $2d$ or larger. The value of d is dictated by the application. Capturing features of scale $2d$ requires sampling the surface with a spatial resolution of d or less. The surface may consist of large areas that can be well approximated by much sparser meshes; however in the absence of *a priori* information we need to begin with a sampling resolution of d or less to guarantee that no feature is missed.

We consider acquisition systems that produce sets of range images, i.e. arrays of depths, each of which covers a subset of the full surface. Because they are height fields with regular sampling, individual range images are easily meshed. The individual meshes can be used to compute an estimated surface normal for each sample point.

An ideal acquisition system would return samples lying exactly on the object surface. Any real measurement system introduces some error. However, if a system returns samples with an error that is orders of magnitude smaller than the minumum feature size, the sampling can be regarded as perfect. A surface can then be reconstructed by finding an interpolating mesh without additional operations on the measured data. Most scanning systems still need to account for acquisition error. There are two sources of error: error in registration; and error along the sensor line of sight. Estimates of actual surface points are usually derived by averaging samples from redundant scans. These estimates are then connected into a triangle mesh.

Most methods for estimating surface points depend on data structures that facilitate the construction of the mesh. Two classes of methods have been used successfully for large datasets; both assume negligible registration error and compute estimates to correct for line-of-sight error. The first of these classes is volumetric methods, such as that introduced by Curless and Levoy [6]. In these methods, individual aligned meshes are used to compute a signed-distance function on a volume grid encompassing the object. Estimated surface points are computed as the points on the grid where the distance funtion is zero. The structure of the volume then facilitates the construction of a mesh using the marching cubes algorithm [7].

The second class of methods are mesh stitching methods, such as the technique of Soucy and Laurendeau [8]. Disjoint height-field meshes are stitched into a single surface. Disjoint regions are defined by finding areas of overlap of different subsets of the set of scans. Estimated surface points for each region are computed as weighted averages of points from the overlapping scans. Estimated points in each region are then retriangulated, and the resulting meshes are stitched into a single mesh. Turk and Levoy developed a similar method [9], which first stitches (or zippers) the disjoint meshes and then computes estimated surface points.

We observe that in both classes of methods, the method of estimating surface points need not be so closely linked to the method for constructing the final mesh. In the volumetric approach, a technique other than marching cubes could be used for finding a triangle mesh passing through the estimated surface points. In the mesh-joining approaches, a technique for finding a mesh connecting all estimated surface points could be used in place of stitching together the existing meshes. Most importantly, with an efficient algorithm for computing a mesh which joins points, any method for computing estimated surface points could be used, including those that do not impose additional structure on the data and do not treat registration and line-of-sight error separately. For example, it has been demonstrated that reducing error in individual meshes before alignment can reduce registration error [10].

We are developing a method that moves samples within known scanner error bounds to conform the meshes to one another as they are aligned. Our current implementation of this method was used to preprocess the data shown in the results section. The method will be described in a future paper.

Finally, it may be desirable to find an interpolating mesh from measured data even if it contains uncompensated error. The preliminary mesh could be smoothed, cleaned, and decimated for use in planning functions. A mesh interpolating measured points could also be used as a starting point for computing consensus points.

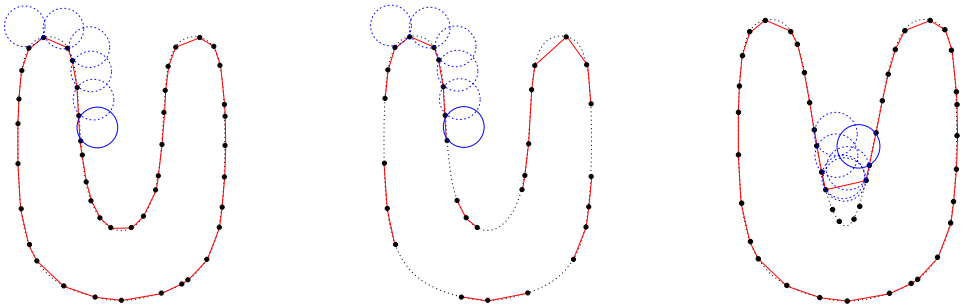
B. State of the Art for Interpolating Meshes

Existing interpolating techniques fall into two categories – sculpting-based [11], [12], [4] and region-growing [13], [14], [15], like the BPA. In sculpting-based methods, a volume tetrahedralization is computed from the data points, typically the 3D Delaunay triangulation. Tetrahedra are then removed from the convex hull to extract the original shape. Region-growing methods start with a seed triangle, consider a new point and join it to the existing region boundary, and continue until all points have been considered.

The strength of sculpting-based approaches is that they often provide theoretical guarantees for the quality of the resulting surface, e.g. that the topology is correct, and that the surface converges to the true surface as the sampling density increases (see e.g. [16], [17]). However, computing the required 3D Delaunay triangulation can be prohibitively expensive in terms of time and memory required, and can lead to numerical instability when dealing with datasets of millions of points. The goal of the BPA is to retain the strengths of previous interpolating techniques in a method that exhibits linear time complexity and robustness on real scanned data.

III. Surface Reconstruction and Ball-Pivoting

The main concept underlying the Ball-Pivoting Algorithm is quite simple. Let the manifold M be the surface of a three-dimensional object and S be a point-sampling of M. Let us assume for now that S is dense enough that a ρ-ball (a ball of radius ρ) cannot pass through the surface without touching sample points (see figure 3 for a 2D example). We start by placing a ρ-ball in contact with three sample points. Keeping the ball in contact with two of these initial points, we "pivot" the ball until it touches another point, as illustrated in figure 2 (more details are given in section IV-C). We pivot around each edge of the current mesh boundary. Triplets of points that the ball contacts



(a) (b) (c)

Fig. 3. The Ball Pivoting Algorithm in 2D. (a) A circle of radius ρ pivots from sample point to sample point, connecting them with edges. (b) When the sampling density is too low, some of the edges will not be created, leaving holes. (c) When the curvature of the manifold is larger than $1/\rho$, some of the sample points will not be reached by the pivoting ball, and features will be missed.

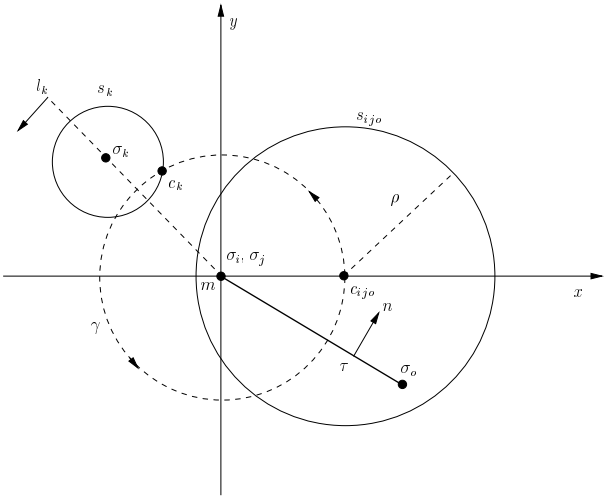


Fig. 2. Ball pivoting operation. See section IV-C for further details. The pivoting ball is in contact with the three vertices of triangle $\tau = (\sigma_i, \sigma_j, \sigma_o)$, whose normal is n. The pivoting edge $e_{(i,j)}$ lies on the z axis (perpendicular to the page and pointing towards the viewer), with its midpoint m at the origin. The circle s_{ijo} is the intersection of the pivoting ball with $z = 0$. The coordinate frame is such that the center c_{ijo} of the ball lies on the positive x axis. During pivoting, the ρ-ball stays in contact with the two edge endpoints σ_i, σ_j, and its center describes a circular trajectory γ with center in m and radius $||c_{ijo} - m||$. In its pivoting motion, the ball hits a new data point σ_k. Let s_k be the intersection of a ρ-sphere centered at σ_k with $z = 0$. The center c_k of the pivoting ball when it touches σ_k is the intersection of γ with s_k lying on the negative halfplane of oriented line l_k.

form new triangles. The set of triangles formed while the ball "walks" on the surface constitutes the interpolating mesh.

The BPA is closely related to alpha-shapes [18], [2]. In fact every triangle τ computed by the ρ-ball walk obviously has an empty smallest open ball b_τ whose radius is less than ρ (see [2], page 50). Thus, the BPA computes a subset of the 2-faces of the ρ-shape of S. These faces are also a subset of the 2-skeleton of the three-dimensional Delaunay triangulation of the point set. Alpha shapes are an effective tool for computing the "shape" of a point set. The surface reconstructed by the BPA retains some of the qualities of alpha-shapes: It has provable reconstruction guarantees under certain sampling assumptions, and an intuitively simple geometric meaning.

However, the 2-skeleton of an alpha-shape computed from a noisy sampling of a smooth manifold can contain multiple non-manifold connections. It is non-trivial to filter out unwanted components. Also, in their original formulation, alpha-shapes are computed by extracting a subset of the 3D Delaunay triangulation of the point set, a data structure that is not easily computed for datasets of millions of points. With the assumptions on the input stated in the introduction, the BPA efficiently and robustly computes a manifold subset of an alpha-shape that is well suited for this application.

In [16], sufficient conditions on the sampling density of a curve in the plane were derived which guarantee that the alpha-shape reconstruction is homeomorphic to the original manifold and that it lies within a bounded distance. The thorem can be easily extended to surfaces (stated here without proof): Suppose that for the smooth manifold M the sampling S satisfies the following properties:

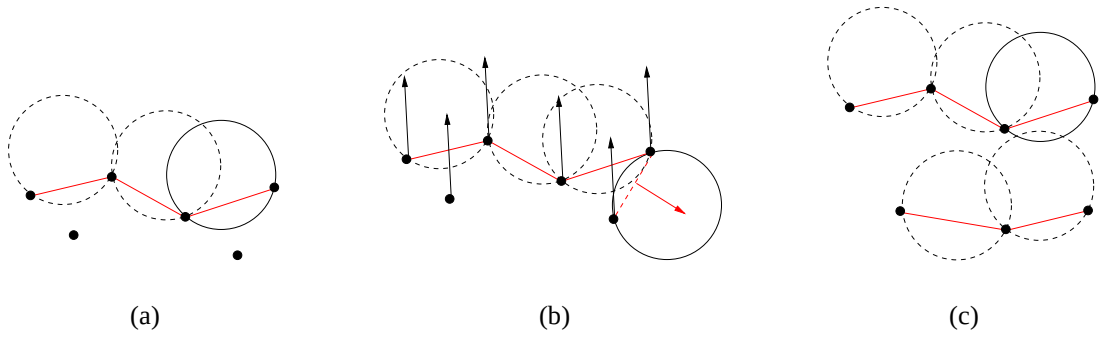
1. The intersection of any ball of radius ρ with the manifold is a topological disk.
2. Any ball of radius ρ centered on the manifold contains at least one sample point in its interior.

The first condition guarantees that the radius of curvature of the manifold is larger than ρ, and that the ρ-ball can also pass through cavities and other concave features without multiple contacts with the surface. The second condition tells us that the sampling is dense enough that the ball can walk on the sample points without leaving holes (see figure 3 for 2D examples). The BPA then produces a homeomorphic approximation T of the smooth manifold M. We can also define a homeomorphism $h : T \mapsto M$ such that the distance $||p - h(p)|| < \rho$.

In practice, we must often deal with less-than-ideal samplings. What is the behavior of the algorithm in these cases? Let us consider the case of real scanned data. Typical problems are missing points, non-uniform density, imperfectly-aligned overlapping range scans, and scanner line-of-sight error.[1]

The BPA is designed to process the output of an accurate registration/conformance algorithm (see section II), and does not attempt to average out noise or residual registration errors. Nonetheless, the BPA is robust in the presence of imperfect data.

[1] Some types of scanners also produce "outliers", points that lie far from the actual surface. These outliers occur more frequently at the boundaries of range images, or in the presence of sharp discontinuities. Outlier removal is best done with device-dependent preprocessing. The scanner used to acquire the data presented in the results section is not affected by this problem.



(a) (b) (c)

Fig. 4. Ball pivoting in the presence of noisy data. (a) Surface samples lying "below" surface level are not touched by the pivoting ball and remain isolated (and are discarded by the algorithm). (b) Due to missing data, the ball pivots around an edge until it touches a sample that belongs to a different part of the surface. By checking that triangle and data point normals are consistently oriented, we avoid generating a triangle in this case. (c) Noisy samples form two layers, distant enough to allow the ρ ball to "walk" on both layers. A spurious small component is created. Our seed selection strategy avoids the creation of a large number of these small components. Remaining ones can be removed with a simple postprocessing step. In all cases, the BPA outputs an orientable, triangulated manifold.

We augment the data points with approximate surface normals computed from the range maps to disambiguate cases that occur when dealing with missing or noisy data. For example, if parts of the surface have not been scanned, there will be holes larger than ρ in the sampling. It is then impossible to distinguish an interior and an exterior region with respect to the sampling. We use surface normals (for which we assume outward orientation) to decide surface orientation. For example, when choosing a seed triangle we check that the surface normals at the three vertices are consistently oriented.

Areas of density higher than ρ present no problem. The pivoting ball will still "walk" on the points, forming small triangles. If the data is noise-free and ρ is smaller than the local curvature, all points will be interpolated. More likely, points are affected by noise, and some of those lying below the surface will not be touched by the ball and will not be part of the reconstructed mesh (see figure 4(a)).

Missing points create holes that cannot be filled by the pivoting ball. Any postprocessing hole-filling algorithm could be employed; in particular, BPA can be applied multiple times, with increasing ball radii, as explained in section IV-F. However, we do need to handle possible ambiguities that missing data can introduce. When pivoting around a boundary edge, the ball can touch an unused point lying close to the surface. Again we use surface normals to decide whether the point touched is valid or not (see figure 4(b)). A triangle is rejected if the dot product of its normal with the surface normal is negative.

The presence of misaligned overlapping range scans can lead to poor results if the registration error is similar to the pivoting ball size. Undesired small connected components lying close to the main surface will be formed, and the main surface affected by high frequency noise (see figure 4(c)). Our seed selection strategy avoids creating a large number of such small components. A simple postprocessing that removes small components and surface smoothing [19] can significantly improve the result in these cases, at least aesthetically.

Regardless of the defects in the data, the BPA is guaranteed to build an orientable manifold. Notice that the BPA will always try to build the largest possible connected manifold from a given seed triangle.

Choosing a suitable value for the radius ρ of the pivoting ball is typically easy. Current structured-light or laser triangulation scanners produce very dense samplings, exceeding our requirement that intersample distance be less than half the size of features of interest. Knowledge of the sampling density characteristics of the scanner, and of the feature size one wants to capture, are enough to choose an appropriate radius. Alternatively, one could analyze a small subset of the data to compute the point density. An uneven sampling might arise when scanning a complex surface, with regions that project into small areas in the scanner direction. The best approach is to take additional scans with the scanner perpendicular to such regions, to acquire additional data. Notice however that the BPA can be applied multiple times, with increasing ball radii, to handle uneven sampling densities, as described in section IV-F.

IV. The Ball-Pivoting Algorithm

The BPA follows the advancing-front paradigm to incrementally build an interpolating triangulation. BPA takes as input a list of surface-sample data points σ_i, each associated with a normal n_i (and other optional attributes, such as texture coordinates), and a ball radius ρ. The basic algorithm works by finding a *seed triangle* (*i.e.*, three data points $(\sigma_i, \sigma_j, \sigma_k)$ such that a ball of radius ρ touching them contains no other data point), and adding one triangle at a time by performing the ball pivoting operation explained in section III.

The *front* $\mathcal{F}$ is represented as a collection of linked lists of edges, and is initially composed of a single loop containing the three edges defined by the first seed triangle. Each edge $e_{(i,j)}$ of the front, is represented by its two endpoints (σ_i, σ_j), the opposite vertex σ_o, the center c_{ijo} of the ball that touches all three points, and links to the previous and next edge along in the same loop of the front. An edge can be *active*, *boundary* or *frozen*. An *active* edge is one that will be used for pivoting. If it is not possible to pivot from an edge, it is marked as *boundary*. The *frozen* state is explained below, in the context of our out-of-core extensions. Keeping all this information with each edge makes it simpler to pivot the ball around it. The reason the front is a collection of linked lists, instead of a single one, is that as the ball pivots along an edge, depending on whether it touches a newly encountered data point or a previously used one, the front changes topology. BPA handles all cases with two simple topological operators, *join* and *glue*, which ensure that at any time the front is a collection of linked lists.

Algorithm *BPA*(S, ρ)

```
1.  while (true)

2.      while (e_(i,j) = get_active_edge(F))
3.          if (σ_k = ball_pivot(e_(i,j)) &&
                    (not_used(σ_k) || on_front(σ_k)))
4.              output_triangle(σ_i, σ_k, σ_j)
5.              join(e_(i,j), σ_k, F)
6 .             if (e_(k,i) ∈ F) glue(e_(i,k), e_(k,i), F)
7 .             if (e_(j,k) ∈ F) glue(e_(k,j), e_(j,k), F)
8 .         else
9 .             mark_as_boundary(e_(i,j))

10.     if ((σ_i, σ_j, σ_k) = find_seed_triangle())
11.         output_triangle(σ_i, σ_j, σ_k)
12.         insert_edge(e_(i,j), F)
13.         insert_edge(e_(j,k), F)
14.         insert_edge(e_(k,i), F)
15.     else
16.         return
```

Fig. 5. Skeleton of the BPA algorithm. Several necessary error tests have been left out for readability, such as edge orientation checks. The edges in the front $\mathcal{F}$ are generally accessed by keeping a queue of active edges. The *join* operation adds two active edges to the front. The *glue* operation deletes two edges from the front, and changes the topology of the front by breaking a single loop into two, or combining two loops into one. See text for details. The *find_seed_triangle* function returns a ρ-exposed triangle, which is used to initialize the front.

The basic BPA algorithm is shown in figure 5. Below we detail the functions and data structures used. In particular, we later describe a simple modification necessary to the basic algorithm to support efficient out-of-core execution. This allows BPA to triangulate large datasets with minimal memory usage.

A. Spatial queries

Both *ball_pivot* and *find_seed_triangle* (lines 3 and 10 in figure 5) require efficient lookup of the subset of points contained in a small spatial neighborhood. We implemented this spatial query using a regular grid of cubic cells, or voxels. Each voxel has sides of size $\delta = 2\rho$. Data points are stored in a list, and the list is organized using bucket-sort so that points lying in the same voxel form a contiguous sublist. Each voxel stores a pointer to the beginning of its sublist of points (to the next sublist if the voxel is empty). An extra voxel at the end of the grid stores a NULL pointer. To visit all points in a voxel it is sufficient to traverse the list from the node pointed to by the voxel to the one pointed to by the next voxel.

Given a point p we can easily find the voxel V it lies in by dividing its coordinates by δ. We usually need to look up all points within 2ρ distance from p, which are a subset of all points contained in the 27 voxels adjacent to V (including V itself).

The grid allows constant-time access to the points. Its size would be prohibitive if we processed a large dataset in one step; but an out-of-core implementation, described in section IV-E, can process the data in manageable chunks. Memory usage can be further reduced, at the expense of a slower access, using more compact representations, such as a sparse matrix data structure.

B. Seed selection

Given data satisfying the conditions of the reconstruction theorem of section III, one seed per connected component is enough to reconstruct the entire manifold (function *find_seed_triangle* at line 10 in figure 5). A simple way to find a valid seed is to:

- Pick any point σ not yet used by the reconstructed triangulation.
- Consider all pairs of points σ_a, σ_b in its neighborhood in order of distance from σ.
- Build potential seed triangles $\sigma, \sigma_a, \sigma_b$.
- Check that the triangle normal is consistent with the vertex normals, *i.e.* pointing outward.
- Test that a ρ-ball with center in the outward halfspace touches all three vertices and contains no other data point.
- Stop when a valid seed triangle has been found.

In the presence of noisy, incomplete data, it is important to select an efficient seed-searching strategy. Given a valid seed, the algorithm builds the largest possible connected component containing the seed. Noisy points lying at a distance slightly larger than 2ρ from the reconstructed triangulation could form other potential seed triangles, leading to the construction of small sets of triangles lying close to the main surface (see figure 4(c)). These small components are an artifact of the noise present in the data, and are usually undesired. While they are easy to eliminate by post-filtering the data, a significant amount of computational resources is wasted in constructing them.

We can however observe the following: If we limit ourselves to considering only one data point per voxel as a candidate vertex for a seed triangle, we cannot miss components spanning a volume larger than a few voxels. Also, for a given voxel, consider the average normal n of points within it. This normal approximates the surface normal in that region. Since we want our ball to walk "on" the surface, it is convenient to first consider points whose projection onto n is large and positive.

We therefore simply keep a list of non-empty voxels. We search these voxels for valid seed triangles, and when one is found, we start building a triangulation using pivoting operations. When no more pivoting is possible, we continue the search for a seed triangle from where we had stopped, skipping all voxels containing a point that is now part of the triangulation. When no more seeds can be found, the algorithm stops.

C. Ball Pivoting

A pivoting operation (line 3 in figure 5) starts with a triangle $\tau = (\sigma_i, \sigma_j, \sigma_o)$ and a ball of radius ρ that touches its three vertices. Without loss of generality, assume edge $e_{(i,j)}$ is the pivoting edge. The ball in its initial position (let c_{ijo} be its center) does not contain any data point, either because τ is a seed triangle, or because τ was computed by a previous pivoting operation. The pivoting is in principle a continuous motion of the ball, during which the ball stays in contact with the two endpoints of $e_{(i,j)}$, as illustrated in figure 2. Because of this contact, the motion is constrained as follows: The center c_{ijo} of the ball describes a circle γ which lies on the plane perpendicular to

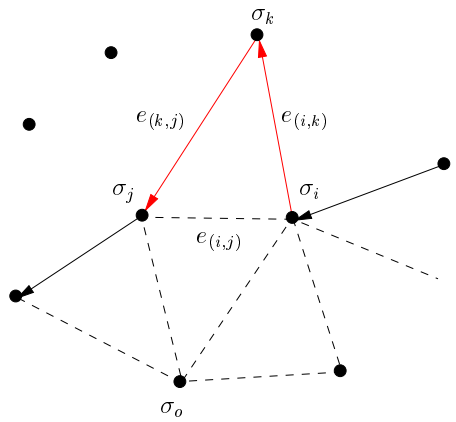


Fig. 6. A *join* operation simply adds a new triangle, removing edge $e_{(i,j)}$ from the front and adding the two new edges $e_{(i,k)}$ and $e_{(k,j)}$.

$e_{(i,j)}$ and through its midpoint $m = \frac{1}{2}(\sigma_j + \sigma_i)$. The center of this circular trajectory is m and its radius is $||c_{ijo} - m||$. During this motion, the ball may hit another point σ_k. If no point is hit, then the edge is a boundary edge. Otherwise, the triangle $(\sigma_i, \sigma_k, \sigma_j)$ is a new valid triangle, and the ball in its final position does not contain any other point, thus being a valid starting ball for the next pivoting operation.

In practice we find σ_k as follows. We consider all points in a 2ρ-neighborhood of m. For each such point σ_x, we compute the center c_x of the ball touching σ_i, σ_j and σ_x, if such a ball exists. Each c_x lies on the circular trajectory γ around m, and can be computed by intersecting a ρ-sphere centered at σ_x with the circle γ. Of these points c_x we select the one that is first along the trajectory γ. We report the first point hit and the corresponding ball center. Trivial rejection tests can be added to speed up finding the first hit-point.

D. The join and glue operations

These two operations generate triangles while adding and removing edges from the front loops (lines 5-7 in figure 5).

The simpler operation is the *join*, which is used when the ball pivots around edge $e_{(i,j)}$, touching a *not_used* vertex σ_k (*i.e.*, σ_k is a vertex that is not yet part of the mesh). In this case, we output the triangle $(\sigma_i, \sigma_k, \sigma_j)$, and locally modify the front by removing $e_{(i,j)}$ and adding the two edges $e_{(i,k)}$ and $e_{(k,j)}$ (see figure 6).

When σ_k is already part of the mesh, one of two cases can arise:

1. σ_k is an internal mesh vertex, (*i.e.*, no front edge uses σ_k). The corresponding triangle cannot be generated, since it would create a non-manifold vertex. In this case, $e_{(i,j)}$ is simply marked as a boundary edge;
2. σ_k belongs to the front. After checking edge orientation to avoid creating a non-orientable manifold, we apply a *join* operation, and output the new mesh triangle $(\sigma_i, \sigma_k, \sigma_j)$. The *join* could potentially create (one or two) pairs of coincident edges (with opposite orientation), which are removed by the *glue* operation.

The *glue* operation removes from the front pairs of coincident edges, with opposite orientation (coincident edges with the same orientation are never created by the algortihm). For example, when edge $e_{(i,k)}$ is added to the front by a *join* operation (the same applies to $e_{(k,j)}$), if edge $e_{(k,i)}$ is on the front, *glue* will remove the pair of edges $e_{(i,k)}, e_{(k,i)}$ and adjust the front accordingly. Four cases are possible, as illustrated in figure 7.

A sequence of *join* and *glue* operations is illustrated in figure 8.

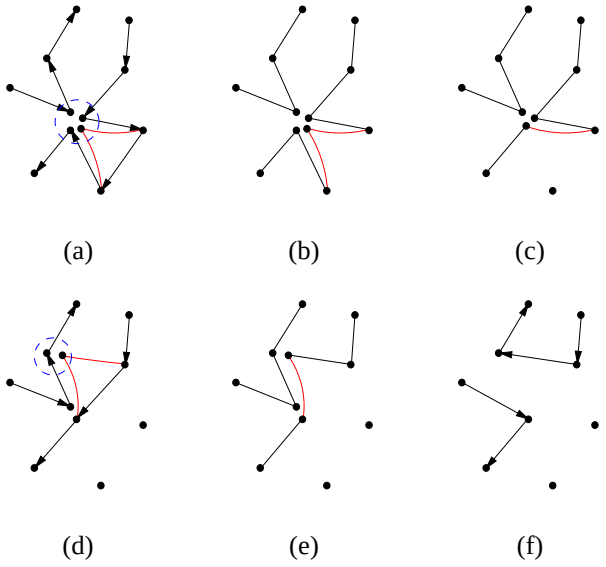


Fig. 8. Example of a sequence of *join* and *glue* operations. (a) A new triangle is to be added to the existing front. The four front vertices inside the dashed circle all represent a single data point. (b) A *join* removes an edge and creates two new front edges, coincident with existing ones. (c), (d) Two *glue* operations remove coincident edge pairs. (d) also shows the next triangle added. (e) Only one of the edges created by this *join* is coincident with an existing edge. (f) One *glue* removes the duplicate pair.

E. Out-of-core extensions

Being able to use a personal computer to triangulate high-resolution scans allows inexpensive on-site processing of data. Due to their locality of reference, advancing-front algorithms are suited to very simple out-of-core extensions.

We employed a simple data-slicing scheme to extend the algorithm shown in figure 5. The basic idea is to cache the portion of the dataset currently being used for pivoting, to dump data no longer being used, and to load data as it is needed. In our case, we use two axis-aligned planes π_0 and π_1 to define the active region of work for pivoting. We initially place π_0 in such a way that no data points lie "below" it, and π_1 above π_0 at some user-specified distance. As each edge is created, we test if its endpoints are "above" π_1; in this case, we mark the edge *frozen*. When all edges remaining in the queue are *frozen*, we simply shift π_0 and π_1 "upwards", and update all *frozen* into *active* edges, and so on. A subset of data points is loaded and discarded from memory when the corresponding bounding box enters and exits the active slice. Scans can easily be preprocessed to break them up into smaller meshes, so that they span only a few slices, and memory load remains low.

The only change required in the algorithm to implement this refinement is an outer loop to move the active slice, and the addition of the instructions to unfreeze edges between lines 1–2 of figure 5.

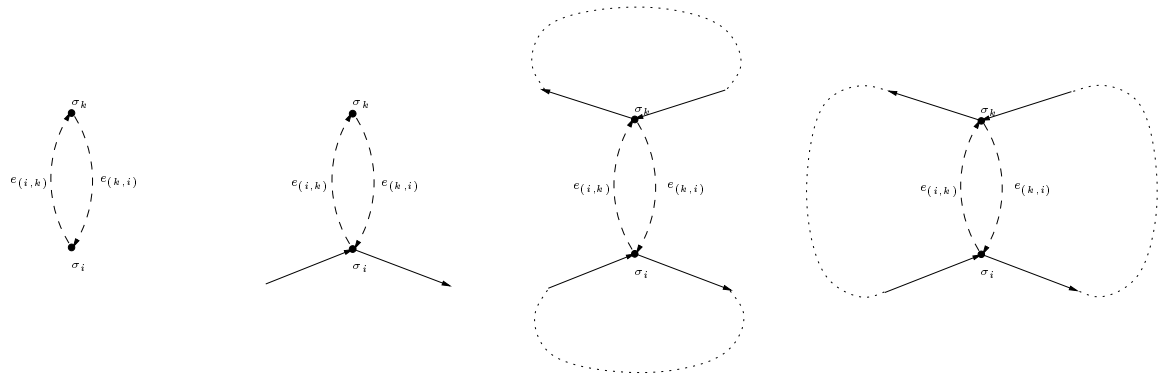


Fig. 7. A *glue* operation is applied when *join* creates an edge identical to an existing edge, but with opposite orientation. The two coincident edges are removed, and the front adjusted accordingly. There are four possible cases: (a) The two edges form a loop. The loop is deleted from the front. (b) Two edges belong to the same loop and are adjacent. The edges are removed and the loop shortened. (c) The edges are not adjacent, and they belong to the same loop. The loop is split into two. (d) The edges are not adjacent and belong to two different loops. The loops are merged into a single loop.

F. Multiple passes

To deal with unevenly sampled surfaces, we can easily extend the algorithm to run multiple passes with increasing ball radii. The user specifies a list of radii $\{\rho_0, \ldots, \rho_n\}$ as input parameters. In each slice, for increasing $\rho_i, i = 0, \ldots, n$, we start by inserting the points in a grid of voxel size $\delta = 2\rho_i$. We let BPA run until there are no more active edges in the queue. At this point we increment i, go through all front edges, and check whether each edge with its opposite vertex σ_o forms a valid seed triangle for a ball of radius ρ_i. If it is, then it is added to the queue of active edges. Finally, the pivoting is started again.

G. Remarks

The BPA algorithm was implemented in C++ using the Standard Template Library. The whole code is less than 4000 lines, including the out-of-core extensions.

The algorithm is linear in the number of data points and uses linear storage, under the assumption that the data density is bounded. This assumption is appropriate for scanned data, which is collected by equipment with a known sample spacing. Even if several scans overlap, the total number of points in any region will be bounded by a known constant.

Most steps are simple $O(1)$ state checks or updates to queues, linked lists, and the like. With bounded density, a point need only be related to a constant number of neighbors. So, for example, a point can only be contained in a constant number of loops in the advancing front. The two operations *ball_pivot* and *find_seed_triangle* are more complex.

Each *ball_pivot* operates on a different mesh edge, so the number of pivots is $O(n)$. A single pivot requires identifying all points in a 2ρ neighborhood. A list of these points can be collected from 27 voxels surrounding the candidate point in our grid. With bounded density, this list has constant size B. We perform a few algebraic computations on each point in the list and select the minimum result, all $O(1)$ operations on a list of size $O(1)$.

Each *find_seed_triangle* picks unused points one at a time and tests whether any incident triangle is a valid seed. No point is considered more than once, so this test is performed only $O(n)$ times. To test a candidate point, we gather the same point-list discussed above, and consider pairs of points until we either find a seed triangle or reject the candidate. Testing one of these triangles may require classifying every nearby point against a sphere touching the three vertices, in the worst case, $O(B^3) = O(1)$ steps. In practice, we limit the number of candidate points and triangles tested by the heuristics discussed in section IV-B.

An in-core implementation of the BPA uses $O(n + L)$ memory, where L is the number of cells in the voxel grid. The $O(n)$ term includes the data, the advancing front (which can only include each mesh edge once), and the candidate edge queue. Our out-of-core implementation uses $O(m + L')$ memory, where m is the number of data points in the largest slice and L' is the size of the smaller grid covering a single slice. Since the user can control the size of slices, memory requirements can be tailored to the available hardware. The voxel grid can be more compactly represented as a sparse matrix, with a small (constant) increase in access time.

V. Experimental Results

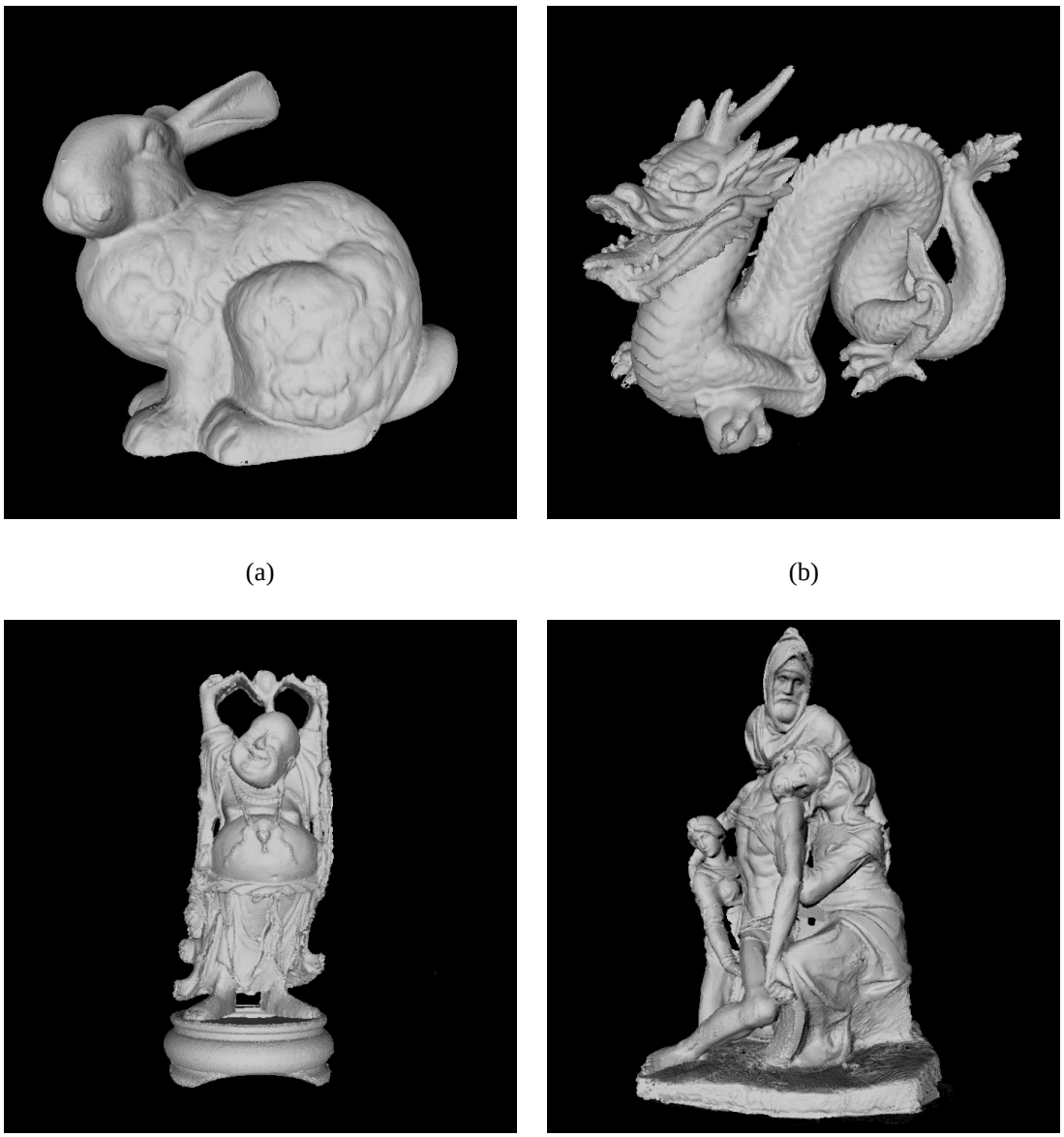
Our experiments for this paper were all conducted on one 450MHz Pentium II Xeon processor of an IBM IntelliStation Z Pro running Windows NT 4.0.

In our experiments we used several datasets: "clean" dataset (*i.e.*, points from analytical surface, see figure 9); the datasets from the Stanford scanning database (see figure 11(a)-(c)); and a very large dataset we acquired ourselves (and the main motivation of this work), a model of Michelangelo's Florentine Pietà [3] (see figure 11(d)).

To allow flexible input of multiple scans and out-of-core execution, our program reads its input in four parts: a list of individual scans to be converted into a single coherent triangle mesh; and for each scan, a transformation matrix, a post-transform bounding box (used to quickly estimate the mesh position for assignment to a slice), and the actual scan, which is loaded only when needed.

Dataset	# Pts	# Scans	ρ	# Slices	# Triangles	Mem. Usage	I/O Time	CPU Time
Clean	11K	1	4	-	22K	4	1.2secs	1.8secs
Bunny	361K	10	0.3, 0.5, 2	-	710K	86	4.5secs	2.1
Dragon	2.0M	71	0.3, 0.5, 1	-	3.5M	228	22secs	10.1
Buddha	3.3M	58	0.2, 0.5, 1	-	5.2M	325	48secs	16.9
Out of core								
Dragon	2.1M	1452	0.3, 0.5, 1	23	3.5M	137	1.0	19.8
Buddha	3.5M	1122	0.2, 0.5, 1	24	5.2M	155	2.1	26.8
Pietà	7.2M	770	1.5, 3, 6	24	14M	180	2.5	28.5

Fig. 10. Summary of results. *# of Pts* and *# of Scans* are the original number of data points and range images respectively. ρ lists the radii of the pivoting balls, in *mm*. Multiple radii mean that multiple passes of the algorithm, with increasing ball size, were used. *# Slices* is the number of slices into which the data is partitioned for out-of-core processing. *# of Triangles* is the number of triangles created by BPA. *Mem. Usage* is the maximum amount of memory used at any time during mesh generation, in MB. *I/O Time* is the time spent reading the input binary files; it also includes the time to write the output mesh, as an indexed triangle set, in binary format. *CPU Time* is the time spent computing the triangulation. All times are in minutes, except where otherwise stated. All tests were performed on a 450MHz Pentium II Xeon.



(a) (b)

(c) (d)

Fig. 11. Results. (a) Stanford bunny. (b) Stanford dragon. (c) Stanford Buddha. (d) Preliminary reconstruction of Michelangelo's Florentine Pietà.

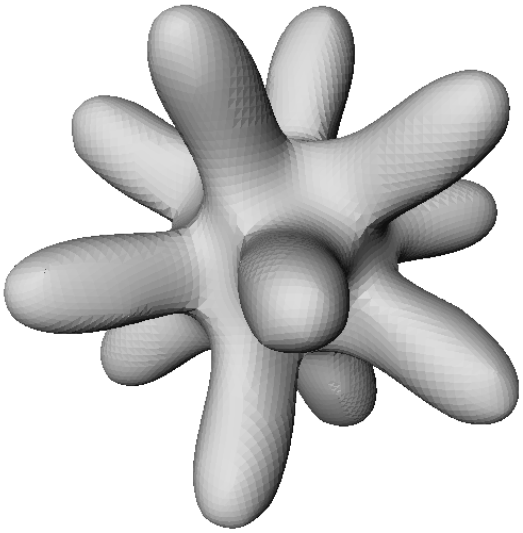


Fig. 9. Results. “Clean” data computed from an analytical surface.

A. Experiments

The table in figure 10 summarizes our results. The “clean” dataset is a collection of points from an analytical surface.

The Stanford Bunny, Dragon and Buddha datasets are multiple laser range scans of small objects. The scanner used to acquire the data was a CyberWare 3030MS.

These data required some minor preprocessing. We used the Standord *vrip* program to connect the points within each individual range data scan to provide estimates of surface normals. We also removed the plane carvers, large planes of triangles used for hole-filling by algorithms described in [6]. This change was made only for aesthetic reasons; BPA has no problem handling the full input.

In order to confirm the effectiveness of our out-of-core capabilities, we modified the Stanford Dragon by subdividing each range mesh into several pieces, multiplying the original 71 meshes to over 7500. A similar preprocessing was also applied to the Buddha dataset. We note that such decompositions can be performed efficiently for arbitrarily large range scans (which do not necessarily need to fit in memory) by the techniques described in [20].

The Pietà data has undergone extensive preprocessing during and after scanning and registration that is out of the scope of this paper. The data is large enough that it cannot be processed in-core, and is only processed in slices. The scanning of the Pietà also included the capture of multiple color images with calibrated lighting, from which reflectance and normals maps to augment the geometric data are computed (see [21]).

VI. Conclusions

In this paper, we introduced the Ball-Pivoting Algorithm, an advancing-front algorithm to incrementally build an interpolating triangulation of a given point cloud. BPA has several desirable properties:

- **Intuitive:** BPA triangulates a set of points by “rolling” a ρ-ball on the point cloud. The user chooses only a single parameter.
- **Flexible, efficient, and robust:** Our test datasets ranged from small synthetic data to large real-world scans. We have shown that our implementation of BPA works on datasets of millions of points representing actual scans of complex 3D objects. For our Pietà data, we found that on a Pentiun II PC the algorithm generates and writes to disk the output mesh at a rate of roughly 500K triangles per minute.
- **Theoretical foundation:** BPA is related to alpha-shapes [2], and given sufficiently dense sampling, it is guaranteed to reconstruct a surface homeomorphic to and within a bounded distance from the original manifold.

There are some avenues for further work. It would be interesting to evaluate whether BPA can be used to triangulate surfaces sampled with particle systems. This possibility was left as an open problem in [22], and further developed in [23] in the context of isosurface generation.

By using weighted points, we might be able to generate triangulations of adaptive samplings. The sampling density could be changed depending on local surface properties, and point weights accordingly assigned or computed. An extension of our algorithm along the lines of the weighted generalization of alpha-shapes [18] should be able to generate a more compact, adaptive, interpolating triangulation.

We have done some initial experiments in using a smoothing algorithm adapted from [19] to pre-process the data and to compute consensus points from multiple overlapping scans to be used as input to the BPA, while at the same time making small refinements to the rigid alignment of the scans to each other. Datasets used in this paper were pre-processed using our current implementation of this algorithm.

Acknowledgments. Thanks to the Stanford University Computer Graphics Laboratory, for making some of the range data used in this paper publicly available. The Museo dell’Opera del Duomo in Florence, Italy allowed us to scan Michelangelo’s Pietà. We acknowledge their kind collaboration.

References

[1] K. Pulli, T. Duchamp, H. Hoppe, J. McDonald, L. Shapiro, and W. Stuetzle, “Robust meshes from multiple range maps,” in *Intl. Conf. on Recent Advances in 3-D Digital Imaging and Modeling*. May 1997, pp. 205–211, IEEE Computer Society Press.

[2] H. Edelsbrunner and E. P. Mücke, “Three-dimensional alpha shapes,” *ACM Trans. Graph.*, vol. 13, no. 1, pp. 43–72, Jan. 1994.

[3] J. Abouaf, “The Florentine Pietà: Can visualization solve the 450-year-old mystery?,” *IEEE Computer Graphics & Applications*, vol. 19, no. 1, pp. 6–10, Feb. 1999.

[4] F. Bernardini, C. Bajaj, J. Chen, and D. Schikore, “Automatic reconstruction of 3D CAD models from digital scans,” *International Journal of Computational Geometry and Applications*, vol. 9, no. 4 & 5, pp. 327–370, Aug.-Oct. 1999.

[5] R. Mencl and H. Müller, “Interpolation and approximation of surfaces from three-dimensional scattered data points,” in *Proceeding of Eurographics ’98*. Eurographics, 1998, State of the Art Reports.

[6] B. Curless and M. Levoy, “A volumetric method for building complex models from range images,” in *Computer Graphics Proceedings*, 1996, Annual Conference Series. Proceedings of SIGGRAPH 96, pp. 303–312.

[7] W. Lorensen and H. Cline, “Marching cubes: a high resolution 3d surface construction algorithm,” *Comput. Graph.*, vol. 21, no. 4, pp. 163–170, 1987.

[8] M. Soucy and D. Laurendeau, “A general surface approach to the integration of a set of range views,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, no. 4, pp. 344–358, Apr. 1995.

[9] G. Turk and M. Levoy, “Zippered polygonal meshes from range images,” in *Computer Graphics Proceedings*, 1994, Annual Conference Series. Proceedings of SIGGRAPH 94, pp. 311–318.

[10] C. Dorai, G. Wang, A. K. Jain, and C. Mercer, "Registration and integration of multiple object views for 3D model construction," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 1, pp. 83–89, Jan. 1998.

[11] C. Bajaj, F. Bernardini, and G. Xu, "Automatic reconstruction of surfaces and scalar fields from 3D scans," in *Computer Graphics Proceedings*, 1995, Annual Conference Series. Proceedings of SIGGRAPH 95, pp. 109–118.

[12] N. Amenta, M. Bern, and M. Kamvysselis, "A new voronoi-based surface reconstruction algorithm," in *Proc. SIGGRAPH '98*, July 1998, Computer Graphics Proceedings, Annual Conference Series, pp. 415–412.

[13] J.-D. Boissonnat, "Geometric structures for three-dimensional shape representation," *ACM Trans. Graph.*, vol. 3, no. 4, pp. 266–286, 1984.

[14] R. Mencl, "A graph-based approach to surface reconstruction," *Computer Graphics Forum*, vol. 14, no. 3, pp. 445–456, 1995, Proc. of EUROGRAPHICS '95.

[15] A Hilton, A Stoddart, J Illingworth, and T Windeatt, "Marching triangles: Range image fusion for complex object modelling," in *Proc of IEEE International Conference on Image Processing*, Laussane, 1996, vol. 2, pp. 381–384.

[16] F. Bernardini and C. Bajaj, "Sampling and reconstructing manifolds using alpha-shapes," in *Proc. of the Ninth Canadian Conference on Computational Geometry*, Aug. 1997, pp. 193–198, Updated online version available at www.qucis.queensu.ca/cccg97.

[17] Nina Amenta and Marshall Bern, "Surface reconstruction by voronoi filtering," in *Proc. 14th Annual ACM Sympos. Comput. Geom.*, 1998, pp. 39–48.

[18] H. Edelsbrunner, "Weighted alpha shapes," Technical Report UIUCDCS-R-92-1760, Dept. Comput. Sci., Univ. Illinois, Urbana, IL, 1992.

[19] Gabriel Taubin, "A signal processing approach to fair surface design," in *Proc. of SIGGRAPH '95*. ACM SIGGRAPH, Aug. 1995, Computer Graphics Proceedings, Annual Conference Series, pp. 351–358.

[20] Yi-Jen Chiang, Cláudio T. Silva, and William Schroeder, "Interactive out-of-core isosurface extraction," in *Proc. IEEE Visualization ́98*, Nov. 1998, pp. 167–174.

[21] H. Rushmeier and F. Bernardini, "Computing consistent normals and colors from photometric data," in *Proc. of the Second Intl. Conf. on 3-D Digital Imaging and Modeling*, Ottawa, Canada, October 1999, To appear.

[22] Andrew P. Witkin and Paul S. Heckbert, "Using particles to sample and control implicit surfaces," in *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Andrew Glassner, Ed. ACM SIGGRAPH, July 1994, Computer Graphics Proceedings, Annual Conference Series, pp. 269–278, ACM Press, ISBN 0-89791-667-0.

[23] Patricia Crossno and Edward Angel, "Isosurface extraction using particle systems," in *IEEE Visualization ́97*, Roni Yagel and Hans Hagen, Eds. IEEE, Nov. 1997, pp. 495–498.