An Efficient Visual Hull Computation Algorithm

Wojciech Matusik
Chris Buehler
Leonard McMillan
Laboratory for Computer Science
Massachusetts institute of Technology
(wojciech, cbuehler, mcmillan)@graphics.lcs.mit.edu

Steven J. Gortler
Division of Engineering and Applied Sciences
Harvard University
sjg@cs.harvard.edu

ABSTRACT
In this paper we describe an efficient algorithm for computing the visual hull of an object. This problem is equivalent to computing the intersection of generalized cones. The naïve visual hull computation algorithm requires intersecting 3D polyhedra. We exploit the special structure of generalized cone polyhedra and show how to reduce this computation to a set of intersections in 2D. Moreover, we describe how the 2D intersections can be carried out efficiently.

Keywords
Visual Hulls, Line Hulls, Efficient Algorithms

1. INTRODUCTION
Many researchers [7,10] have used silhouette information to distinguish regions of 3D space where an object is and is not present. Suppose that some original 3D object is viewed from a set of reference views R. Each reference view $r \in R$ has the silhouette contour s_r whose interior is covered by the object. For view r, one creates the cone-like volume vh_r defined by all the rays starting at the image's center of projection p_r and passing through the interior of the silhouette. It is guaranteed that the actual object must be contained in vh_r. This statement is true for all r; thus, the object must be contained in the volume $vh_R = \cap_{r \in R} vh_r$. As the size of R goes to infinity, and includes all possible views, vh_R converges to a shape known as the visual hull vh_∞ of the original geometry [5]. The visual hull is also commonly known as the line hull.

The visual hull is not guaranteed to be the same as the original object since concave surface regions can never be distinguished using silhouette information alone. Moreover, in practice, one must construct approximate visual hulls using only a finite number of views. Given the set of views R, we would like to efficiently compute vh_R – the intersection of all cones defined by all silhouettes in R. Computing visual hulls quickly has many potential applications including structure-from-silhouettes, gesture recognition, 3D photography, and real-time geometry capture.

2. PREVIOUS WORK
Typically visual hulls have been computed using volume carving methods. These methods remove unoccupied regions from an explicit volumetric representation. All voxels falling outside of the silhouette view are eliminated from the volume. This process is repeated for each reference image. The resulting volume is a quantized representation of the visual hull according to the given volumetric grid.

There has been considerable work [4,8] on Boolean operations on the 3D polyhedra. Most the algorithms require decomposing the input polyhedra to convex polyhedra. Then, the operations are carried out on the convex polyhedra.

It has been also shown that the exact sampling of the visual hull (i.e., an image of the visual hull) from arbitrary views can be computed efficiently at interactive frame rates [6]. However, this algorithm does not build the explicit polyhedral representation. Our algorithm is similar to that in [6], except that we achieve comparable speed while computing an explicit polyhedral model, which is more useful in some applications.

The work in [9] also computes an explicit polyhedral model of the visual hull. They use a slightly different 2D reduction of the problem, which appears to result in lower performance.

3. Visual Hull Computation Algorithm

3.1 Inputs
For simplicity, we assume that each silhouette is specified by a set of convex or non-convex 2D polygons. These polygons can have holes. Each polygon consists of a set of consecutive vertices in its contours. We also assume that the degree of each vertex is equal to two. Moreover, for each silhouette we know the location of its image plane and the location of the center of projection – the apex of the each cone. The image plane is the 2D plane that contains the silhouette polygons.

Throughout this paper we use the following notations: we let k be the number of input silhouettes; we let n be the number of edges in each silhouette; and we let l be the maximum number of intersections of a projected line with a silhouette.

3.2 Algorithm Outline
In order to compute the visual hull with respect to the input silhouettes we need to compute the intersection of the cones defined by the input silhouettes. The resulting polyhedron is described by all its faces. The faces of this polyhedron can only lie on the faces of the original cones. The faces of the original cones are defined by the center of projections and the edges in the input silhouettes. The naïve algorithm for computing the visual hull would do the following: For each input silhouette s_i and for each edge e in the input silhouette s_i we compute the face of the cone. Then we intersect this face with the cones of all other input silhouettes. The result of these intersections is a set of polygons

that define the surface of the visual hull. The pseudocode for the algorithm is given below.

```
VHISECT(Input Silhouettes S)
(1)  PolygonSet VHFaces : = ∅
(2)  for each input silhouette s_i in S
(3)    for each edge e in silhouette s_i
(4)      for each reference silhouette s_j in S\{s_i}
(5)        PolygonSet faces_j := ConeFace(e) ∩ Cone(s_j)
(6)      PolygonSet EdgeFaces := ∩ (faces_m, m = 1..k, m≠i)
(7)      VHFaces :=  VHFaces ∪ EdgeFaces
```

3.3 Reduction to 2D Intersections

The intersection of a face of a cone with other cones is a 3D operation (these are polygon-polyhedron intersections). It was observed by [3,6,9] that these intersections can be reduced to simpler intersections in 2D. This is because each of the silhouette cones has a fixed scaled cross-section – it is defined by the 2D silhouette. Reduction to 2D also allows for less complex 2D data structures to accelerate the intersections.

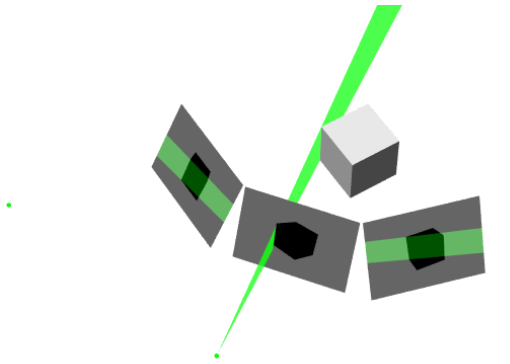


Figure 1: One face of the center cone is projected onto the image planes of two other silhouettes.

To compute the intersection of a face f of a cone $cone(s_i)$ with a cone $cone(s_j)$ we project f onto the image plane of silhouette s_j (see Figure 1). Then we compute the intersection of projected face f with silhouette s_j. Finally, we project back the resulting intersection onto the plane of face f. The pseudocode for the algorithm is given below.

```
VHISECT(Input Silhouettes S)
(1)  PolygonSet VHFaces : = ∅
(2)  for each input silhouette s_i in S
(3)    for each edge e in silhouette s_i
(4)      for each reference silhouette s_j in S\{s_i}
(5)        Polygon p := project ConeFace(e)onto s_j
(6)        PolygonSet ps := p ∩ s_j
(5)        PolygonSet faces_j := project ps onto ConeFace(e)
(6)      PolygonSet EdgeFaces := ∩ (faces_m, m = 1..k, m≠i)
(7)      VHFaces :=  VHFaces ∪ EdgeFaces
```

3.4 Efficient Projected Cone – Silhouette Intersection

In this section we show how to compute the intersection of the projected cone c_i with the silhouette s_j of some other cone c_j.

3.4.1 Construction of the Edge-Bins

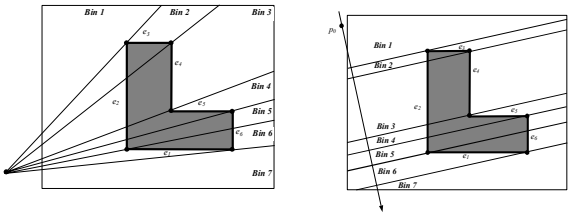
In order to perform the intersections efficiently we use an Edge-Bin data structure. First, we observe that in case of perspective projection all rays on the surface of the cone c_i project to a pencil of lines sharing a common point p_0 in the image plane of s_j. We can parameterize all projected lines based on the slope α that these lines make with some reference line. Given this parameterization we partition the domain of $\alpha = (-\infty, \infty)$ into ranges such that any projected line with the slope falling inside of the given range always intersects the same set of edges of the silhouette s_j. We define a bin b_i to be a three-tuple: the start α_{start}, the end α_{end} of the range, and a corresponding set of edges S_i, $b_i=(\alpha_{start}, \alpha_{end}, S_i)$. We note that each silhouette vertex corresponds to a line that defines a range boundary.

In certain configurations, all rays project to a set of parallel lines in the image plane of s_j. When this case occurs, we use a line $p(s)= p_0+d\alpha$ to parameterize the lines, where p_0 is some arbitrary point on the line p and d is a vector perpendicular to the direction of the projected rays. To define bins, we use the values of the parameter α at the intersection points of the line p with the lines in the direction of the lines passing through silhouette vertices. In this way we can describe the boundary of the bin using two values α_{start} and α_{end}, where α_{start}, α_{end} are the values of α for lines passing through two silhouette vertices that define the region.

The edge-bin construction involves two steps. First, we sort the silhouette vertices based on the value of the parameter α. The lines that pass through the silhouette vertices define the bin boundaries. This step has a bound of $O(n \log n)$.

Next, we observe that two consecutive slopes in the sorted list define α_{start} and α_{end} for each bin. To compute a set of edges assigned to each bin we traverse the sorted list of silhouette vertices. At the same time we keep the list of edges in the current bin. When we visit a vertex of the silhouette we remove from the current bin an edge that ends at this vertex and we add an edge that starts at the vertex. A start of an edge is defined as the edge endpoint that has a smaller value of parameter α. In Figure 2 we show a simple silhouette, bins, and corresponding edges for each bin. The running time of the above algorithm is $O(nl)$ – proportional to the total number of edges in all bins.

The edges in each bin need to be sorted based on the increasing distance from the point p_0 (or the distance from parameterization line $p(s)$ in case of the parallel lines). The efficient algorithm first performs a partial ordering on all the edges in the silhouette such that the edges closer to the point p_0 are first in the list. Then, when the bins are constructed the edges are inserted in the bins in the correct order. The time to construct the bins is $O(n \log n + nl)$ for one silhouette.



Bin	Edges
1	∅
2	e_2, e_3
3	e_2, e_4
4	e_2, e_5
5	e_2, e_6
6	e_1, e_6
7	∅

Figure 2: Edge-Bins and corresponding edges.

3.4.2 Efficient Intersection of the Projected Cone Faces with a Silhouette

Using the edge bin data structure we can compute efficiently the intersection of the projected cone c_i with the silhouette s_j of some other cone c_j. In order compute the intersection we process the consecutive faces of cone c_i. We start by projecting the face f_1 onto the plane of silhouette s_j. The projected face f_1 is defined by its boundary lines with the values α_{p1} , α_{p2}. First, we need to find a bin $b = \{\alpha_{start}, \alpha_{end}, S\}$ such that $\alpha_{p1} \in (\alpha_{start}, \alpha_{end})$. Then, we intersect the line α_{p1} with all the edges in S. Since the edges in S are sorted based on the increasing distance from the projected vertex of cone c_i (or distance from line $p(s)$ in case of parallel lines) we can immediately compute the edges of the resulting intersection that lie on line α_{p1}. Next, we traverse the bins in the direction of the value α_{p2}. As we move across the bins we build the intersection polygons by adding the vertices that define the bins. When we get to the bin $b' = \{\alpha'_{start}, \alpha'_{end}, S'\}$ such that $\alpha_{p2} \in (\alpha'_{start}, \alpha'_{end})$ we intersect the line α_{p2} with all edges in S' and compute the remaining edges of the resulting polygons. It is important to note that the next projected face f_2 is defined by the boundary lines α_{p2}, α_{p3}. Therefore, we do not have to search for the bin α_{p2} falls into. In this manner we compute the intersection of all projected faces of cone c_i with the silhouette s_j. The running time of this intersection operation is optimal since it is $O(m)$, where m is the number of vertices of the resulting cone intersection. Therefore, the total running time for intersecting two silhouette cones is $O(n \log n+nl +m)$, or $O(n \log n + nl)$ since m is bounded by nl.

3.5 Calculating Visual Hull Faces

In the previous section we described how to perform the intersection of two cones efficiently. Performing the pairwise intersection on all pairs of cones results in *k-1* polygon sets for each face of each cone. The faces of the visual hull are the intersections of these polygon sets at each cone face. We perform the intersection of these polygon sets using standard algorithms for Boolean operations [1,2]. It is important to note that the polygons in these sets are possibly non-convex, have holes, and have no high-degree vertices. The time to intersect two polygon sets each with q vertices is $O((q+a) \log q)$, where a is the number of vertices in the resulting intersection. In our case we have *k-1* polygon sets for each cone face that we need to intersect. The intersection of the *k-1* polygons each with q vertices can be implemented in $O((kq +b)\log kq)$ where b is the number of all intersections of polygon edges.

Our resulting representation includes redundant copies of each vertex in the resulting polyhedron (in fact, the number of copies of each vertex is equal to the degree of the vertex divided by 2). To optionally eliminate the redundant copies, we simply merge identical vertices. This allows us to obtain a watertight model.

3.6 Complexity Analysis

We analyze the time complexity in terms of l. Recall that l is defined to be the maximum number of intersections of a projected line with a silhouette. Note that l is always less than n, and in practice l is generally much smaller than n.

In Section 3.4, it is shown that the complexity of intersecting two cones is $O(n \log n+nl)$. Thus, for all pairwise intersections between cones we have $O(k^2(n \log n + nl))$.

The time to intersect *k-1* polygons sets, each of them with at most n vertices, is $O((kn+b) \log kn)$, where b is the number of vertices in the intersection. In the worst case, each of the k silhouettes has at most l faces with n vertices in each of the *k-1* polygon sets. This is because the number of vertices in the intersection of two cones is bounded by $O(nl)$. Therefore, the part of the algorithm described in Section 3.5 takes $O(kl(kn+b) \log kn)$.

Combining the analyses of Sections 3.4 and 3.5, the complete time complexity is $O(k^2(n \log n + nl) + kl(kn+b) \log kn)$.

4. Implementation and Results

We have implemented and tested the algorithm on a variety of both synthetic and real silhouettes of objects. The silhouettes of the real objects are obtained using a system composed of a digital camera and a rotating platform. We can vary the complexity of the silhouettes by arbitrarily approximating their contours. The sample results of the visual hulls are shown in Figure 3. Table 1 shows the running time of the algorithm depending on the number of input silhouettes. Table 2 shows the running time of the algorithm depending on the number of edges in the input silhouettes. All running times are measured on a 1GHz Pentium III machine with 1GB of RAM.

These running times illustrate the performance of the algorithm in the common case, when l is small relative to n. We can see that the running times are nearly quadratic in the number of silhouettes and nearly linear in the number of edges.

Table 1. Running Time vs Number of Cones (100 edges per silhouette)

Number of Cones	Running Time (s)
2	**0.015**
3	**0.031**
4	**0.047**
5	**0.078**
6	**0.125**
8	**0.235**
10	**0.375**

continued from last page	
12	**0.563**
16	**0.984**
20	**1.547**
24	**2.187**
28	**2.953**
32	**3.828**
36	**4.797**

Table 2. Running Time vs Edges in Silhouettes (8 cones per visual hull)

Edges per Silhouette	Running Time (s)
9	**0.031**
15	**0.047**
23	**0.078**
35	**0.094**
41	**0.109**
55	**0.141**
67	**0.156**
74	**0.174**
86	**0.203**
99	**0.234**
113	**0.265**
135	**0.313**
173	**0.421**
241	**0.609**
319	**0.860**
459	**1.344**
590	**1.797**
641	**2.000**

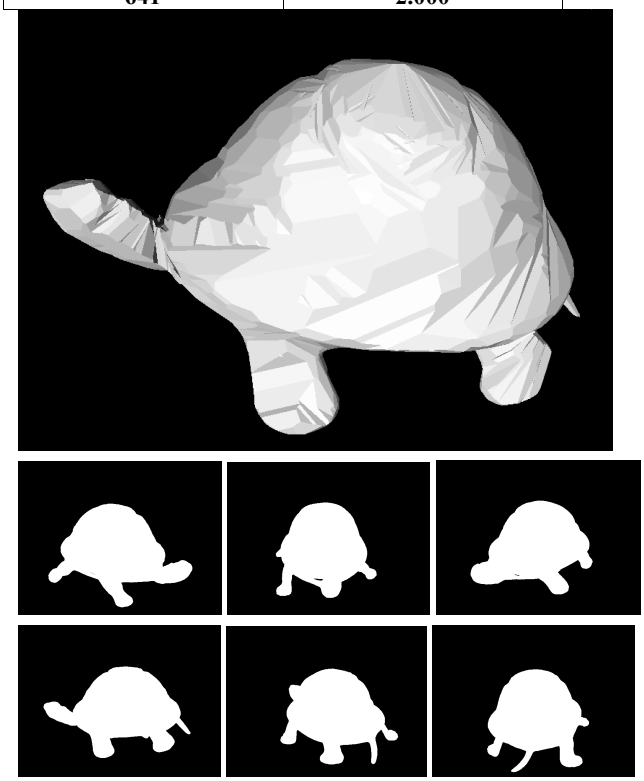


Figure 3: Shaded image of the visual hull and 6 out of 26 input silhouettes that generated it.

5. Conclusion

In this paper we have presented an efficient algorithm for constructing polyhedral representation of the visual hull. To our knowledge, this is the first algorithm that is capable of computing the visual hull polyhedral representation in real time or at least at interactive frame rates. We believe it is well suited for a variety applications especially those in computer graphics and computer vision.

REFERENCES

[1] Balaban, I. J., "An optimal algorithm for finding segments intersections." *Proc. 11th Annual ACM Sympos. on Computational geometry*,1995, p. 211-219.

[2] Bentley, J., Ottmann, T., "Algorithms for Reporting and Counting Geometric Intersections." *IEEE Trans. Comput*. C-28, 9 (Sept. 1979), pp. 643-647

[3] Buehler, C., Matusik, W., Gortler, S.J., and McMillan, L., "Creating and Rendering Image-Based Visual Hulls." *Technical Report 780, Laboratory for Computer Science*, Massachusetts Institute of Technology, June 1999.

[4] Chazelle, B., "An optimal Algorithm for Intersecting Three-Dimensional Convex Polyhedra" , *SIAM J. Computing*, 21 (1992), 671-696

[5] Laurentini, A. "The Visual Hull Concept for Silhouette Based Image Understanding." *IEEE PAMI* 16,2 (1994), 150-162.

[6] Matusik, W., C. Buehler, R. Raskar, S. Gortler, and L. McMillan. "Image-Based Visual Hulls" *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, July 2000, pp. 369-374.

[7] Potmesil, M. "Generating Octree Models of 3D Objects from their Silhouettes in a Sequence of Images." *CVGIP* 40 (1987), 1-29.

[8] Rappoport, A., and S. Spitz. "Interactive Boolean Operations for Conceptual Design of 3D solids." *SIGGRAPH 97*, 269-278.

[9] Rozenoer, Max & Shlyakhter, Ilya "Reconstruction of 3D Tree Models from Instrumented Photographs." Master of Engineering Thesis, Massachusetts Institute of Technology, 1999.

[10] Szeliski, R. "Rapid Octree Construction from Image Sequences." *CVGIP: Image Understanding* 58, 1 (July 1993), 23-32.