

Object Space Silhouette Algorithms

Ashley Hartner
University of Utah

Mark Hartner
University of Utah

Elaine Cohen
University of Utah

Bruce Gooch
University of Utah

Abstract

In computer graphics, silhouette extraction and rendering has a critical role in a growing number of applications. This paper examines five object space silhouette extraction algorithms for polygonal models. The algorithms are applied to a variety of models and compared in terms of code complexity and run time performance. The purpose of this paper is to inform programmers who must choose from among these five algorithms.

Note: The computer code generated for this project is available online at: <http://cs.utah.edu/~hartner>

1 Introduction

Silhouette drawings are a simple form of line art used in cartoons, technical illustrations, architectural design and medical atlases. Silhouette curves of a polygonal model are useful in realistic rendering, in interactive techniques, and in non-photorealistic rendering (NPR).

In realistic rendering silhouettes are used to simplify shadow calculation. Sander et al. demonstrate that complex models can be rendered at interactive rates by clipping the polygons of a coarse geometric approximation of a model along the silhouette of the original model [Sander et al. 2000]. Hertzmann and Zorin have shown that silhouettes can be used as an efficient means to calculate shadow volumes [Hertzmann and Zorin 2000]. Haines demonstrates an algorithm using silhouettes for rapidly rendering soft shadows on a plane [Haines 2001]. Silhouettes are used for interactive haptic rendering [Johnson and Cohen 2001]. Some authors, [Jensen et al. 2002; Chung et al. 1998] have described the use of silhouettes in CAD/CAM applications. Systems have also been built which use silhouettes to aid in modeling and motion capture tasks [Fua et al. 1999; Lee et al. 2000; Bottino and Laurentini 2001].

In NPR, complex models and scenes are often rendered as line drawings using silhouette curves. Lake et al. present interactive methods to emulate cartoons and pencil sketching [Lake et al. 2000]. Gooch et al. built a system to interactively display technical drawings [Gooch et al. 1999]. Rheingans and Ebert and Lum and Ma have built a NPR volume visualization system which uses silhouettes to emphasize key data in volume renderings [Rheingans and Ebert 2001; Lum and Ma 2002].

The silhouette set of a polygonal model can be computed in object space or in screen space. Object space algorithms require computations in three dimensions and produce a list of silhouette edges for a given viewpoint. Screen space algorithms are usually based on 2D image processing techniques and are useful if rendering silhouettes is the only goal of the algorithm. This paper examines five software-based object space algorithms in terms of runtime speed, code complexity, pre-process timing and complexity, scalability, and memory usage.

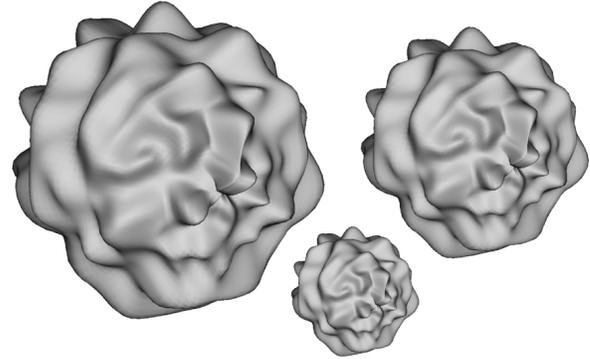


Figure 1: Silhouette algorithms are evaluated using various tessellations of an irregular sphere model. This model was chosen because; it has a high silhouette complexity, it contains interior silhouettes for every view angle, and it contains regions with similar normals which are not spatially close, these properties make this model a difficult case for all of the evaluated algorithms.

1. Brute Force – Iterate through each edge in a polygonal model and test whether it is a silhouette edge.
2. Edge Buffer – Using the “Edge Buffer” data structure of Buchanan and Sousa [Buchanan and Sousa 2000] to iterate over facets instead of edges.
3. Probabilistic – An edge tracing method that chooses a finite number of “seed” edges of each viewpoint based on a measure of the likelihood that the “seed” edges are silhouettes [Markosian et al. 1997].
4. Gauss Map Arc Hierarchy – The angles of arcs between front and back facing polygons are stored in a tree structure [Gooch et al. 1999; Benichou and Elber 1999].
5. Normal Cone Hierarchy – Polygon normals are grouped into cones and these cones are stored in a tree structure [Sander et al. 2000; Johnson and Cohen 2001; Hertzmann and Zorin 2000; Pop et al. 2001].

2 Definition of a Silhouette

Given $E(u, v)$ as the eye vector, a point on a surface $\sigma(u, v)$ with surface normal $N(u, v)$ is a silhouette point if $E(u, v) \cdot N(u, v) = 0$, that is, the angle between $E(u, v)$ and $N(u, v)$ is 90 degrees. This relationship is demonstrated in Figure 3. This definition includes internal silhouettes as well as the object’s outline, or halo. It is important to note that the silhouette set of an object is view dependent, that is the edges of a model that are silhouettes change based on the point from which the object is viewed.

Additional important feature lines exist for three dimensional models. These lines include; texture boundaries, creases, and object boundaries. These additional feature lines are view independent,

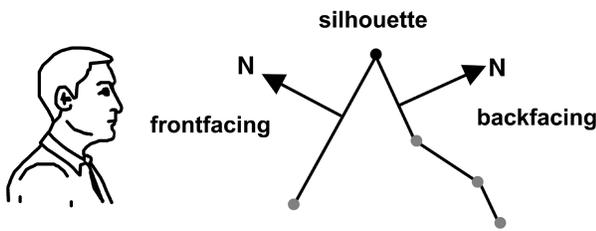


Figure 2: Silhouettes on a polygonal surface.

and can therefore be completely specified prior to runtime. In this work we evaluate only runtime silhouette extraction algorithms.

3 Silhouettes for Polygonal Models

The silhouette set for a polygonal model is defined to be all edges in the model which are shared by both a front-facing and a back-facing polygon, as illustrated in Figure 2.

For uniformity throughout this work it is assumed that polygon normals point outward from surfaces. This assumption yields the following:

if $N \cdot E < 0$ then the polygon is front-facing

if $N \cdot E > 0$ then the polygon is back-facing

if $N \cdot E = 0$ then the polygon is perpendicular to the view direction

4 Algorithms

In order to make a fair comparison of the various algorithms at runtime a test suite of polygonal models was built using multiple tessellations of an modified sphere model as shown in Figure 1. This model was chosen because; it has a high silhouette complexity, it contains interior silhouettes for every view angle, and the model contains regions with similar normals which are not spacially close to each other. These properties make the irregular sphere model a complex case for each of the evaluated silhouette algorithms.

The five object space silhouette algorithms were implemented using C++ and the Standard Template Library (STL). Each silhouette method consists of a pre-process routine which is executed only once per model, and a runtime routine which is executed every time a frame is rendered. The test program choose an algorithm at random and a model tessellation at random, then rotated about a number of arbitrary axis alignments with an angular velocity such that the model and silhouette lines can be rendered without a noticeable skip in the rotation of the model. The measure of runtime speed reported here includes silhouette extraction time only. The time required to render the model and silhouette lines is not included in this measure. The observed runtime speed is reported in frames per second and is calculated by dividing 1080 frames by the number of seconds needed to calculate those frames. The runtime tests were performed on an AMD Athlon 1.4Ghz machine with 512MB of RAM and a GeForce3 (64MB) graphics card running Linux 2.4.18. For each method we present:

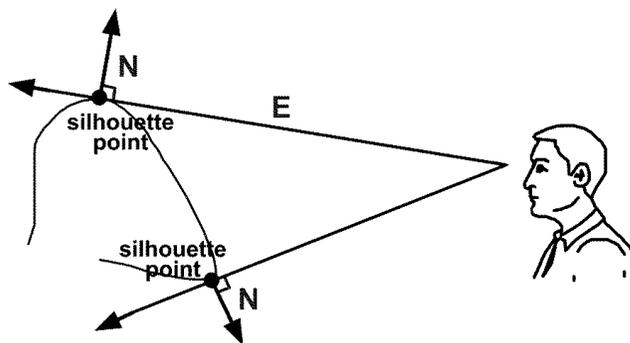


Figure 3: Silhouettes on a smooth surface.

1. Method – A summary of the silhouette extraction algorithm.
2. Pre-process – A summary of steps performed prior to runtime.
3. Runtime – The frames per second for test suite of polygonal irregular sphere models.
4. Code Complexity – We report two measures of code complexity. One is the amount of time needed for an advanced undergraduate student to write and debug the code. The second is the number of lines of code in both the pre-process and the runtime routine.
5. Results and Observations

4.1 Brute Force

Method

The brute force method of silhouette extraction requires testing each edge in the polygonal mesh sequentially to verify whether or not it is a silhouette.

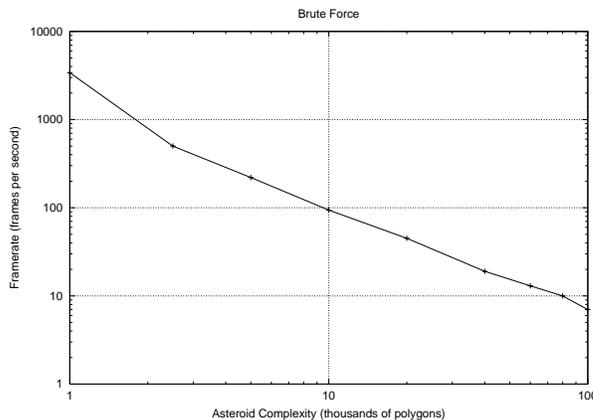


Figure 4: Brute force silhouette extraction runtime performance

Pre-Process

Create a data structure which holds the information about an edge and contains pointers to the normal vectors of both polygons associated with that edge. Then create an edge list using a static array of these edge data structures.

Runtime

At runtime, for every frame, traverse the edge list. Each polygon adjacent to the current edge is tested to determine whether it is front-facing or back-facing with respect to the current eye point. If one polygon is front-facing and the other back-facing a silhouette line is rendered.

Results and Observations

There is a steep dropoff initially due to cache performance, then the brute-force silhouette method's runtime complexity scales linearly with the number of edges. Because of the simplicity of the brute force method, it is easy to implement using a single iterative loop, thus eliminating any function call overhead.

4.2 Edge Buffer

Method

The second silhouette algorithm tested is the "edge buffer" [Buchanan and Sousa 2000]. Instead of iterating over each edge and testing both adjacent polygon normals for front/back facing, the Edge Buffer method iterates over the polygons. Since the number of polygons is always fewer than the number of edges, the edge buffer method should run faster than the brute force algorithm.

Pre-Process

Create an edge list similar to the data structure used in the brute force method, adding of a front facing flag and a back facing flag for each edge. The front facing and back facing flags for each edge are initialized to 0. In addition, create a data structure which maps polygons back to edges shared by the respective polygon.

Runtime

For each polygon in the polygon list, test to see whether the polygon is front-facing. If the polygon is front facing, XOR a 1 with the front facing flag for each edge shared by that polygon. Likewise, if the polygon is back facing, XOR a 1 with the back facing flag for each edge shared by that polygon. Upon completion, each edge that shares exactly one front-facing polygon and one back-facing polygon will have both flags set. Finally, iterate through the edge list and draw only the edges that have both their flags set, also reset both flags for all the edges to 0.

Results and Observations

In practice this implementation of the edge buffer algorithm runs slightly slower than brute force. Although the edge buffer method is less complex than brute force in terms of floating point operations, the edge buffer has a higher overhead cost because of the XOR

operations, in addition to the dot products computed to test whether polygons are front or back facing.

As outlined in the edge buffer paper [Buchanan and Sousa 2000], it is necessary to do a large number of edge table lookups at runtime. An optimization of this method is to create a polygon list during the pre-process stage to eliminate table lookups at runtime. Without this optimization, the edge buffer runs much slower than brute force. The edge buffer's runtime routine is implemented using a single loop, eliminating function call overhead.

4.3 Probabilistic

Method

[Markosian et al. 1997] present a probabilistic silhouette finding algorithm. In this method a small number of edges are chosen based on the probability that they are silhouette edges, then tested to see if they are silhouette edges. Edges with higher dihedral angles have a higher probability of being silhouettes as shown in Figure 6. Silhouette edges from the previous frame also have a high probability of being silhouettes, or being close to the new silhouette edges. When a silhouette edge is found, its adjacent edges are tested to see if any of these edges is also a silhouette edge.

Pre-Process

Compute the dihedral angle of each edge in a model, sort the edges by their dihedral angle, and place the sorted edges into a list. This list is called the dihedral angle list. As shown in Figure 6, the probability that an edge is a silhouette is (θ/π) where θ is the dihedral angle between the edges two adjacent polygons [?]. At runtime choose random edges to test as silhouettes weighting the random search to look at edges with a high probability of being silhouettes. The data structure used for individual edges is modified from the brute force data structure with the addition of pointers to all adjacent edges.

Runtime

A collection of edges is chosen and tested to see if they are silhouette edges. First test all the silhouette edges from the previously rendered frame. Next, a number of randomly chosen edges from the dihedral angle list are tested to determine if they are silhouette

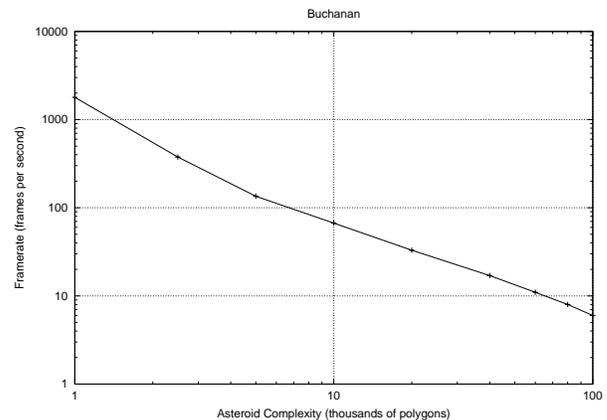


Figure 5: Edge Buffer silhouette extraction runtime performance

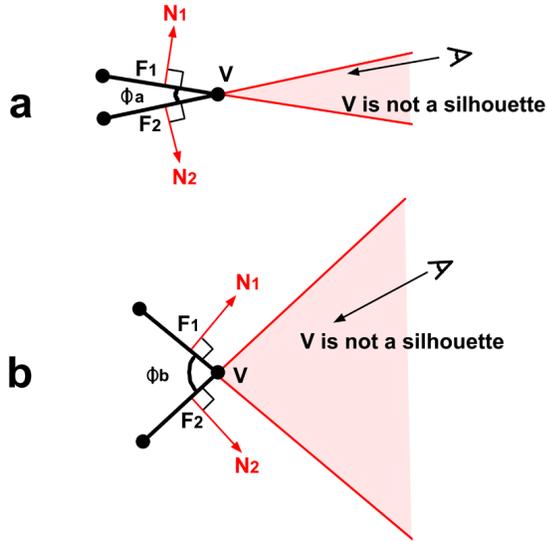


Figure 6: The small dihedral angle corresponds to a small view area where edge V is not a silhouette. The large dihedral angle corresponds to a large view area where edge V is not a silhouette. Thus an edge with a smaller dihedral angle has a higher probability of being a silhouette edge.

edges. Each edge that is found to be a silhouette edge for the current viewpoint is then traced.

If the edge is a silhouette, trace through the edge adjacency pointers in the dihedral angle list to check if any adjacent edges are silhouette edges. Adjacent edges are recursively tested until there are no adjacent silhouette edges. In order to avoid testing silhouette edges that have already been found, a reference to each silhouette edge is hashed into a table. Every time a new silhouette edge is found it is checked for inclusion in this hash table. In order to avoid having to reset the hash table after each frame, frame numbers are stored in the hash table and compared to the current frame number.

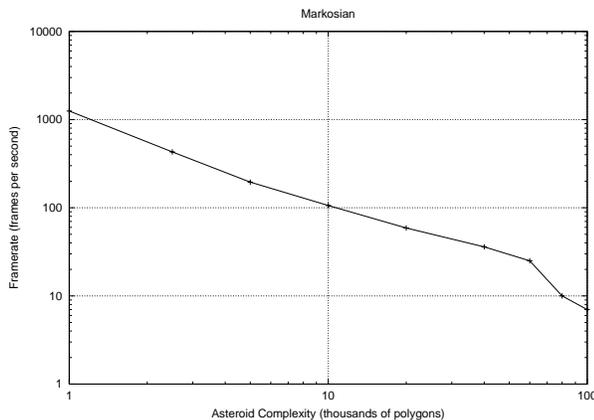


Figure 7: Probabilistic silhouette extraction runtime performance

Results and Observations

A speedup over the brute force algorithm is observed, although a small number of silhouette edges may be missed for a particular frame. It was observed that frame-to-frame coherence of silhouette edges works very well, but can become more difficult as the model becomes large because silhouette curves tend to move across many polygons per frame and may be missed by the tracing algorithm. For good performance scaling, the number of random edges chosen at the beginning of each frame should be proportional to the square root of the total number of edges.

4.4 Gauss Map Arcs

Method

A modified Gauss map can be used to calculate the silhouette edges of a polygonal model [Gooch et al. 1999; Benichou and Elber 1999]. The model is placed at the origin of a bounding sphere (Gauss map) and each edge of the model maps to an arc on the sphere. Silhouette edges are extracted from the Gauss map by intersecting the Gauss map with a plane. The intersecting plane is defined as passing through a point at the origin of the bounding sphere and perpendicular to the viewing vector. The arcs on the Gauss map which are intersected by the plane correspond to silhouette edges in the original model. Since the Gauss map only takes into consideration the viewing direction, and not the viewing distance, it will not work for perspective. A 2D Gauss map example is shown in Figure 8.

Pre-Process

Begin by mapping the model edges onto the Gauss map. Each edge in the model has two adjacent facets, F_1 and F_2 which have normals N_1 and N_2 . Model edges are mapped to the Gauss surface by placing N_1 and N_2 at the origin of the Gauss Map and sweeping N_1 across the Gauss surface to N_2 . For a Gauss sphere, N_1 and N_2 sweep out an arc on the surface of the sphere.

For simplicity the Gauss map can be represented as a bounding cube. With the Gauss map represented as a cube, the Gauss map arcs become straight lines on one or more of the cube faces. Each face of the cube is divided into a 20 by 20 grid of buckets, and each edge maps to several of these buckets, see Figure 9. After all edges have been mapped each bucket contains a list of zero or more edges.

Runtime

To extract the set of silhouette edges for a given viewing direction a plane is intersected with the Gauss map. The intersecting plane is defined by a point P at the origin of the Gauss map and a vector V which is the viewing direction. Since the Gauss map is represented as a grid of buckets, the intersecting plane corresponds to a list of buckets intersected in the Gauss map. Each bucket contains a possibly empty list of edges which are silhouette edges for the current viewing direction.

Results and Observations

This algorithm is simple to implement when the Gauss sphere is approximated by a cube. However, the data structures used in this algorithm can become large depending on the bucket resolution, and this technique works only for orthogonal projection.

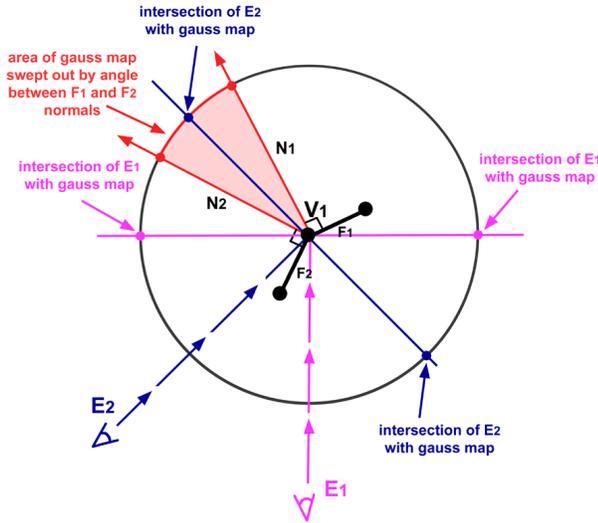


Figure 8: 2D Gauss Map Example. During a pre-process, the edge V_1 maps to an arc on the Gauss Map. At runtime, a line perpendicular to the eye point and which passes through the origin is used to intersect the Gauss Map. If the line intersects the arc defined for V_1 , then V_1 is a silhouette point. In this example V_1 is a silhouette as seen from the eye point E_2 .

4.5 Hierarchal Culling

Method

There are several silhouette extraction methods based on hierarchal culling. Hertzmann et al. use dual surface intersections [Hertzmann and Zorin 2000]. Pop et al. use a wedge hierarchy [Pop et al. 2001]. Sander et al. introduce the idea of using two open-ended normal cones to compute whether or not an edge is a silhouette edge [Sander et al. 2000].

Each of these algorithms requires descending a hierarchal data structure at runtime. We choose to implement the normal cone method of Sander et. al because their method requires only a single dot product to be computed at each level of decent. The methods of Hertzmann et al. [Hertzmann and Zorin 2000] and Pop et al. [Pop et al. 2001] interpolate between edges to find the exact zero crossing of the silhouettes and require more complex calculations.

Sander et al. [Sander et al. 2000] create a hierarchy of normal cones during a pre-process, which can be used at runtime to cull large numbers of edges which are not silhouettes. Any cones which intersect the current eye vector are discarded as not being silhouette edges. Cones that do not intersect the eye vector have to be analyzed. (See Figure 3.) The cones are organized in a hierarchal search tree. If the polygons in a node can be determined to be all front-facing or all back-facing, the node can be discarded as not containing silhouettes.

Pre-Process

Create a cone hierarchy containing all edges. The algorithm of Sander et al. uses weighting functions based on linear programming to determine which cones can be combined when forming the search tree. These weighting functions are computationally expensive and cause the pre-process algorithm to take anywhere from

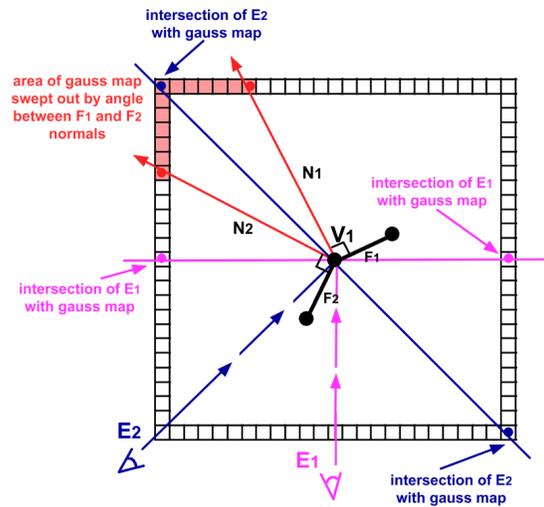


Figure 9: 2D Example of our Gauss Map implementation. Create a containing cube. For each triangle in mesh (a) intersect polygon normal with cube (b) draw a point on cube representing polygon (c) connect adjacent polygons, or points, with a line.

30 minutes to 24 hours to load a polygonal model. Instead of using weighting functions, we implemented a sort and divide routine based on the observation that cones whose edges have similar dihedral angles, have similar normals, and are spatially close together combine to form good cones. An example of a cone hierarchy is shown in Figure 13.

First, divide the edges into groups based on their dihedral angles. Next, divide the unit sphere into eight regions and build normal cones for each of these regions. Each dihedral angle group is now sorted by normal to form the next level of the hierarchy. Next sort each of the normal cones with respect to spatial proximity to build the next level. Recursively continue the cone normal and spatial proximity sort process until each cone contains a single edge. A more detailed reporting of this process, along with analysis of pre-process timing and runtime evaluation is currently under submission [Hartner et al. 2002].

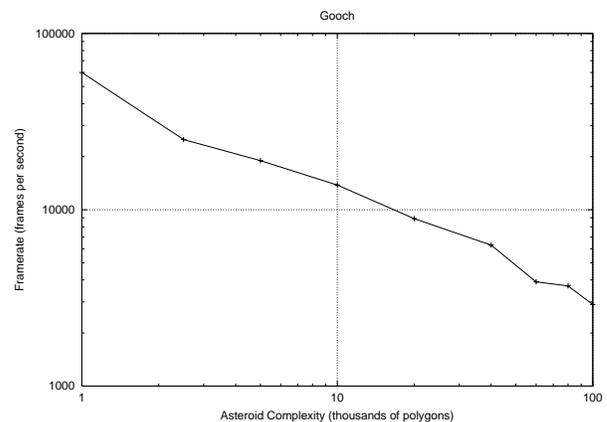


Figure 10: Gauss map silhouette extraction runtime performance

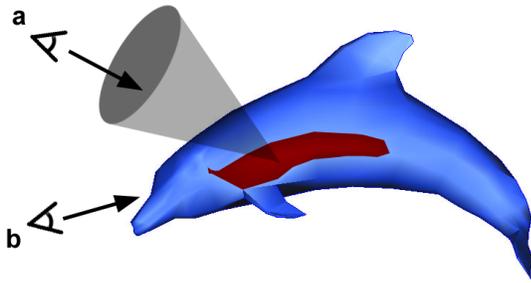


Figure 11: A dolphin model in which a cone is associated with some of the polygon edges. (a) view from which marked polygons are not silhouettes (b) view from which marked polygons must be tested

Runtime

During runtime, traverse the cone hierarchy testing cones to find if they include the current eye vector. This test can be done with two dot products:

The eye vector is inside the cone if

$$(E - O) \cdot (N) \geq 0 \text{ and } ((E - O) \cdot (N))^2 \geq \|E - O\|^2$$

Where E is the vector from the eyepoint as in 2, O is the cone origin and N is the scaled cone normal $N = \text{coneNormal} / \cos(\text{coneAngle})$. If a cone contains the eye point, we can discard it and all its sub-cones. Otherwise we traverse down the cone hierarchy until we are left with only edges. Each edge that is not culled at this point must be checked individually to determine whether or not it is a silhouette edge.

Results and Observations

A significant improvement in runtime speed for silhouette extraction is observed using this method. Because cone hierarchy is constructed using a top-down approach, alternating grouping normals by angle and by distance, edges become closer together both in terms of edge normal and spatial locality at each level of the hierarchy.

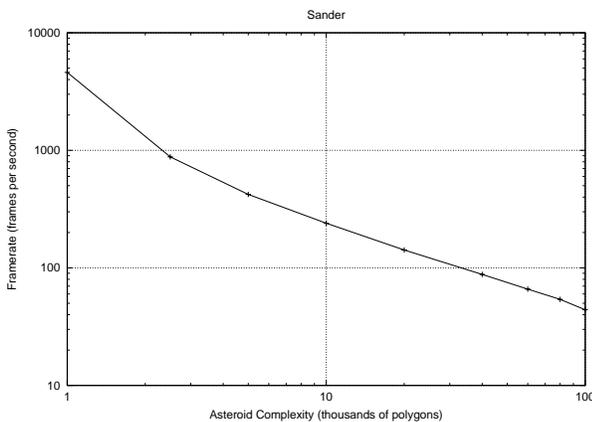


Figure 12: Hierarchical Culling silhouette extraction runtime performance

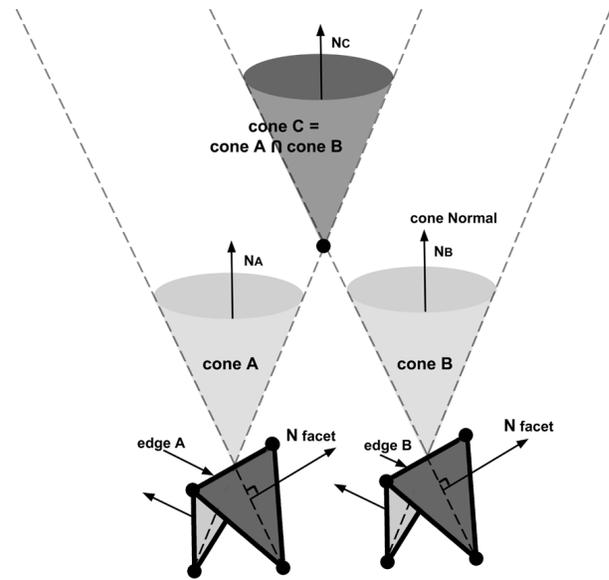


Figure 13: Example of a cone hierarchy built from cones that have similar dihedral angles, have similar cone normals, and are spatially close to each other.

5 Discussion and Future Work

We found that for small models, under 10,000 polygons for our hardware, brute force silhouette extraction is easy to implement and runs nearly as fast much more complex methods. For more complex models it may be worth the time implementing more complex methods. If orthographic is all that is needed, then Gauss Maps are easiest to implement and also very fast. For large models with perspective, methods based upon hierarchical culling may be necessary, but are generally difficult to program. Memory management seems to be more important than the algorithm. All of the methods scale at the same rate as soon as cache misses began to occur.

We found that silhouette extraction methods are more sensitive to size than to complexity in the polygonal models. In all of the tests performed model size dominated the runtime speed of the algorithms. We tested the algorithms on the complexity based test suite of Kettner and Welzl [Kettner and Welzl 1997] we found no significant difference in the runtime of the algorithms on models of differing complexity but with similar polygon counts. This may be due to the fact that all of the models in the Kettner and Welzl test suite have less than 15,000 polygons. The complexity based model test suite is available online at: www.cs.unc.edu/kettner/proj/obj3d/index.html

The algorithms reviewed in this paper represent the current best in the field. However, an ideal silhouette extraction algorithm has yet to be found. Some desirable characteristics for an ideal silhouette algorithm are given below.

- The ability to handle non-closed models and multiple objects.
- The ability to perform visibility culling on the extracted silhouette set.
- The ability to perform occlusion clipping on the extracted silhouette set.
- The ability to simplify coincident silhouette edges, i.e. edges which overlap or occlude each other when projected to screen space.

6 Acknowledgements

We would like to thank Aaron Hertzmann for sharing his code with us. We would also like to thank Peter Pike Sloan, Amy Gooch, Lee Markosian, Mario Costa Sousa, Gershon Elber and Adam Finkelstein for their time and consideration in talking with us and answering our many questions on this project. We would also like to thank Amy Gooch for her help in constructing models.

Thanks to Alias|Wavefront for their generous donation of Maya Complete, which we used to create the geometric models for this paper.

References

BENICHO, F., AND ELBER, G. 1999. Output sensitive extraction of silhouettes from polygonal geometry. In *Pacific Graphics '99*.

BOTTINO, A., AND LAURENTINI, A. 2001. Experimenting with noninvasive motion capture in a virtual environment. *The Visual Computer* 17, 1, 14–29. ISSN 0178-2789.

BUCHANAN, J. W., AND SOUSA, M. C. 2000. The edge buffer: A data structure for easy silhouette rendering. In *NPAP 2000 : First International Symposium on Non Photorealistic Animation and Rendering*, ACM SIGGRAPH / Eurographics, 39–42.

CHUNG, Y. C., PARK, J. W., SHIN, H., AND CHOI, B. K. 1998. Modeling the surface swept by a generalized cutter for nc verification. *Computer-aided Design* 30, 8, 587–594.

FUA, P., PLANKERS, R., AND THALMANN, D. 1999. From synthesis to analysis: Fitting human animation models to image data. In *Computer Graphics International '99*, IEEE CS Press. ISBN ISBN 0-7695-018.

GOOCH, B., SLOAN, P.-P. J., GOOCH, A., SHIRLEY, P. S., AND RIESENFELD, R. 1999. Interactive technical illustration. In *1999 ACM Symposium on Interactive 3D Graphics*, ACM SIGGRAPH, 31–38. ISBN 1-58113-082-1.

HAINES, E. 2001. Soft planar shadows using plateaus. *Journal of Graphics Tools* 6, 1, 19–27.

HARTNER, A., HARTNER, M., COHEN, E., AND GOOCH, B. 2002. A fast method for normal cone hierarchy construction. *Submitted to The Journal of Graphics Tools*.

HERTZMANN, A., AND ZORIN, D. 2000. Illustrating smooth surfaces. In *Proceedings of ACM SIGGRAPH 2000*, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, Computer Graphics Proceedings, Annual Conference Series, 517–526. ISBN 1-58113-208-5.

JENSEN, C. G., RED, W. E., AND PI, J. 2002. Tool selection for five-axis curvature matched machining. *Computer-Aided Design* 34, 3 (March), 251–266. ISSN 0010-4485.

JOHNSON, D. E., AND COHEN, E. 2001. Spatialized normal cone hierarchies. In *2001 ACM Symposium on Interactive 3D Graphics*, 129–134. ISBN 1-58113-292-1.

KETTNER, L., AND WELZL, E. 1997. Contour edge analysis for polyhedron projections.

LAKE, A., MARSHALL, C., HARRIS, M., AND BLACKSTEIN, M. 2000. Stylized rendering techniques for scalable real-time 3d animation. In *NPAP 2000 : First International Symposium on Non Photorealistic Animation and Rendering*, ACM SIGGRAPH / Eurographics, 13–20.

LEE, W., GU, J., AND MAGNENAT-THALMANN, N. 2000. Generating animatable 3d virtual humans from photographs. *Computer Graphics Forum* 19, 3 (August). ISSN 1067-7055.

LUM, E., AND MA, K.-L. 2002. Hardware-accelerated parallel non-photorealistic volume rendering. In *NPAP 2002*, ACM SIGGRAPH / Eurographics.

MARKOSIAN, L., KOWALSKI, M. A., TRYCHIN, S. J., BOURDEV, L. D., GOLDSTEIN, D., AND HUGHES, J. F. 1997. Real-time nonphotorealistic rendering. In *Proceedings of SIGGRAPH 97*, ACM SIGGRAPH / Addison Wesley, Los Angeles, California, Computer Graphics Proceedings, Annual Conference Series, 415–420. ISBN 0-89791-896-7.

POP, M., DUNCAN, C., BAREQUET, G., GOODRICH, M., HUANG, W., AND KUMAR, S. 2001. Efficient perspective-accurate silhouette computation and applications. In *Proceedings of the seventeenth annual symposium on Computational geometry*, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 60 – 68. ISBN:1-58113-357-X.

RHEINGANS, P., AND EBERT, D. 2001. Volume illustration: non-photorealistic rendering of volume models. *IEEE Transactions on Visualization and Computer Graphics* 7, 3 (July - September), 253–264. ISSN 1077-2626.

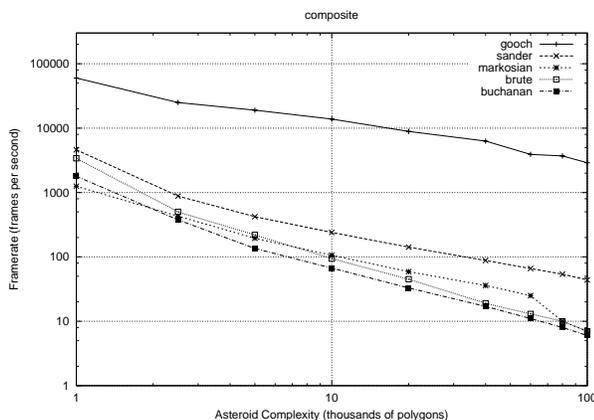


Figure 14: Method comparison for silhouette extraction runtime performance

Polygons	Edges	Silhouette Edges
1000	1425	290
2500	3627	530
5000	7260	860
10000	14742	1300
20000	129700	2000
40000	59436	2850
60000	89388	3300
80000	119340	4200
100000	148941	4600

Table 1: An overview of the polygon count, the number of edges, and the average number of silhouette edges in the "irregular sphere test suite" models. The "silhouette edge" number was calculated by averaging the number of silhouettes from 100 random perspective models.

SANDER, P. V., GU, X., GORTLER, S. J., HOPPE, H., AND SNYDER, J. 2000. Silhouette clipping. In *Proceedings of ACM SIGGRAPH 2000*, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, Computer Graphics Proceedings, Annual Conference Series, 327–334. ISBN 1-58113-208-5.

Polys	Brute F.	Edge B.	Probabilistic	G. Map	N. Cone
1000	326	320	336	340	348
2500	325	315	332	335	345
5000	194	129	185	330	335
10k	92	64	103	330	210
20k	45	32	57	330	129
40k	19	16	34	330	82
60k	13	11	24	315	62
80k	10	8	10	310	51
100k	7	6	7	310	42

Table 2: This table shows the the framerate of the algorithms for silhouette extraction and rendering.

Polys	Brute F.	Edge B.	Probabilistic	G. Map	N. Cone
1000	3400	1790	1250	60000	4600
2500	500	376	430	25000	880
5000	220	135	195	19000	422
10k	94	67	106	13800	240
20k	45	33	59	8900	142
40k	19	17	36	6300	88
60k	13	11	25	3900	66
80k	10	8	10	3700	54
100k	7	6	7	2900	44

Table 3: An overview of the framerate of the algorithms for only silhouette extraction. We show these numbers because silhouette algorithms can be used in haptic applicaitons, for example, without the need to render the silhouette edges.

Polys	Brute F.	Edge B.	Probabilistic	G. Map	N. Cone
1000	5.8	5.4	6.1	15	5.8
2500	6.4	5.7	7	29	6.4
5000	7.6	6.2	8.	48	7.4
10k	9.7	7.2	11	62	9.6
20k	14	9.1	18	103	14
40k	23	13	30	133	24
60k	32	17	42	160	32
80k	40	20	55	190	40
100k	49	24	68	219	49

Table 4: An overview of the memory requiernments, in megabytes, of the algorithms for silhouette extraction and rendering.