Object Space Silhouette Algorithims

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 central role in a growing number of applications. This paper examines five object space silhouette extraction algorithms for polygonal models. The algorithms are compared in terms of code complexity, necessary system resources, and run time performance on a variety of polygonal models. The goal of this paper is to become an informative source for programmers making a choice between one of these five algorithms.

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

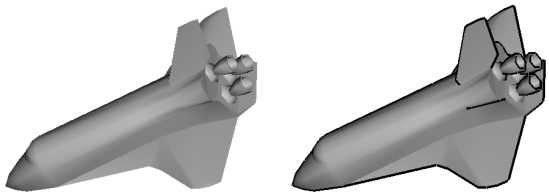

Figure 1: A polygonal space shuttle rendered both without, and with silhouette lines.

1 Introduction

Silhouette drawings are a simple form of line art used in cartoons, technical illustrations, architectural design and medical atlases [Hertzmann and Zorin 2000]. The silhouette curves of a polygonal model are useful in realistic rendering, in interactive techniques, and in non-photorealistic rendering (NPR).

In realistic rendering silhouettes can be used for model simplificaiton and for 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].

In interactive rendering silhouettes are used for haptic rendering. Johnson and Cohen show that haptic rendering can be facilitated using silhouette information [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 systems which use silhouettes to emphasize key data in volume renderings [Rheingans and Ebert 2001; Lum and Ma 2002].

The silhouette set of a polygonal model can either be computed in object space or in screen space. Object space algorithms involve computations in three dimensions and produce a list of silhouette edges for a given viewpoint. Screen space algorithms usually involve image processing techniques and are useful if rendering silhouettes is the only goal. 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.

1. Brute Force – Iterate through each edge in a polygonal model and test whether each edge 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 where a finite number of "seed" edges are chosen of reach 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

At a point on a surface, $\sigma(u,v)$, given $E(u,v)$ as the eye vector and $N(u,v)$ as the surface normal, a silhouette point is defined as the point on the surface where $E(u,v) \cdot N(u,v) = 0$, or that the angle between $E(u,v)$ and $N(u,v)$ is 90 degrees. This relationship is demonstrated in Figure 2. 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 do exist for three dimensional models. These lines include; texture boundaries, creases, and object boundaries. These additional feature lines have the convenient property of being view independent, and can therefore be completely specified prior to runtime. In this work we evaluate runtime silhouette extraction algorithms.

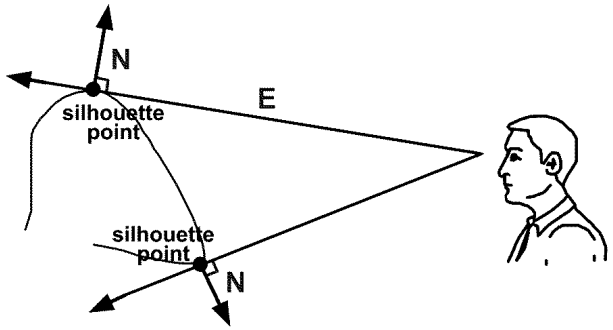

Figure 2: 2-D silhouette example for a smooth surface.

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 3.

For uniformity throughout this work we assume 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 algorithims at runtime we built a test suite using multiple tesselations of an modified sphere model as shown in Figure 9. This model was chosen because; it has a high silhouette complexity, it contains interior silhouettes for every veiw angle, and containins regions with similar normals which are not spacialy close to each other, these properties make this model a worst case for all of the evaluated algorithims. These properties make this model a worst case for all of the evaluated algorithims. The observed runtime speed we report is the average number of frames per second for 1080 frames which represent traversing the model along circular paths around the X, Y, and Z axies.

We implemented five object space silhouette algorithms 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. For the measure of theoretical runtime complexity we treat silhouette extraction and rendering as seperate processes and report only on extraction. The observed runtime speed we report is calculated by dividing 1080 frames by the number so seconds needed to render those frames. The 1080 frames represent traversing the model along circular paths around the X, Y, and Z axies respectively. The runtime tests were performed on an AMD Athlon 1.4Ghz machine with 512MB of RAM and a GeForce3 (not TI) grachics card running Linux 2.4.18. For each method we present:

1. Theoretical Complexity – We report the theoretical runtime complexity of the algorithms based on analysis of our implementation.
2. Observed Runtime Speed – We report the frames per second on a 69,473 polygon model of the “Stanford Bunny”.
3. Code Complexity – We report two measures of code complexity, the amount of time needed for an advanced undergraduate student to write and debug the code, and the number of lines of code in both the pre-process and the runtime routine.
4. Pre-Process Speed – The amount of time the pre-process runs.
5. Memory Usage – We report memory usage of the algorithm on a 69,473 polygon model of the “Stanford Bunny”.

4.1 Brute Force

Method

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

Pre-Process

We 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. We then create an edge list using a static array of these edge data structures.

Runtime

At runtime, for every frame, we traverse the edge list, testing whether each edges two adjacent polygons are front-facing or back-facing with respect to the current eye point. If one polygon is front-facing and the other back-facing, the edge is then rendered.

Results and Observations

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.

Theoretical Complexity – Linear complexity based on the number of edges.

Observed Runtime Speed – 11 FPS.

Code Complexity – One hundred lines of code, written in six hours

Pre-Process Speed – The pre-process runs in less than one second.

Memory Usage – 35 megabytes of memory used.

4.2 Edge Buffer

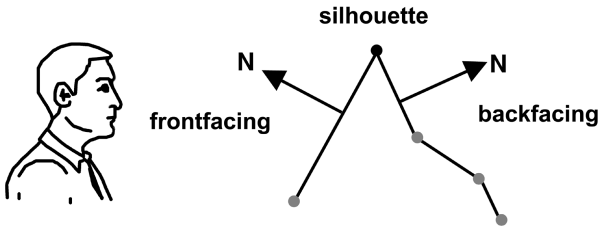

Figure 3: 2-D silhouette example for a polygonal surface.

Method

The second silhouette algorithm we test is the "edge buffer" introduced by Buchanan and Sousa [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. Due to the fact that the number of polygons is always lower than the number of edges, the edge buffer method should run faster than brute force.

Pre-Process

We create an edge list similar to data structure used in the brute force method, with the addition 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 we create polygon list which is composed of a static array of polygon data structures. Each polygon entry contains a pointer to each edge shared by that polygon.

Runtime

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

Results and Observations

In practice our implementation of the edge buffer algorithm actually 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 the XOR operations are being performed in addition 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. We created the polygon list during the pre-process stage to eliminate all table lookups during 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, thus eliminating function call overhead.

Theoretical Complexity – Linear complexity based on the number of polygons.

Observed Runtime Speed – 9 FPS.

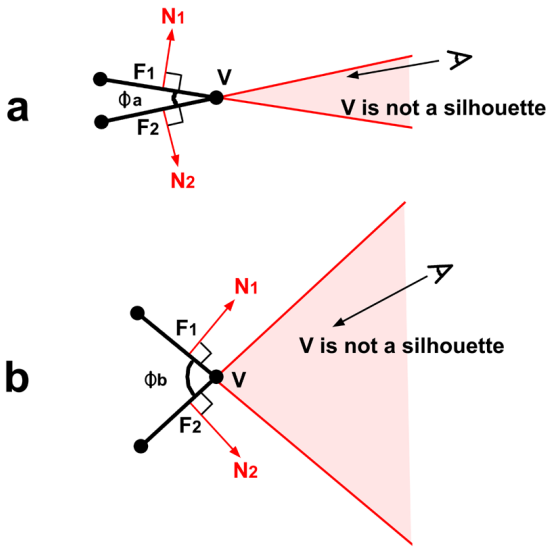

Figure 4: a) The small dihedral angle corresponds to a small view area where edge V is not a silhouette. b) The large dihedral angle corresponds to a large view area where edge V is not a silhouette. Thus an edge with a smaller dihedral angle has a higher probability of being a silhouette edge.

Code Complexity – Three hundred lines of code, written in eight hours

Pre-Process Speed – The pre-process runs in less than one second.

Memory Usage – 18 megabytes of memory used.

4.3 Probabilistic

Method

Markosian et al. [Markosian et al. 1997] present a probabilistic silhouette finding algorithm. 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 4.3. 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

We compute the dihedral angle of each edge in a model, sort the edges by their dihedral angle, and place the sorted edges into an edge list data structure. We all this list the dihedral angle list. This sort is preformed because the probibility that a edge is a silhouette is $(theta/pi)$ where $theta$ is the dihedral angle between the edges two adjacent polygons [**?**]. At runtime when we choose random edges to test as silhouettes we weight the random search to look at edges with a high probibility 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. We first test 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 edges. Each edge that is found to be a silhouette edge for the current viewpoint is then "traced".

If the edge is a silhouette, we 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.

Results and Observations

We observed a marginal speedup over the brute force algorithm at the cost of missing some silhouette edges. During runtime, missed edges cause the model to shimmer. For good visual performance the number of random edges chosen at each time step must be proportional to the total number of edges. For this reason the runtime complexity seems to be proportional to the total number of edges. We also noticed that the random nature of the algorithm caused it to scale poorly to large models due to problems with cache coherence.

Theoretical Complexity – Linear complexity based on the number of silhouette edges.

Observed Runtime Speed – 23 FPS.

Code Complexity – Four hundred lines of code, written in sixteen hours

Pre-Process Speed – The pre-process runs in less than one second.

Memory Usage – 48 megabytes of memory used.

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 by the point at the origin of the bounding sphere and 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 5.

Pre-Process

We begin by mapping the model edges onto the Gauss map. Each edge in the model has two ajacent facets, F1 and F2 which have normals N1 and N2. Model edges are mapped to the Gauss surface by placing N1 and N2 at the origin of the Gauss Map and sweeping

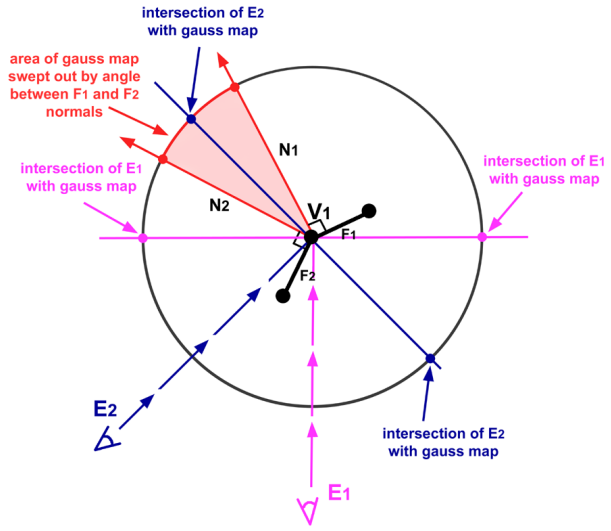

Figure 5: 2D Gauss Map Example. During a pre-process, the edge V_1 maps to an arc on the Gauss Map. At runtime, a line perpendicular to the eye point and which passes through the origin is used to intersect the Gauss Map. If the line intersects the arc defined for V_1, then V_1 is a silhouette point. In this example V_1 is a silhouette as seen from the eye point E_2.

N1 across the Gauss surface to N2. For a Gauss sphere, N1 and N2 sweep out an arc on the surface of the sphere.

To think of this mathematically, take a vector V perpendicular to both N1 and N2, and a point P at the origin of the Gauss Map. The vector V and point P define a surface S which intersects the Gauss Map. A subset of the plane/surface intersection which spans the edge dihedral angle defines the arc.

In our implementation we represent the Gauss map 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 our cube is divided into a 20 by 20 grid of buckets, and each edge maps to several of these buckets, see Figure 6. 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 used to intersect 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 our 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 algorithim can become large depending on the bucket resolution, and this technique works only for orthogonal projection. However, the Gauss Map method is ideal when sufficient quantities of memory are available.

Theoretical Complexity – Constant complexity based on the number of bins in the Gauss map.

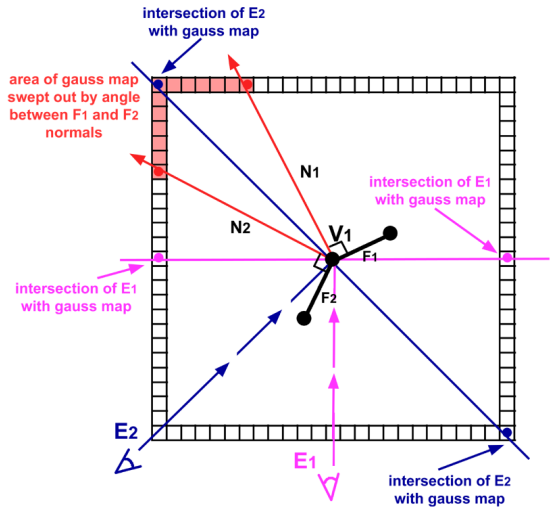

Figure 6: 2D Example of our Gauss Map implementation. Create a containing cube. For each triangle in mesh (a) intersect polygon normal with cube (b) draw a point on cube representing polygon (c) connect adjacent polygons, or points, with a line.

Observed Speed – 315 FPS.

Code Complexity – One thousand lines of code, written in sixty hours

Pre-Process Speed – The pre-process runs in less than one second.

Memory Usage – 166 megabytes of memory used.

Note: This method will only work for orthographic projection.

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 due to the fact that their method requires only a single dot to be computed at each level of decent. The other two methods are more complex runtime calculations, resulting in slower run-times. The methods of Hertzmann et al. [Pop et al. 2001] and Pop et al. [Pop et al. 2001] require more complex calculation because their methods interpolate between edges to find the exact zero crossing of the silhouettes.

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

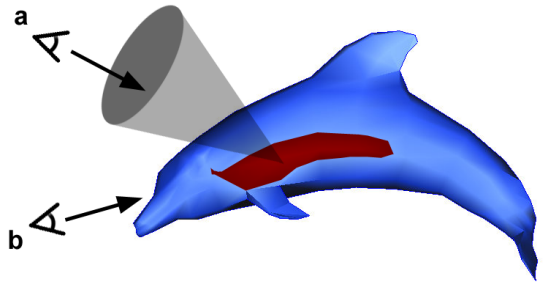

Figure 7: A dolphin model in which a cone is associated with some of the polygon edges. (a) view from which marked polygons are not silhouettes (b) view from which marked polygons must be tested

analyzed. (See Figure 2.) The cones are organized in a hierarchical 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

We create a cone hierarchy containing all edges. Sanders algorithm 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 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 8.

First, we divide the edges into groups based on their dihedral angles. Next, we 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. We next sort each of the normal cones with respect to spatial proximity to build the next level. We 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].

Runtime

During runtime, we traverse the cone hierarchy and test 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

$(eyePoint - coneOrigin)dot(scaledConeNormal) >= 0$ and $((eyePoint - coneOrigin)dot(scaledConeNormal))^2 >= ||eyePoint - coneOrigin||^2$ where $scaledConeNormal = coneNormal/cos(coneAngle)$

If a cone contains the eye point, we can discard it and all its subcones. 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.

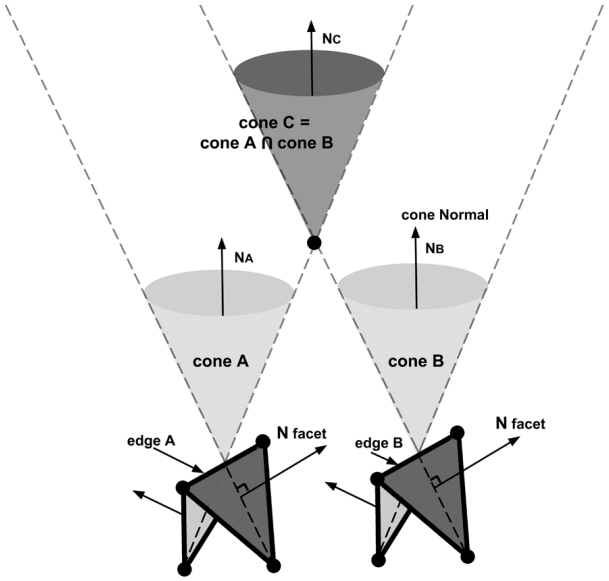

Figure 8: Example of a cone hierarchy built from cones that have similar dihedral angles, have similar cone normals, and are spatially close to each other.

Results and Observations

We observe logarithmic runtime speed for silhouette extraction. Because we construct our cone hierarchy 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.

Theoretical Complexity – Logarithmic complexity based on the number of edges.

Observed Speed – 58 FPS.

Code Complexity – Three thousand lines of code (with libraries), written in two hundred hours.

Pre-Process Speed – The pre-process runs in less than ten seconds.

Memory Usage – 36 megabytes of memory used.

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 off of heirarchical culling may be worth it, but are generally difficult to program.

We found that silouette extraction methods are more sensitive to size than to complexity in the polygonal models. In all of the tests we performed model size dominated the runtime speed of the algorithims. When tested the algorithims on the complexity based test suite of Kettner and Welzl [Kettner and Welzl 1997] we found no significant difference in the runtime of the algorithims on models of differing complexity but with similar polygon counts. This

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 overveiw of the polygon count, the number of edges, and the average number of silhouette edges in the "astroid test suite" models. The "silhouette edge" number was calculated by averageing the number of silhouettes from 100 randorespectivelythe model.*

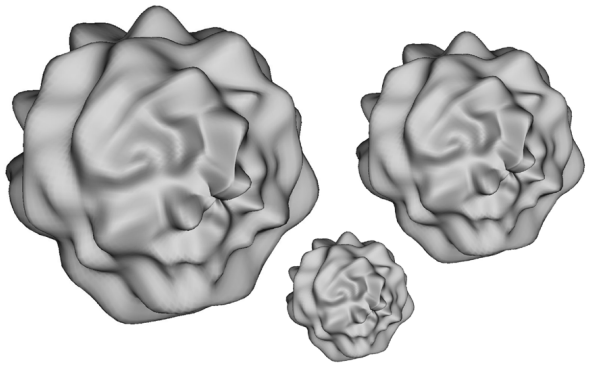

Figure 9: We test the runtime speed of the silhouette methods using various tesselations of an irregular sphere model. This model was chosen because; it has a high silhouette complexity, it contains interior silhouettes for every veiw angle, and containins regions with similar normals which are not spacialy close to each other, these properties make this model a worst case for all of the evaluated algorithims.

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: *w*ww.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 visability culling on the extracted silhouette set.

The ability to perform occlusion clipping on the extracted silhouette set.

The ability to simplifiy coincident silhouette edges, i.e. edges which overlap or occlude each other when projected to screen space.

A constant time algorithm which handles perspective projection and scales well to large models.

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 algorithims 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 overveiw of the framerate of the algorithims for only silhouette extraction. We show these numbers because silhouette algorithims 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 overveiw of the memory requiernments, in megabytes, of the algorithims for silhouette extraction and rendering.*

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 Finkelsein 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.

References

BENICHOU, 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 nonintrusive 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 *NPAR 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 *NPAR 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 *NPAR 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.

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.