A Developer's Guide to Silhouette Algorithms for Polygonal Models

Tobias Isenberg, Bert Freudenberg, Nick Halper, Stefan Schlechtweg, and Thomas Strothotte
Otto-von-Guericke University of Magdeburg, Germany

Generating object silhouettes lies at the heart of nonphotorealism. Algorithms for computing polygonal model silhouettes are surveyed to help find the optimal approach for developers' specific needs.

For a long time, line drawings have been a part of artistic expression (for example, any pencil or pen-and-ink drawing), scientific illustrations (medical or technical), or entertainment graphics (such as in comics). Hence, computer graphics researchers have extensively studied the automatic generation of such lines. In particular, the area of nonphotorealistic rendering has focused on two main directions of research in this respect: the generation of hatching that conveys illumination as well as texture in an image and the computation of outlines and silhouettes.

Silhouettes play an important role in shape recognition because they provide one of the main cues for figure-to-ground distinction. However, since silhouettes are view dependent, they need to be determined for every frame of an animation. Finding an efficient way to accomplish this is nontrivial. Indeed, a variety of different algorithms exist that compute silhouettes for geometric objects. This article provides a guideline for developers who need to choose between one of these algorithms for his or her application.

Here, we restrict ourselves to discussing only those algorithms that apply to polygonal models, because these are the most commonly used object representations in modern computer graphics. (For an algorithm to compute the silhouette for free-form surfaces see, for example, Elber and Cohen.[1]) Thus, we can use all algorithms discussed here to take a polygonal mesh as input and compute the visible part of the silhouette as output. Some algorithms, however, can also help compute the silhouette only, without additional visibility culling. The silhouette's representation might vary depending on the algorithm class—that is, the silhouette might take the form of a pixel matrix or a set of analytic stroke descriptions.

Definitions and terminology

The silhouette S of a free-form object is typically defined as the set of points on the object's surface where the surface normal is perpendicular to the vector from the viewpoint.[2] Mathematically, this means that the dot product of the normal $\mathbf{n_i}$ with the view vector at a surface vertex P's position p_i is zero: $S = \{P : 0 = \mathbf{n_i} \cdot (\mathbf{p_i} - \mathbf{c})\}$, with c being the center of projection. In case of orthographic projection $(\mathbf{p_i} - \mathbf{c})$ is exchanged with the view direction vector $\mathbf{v}$.

Unfortunately, for polygonal objects this definition can't be applied because normals are only well defined for polygons and not for arbitrary points on the surface. Hence, typically no points exist on the surface where the normal is perpendicular to the viewing direction. However, we can find silhouettes along those edges in a polygonal model that lie on the border between changes of surface visibility. Thus, we define the silhouette edges of a polygonal model as edges in the mesh that share a front- and a back-facing polygon.

Some authors refer to the contour rather than the silhouette. We define a contour as the subset of the silhouette that separates the object from the background. Also, the subset of the silhouette, which in 2D divides one portion of the object from another one of the same object, is not part of the contour. Those lines are sometimes called internal silhouettes.

Other lines significant in the context of silhouettes are creases, borders, and self-intersections. Creases are edges that should always be drawn, for instance the edges of a cube. Typically, we can identify a crease by comparing the angle between its two adjacent polygons with a certain threshold. Border lines only appear in models where the mesh is not closed and are those edges with only one adjacent polygon. Lastly, self-intersection lines are where two polygons of a model intersect. They aren't necessarily part of the model's edges but are important for shape recognition. In the literature, these three additional line categories together with the silhouette lines are often called feature lines. Some silhouette detection methods make no algorithmic distinction between these types of lines. This is because you must determine

 Published by the IEEE Computer Society

the visibility for all of these line types to generate an image. Hence, an algorithm's visibility aspect usually treats the lines in the same way. The visibility test yields visible segments of silhouette edges and potentially also feature lines. Sometimes visible segments are joined together into visible strokes.

Classification

Every silhouette algorithm must solve two major problems. First, we must detect the set of silhouette edges. Next, we determine the visible subset thereof (visibility culling). For the purpose of this article, we classify the approaches according to how they solve each of these problems. With respect to solving the edge detection problem, we distinguish between image space algorithms that only operate on image buffers, hybrid algorithms that perform manipulations in object space but yield the silhouette in an image buffer, and object space algorithms that perform all calculations in object space with the resulting silhouette represented by an analytic description of silhouette edges. The second problem—visibility culling—is inherently solved within the algorithm for both image space and hybrid approaches of silhouette detection. However, for object space methods, we must approach this problem separately. We can categorize the algorithms that perform this visibility culling into image-space, object-space, and hybrid approaches. Besides belonging to one of these categories, the algorithms differ from each other in other aspects more relevant in practice, such as:

- whether they solve the edge detection and edge visibility problem in one step or separately,
- how they represent results (that is, an image with the lines of a specific color or an analytic set of edges),
- how precise the results are (image, subpixel, or exact precision),
- how complete their results are (finding all silhouette edges or only a subset of them),
- how much computation time the algorithm takes (real-time, interactive rates, or offline rendering),
- how much memory the algorithms needs,
- whether they allow animation of the model, and
- whether they are easy to implement.

According to these measures, we give recommendations for which algorithm to use in a certain situation in Table 1.

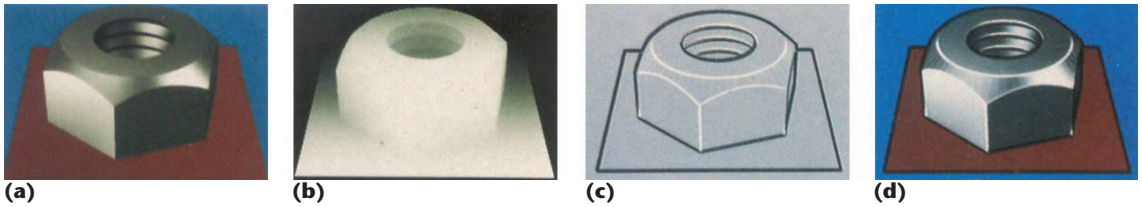
Table 1. Choosing the best silhouette detection algorithm.

Requirements	Recommendations
Real-time or interactive frame rates.	Image space and hybrid algorithms. Object space methods that use precomputation for silhouette detection and an image space or hybrid method for visibility culling. Brute-force object space silhouette detection or silhouette visibility test not recommended.
Silhouettes in an analytic description.	Object space silhouette algorithms with an object space or a hybrid visibility test.
Exact results; pixel or subpixel accuracy is not sufficient.	Object space silhouette algorithms with an object space visibility test. (This usually slows down the program significantly.)
Guarantee that all silhouette edges can be found.	An object space algorithm for all edges sharing a front- and a back-facing polygon; also image space and hybrid algorithms for all silhouette edges that contribute visually to the silhouette. A stochastic silhouette edge detection approach is not appropriate.
Animate model in real time beyond a mere flight through the scene.	Real-time technique. Object space silhouette detection method that uses preprocessing is not appropriate.
Stylized silhouettes.	Object space silhouette detection algorithm with an object space or a hybrid visibility test. Hybrid silhouette detection algorithms can also produce somewhat stylized silhouettes.
Cope with all model types, even those with errors.	Image space or a hybrid silhouette detection algorithm typically don't need connectivity information (polygon soups can be handled) and can deal with errors.
Silhouettes for a huge data set (with millions of polygons).	Image space or a hybrid silhouette detection algorithm for interactive or real-time applications. All other algorithms, but these probably won't scale linearly in their computation times.
Least amount of memory.	Image space or hybrid algorithm (these usually don't require an additional data structure besides the geometry).
Compute just the contour.	Rustagi's hybrid contour algorithm.[8] Object space silhouette detection algorithm with an adapted hybrid silhouette visibility test.

Image space algorithms

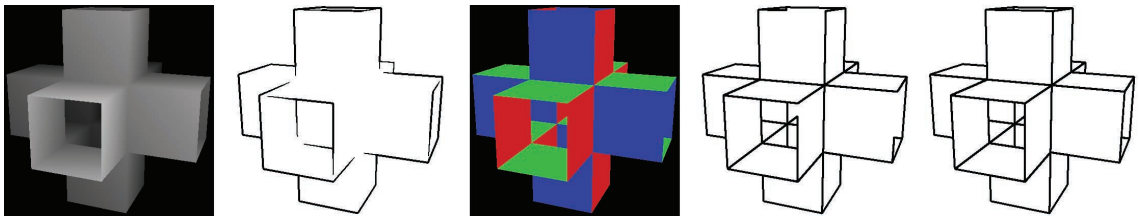
Image space algorithms exploit discontinuities in the image buffer(s) that result from conventional rendering and extract them using image-processing methods. These methods provide a silhouette represented as features in a pixel matrix.

The easiest way to find significant lines would be to detect edges in the color buffer. This, however, doesn't necessarily detect silhouettes since changes in shading and texturing can cause erroneous edge detection. Saito and Takahashi suggest using the z-buffer instead and applying an edge detector such as the Sobel operator.[3] This has the advantage of only finding object-relevant edges such as silhouette lines including contours, because at most of the places where silhouette lines are in the image there is a (C^0) discontinuity in the z-buffer



(a) (b) (c) (d)

1 Image space silhouette detection based on edge detection operators on the *z*-buffer. (a) Original image, (b) *z*-buffer, (c) detected edges, and (d) composed image. (Courtesy of Saito and Takahshi[3] © 1990 ACM. Reprinted with permission.)



(a) (b) (c) (d) (e)

2 Image space silhouette detection using edge detection operators on the *z*-buffer and the normal buffer. (a) *z*-buffer, (b) edges extracted from the *z*-buffer, (c) normal buffer, (d) edges extracted from the normal buffer, and (e) combination of both edge buffers. (Courtesy of Aaron Hertzmann[4] © 1999 ACM. Reprinted with permission.)

(see Figure 1). Hertzmann extends this method by using a normal buffer instead.[4] This can also detect C^1 discontinuities. Combining both approaches yields a pleasing result (see Figure 2).

Deussen and Strothotte use a simplified version of Hertzmann's algorithm to compute pen-and-ink illustrations of trees.[5] They render the tree's foliage as primitives (such as oriented disks) only to the *z*-buffer and look for discontinuities larger than a specified threshold. Depending on the primitives' size and the depth threshold this achieves the special look of pen and ink.

The advantage of image space algorithms is that we can use existing graphics hardware and rendering packages to generate the buffers on which the algorithms operate. This makes these algorithms relatively simple to implement. The computational complexity of silhouette detection in this manner depends on the number of pixels that comprise the image, rather than the number of polygons in the model, and thus is constant provided that image resolution remains constant. (Although silhouettes are not computed from polygons, these polygons must be rendered first into the required buffers.) Thus, these algorithms are usually fast, and appropriate graphics hardware can accelerate them even more. Mitchell suggests using the pixel shader technique on newer graphics cards for hardware acceleration of image space silhouette detection.[6] Finally, an image space algorithm deals with silhouette edges in the same manner as it does with feature lines. In contrast to other methods, these algorithms can automatically find self-intersections. They also share similarities with *z*-buffer rendering; the algorithms are generic and robust to errors in the models.

The main disadvantage of image space algorithms is that the user has little control over the resulting lines' attributes. The only way to directly influence the resulting lines is by choosing an edge detection operator and a source buffer on which to apply the operator. A second disadvantage is that the silhouette is not available in the form of an analytic line description. Hence, silhouettes can typically not be stylized or used for further processing. Thus, we can't readily apply many techniques simulating traditional drawing or painting utensils because they usually require this analytic information. On the other hand, some approaches extract curves from the silhouette pixels such as presented by Loviscach, who fits Bezier curves to the pixels.[7] Loviscach's approach allows for subsequent stylization of the resulting curves. However, the process of fitting curves to the pixel silhouettes might introduce new artifacts and inaccuracies. In addition, this approach is too slow for interactive or real-time applications.

Inherent in the use of image processing operators is that the resulting silhouettes don't have distinct borders. On the contrary, the intensity—the gray value—of a silhouette pixel usually depends on the derivative and thus on the intensity of features detected in the original buffer. However, this also means that the resulting images tend to not have significant aliasing problems. Another characteristic of these algorithms is that they are limited to pixel precision. This means that some fine details of less-than-pixel size might be hidden. However, for visual appearance this usually makes no difference.

Hybrid algorithms

Hybrid algorithms are characterized by operations in object space that are followed by rendering the modified polygons in a more or less ordinary way using the *z*-buffer. This usually requires two or more rendering passes. The result is similar to image space algorithms in that the silhouette is represented in a pixel matrix.

Rustagi presents a simple algorithm using the stencil

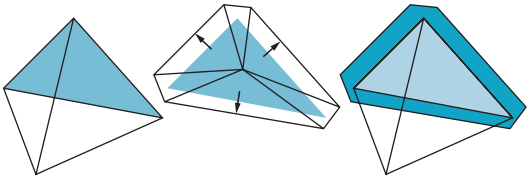
buffer that delivers the contour—not the complete silhouette—of an object.[8] The algorithm renders the object's mesh four times, each time translating the objects by one pixel in screen coordinates in the positive or negative x or y directions. During each pass it increments the stencil buffer where the object fills the viewport. After these four passes, the object's interior pixels will have a stencil value of four, the perimeter pixels will have values of two or three, and the exterior will have values of zero or one. Finally, setting the stencil function to pass if the stencil value is two or three and rendering a primitive larger than the object will result in only the outline pixels being changed.

Rossignac and Emmerik present a method based on z-buffer rendering that yields silhouettes and not only contours.[9] They show four algorithms that differ slightly from each other and that render polygonal objects either in wireframe or silhouette mode with the hidden lines either removed or dashed. For generating an image with visible silhouette and feature lines only, they first draw the object's faces into the z-buffer. Then they translate the mesh away from the viewer by a small distance and render a wireframe representation using wide lines. This ensures that only silhouette edges are visible since the previously rendered faces in the z-buffer will occlude the other lines. Finally, they translate the object forward again by twice the former distance and render the feature lines of the objects in regular line width.

Raskar and Cohen generalize this approach in their work.[10] For automatically determining front- and back-facing polygons that define the silhouette they use traditional z-buffering along with back- or front-face culling, respectively. Similar to Rossignac and Emmerik, by first rendering all polygons in white on a white background with back-face culling enabled, they fill the z-buffer with the depth data of front-facing polygons. Afterwards, they render all polygons again, but this time in the desired silhouette color and using front-face culling. Hence, only the back-facing polygons affect the frame buffer during this second pass. By employing the *equal to* depth function, Raskar and Cohen's approach draws only the edges where the two groups of polygons meet and thus yield the model's silhouette. Rendering the wireframe representation in the second pass achieves similar results. This allows silhouette rendering with a certain line width by using thicker lines for the wireframe rendering.

Performing additional transformations before rendering the back-facing polygons can improve this general technique. A translation of the back-facing polygons toward the viewer yields thicker silhouette lines in the resulting image. However, these lines don't have a constant line width. For silhouettes with adjustable but constant width, the back-facing polygons are enlarged depending on the distance from the viewer and the angle with the viewing direction (see Figure 3).

Gooch et al. present a similar technique that also uses hardware-accelerated rendering.[11] Instead of rendering directly to a color buffer, their method draws lines into a stencil buffer. This stencil buffer acts as a mask for drawing the back-facing polygons in a second pass. Also, it can render creases in a different color than silhouettes.



3 Enlargement of a back-facing polygon to achieve wide silhouette lines. (Courtesy of Ramesh Raskar and Michael Cohen[10] © 1999 ACM. Reprinted with permission.)

4 Silhouette generated with a hybrid algorithm. (Courtesy of Bruce Gooch,[11] University of Utah © 1999 ACM. Reprinted with permission.)

Raskar proposes a one-pass hardware implementation that basically adds borders around each triangle.[12] This method arranges the borders in such a way that they disappear between two neighboring front-facing polygons during rendering. It also inherently includes generating crease lines where the angle between the faces exceeds a given threshold.

Another technique introduced by Gooch et al. uses environment mapping in addition to regular shading.[11] By using a partially darkened environment map, this method assigns dark values to vertices with normals that are almost perpendicular to the viewing direction, whereas other vertices remain unchanged. This method achieves a stylistic effect (see Figure 4).

The advantage of hybrid over image space algorithms is the typically higher degree of control over the outcome. The choice of algorithm and parameters such as translation or enlargement factors that control line width provide this control. The visual appearance of the generated images tends to be a more stylistic one. Also, in contrast to image space algorithms, the silhouettes inherently have distinct borders, which might be a desired trait. On the other hand, the distinct borders can cause aliasing artifacts in the image. However, we can avoid these artifacts by employing well-known antialiasing techniques.

Computation times for hybrid algorithms generally don't differ much from those for image space methods and run at interactive to real-time frame rates. Some algorithms need two or more render passes but, in return, don't require additional manipulation of the generated buffers. Object space manipulations needed for some of the algorithms might add computation time in addition to the number of rendering passes. However, the algorithms can easily make use of commonly available graphics hardware to speedup the rendering process.

The drawbacks of a pixel matrix representation of the

detected silhouette lines—as mentioned for image-space algorithms—apply to hybrid algorithms also. Similarly, the silhouette lines have pixel resolution and don't facilitate further stylization. In addition, the algorithms might have some numerical problems due to limited z-buffer resolution.

Object space algorithms

To apply further stylization to the lines an analytic representation of the silhouette is needed. This is not achievable with the previously discussed algorithms because of the disadvantage of a pixel matrix representation of the computed silhouette. A good way to overcome this problem is to employ an object space algorithm. The computation of silhouettes in this group of algorithms, as the name suggests, takes place entirely in object space. In contrast to hybrid and image space algorithms, object space algorithms deal with the problems of edge detection and edge visibility in separate stages. Thus, we discuss these algorithms in two parts: methods for silhouette edge detection and, because parts of the silhouette edges might be occluded, finding the visible subset of those edges.

Silhouette edge detection

A straightforward way to determine a model's silhouette edges follows directly from the definition of a silhouette. Approaches that speed up the process use some precomputed data structures, while other algorithms achieve faster execution by employing stochastic methods.

Trivial method. The trivial algorithm is based on the definition of a silhouette edge. The algorithm consists of two basic steps. First, it classifies all the mesh's polygons as front or back facing, as seen from the camera. Next, the algorithm examines all model edges and selects only those that share exactly one front- and one back-facing polygon. The algorithm must complete these two steps for every frame.

Buchanan and Sousa suggest using a data structure called an *edge buffer* to support this process.[13] (The edge buffer optimizes platforms such as game consoles that have certain hardware restrictions.) In this data structure they store two additional bits per edge, F and B for front and back facing. When going through the polygons and determining whether they face front or back, they XOR the respective bits of the polygon's adjacent edges. If an edge is adjacent to one front- and one back-facing polygon the FB bits are 11 after going through all polygons and we can use them for silhouette rendering. Buchanan and Sousa extend this idea to support border edges and other feature edges.

This simple algorithm—with or without using the edge buffer data structure—works for both perspective and orthographic projections. Also, it's guaranteed to find all silhouette edges in the model, be easy to implement, and be well suited for applications that only use small models. However, it's computationally expensive for common hardware-based rendering systems and the model sizes typically used with them. (For software-based rendering systems, however, the silhouettes computed with the brute-force approach come at little extra cost.) Even if we suppose an effective data structure that delivers local connectivity information in constant time, an algorithm must look at every face, determine face orientation (using one dot product per face; for perspective projection it must recompute the view vector for every face), and look at every edge. This is a linear method in terms of the number of edges and faces but too slow for interactive rendering of reasonably sized models. When we also consider that only a small number of edges typically exist that are in fact silhouette edges, testing each one also seems unnecessary.

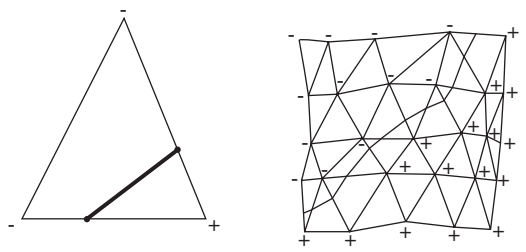
To speed up the brute-force algorithm, Card and Mitchell suggest employing user-programmable vertex processing hardware.[14] For every potential silhouette edge in the model a separate quad is generated along the edge. In addition, each vertex of such an edge stores the normals of both faces adjacent to the edge. When rendering the quads, a vertex program tests whether the normals are front or back facing. Only in cases where the result is different for both normals is the quad actually drawn.

Subpolygon silhouettes. Because a polygonal mesh is usually only an approximation of a free-form object, silhouettes of polygonal meshes typically have some artifacts (for example, zigzags or silhouette edge clusters). Hence, the expected silhouette of the real object can differ significantly from the silhouette that the trivial algorithm yields. Therefore, besides the quest for a fast algorithm, there are approaches that try to find a more exact silhouette, similar to that of the real object.

Hertzmann and Zorin consider the silhouette of a free-form surface approximated by a polygonal mesh.[2] To find this silhouette, they recompute the normal vectors of the approximated free-form surface in the vertices of the polygonal mesh. Using this normal, they compute its dot product with the respective viewing direction. Then, for every edge where the sign of the dot product is different at both vertices, they use linear interpolation along this edge to find the point where it is zero. These points connect to yield a piecewise linear subpolygon silhouette line (see Figure 5). The resulting silhouette line is likely to have far fewer artifacts. Also, the result is much closer to the real silhouette than the result of a traditional polygonal silhouette extraction method. Hence, it's well suited for later stylization of the lines. Figure 6 shows an example of this method.

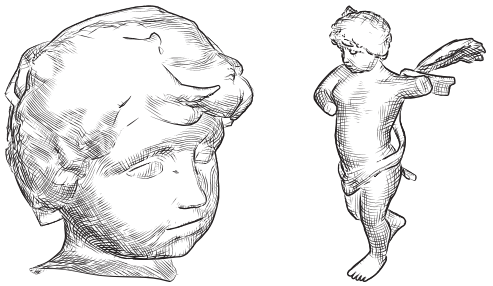
Precomputation methods. To speed up the process of silhouette edge determination, various authors have developed methods that accomplish some type of preprocessing. The preprocessing stage sets up data structures used to find silhouette edges more quickly.

Gooch et al.[11] and Benichou and Elber[15] present a preprocessing procedure based on projecting face normals onto a Gaussian sphere. Here, every mesh edge corresponds to an arc on the Gaussian sphere, which connects the normal's projections of its two adjacent polygons (see Figure 7). For orthographic projection, a view of the scene is equivalent to a plane through the origin of the Gaussian sphere. They further observe that every

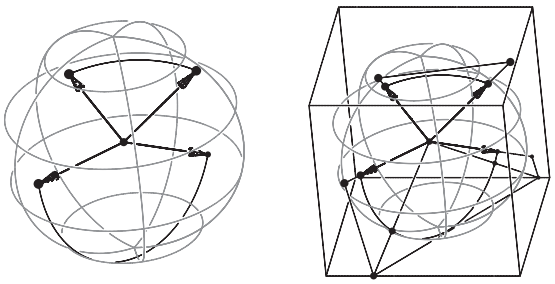


(a) (b)

5 Computation of a subpolygon silhouette for (a) a single triangle and (b) a mesh. The dot product between normal vector and view direction is positive at vertices with plus signs and negative at vertices with minus signs. Between a positive and a negative vertex linear interpolation is used to find the silhouette. (Courtesy of Aaron Hertzmann[4] © 1999 ACM. Reprinted with permission.)

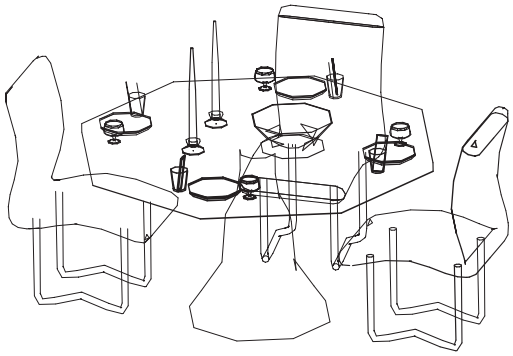


6 Two images generated with Hertzmann and Zorin's subpolygon method. The silhouettes are combined with a hatching technique. (Courtesy of Aaron Hertzmann and Denis Zorin[2] © 2000 ACM. Reprinted with permission.)



(a) (b)

7 Preprocessing using projection of the normals of two faces adjacent to an edge onto (a) the Gaussian sphere and (b) its surrounding cube. (Courtesy of Gershon Elber and Fabien Benichou[15] © 1999 IEEE.)



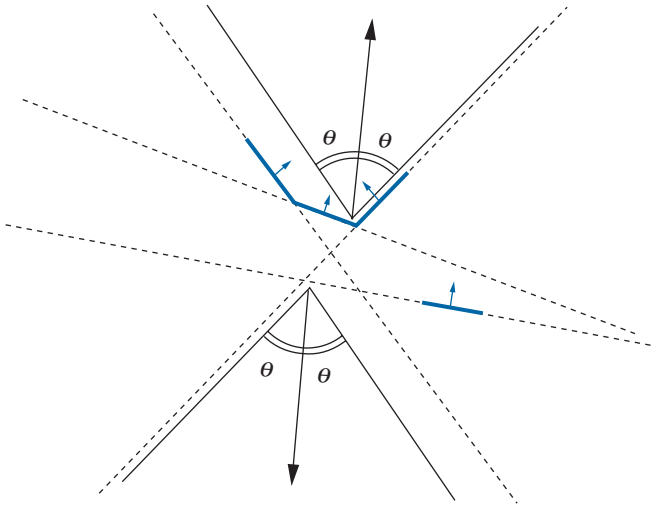
8 Example scene rendered with the Gaussian sphere preprocessing algorithm. No visibility test was used. (Courtesy of Gershon Elber and Fabien Benichou.)

arc intersected by this plane is a silhouette edge in the corresponding view. Applying this observation to silhouette edge extraction removes the need to check for each frame if every face is front or back facing. The arcs are computed in a preprocessing step and at runtime only the intersections with the view plane are tested. Figure 8 shows an example rendering.

To further limit the number of arcs tested, Gooch et al. use a hierarchical decomposition of the sphere. They start with a platonic solid and consecutively apply subdivision to solid's sides.[11] The arcs are stored in the lowest possible level. Benichou and Elber, on the other hand, map the Gaussian sphere and the arcs to a cube surrounding the sphere.[15] The arcs are equivalent to line segments on the cube. This method maps the view plane's great arc onto the cube, resulting in a set of line segments, which simplifies the intersection test. To reduce the number of arcs tested, this approach decomposes the sides of the cube into a grid and only tests for intersection those edge line segments in grid cells that touch a viewplane line segment.

Hertzmann and Zorin[2] present a method that uses a data structure also based on a dual representation, which is, in fact, similar in principle to the one by Benichou and Elber. This approach, however, constructs a dual representation of the mesh in 4D space based on the position and tangent planes of every vertex. The viewpoint's dual (a plane in 4D) intersects with the mesh triangles' dual. Beforehand, the approach normalizes the dual vertices using the l_∞ norm so that the vertices end up on one of the unit hypercube's sides. (The normalization does not make a difference because the viewpoint's dual plane goes though the origin.) This reduces the problem to intersecting the triangles on a hypercube's sides with the viewpoint's dual plane. This means you need to intersect triangles in eight 3D unit cubes (the eight hypercube sides) with a plane. Here, again, speedup occurs by employing space partitioning, namely by using an octree for each hypercube sides. At runtime, the approach only computes the viewpoint's dual plane and then intersects it with each hypercube side, resulting in edges that intersect the silhouette. The major advantage of this approach over methods discussed earlier in this section is that it works for orthographic as well as perspective projections. For orthographic projection you simply place the viewpoint at infinity.

Pop et al. present another algorithm based on a dual representation.[16] Similar to the previous approach but this time in 3D, this method constructs a dual space with



9 Preprocessing for faster silhouette detection through arrangement of face clusters in anchored cones (visualized in 2D) by Sander et al.[17] (© 2000 ACM. Reprinted with permission.)

vertices having planes as duals and vice versa. This reduces the silhouette problem to finding intersections of the viewpoint's dual plane with the duals of the mesh edges. This, unfortunately, is still expensive to compute for every frame. These authors note, however, that you only need to find silhouette changes in two consecutive frames. To accomplish this they identify the dual edges with one vertex inside and the other vertex outside the wedge formed by the dual planes of two consecutive viewpoints. Similar to Hertzmann and Zorin's method, to speed up this process they use an octree data structure. The advantage of this approach is that it works in 3D only. Hence, it performs the search for intersections only once per frame as opposed to eight times. Also, the approach only detects silhouette changes. However, although Hertzmann and Zorin's approach requires eight octrees instead of just one, each of the eight octrees only contains one-eighth of the faces. Also, both methods should have an expected extraction time linear to the number of found silhouette edges.

Sander et al. use a different method for silhouette edge detection.[17] This approach constructs a hierarchical search tree that stores the mesh's edges. All the faces attached to edges that are stored in a node or its associated subtree make up a face cluster. At runtime, the method searches the tree recursively to find face clusters that are entirely front or back facing. All edges of such a face cluster can be discarded and the search for the associated subtree stopped. To effectively test whether a face cluster is entirely front or back facing in constant time, Sander et al. store two anchored cones in every node (see Figure 9). One cone represents those positions where the viewpoint can be located for the entire face cluster to be front facing, the other cone for the face cluster to be back facing.

Summarizing, all precomputation methods presented here reduce the number of triangles or edges checked at runtime, speeding up the silhouette detection process without trading accuracy. They accomplish this by establishing an efficient data structure during preprocessing. All authors claim to achieve at least interactive frame rates for reasonably sized models. However, these methods make animation of the models inefficient since the precomputed data structure stores information about the visibility of polygons to quickly identify silhouettes for the moving viewpoint. If a model were animated beyond a flight through the scene, this precomputed data structure would become invalid and result in new precomputation steps for every frame. On current hardware, this would reduce the frame rate to below interactive rates. In addition, precomputation algorithms need a separate elaborate data structure besides the actual geometry for achieving their speedups.

Stochastic method. In contrast to precomputation, Markosian et al. suggest a stochastic algorithm to gain faster runtime execution of silhouette detection.[18] They observe that only a few edges in a polygonal model are actually silhouette edges. (Typically $O(\sqrt{n})$ edges of the n polygons according to Sander et al.[17].) In the hope of finding a good initial set of candidates for front- and back-face culling, they randomly select a small fraction of the edges and exploit spatial coherence. Once they detect a silhouette edge, they recursively test adjacent edges until they reach the end of the silhouette line. In addition, they also exploit spatial coherence, as the silhouette in one frame is typically not far from the (visually) similar silhouette in the next frame.

The combination of these two parts of the algorithm yields most of the silhouette edges in one image. By exploiting spatial and temporal coherence, Markosian et al. achieve fast runtime execution for interactive or real-time applications. Also, the method is not restricted to static objects, so animation does not pose a problem. However, in contrast to the precomputation algorithms we discussed previously, the algorithm can't guarantee finding the entire set of silhouette edges for a certain view on the scene.

Line visibility determination

In most cases the process of computing object space silhouettes produces the problem of visibility culling of the silhouette edges. This problem is the classic computer graphics problem of hidden line removal. Similar to the problem of silhouette detection, three general ways exist to attack this problem. A fast way to solve it is an image space approach. A highly precise solution employs an object space method that yields visible silhouette segments in an analytic description. Finally, by combining both approaches you can use a hybrid method that's faster than an object space method but still yields analytic descriptions of the visible silhouette lines.

Image space. A trivial and fast way to determine the visibility of silhouette edges in image space is to use the z-buffer. The simplest method is to render the sil-

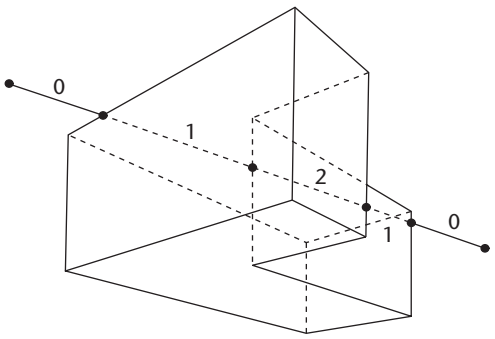
houette edges and let the z-buffer remove the hidden lines. The result is comparable to Raskar and Cohen's hybrid method using wireframe rendering in the second pass. A more advanced algorithm renders the silhouettes in a certain line width and perpendicular to the viewing direction. Unfortunately, this generates images with thick internal silhouettes and thin contour lines because the object itself partly occludes contour lines but not internal silhouettes.

Besides the limitation to pixel accuracy, the main disadvantage of this image space approach is that when you apply style variations to the silhouette the line might be partially occluded.

Object space. Researchers have proposed many algorithms for hidden line removal in object space (see, for example, Sutherland et al.[19]). Most of these also solve the problem of hidden surface removal. It would go beyond the scope of this article to address all of these algorithms, but you could use many of them to easily determine silhouette visibility. The visible line algorithm presented by Appel,[20] however, deserves further attention since it's frequently used in the context of silhouette algorithms. The algorithm is based on the notion of quantitative invisibility (QI)—the number of front-facing polygons between a point on the edge to be rendered and the viewer—which is determined for all edge segments (see Figure 10). All edge segments with a QI value of zero are visible; all others are invisible. The fact that the QI value only changes when the edge to be rendered intersects a silhouette edge in the 2D projection allows for propagation along connected sets of edges. (However, it can also change at a vertex if the vertex lies on a silhouette edge, which causes some complications for the computation.) Therefore, for every connected set of edges tested the QI value is initially established for one point using, for example, ray tracing. The QI value is then propagated along the edges to be rendered determining whether it increases or decreases each time the edge passes behind a silhouette edge.

Markosian et al.[18] use a modified version of Appel's algorithm to improve computation time. They redefine the QI value as the total number of faces between a point and the viewer. This simplifies Appel's algorithm in that the QI value can now only change at a vertex if that vertex is a so-called cusp vertex. In addition, they avoid many of the required ray tests by first finding out how the QI value changes by establishing relative QI values when traversing a connected set of edges before executing the ray test. Sometimes, this marks as invisible the whole connected set of edges, making it unnecessary to establish initial QI values. In the remaining cases, they avoid even more tests by inferring from QI values of one set of connected edges to those of others by using the relative QI values determined previously.

Hertzmann and Zorin apply this approach to their algorithm for subpolygon silhouettes.[2] First, they divide their silhouette curves into segments at points where the visibility can possibly change and determine visibility individually for each of these segments. Then, in addition to their subpolygon silhouette edges, they also determine regular (non-subpolygon) silhouette edges



10 Example for Appel's notion of quantitative invisibility for a line passing behind the object. The dots denote the positions where the QI value changes. Adapted from Appel.[20] (© 1967 ACM. Reprinted with permission.)

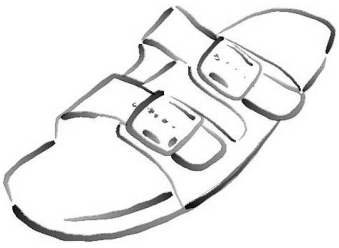
that are a subset of the input mesh. They determine edge visibility using the adapted Appel's algorithm described previously. Finally, they use the visibility of the majority of edges adjacent to a subpolygon silhouette edge segment to infer the segment's visibility. For finely tessellated objects this yields a sufficiently correct visible silhouette.

The advantage of analytic algorithms is they typically yield highly precise visibility information. However, since their computational complexity is not constant, they usually take longer to compute than image space approaches.

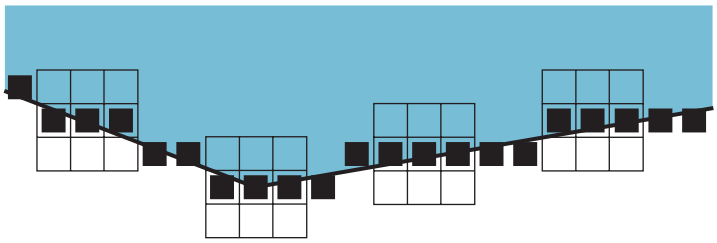
Hybrid. As we just showed, because they precisely solve the hidden line problem, object space silhouette visibility tests are usually expensive. A fast but less accurate way of determining silhouette edge visibility is an image space approach. However, in many applications only pixel accuracy is necessary. Thus, combining object space and image space approaches in a hybrid algorithm can achieve significant speedup for the visibility test.

In addition to regular rendering with a z-buffer, Northrup and Markosian use an ID buffer to determine silhouette edge visibility.[21] A unique color identifies each triangle and silhouette edge in this ID buffer. For each frame, the ID buffer is read from the graphics hardware and all reference image pixels are examined to extract all silhouette edges represented by at least one pixel. The approach then scan converts and checks for visibility the remaining silhouette edges according to whether a pixel with the edge's unique color exists in the ID buffer. Figure 11 shows an example rendering.

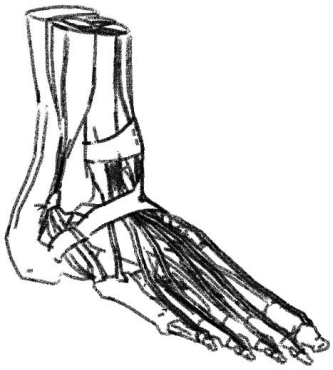
Isenberg et al. use a similar approach, in principal.[22]



11 Image generated using Northrup and Markosian's hybrid visibility test. Line styles are used for rendering silhouette strokes. (Courtesy of J.D. Northrup[21] © 2000 ACM. Reprinted with permission.)



12 Testing visibility by looking at the *z*-buffer values of the pixel and its 8-pixel neighborhood. Not every pixel is tested (in this case every fifth pixel is examined). (Courtesy of Tobias Isenberg, Nick Halper, and Thomas Strothotte, *Computer Graphics Forum*, published by Blackwell Publishing.[22] Reprinted with permission.)



13 Image generated by applying the hybrid *z*-buffer visibility test. (Courtesy of Tobias Isenberg, Nick Halper, and Thomas Strothotte, *Computer Graphics Forum*, published by Blackwell Publishing.[22] Reprinted with permission.)

However, they base their visibility test on the *z*-buffer instead of an additional ID buffer. This saves time for the second render pass otherwise required for the ID buffer. In addition, they note that simply scan converting silhouette edges and comparing the computed pixels with values in the *z*-buffer is somewhat numerically unstable. Thus, they suggest not only looking at the pixel's exact position but also at its 8-pixel neighborhood for pixels that are further away than the tested pixel (see Figure 12). This significantly reduces the numerical problems. To further speed up the process they only look at every *n*th pixel. This introduces a trade-off between accuracy and speed.

The result of either of these techniques is—as with object space visibility algorithms—a set of visible silhouette lines. We can now use them, for example, for stylized silhouette rendering. After concatenating the visible silhouettes to form paths, we can apply line styles and render them without an additional visibility test (Figure 13 shows an example). In contrast to image space visibility-tested silhouettes, style features that significantly distort the path are now completely visible.

Future work

The various algorithms presented here can guide developers who need to find a polygonal model's silhouette. Goals for future research could include, for example, making object animation compatible with certain precomputation algorithms for object space silhouette detection—that is, to find ways to efficiently update the data structure. This would open the category of object space silhouette detection to general applications that require large models and animation. Also, recent advances in computer graphics hardware could speed the process of silhouette visibility detection in hybrid methods or even support the computation of silhouettes directly. ■

Acknowledgments

We thank Stefan Schirra for many interesting discussions on the topic and Bert Vehmeier for his help with the images.

References

1. G. Elber and E. Cohen, "Hidden Curve Removal for Free Form Surfaces," *Proc. Siggraph 90*, *Computer Graphics* (Proc. Ann. Conf. Series), vol. 24, ACM Press, 1990, pp. 95-104.
2. A. Hertzmann and D. Zorin, "Illustrating Smooth Surfaces," *Proc. Siggraph 00*, *Computer Graphics* (Proc. Ann. Conf. Series), S.N. Spencer, ed., ACM Press, 2000, pp. 517-526.
3. T. Saito and T. Takahashi, "Comprehensible Rendering of 3-D Shapes," *Proc. Siggraph 90*, *Computer Graphics* (Proc. Ann. Conf. Series), F. Baskett, ed., ACM Press, 1990, pp. 197-206.
4. A. Hertzmann, "Introduction to 3D Non-Photorealistic Rendering: Silhouettes and Outlines," *Non-Photorealistic Rendering* (Siggraph 99 Course Notes), S. Green, ed., ACM Press, 1999.
5. O. Deussen and T. Strothotte, "Computer-Generated Pen-and-Ink Illustration of Trees," *Proc. Siggraph 2000*, *Computer Graphics* (Proc. Ann. Conf. Series), vol. 34, ACM Press, 2000, pp. 13-18.
6. J.L. Mitchell, C. Brennan, and D. Card, "Real-Time Image-Space Outlining for Non-Photorealistic Rendering," *Siggraph 02 Conf. Abstracts and Applications*, ACM Press, 2002, p. 239.
7. J. Loviscach, "Rendering Artistic Line Drawings Using Off-the-Shelf 3-D Software," *Proc. Eurographics: Short Presentations*, I.N. Alvaro and P. Slusallek, eds., Blackwell Publishers, 2002, pp. 125-130.
8. P. Rustagi, "Silhouette Line Display from Shaded Models," *Iris Universe*, Fall 1989, pp. 42-44.
9. J.R. Rossignac and M. van Emmerik, "Hidden Contours on a Frame-Buffer," *Proc. 7th Eurographics Workshop Computer Graphics Hardware*, Eurographics, 1992, pp. 188-204.
10. R. Raskar and M. Cohen, "Image Precision Silhouette Edges," *Proc. 1999 ACM Symp. Interactive 3D Graphics*, S.N. Spencer, ed., ACM Press, 1999, pp. 135-140.
11. B. Gooch et al., "Interactive Technical Illustration," *Proc. 1999 ACM Symp. Interactive 3D Graphics*, ACM Press, 1999, pp. 31-38.
12. R. Raskar, "Hardware Support for Non-Photorealistic Rendering," *Proc. 2001 Siggraph/Eurographics Workshop on Graphics Hardware*, ACM Press, 2001, pp. 41-46.
13. J.W. Buchanan and M.C. Sousa, "The Edge Buffer: A Data Structure for Easy Silhouette Rendering," *Proc. 1st Int'l Symp. Non-Photorealistic Animation and Rendering*, ACM Press, 2000, pp. 39-42.
14. D. Card and J.L. Mitchell, "Non-Photorealistic Rendering with Pixel and Vertex Shaders," *Vertex and Pixel Shaders Tips and Tricks*, W. Engel, ed., Wordware, 2002.

15. F. Benichou and G. Elber, "Output Sensitive Extraction of Silhouettes from Polygonal Geometry," *Proc. 7th Pacific Graphics Conf.*, IEEE CS Press, 1999, pp. 60-69.
16. M. Pop et al., "Efficient Perspective-Accurate Silhouette Computation," *Proc. 17th Ann. ACM Symp. Computational Geometry*, ACM Press, 2001, pp. 60-68.
17. P.V. Sander et al., "Silhouette Clipping," *Proc. Siggraph 2000*, *Computer Graphics* (Proc. Ann. Conf. Series), S.N. Spencer, ed., ACM Press, 2000, pp. 327-334.
18. L. Markosian et al., "Real-Time Nonphotorealistic Rendering," *Proc. Siggraph 97*, *Computer Graphics* (Proc. Ann. Conf. Series), T. Whitted, ed., Addison Wesley, 1997, pp. 415-420.
19. I.E. Sutherland, R.F. Sproull, and R.A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms," *Computing Surveys*, vol. 6, no. 1, 1974, pp. 1-55.
20. A. Appel, "The Notion of Quantitative Invisibility and the Machine Rendering of Solids," *Proc. ACM National Conf.*, Thompson Books, 1967, pp. 387-393.
21. J.D. Northrup and L. Markosian, "Artistic Silhouettes: A Hybrid Approach," *Proc. 1st Int'l Symp. Non-Photorealistic Animation and Rendering*, J.-D. Fekete and D.H. Salesin, eds., ACM Press, 2000, pp. 31-37.
22. T. Isenberg, N. Halper, and T. Strothotte, "Stylizing Silhouettes at Interactive Rates: From Silhouette Edges to Silhouette Strokes," *Computer Graphics Forum* (Proc. Eurographics 2002), vol. 21, no. 3, 2002, pp. 249-258.

Tobias Isenberg *is a PhD candidate in the Department of Simulation and Graphics at the Otto-von-Guericke University of Magdeburg, Germany. His research interests include feature detection on 3D shapes and nonphotorealistic rendering with an emphasis on silhouette algorithms, line stylization, and hybrid rendering. Isenberg received a BSc from the University of Wisconsin, Stevens Point, and a Diplom from the Otto-von-Guericke University of Magdeburg.*

Bert Freudenberg *is a PhD candidate in the Department of Simulation and Graphics at the Otto-von-Guericke University of Magdeburg. His research interests include real-time rendering, nonphotorealistic computer graphics, and interactive educational environments. Freudenberg received a Diplom from the Otto-von-Guericke University of Magdeburg.*

Nick Halper *is a PhD candidate in the Department of Simulation and Graphics at the Otto-von-Guericke University of Magdeburg. His research interests include nonphotorealistic rendering (modular systems, user interfaces, evaluating the influence of nonphotorealistic rendering), camera presentation (declarative specification, real-time techniques), and computer games (action summarization, nonphotorealistic rendering in games). Halper received an MEng in computer systems and software engineering from the University of York, England.*

Stefan Schlechtweg *is an assistant professor in the Department of Simulation and Graphics at the Otto-von-Guericke University of Magdeburg. His research interests include nonphotorealistic rendering, the application of nonphotorealistic techniques in interactive systems, and visualization. Schlechtweg received a PhD in computer science from the Otto-von-Guericke University of Magdeburg.*

Thomas Strothotte *is a full professor of computer science in the Department of Simulation and Graphics at the Otto-von-Guericke University of Magdeburg, where he is the Chair for Graphics and Interactive Systems. His research interests include nonphotorealistic rendering, image-text coherence in interactive systems, and techniques for rendering illustrations. Strothotte received a BSc and MSc from Simon Fraser University, a PhD in computer science from McGill University, as well as a Habilitation degree from the University of Stuttgart.*

Readers may contact Tobias Isenberg at the Otto-von-Guericke University of Magdeburg, Dept. of Simulation and Graphics, Universitatsplatz 2, 39106 Magdeburg, Germany; isenberg@isg.cs.uni-magdeburg.de.