Exact Polyhedral Visual Hulls

Jean-Sébastien Franco and Edmond Boyer
INRIA Rhône-Alpes
ZIRST Montbonnot, 655, av. de l'Europe, 38334 St-Ismier CEDEX, France
jean-sebastien.franco@inrialpes.fr *edmond.boyer@inrialpes.fr*

Abstract

We propose an exact method for efficiently and robustly computing the visual hull of an object from image contours. Unlike most existing approaches, ours computes an exact description of the visual hull polyhedron associated to polygonal image contours. Furthermore, the proposed approach is fast and allows real-time recovery of both manifold and watertight visual hull polyhedra. The process involves three main steps. First, a coarse geometrical approximation of the visual hull is computed by retrieving its viewing edges, an unconnected subset of the wanted mesh. Then, local orientation and connectivity rules are used to walk along the relevant viewing cone intersection boundaries, so as to iteratively generate the missing surface points and connections. A final connection walkthrough allows us to identify the planar contours for each face of the polyhedron. Implementation details and results with synthetic and real data are presented.

1 Introduction

Visual hulls are object shape approximations which can be determined from object silhouettes in images. Such approximations capture all the geometric information given by the image silhouettes. Visual hulls are extensively used in a number of modeling applications including human modeling systems. Their popularity is largely due to the fact that straightforward approaches exist and are easy to implement. However, existing approaches are only partial solutions to the visual hull estimation problem and do not address all the essential criteria in modeling: exactness, robustness and fastness. Our motivation is therefore to propose an exact approach which computes the visual hull polyhedron associated to a finite number of discrete silhouettes in a fast and robust way.

Silhouettes were first considered by Baumgart [1] who proposed to compute polyhedral shape approximations by intersecting silhouette cones. The term visual hull was later coined by Laurentini [9] to describe the maximal volume compatible with a set of silhouettes. Following Baumgart's work, a number of modeling approaches based on silhouettes have been proposed. They can be roughly separated into two categories : volume based approaches and surface based approaches.

The first category includes methods that approximate visual hulls by collections of elementary cells called voxels. An early approach in this category was proposed by Martin and Aggarwal [11] who used parallelepipedic cells aligned with the coordinate axis. Later on, octrees were proposed [4] as adaptive data structures for representing visual hulls and efficient approaches [16, 13, 3] were presented to compute voxel-based representations.

See [14] and [7] for reviews on volume based modeling approaches. All these approaches are based on regular voxel grids and can handle objects with complex topologies. However, the space discretizations used lead to approximations only, with a poor precision to complexity trade-off.

As shown in [10], the visual hull surface is a projective topological polyhedron made of curve edges and faces connecting them. In the case of piecewise-linear image contours, it becomes a regular polyhedron. The second category of approaches estimates elements of this polyhedron by intersecting silhouette cones. This includes several works which focus on individual points reconstructed using local second order surface approximations, see [5] for a review. Approaches have also been proposed to compute surface patches[15], or individual strips [12] of the visual hull. In the latter work, the computed strips are exact parts of the visual hull, however the approach duplicates cone intersection operations and requires an additional step to connect these different parts, with no topological guarantee. We will show in this paper that the complete visual hull polyhedron can be recovered as a whole with a reduced number of operations. Most of the mentioned surface based approaches suffer from numerical instabilities around frontier points as identified in [10]. Consequently, they often lead to surface models which are incomplete or corrupted, in particular when considering objects with complex topologies.

The method that we propose follows recent work [2] which separates the visual hull computation into two steps. A first step computes viewing edges of the visual hull and a second approximates its faces through a Delaunay triangulation. Our approach improves this scheme to produce, with less time complexity, the exact polyhedron that is silhouette consistent. To this aim, we replace the second step mentioned above with an algorithm which straightforwardly recovers mesh connectivity. Our main contribution with respect to all the mentioned approaches is to provide an algorithm which is exact given polygonal silhouettes and low in time complexity.

The paper is organized as follows. Section 2 introduces definitions and describes how viewing edges are computed. Section 3 discusses the local polyhedron orientation properties relevant to our task. Section 4 describes how these properties are used to follow the cone intersections and generate the visual hull mesh. Section 5 presents our results and future work.

2 Preliminaries

2.1 Definitions

Assume that a scene, composed of several objects, is observed by a set of pinhole cameras. The objects' surfaces are supposed to be orientable closed surfaces, smooth or polyhedral with possibly non-zero genus. *Rims* are locus of points, on the object surface, where viewing rays are tangent to the surface. Rims project onto image curves, called the *occluding contours* [11], which border the object silhouettes in the image plane. Occluding contours are oriented in the images. Their orientation is such that the object silhouette lies on the left of the oriented contour. Hence, exterior contours are oriented counterclockwise and interior contours are oriented clockwise. We will call the *inside region* of an occluding contour the closed region of the image plane delimited by the contour and containing the silhouette, and we will call the *outside region* its complement in the image plane. Note that in the following, we will consider that occluding contours are polygonal contours.

A *viewing cone* is a generalized cone in $\mathbb{R}^3$ whose apex is the image center and whose base is the inside region of an occluding contour. More formally, the *viewing cone* V associated with the occluding contour O is the closure of the set of rays passing through points inside O and through the camera center. V is thus tangent to the corresponding object surface along the rim that projects onto O. According to the orientation of O, exterior or interior, the viewing cone V is an acute or obtuse cone of $\mathbb{R}^3$ respectively. Viewing cone boundaries intersect along space curves which do not lie on the surface, except at *frontier points* where rims intersect. Note that in the case of polyhedral surfaces, frontier points are not necessarily isolated points and can form frontier edges.

We assume that using some standard background subtraction pre-process, images are transformed into sets of silhouettes which are themselves sets of polygonal contours. Recall the topological nature of the *visual hull* [10], a projective structure, with *frontier points* where rims intersect; *triple points* where three cones intersect; *cone intersection curves*; and *strips* corresponding to possibly unconnected contour contributions to the visual hull surface. Visual hulls are usually defined as the intersection of the viewing cones associated to all image contours, however such an intersection must be performed in full consistency with the silhouette information. To this purpose, two definitions of the visual hull VH can actually be considered [2]:

$$VH = \bigcap_{Images} \Big(\bigcup_{Silhouettes} \Big(\bigcap_{Contours} V \Big)\Big), \tag{1}$$

or:

$$VH^c = \bigcup_{Images} \Big(\bigcap_{Silhouettes} \Big(\bigcup_{Contours} D \setminus V \Big)\Big), \tag{2}$$

where VH^c is the visual hull complement in $\mathbb{R}^3$, D is the image visibility domain in $\mathbb{R}^3$ and $D \setminus V$ is the complement of V relative to this domain. Considering the visual hull or its complement is equivalent since the surface of interest borders both regions. The visual hull complement is in fact what is implicitly considered when carving voxels with volumetric methods. The first definition above is equivalent to the set of points in $\mathbb{R}^3$ that project inside one silhouette in every image while the second limits the projection constraint to the images where the points are visible. They differ by the fact that objects under consideration should be seen in every image with the first definition but not necessarily with the second. Both definitions may add independent virtual objects that do not appear in the original scene, but another difference is that the second definition may add more virtual objects as a consequence of the projection constraint relaxation. Note that polygonal contours such as those we take as input induce a polyhedral visual hull. Our algorithm computes exactly this polyhedron, which we will later refer to as being the *exact visual hull* in the context of piecewise linear silhouettes.

2.2 Computing viewing edges

Viewing edges are intervals along viewing lines. They correspond to viewing lines contributions to the visual hull surface and are thus associated to image points on occluding contours. We use this as the input of the second step in our algorithm. There are several advantages in doing so: computing such a set of edges has proven to be fast, simple and well-defined, and has already been used in different reconstruction applications [12, 2, 3]. Also, since edges computed by this method are already part of the visual hull polyhedron, it is a greedy initialization step. We recall in this section how to compute them efficiently given polygonal occluding contours.

For each occluding contours vertices in each image, we can compute its viewing line. The viewing edges associated to this viewing line are then defined by the intervals of points which satisfy any of the definitions introduced in 2.1. These intervals can be determined iteratively, using intersections with the silhouettes in every other image. At each iteration, intervals are updated according to their intersections with contributions from new contours or images. Such operation can be easily achieved in 2D by means of the epipolar geometry as shown in fig. 1. Importantly, for each generated intersections, we choose to record what edges in the images generated each intersection.

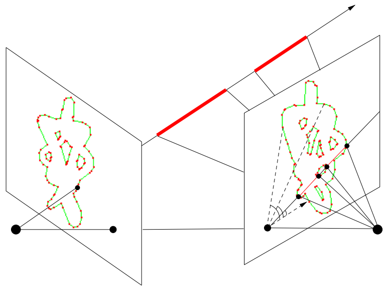


Figure 1: Viewing edges (in bold) along the viewing line. Epipolar line angles can be used to accelerate the search for the image segments intersecting the epipolar line.

Intervals along the viewing line are updated according to definition 1 or 2. A direct application of these definitions may not be straightforward since it requires contours to be grouped according to the silhouettes they belong to. A simpler solution takes advantage of the fact that interior contour contributions are unbounded along the viewing line. Thus, the union of an interior contour contribution with those of any disjoint exterior contour is equivalent to the identity for the former contribution. Following this principle, expressions 1 and 2 become:

$$VH = \bigcap_{Images} \left[\left(\bigcup_{Exteriors} V \right) \bigcap \left(\bigcap_{Interiors} V \right) \right], \tag{3}$$

and:

$$VH^c = \bigcup_{Images} \left[\left(\bigcap_{Exteriors} D \setminus V \right) \bigcup \left(\bigcup_{Interiors} D \setminus V \right) \right], \tag{4}$$

which require the contour orientations only. Note that in real situations, viewing cones are not necessarily tangent and hence, not all contour vertices give birth to viewing edges. The viewing edges computed during this initialization step form the initial seed for the subsequent steps in the algorithm.

2.3 Algorithm outline

The algorithm we propose consists in three main steps. The first step is to compute the viewing edges as explained above. However, this initial representation is incomplete. Typically, triple points, where three viewing cones intersect, project on each image contour at a point which is not necessarily one of the initial image contour vertices considered for viewing edge computation. As such, they are not part of the initial viewing edge vertices and need to be computed. Thus the goal of the second step is to recover all missing vertices and connections, so as to construct the complete oriented mesh describing the polyhedron. We have devised a scheme for following the cone intersection boundaries on the surface, using the geometry and orientation properties of the image contours discussed in section 3. With such properties, we can identify where connections are missing, and in what local direction to look for the next intersection boundary vertices. We can therefore iteratively generate the missing triple points, by detecting intersections with viewing cones, as described in section 4. As a third and final step, explained in section 4.3, the algorithm walks through the previously generated edge graph to identify each face.

3 Recovering the Local Orientation

3.1 Strip Orientation

The viewing edges we computed as a first step are a discrete representation of the visual hull strips (fig. 2a). We will not attempt to explicitly reconstruct frontier points or recover any topological features specific to theoretical visual hulls, as they do not extend to real visual hull surfaces as defined in section 2.1: because of inherently noisy calibrations and discretization steps, the perfect cone tangency that occurs at frontier points for theoretical visual hulls is lost. Note however that there still are regions of very close tangency, where both topology and orientation between strips may be complex and unstable.

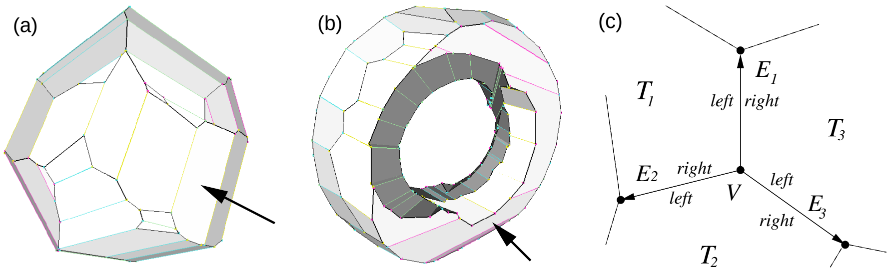


Figure 2: (a) Visual Hull of a sphere from 4 views as generated by our algorithm. Notice the strip structure and viewing edges, in light color. Symptoms of lost cone tangency: indicated strip covers all others, in a region where there should have been frontier points. (b) Visual hull of a torus from 4 images. Notice the branching of the indicated strip. (c) The relationship between vertex V's orientation (the ordering of pairs (E_i, T_i) around it) and orientation (left and right) of edges leaving from V on the visual hull polyhedron. Note that vertices are trivalent in the degenerate case, being the intersection of three planes.

Strips can have bifurcations (fig. 2b), and more generally can have several components, as soon as there are images with distinct silhouettes or hyperbolic contour portions. Nevertheless, each viewing edge we computed can be oriented with respect to the image it was backprojected from. We know that outer contours are oriented counter-clockwise and inner contours clockwise (see 2). For any contour vertex q in a given image, there are two edges e_1 and e_2 incident to q, respectively preceding and succeeding q according to the contour orientation. Note that any vertex q backprojects in space to a viewing line V_q. Similarly, any edge e of the contour backprojects into space to an infinite triangular planar patch T_e. The image contour orientation extends to these planar patches, and we can call *up* the direction on the strip induced by the positive direction of the corresponding image contour. Notice that with this definition of up and down on a strip, every viewing edge also inherits definitions for *front* and *back*, front always being the direction pointing toward the camera center. Moreover, since each viewing edge is defined by two points in space, these points can be labeled P_{front} and P_{back}. These image and strip orientability features are the foundations upon which we can build local surface orientation.

3.2 Polyhedron Point and Edge Orientation

In order to generate a consistent mesh, we must ensure that consistent orientation properties are propagated during processing. The initial stable and robust orientation property is the strip orientability described above. It is used as a basis to construct and propagate the local orientation for each primitive we will generate on the mesh. For oriented edges, this orientation information reduces to knowing what is left and what is right; for vertices, it reduces to knowing how incident edges are ordered around it.

Let us first consider an oriented edge of the visual hull polyhedron. Such an edge borders two visual hull surface regions, one locally on its left and the other locally on its right. These surface regions are a planar subset of the two corresponding triangular planar patches backprojecting from image edges, which we can label T_{left} and T_{right}. Furthermore, the edge itself represents the contribution segment to the visual hull of the line intersection between these two patches. Note that for the initial viewing edges, the (T_{left}, T_{right}) pair can be identified using the strip orientation properties (see 3.1): for example, if you consider the oriented edge $[P_{front}, P_{back}]$, what lies locally left of the edge is locally *up* on the strip. Reciprocally for oriented edge $[P_{back}, P_{front}]$, what lies locally left is locally *down* on the strip. Keeping track of this information for each oriented edge is also the key to maintain a consistent topological ordering between edges leaving from the same vertex, as illustrated in fig. 2c. If a vertex is visited coming from a given edge during a mesh walkthrough, it is then possible to query for existence of neighboring edges, and to make left or right turns at this vertex. We have therefore provided the necessary tools to generate and iterate over surface primitives.

4 Recovering the Missing Edges

We will now present the algorithm to follow cone intersection curves on the surface of the visual hull polyhedron, and generate the complete mesh of the visual hull as output. Existing surfacic approaches [12] have focused on the full edge plane intersections with the visual hull, which computes these curves as a side effect. However this requires numerous polygon intersection and transformation operations. Most edges and points of the intermediate polygons do not contribute to the final visual hull face. Following the intersection curves ensures that we will always focus on the relevant surface primitives, with an immediate complexity gain. Furthermore, all intersection operations in our algorithm resolve in the 2D image planes with one-parameter unknown computations.

Cone intersection curves materialize as a full graph of k triple points joining several viewing edge vertices, as illustrated in fig. 3a. We have developed a simple scheme to recover this graph, with two substeps. First, whenever a missing edge connection is identified on a polyhedron point, we must determine a direction in which the edge is leaving (see 4.1). Then, we use the direction's projection in images to detect triple points, and to identify which vertex this edge leads to (section 4.2).

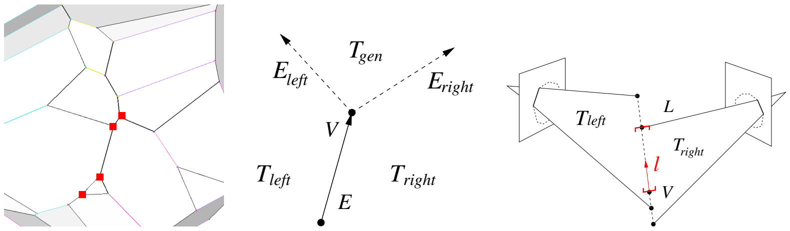
4.1 Recovering the Direction of Missing Edges

Given any vertex V on the polyhedron, the vertex has three edge connections leaving from it, in the non-degenerate case. Let us focus (fig. 3b) on a particular type of vertex V found on the incomplete polyhedron surface, used as a *search seed*. Such a vertex already has one connected edge E, but two of its edges E_{left} and E_{right} are *potentially* still missing. Notice that E is considered oriented *toward* V; it is the direction of the walkthrough. E is fully known: we know what infinite backprojected planar patches T_{left} and T_{right} lie locally left and right of E. We also know what third planar patch T_{gen} generated the vertex V, through intersection with the two patches above. Notice that the initial viewing edges fit exactly this description, each of their two vertices being an initial search seed.

Now, we must recover the missing edge connections E_{left} and E_{right}. Since the search might have been launched from other connected branches of the graph, these edges might already exist. However, when missing, we must determine what vertex the connection leads to, in order to provide a complete edge description. Its nature (triple point or viewing edge vertex) and position must be determined. Solving both of these problems requires knowing in what 3D direction leaving from V this vertex lies, e.g. the *search half-line*. For example, if E_{left} is missing, we compute the half line as the intersection of planar patches T_{left} and T_{gen}, oriented left of E. Reciprocally we use T_{gen} and T_{right} if E_{right} is missing. We then apply the scheme described in the next section to each missing edge.

4.2 Identifying Missing Incident Vertices

We use an intersection search scheme to identify the vertex each missing edge leads to. The context of this search is illustrated in fig. 3c. Note that for both possible missing edges the local (T_{left}, T_{right}) pair is known: in fig. 3b, this pair corresponds to (T_{left}, T_{gen}) for missing edge E_{left}, and (T_{gen}, T_{right}) for E_{right}. For a given missing edge, consider the segment intersection between its two local T_{left} and T_{right} patches. The wanted edge is a subset of this segment, because this segment represents the contribution to the visual hull of two images only: the ones T_{left} and T_{right} backproject from. Use of more images reduces this contribution because of successive intersections. Nevertheless our two initial patches define natural search bounds, the brackets in fig. 3c. The lower bound is trivially given by the position of V, current search seed, and the upper bound by L, the most restrictive of the four viewing lines bordering patches T_{left} and T_{right}.



(a) (b) (c)

Figure 3: (a) Close-up view of a sphere's visual hull. Black edges are the discrete cone intersection curves, squares are triple points. (b) Spawning search directions from a given initial situation on the visual hull surface. (c) Once the search direction l is known for an identified missing connection, the knowledge of the local T_{left} and T_{right} infinite triangular patches results in natural search bounds for the incident vertex along this direction.

It is possible that no other image viewing cone intersects this segment, when its projection lies inside all other silhouettes. In this case these initial bounds exactly describe the wanted edge. Note that the incident vertex then lies on viewing line L, which defined the upper bound. Therefore, it has already been computed in the viewing edges stage, and its instance can be retrieved once identified among all viewing edge points of L. Possibly however, the viewing cone of a third image does intersect this segment, hence creating a triple point. Which is why every third possible image must be considered, in order to iteratively compute the most restrictive upper bound. We therefore reproject the search half-line in each possible third image, and compute the intersection with the silhouette, so as to update the bound. The nature of the incident vertex, viewing edge vertex or triple point, is known only after this process. Indexing triple points from the three planar patches which define it ensures that they are computed only once. Once identified, a newly generated triple point serves as a new search seed for the mesh generation; the process described in 4.1 is recursively applied to it. A search branch is completed once it meets an already computed vertex. We have therefore defined how to follow and generate the cone intersection segments starting with the initial viewing edge structure. The output is the complete polyhedron mesh, such that no polyhedron vertex is computed twice.

4.3 Identifying Faces of the Polyhedron

With the oriented edge computed in step two, we can now apply the third and final step: retrieving face information of the polyhedron. Note that each 2D edge in the initial images potentially contributes to the visual hull polyhedron surface, its contribution lying in its backprojected edge plane. In fact, this contribution is a single general planar polygon, with possibly several inside and outside contours, as noticed in [12]. With the vertex properties described in 3.2, graph walkthroughs can be constrained to follow a given backprojected edge plane and keep it locally left, by taking a left turn at each vertex. We also constrain the traversal of each polyhedron edge, in order to ensure that it will be walked through exactly twice, once in each direction. Hence an $O(v)$ walkthrough (with v the number of vertices in the final polyhedron) enables us to recover the entire contour set associated to every planar polygon previously mentioned. This is also an advantage of our method, the output polygons being described in their most general and concise form. Retrieving triangles for a specific application can be done efficiently by applying existing $O(v \log v)$ planar tessellation algorithms. Given the completeness of the mesh generated in step two, this constrained walkthrough ensures that our final polyhedron satisfies the manifold and watertight properties.

5 Results and Future Work

We have experimented with a preliminary implementation of our algorithm, on a variety of synthetic and real input sets for validation. Contours in each view are discretized to be used as input for our algorithm, using an optimal segment recognition algorithm [6]. This guarantees that the generated piecewise linear contours describe exactly the boundary pixels separating the background and foreground regions. Given that our algorithm computes the exact polyhedron associated to a given input contour set, the generated polyhedron is the best geometric information we can obtain after the background subtraction step.

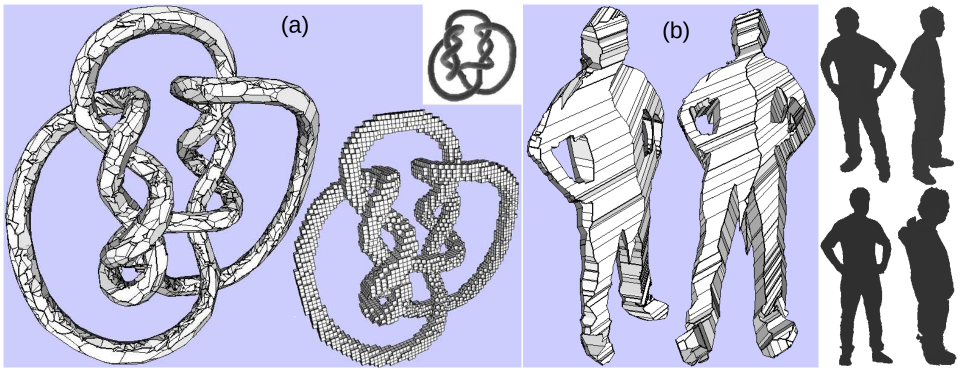


Figure 4: (a) Visual hull as obtained by our method, of the knots object taken from Hoppe's web site [8], from 42 viewpoints surrounding the object. Reconstruction of its 11146 points and 16719 edges took 12.8sec on a 1.8GHz PC. Discretization of silhouettes resulted in 200 contour points per image on average. Its voxel-based counterpart shows much less precision under the same conditions, using a well fitted $64^3 = 262144$ voxel grid. Original model is shown top right. (b) Two renderings of the visual hull of a human shape using 4 acquired 640x480 silhouettes (shown right). 2316 points, 2356 edges, computed in 142msec, with 250 contour points per image silhouettes.

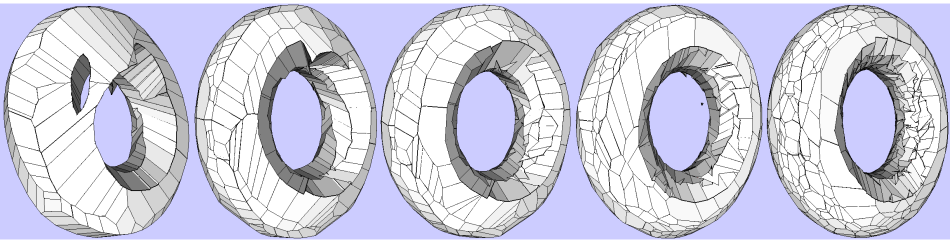
A first experiment places our algorithm in a complex topological situation (fig. 4a), while still yielding artifact-free results. A second experiment involves real image data from four cameras in a virtual reality studio. It results in a correct model, usable for such applications as scene relighting. We have also provided several visual hulls of a torus (fig. 5), which illustrate the impact of the number of views on model complexity. Computation times for this non-optimized version are of the order of 100ms for a dozen viewpoints, making it suitable for real-time applications. Implementation speedups are still possible. We will explore complexity reduction through simplification of the input contours. We will use this algorithm for real-time human modeling from video flows in virtual studios. Also, we will provide complexity estimations in the near future. Arguably this algorithm could be optimal for the visual hull polyhedron computation problem. Clearly it surpasses existing visual hull surface reconstruction algorithms: each operation in the algorithm is a greedy one, each computed primitive is known beforehand to be part of the visual hull, and uses the minimal necessary information to be determined, namely reprojection and intersection in images. Experimentally, the execution time and number of operations of this approach rivals with the voxel-based approaches, while attaining geometrical exactness hardly accessible to those methods.

References

[1] B.G. Baumgart. A polyhedron representation for computer vision. In *AFIPS National Computer Conference*, 1975.

[2] E. Boyer and J.S. franco. A hybrid approach for computing visual hulls of complex objects. In *CVPR'03*, volume I, pages 695–701, 2003.

[3] G. Cheung, S. Baker, and T. Kanade. Visual hull alignment and refinement across time: a 3d reconstruction algorithm combining shape-from-silhouette with stereo. In *CVPR'03*, volume II, pages 375–382, 2003.



4 / 714pts,19ms *8 / 1090pts, 75ms* *12 / 1410pts,125ms* *16 / 1656pts,217ms* *42 / 3126pts,1.44s*

Figure 5: Visual hulls of a torus under an increasing number of viewpoints, with 60 contour points per silhouette. Our method accounts for all available geometric information, with an increasingly complex and precise model. Very high numbers of viewpoints don't offer a good complexity/precision trade-off, since the visual hull inherently becomes more sensitive to image calibration and discretization noise. 10 to 20 viewpoints offer very decent results.

[4] C.H. Chien and J.K. Aggarwal. Volume/surface octress for the representation of three-dimensional objects. *Computer Vision, Graphics and Image Processing*, 36(1):100–113, 1986.

[5] R. Cipolla and P.J. Giblin. *Visual Motion of Curves and Surfaces*. Cambridge University Press, 1999.

[6] I. Debled-Rennesson and J.P. Reveillès. A linear algorithm for segmentation of digital curves. *International Journal of Pattern Recognition and Artificial Intelligence*, 9(4):635–662, 1995.

[7] C.R. Dyer. Volumetric Scene Reconstruction from Multiple Views. In L.S. Davis, editor, *Foundations of Image Understanding*, pages 469–489. Kluwer, Boston, 2001.

[8] H.Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Surface reconstruction from unorganized points. In *SIGGRAPH'92*, volume 26(2), pages 71–78, 1992.

[9] A. Laurentini. The Visual Hull Concept for Silhouette-Based Image Understanding. *IEEE Transactions on PAMI*, 16(2):150–162, February 1994.

[10] S. Lazebnik, E. Boyer, and J. Ponce. On How to Compute Exact Visual Hulls of Object Bounded by Smooth Surfaces. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, Kauai, (USA)*, volume I, pages 156–161, December 2001.

[11] W.N. Martin and J.K. Aggarwal. Volumetric description of objects from multiple views. *IEEE Transactions on PAMI*, 5(2):150–158, 1983.

[12] W. Matusik, C. Buehler, and L. McMillan. Polyhedral Visual Hulls for Real-Time Rendering. In *Eurographics Workshop on Rendering*, 2001.

[13] W. Niem. Automatic Modelling of 3D Natural Objects from Multiple Views. In *European Workshop on Combined Real and Synthetic Image Processing for Broadcast and Video Production, Hamburg, Germany*, 1994.

[14] G. Slabaugh, B. Culbertson, T. Malzbender, and R. Schafe. A Survey of Methods for Volumetric Scene Reconstruction from Photographs. In *International Workshop on Volume Graphics*, 2001.

[15] S. Sullivan and J. Ponce. Automatic Model Construction, Pose Estimation, and Object Recognition from Photographs using Triangular Splines. *IEEE Transactions on PAMI*, pages 1091–1096, 1998.

[16] R. Szeliski. Rapid Octree Construction from Image Sequences. *CVGIP*, 58(1):23–32, 1993.