EUROGRAPHICS 2002 / G. Drettakis and H.-P. Seidel
(Guest Editors)

Volume 21 (*2002*), *Number 3*

Stylizing Silhouettes at Interactive Rates: From Silhouette Edges to Silhouette Strokes

Tobias Isenberg, Nick Halper, and Thomas Strothotte

Department of Simulation and Graphics
Otto-von-Guericke University Magdeburg
{isenberg|nick|tstr}@isg.cs.uni-magdeburg.de

Abstract

A way to create effective stylized line drawings is to draw strokes that start and stop at visible portions along the silhouette of an object to be portrayed. In computer graphics to date, algorithms to extract silhouette edges are many, although putting these edges into a form such that stylized strokes may be applied to them has not been greatly covered, so that existing methods are either time-consuming or presented vaguely. In this paper, we introduce an algorithm that takes a set of silhouette edges originating from polygonal meshes and efficiently computes the visible parts of the edges before connecting them to form long smooth silhouette strokes to which stylization algorithms may be effectively applied. Features of our algorithm that gain efficiency and accuracy over existing methods is that we directly exploit the analytic connectivity information of the mesh in combination with the available z-buffer information during rendering, and filter artifacts in connected edges during the process to improve the visual quality of strokes after stylization.

Categories and Subject Descriptors (according to ACM CCS):
I.3.3 [Computer Graphics]: Picture/Image Generation—Line and curve generation
I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Hidden line/surface removal

1. Introduction

Line drawings have been an important part in the area of artistic expression and scientific visualization (see, for example, HODGES[9]) for a long time. In particular, silhouette drawings allow artistic expression even with a very limited number of strokes. One can also emphasize specific parts of an image, such as by using stylized lines, in order to focus the viewer's attention[17, 20]. Although there has been considerable research in the area of computer-generated line drawings there is still a lot to be done in order to create stylized line drawings for interactive or even real-time applications.

In order to successfully apply styles across silhouettes, the silhouettes of a scene must be presented to a line stylization algorithm as a series of connected line segments. In this way, stylizations applied to a silhouette of an object to be drawn start and stop at visible portions (those that are not occluded by other parts of the scene) along the silhouette, which simulates long smooth strokes that an artist may have drawn. In addition, the importance of determining visible line segments becomes apparent when we consider that line styles may apply perturbations to the original line. This then allows lines to be rendered to the scene without depth testing because the perturbations may overlap or cross over other objects (such as thicker lines or path variations).

For clarity, we define those edges that mark the border between front-facing and back-facing polygons as *silhouette edges* and the visible parts thereof as *visible silhouette segments*. Since the visibility test used in this paper can be applied to silhouette edges and certain feature edge lines, visible silhouette segments may also comprise those visible portions of edges that are tagged as sharp (often called creases or valley and ridge lines), so we do not distinguish between these. Connected visible silhouette segments form *silhouette strokes*. A *silhouette* applies to the union of visible silhouette segments specific to an object or the scene. Note that a silhouette can also include inner lines (those that would not show up in an objects shadow) as opposed to a contour (that traces an object's shadow).

In this paper we introduce a new approach to determine which portions of silhouette edges are visible based on a *z*-buffer[4] lookup, and manipulating the resulting set of silhouette segments to produce a set of silhouette strokes that are adapted for a stylization algorithm in such a manner that they can be rendered 'on top' of a scene. We begin by highlighting the major contributions of the paper:

- a fast and robust image-precision hidden line removal algorithm adapted for silhouette edges that scan-converts edges and tests against depth information available in the *z*-buffer, requiring no additional rendering of the scene such as an ID-buffer,
- the retainment of local edge connectivity information from silhouette segment computation for efficient concatenation into silhouette strokes,
- the adaptation of silhouette strokes for stylization algorithms to increase visual quality by filtering artifacts, and
- the combined computation time of our algorithm allows for interactive frame-rates.

Section 2 covers related work in the area of silhouette rendering. In Section 3 we describe the silhouette stroke generation algorithm in detail. The performance and the yielded results of the work presented is examined in Section 4. Section 5 draws a conclusion from the discussion of the algorithm and in Section 6 we evaluate directions of future work in the area.

2. Related Work

For our work we require that we have found silhouette edges. A number of techniques are able to be used for this purpose. EFRAT et al.[5] discuss the computational bounds for silhouette computation. However, they impose tight restrictions on the objects they consider (only convex polytopes) and the movement of the viewpoint (only a straight line or an algebraic curve). MARKOSIAN et al.[11, 10] use a randomized approach that is fast but does not guarantee to find all silhouette edges. BENICHOU and ELBER[2], GOOCH et al.[6], HERTZMANN and ZORIN[8], SANDER et al.[18] as well as POP et al.[15] use pre-computation for optimized silhouette edge extraction, although this is only applicable to objects that do not change form, requiring re-computation whenever an object's shape is altered. BUCHANAN and SOUSA[3] compute silhouette edges by use of an edge-buffer that optimizes with respect to cache hits. Any of these techniques can be used for generating silhouette edges as the input our algorithm. For a survey of silhouette extraction algorithms see HERTZMANN[7].

Computerized extraction of visible silhouette segments has been studied in detail only recently. There are two major techniques to do this, image-based and analytic. Image-based techniques take advantage of the image buffers that can be generated quickly using hardware acceleration. RASKAR, for example, renders front-facing and back-facing polygons separately so that the back-facing polygons render the edge lines, and the front-facing polygons occlude those edge lines that do not form part of the silhouette[16]. However, because this is all happening in hardware to the image buffer, lines cannot be stylized beyond thickness or color.

Analytic algorithms (as, for example, described by SECHREST and GREENBERG[19]) have a high computational complexity and are not suited for interactive applications. MARKOSIAN et al.[11] use an adaption of APPEL's algorithm that is based on determining the *quantitative invisibility* (QI) for points. The QI value might only change when one silhouette line crosses another one in 2D or at certain so-called *cusp vertices*. Determining visible parts requires clipping of silhouette edges against others and several *ray tests* to establish initial QI values. This technique works best for smooth surfaces.

Silhouette strokes have been implemented in some off-line systems that analytically generate high quality stylized renderings (such as the dali! system[12, 13]). Only recently techniques have been introduced for interactive renderings. NORTHRUP and MARKOSIAN describe a hybrid approach[14]. Visibility of edges is determined by searching in a reference image (ID buffer) for connected paths of edges close together. However, their algorithm does not make use of the analytic connectivity information that was available from the original mesh. In contrast, the algorithm presented in this paper makes direct use of the available analytic edge information present in a winged-edge data structure[1], thereby sparing the computationally expensive reconstruction of connectivity information by searching for near-by edges in image space.

3. Silhouette Stroke Generation

Given an algorithm that generates all the relevant silhouette edges, this section describes the way we determine the set of visible silhouette segments from silhouette edges, how to connect them to form silhouette strokes, and how to reduce artifacts that might occur during the process.

This is broken down into four steps:

1. removing redundant silhouette edges that could cause artifacts,
2. determining visible silhouette segments (visibility test),
3. generating silhouette strokes from visible silhouette segments, and
4. removing image-plane artifacts of silhouette strokes.

Determining relevant silhouette edges can be done by using any of the algorithms mentioned in Section 2. We are using a brute force approach since we want to guarantee that all silhouette edges are found and so that results can be tested on animations. We describe the determination of visible silhouette segments and stroke generation (steps 2 and 3, respectively) first, and then focus on the removal of artifacts (steps 1 and 4).

3.1. Determining Visible Silhouette Segments

From our set of silhouette edges we need to determine which segments of those edges are visible to the viewer. We introduce a new technique that does not require rendering of an ID-buffer (such as done by NORTHRUP and MARKOSIAN[14]), but makes direct use of available *z*-buffer information. Therefore, we may render the scene as normal (or enable writing to the *z*-buffer only), and then use that information for visibility testing. Thus, if we want to render the scene geometry in combination with silhouettes, we obtain the generation of *z*-depth information for free.

To determine visible silhouette segments, we take each silhouette edge separately, and scan-convert the edges in relation to the *z*-buffer, testing for point visibility along the edge. In a naïve approach, to determine the visibility of a point, one could just compute its *z*-value and compare it with the value in the *z*-buffer. However, the results are highly instable due to *z*-buffer precision and pixel quantization. Since we want to check the visibility of silhouette edges we will often look at parts of the image buffer where many faces are almost parallel to the viewing direction. Hence, these faces may share the same *z*-buffer locations as the silhouette edges, but will overwrite those locations with values closer to the viewer. The naïve test will fail under these circumstances, and falsely detect an edge as occluded (see Figure 1).

To take advantage of the availability of the rendered *z*-buffer, we choose to employ a more reliable point visibility test along a silhouette edge. We adapt *z*-buffer silhouette extraction techniques (e.g. SAITO and TAKAHASHI[17]) so that we not only look at the exact position of a point to determine its visibility but to also look at its 8-neighborhood. If there is any point in this 8-neighborhood where the *z*-buffer value is farther away than the computed *z*-value of the point, we declare it visible, and not visible otherwise (see Figure 2).

This works because silhouette edges occur in regions with a discontinuity in *z*-depth. Hence, it is almost certain that we find one of these background pixels by looking in the 8-neighborhood. Those silhouette edges that are occluded by larger areas will not find any pixels further away, and be correctly identified as occluded. Therefore, the numeric instability of the *z*-value computation is greatly reduced.

For speeding up the process we do not need to test visibility for each pixel in the edge scan conversion. Instead, we are able to parameterize the number of pixels that we may skip along this process (example in Figure 2). The advantage of this is that we are able to tune a speed vs. accuracy trade-off for our needs. Should we find that two adjacent test points in the scan conversion of an edge intersect a boundary between visible and occluded, then we subdivide the scan-conversion process until we reach a one-pixel accuracy for the location where we need to clip the edge. The consequence of skipping pixels could be that we draw a line through an occluder that lies in-between two visibility test points in the scan conversion of an edge.

Additional speed-up is obtained through the optimization of the extraction of *z*-buffer values. This is done both by caching results (eliminating re-accessing the same values) and reading larger chunks at a time (minimizing function-call overhead and bus-rate transfer). The size of the chunks to read is largely dependent on the performance issues of the hardware. On our machine it has shown to be practicable to read chunks of eight by eight pixels at a time and store them into the main memory. Only in cases where the image exhibits a high density and distribution of silhouette edges does it become profitable to read the entire area of the *z*-buffer that the scene covers.

As a result we have the set of clipped silhouette edges that are now defined as the visible silhouette segments. Next we combine the visible silhouette segments into a set of silhouette strokes.

3.2. Generating Silhouette Strokes

A silhouette stroke is a concatenation of visible silhouette segments that pairwise share a common vertex. Therefore, strokes terminate when there are no more segments that may be joined by a common vertex. There are cases where more than two visible silhouette segments share a common vertex (e. g., think of a cusp vertex), and here we adopt the approach that continues the stroke along the segment with the lowest angle to it (as done by NORTHRUP and MARKOSIAN[14]).

To generate these strokes, we make use of a winged-edge data structure that provides local connectivity information. We start with an arbitrary visible silhouette segment, and search in both directions for additional connecting visible silhouette segments, concatenating them in each iteration. Visible silhouette segments that do not comprise a whole silhouette edge may either form a silhouette stroke by themselves (if they are located in the middle of a silhouette edge) or may start or end a silhouette stroke (if they are at the beginning or end of one silhouette edge). This is simple and efficient because the visible silhouette segments that we have generated in the previous stages of our algorithm correspond directly to the original mesh with the edge connectivity information intact. Thus, no additional processing, such as rendering an edge ID buffer, is necessary to find connectivity between edges.

3.3. Artifact Removal

Since meshes do not provide perfect representations of surfaces and due to numerical instabilities in silhouette edge detection we often get overlapping silhouette edges and small zig-zags in image-space (of course, there might also be zig-zags due to the local shape of an object). These have to be removed in order to achieve good looking results when

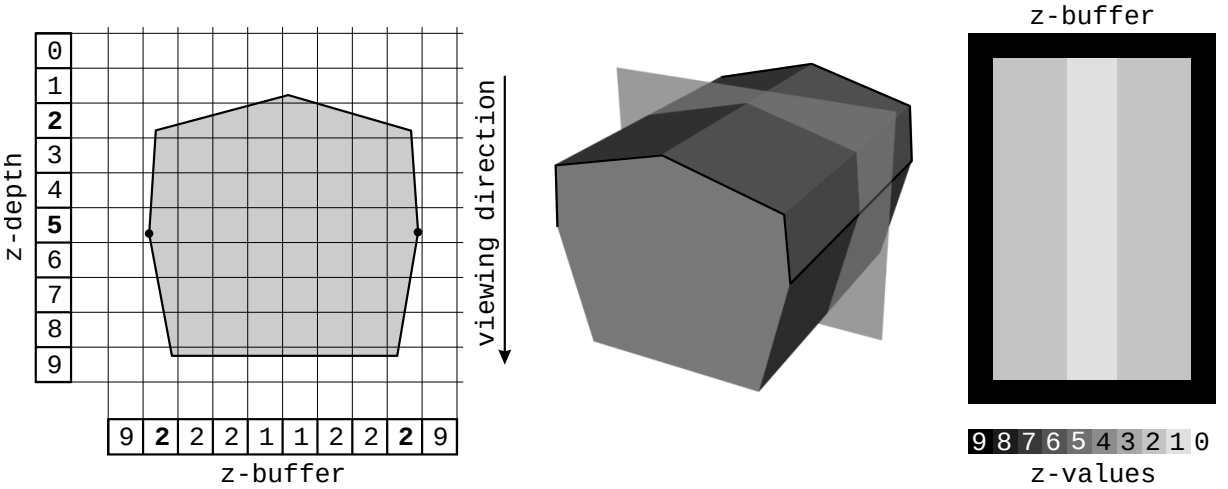


Figure 1: *Problem with faces almost parallel to the viewing direction (indicated by the arrow). Although the z-value of the silhouettes (indicated by dots in the left image) is 5 the value of the related pixels' z-buffer is 2. The left image shows a cut through the model as shown in the middle (with silhouette lines emphasized) and right part shows the resulting z-buffer image.*

applying styles to strokes, as will be described in the following sections. The visibility-check algorithm requires that we scan-convert edges to the rendered information in the *z*-buffer, thus we may not alter the position of the edge vertices prior to this process. Therefore, there are two stages in artifact removal—one before the visibility processing of silhouette edges that retains the original vertices for edges, and one after visible silhouette segments have been found that might merge or alter vertices. The first two artifacts outlined below are treated in the first stage, and the subsequent three are handled after silhouette stroke generation.

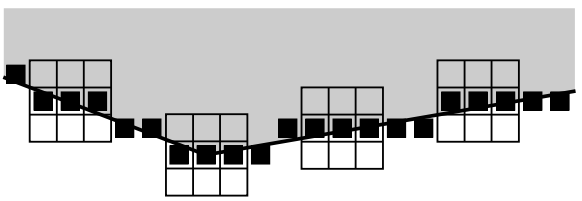


Figure 2: *Testing the visibility by looking at the z-buffer values of the pixel and its 8-neighborhood. Also, not every pixel is tested (in this case every fifth pixel is examined).*

Triangles With Two Visible Edges

Sometimes it happens that two silhouette edges share the same triangle (depicted in Figure 3) such that they cause sharp zig-zag lines when projected to the image plane. Applying a style in form of a texture to these zig-zag lines will result in awkward artifacts, due to the edges turning back on themselves. However, because a triangle with two silhouette edges is most likely to be almost parallel to the viewing direction, we may avoid the artifacts created by un-marking the two formerly marked silhouette edges and marking the remaining edge of the shared triangle instead (see Figure 3). This gives the result that we now have one new silhouette edge that covers the same span as the previous two, but with the zig-zag artifact removed. However, before this artifact reduction is applied, the orientation of the the triangle is tested whether it is oriented parallel to the viewing direction. Only when the angle between triangle and viewing direction is below a certain small angle (we used ≈ 3 degrees) the marked edges are switched. This avoids unwanted effects when both silhouette edges have to be visible (e. g., think of a triangle facet of a cube).

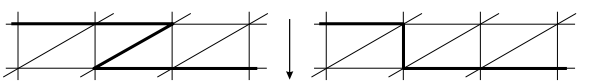


Figure 3: *Example for handling triangles with two silhouette edges before and after the update. The arrow indicates the viewing direction.*

Silhouette Edge Clusters

Another problem also arises due to small numerical errors and triangle surface approximation that can occur when computing the visibility of faces that are almost parallel to the viewing direction. In these cases one might encounter lots of adjacent triangles that alternate between front-facing and back-facing, thereby creating clusters of silhouette edges (Figure 4 shows an example; one can find clusters of silhouette edges, for example, at the silhouette of cylinder-like shaped objects). This may result in many short strokes whereas one long stroke would be favorable.

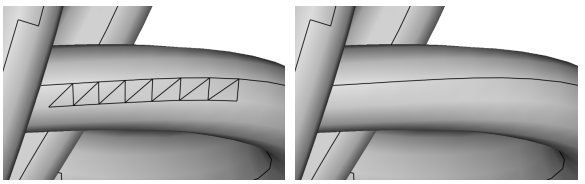


Figure 4: *Example for treating silhouette edge clusters. In the screen-shots the upper part is visible and the lower part is not. The triangles with all edges marked alternate from not visible to visible. To remove the cluster, a path around it is detected, the inner edges are removed, and the longest segment of the path between the two leaving edges is removed. The left and right images show, respectively, the situation before and after the update.*

One way to remove these silhouette edges clusters is to find each triangle that comprises three silhouette edges. Next, we find the closed path of outer silhouette edges around such groupings of triangles. We eliminate the silhouette edges inside the closed path loop. Now because we have a loop, when viewed from the viewpoint, we will get overlapping silhouette edges, so we may remove one side of the loop in a manner so that a single connected path joins all the silhouette edges leaving the group of triangles (as an example, in Figure 4 we remove the path comprising the most silhouette edges). This is achieved by detecting the longest path (the one with the highest number of edges) between two neighboring leaving edges. Of course, if there is only one triangle in a triangle cluster (such as a tetrahedron with only one side visible) this remains untouched.

Silhouette Stroke Zig-Zags

After the projection into 2D, again due to numerical instabilities, there can be zig-zags in the generated strokes (see Figure 5). We classify parts of a stroke as comprising zig-zags for each segment that has two sharp angles formed by its adjacent segments. If we find such a case, we replace the visible silhouette segment by just a vertex located in the middle of this segment. This typically eliminates the zig-zag.

Figure 5: *Example for removing zig-zags, left side before and right side after the update. These zig-zags might originate from cases similar to the one displayed on the right side of Figure 3 due to a viewing direction which was changed just slightly in the counter-clockwise direction and minimal numerical errors during the visibility test. These can occur very easily because the viewing direction is almost parallel to the triangles.*

Sharp Angles

Similar problems as with zig-zag lines can arise when there are acute angles in a stroke depending on the stroke width. Consider Figure 6. On the left side we see that over the length of the segment the normal of the stroke texture is twisted by almost 90 degrees. Also, because of the high stroke width the thickness at the vertex of the sharp angle is much smaller than at the other end. Both effects yield an unappealing image. Therefore, we decide to split the silhouette stroke at the point of the sharp angle. This means to stop the stroke in the vertex of the sharp angle and start a new one there for the remainder of the former stroke. This simulates an artist drawing one stroke, stopping, and then drawing back over his stroke (with a thick drawing utensil).

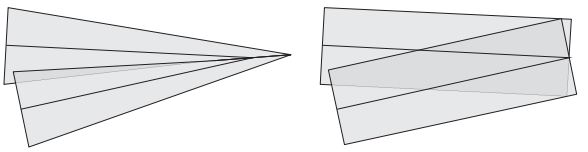


Figure 6: *Example for splitting silhouette strokes at sharp angles. The left side shows the situation before the update where one stroke is twisted awkwardly. On the right side, after the update, the stroke has been split into two separate strokes yielding a better rendering.*

Short Segments

Sometimes stroke segments with screen-projection lengths less than a few pixels are generated. We can improve the stroke quality by merging sub-pixel and short strokes together. Note that this eliminates many sharp angles occurring at the sub-pixel level that would influence the orientation changes and texturing of a stroke, whilst also acting as a level-of-detail feature by reducing geometry (see Figure 7).

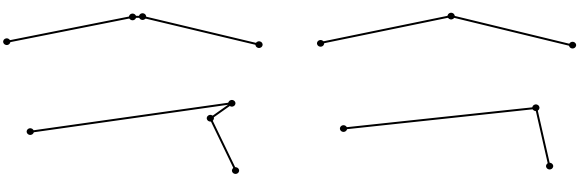


Figure 7: *Example for removing short segments, left side before and right side after the update.*

4. Results

The described algorithms are implemented in C++ within a modular non-photorealistic rendering and animation system. All the proposed stages of the algorithm and artifact removal techniques are implemented as modules that can be combined with stroke stylization modules as part of the rendering pipeline.

4.1. Artifact Reduction

We found that the artifact reduction techniques discussed in Section 3.3 improved the appearance of the final image. The slight displacements to silhouette strokes and segments might alter the real silhouette but do not significantly distort the line drawing. In contrast, for the application in stylized rendering they are necessary to yield appealing images (for applications that rely on an exact silhouette the artifact removal can be skipped). In Figure 8 we can see the local improvements to the silhouette rendering. When no artifacts are removed at the geometric level, the projection of zig-zags and short segments create noticeable unwanted properties in the line stylization as depicted in Figure 8(a). Figures 8(b) and 8(c) show elimination of visual artifacts before and after visible silhouette segment processing, respectively. However, only when we combine both artifact removal stages do we get the smooth consistent outline present in Figure 8(d).

4.2. Computation Times

We provide a tuning mechanism so that we can decide upon a trade-off between accuracy and speed (see Table 1) by providing one parameter that determines the number of pixels we skip in the silhouette edge visibility scan conversion process. Determining this number depends on the resolution of the image (the higher the resolution, the more pixels we may skip) and the geometric properties of the scene (objects that cover small areas would require reducing the number of pixels we skip). The extravagance of styles may also be taken into account (wavy lines, for instance, will overlap nearby objects anyway). We have found that a skipping value of six pixels has negligible visual impact for most cases.

In Tables 2 and 3 we show the computation times for various stages of the rendering pipeline for a number of objects and polygon counts. Here we see that the major bottleneck in the resulting frame-rate is the silhouette edge detection. However, this can be reduced by using any of the more efficient algorithms as reviewed in Section 2.

The computation times for the silhouette stroke generation is dependent on the number of 8-neighborhood tests (i. e., number and length of silhouette edges, the pixel coverage of the rendered object, and the skipping value). In Table 2 we note that computation time across the models is relatively constant, except for the foot model because the silhouette is more complex. Higher resolution representations of models also yield higher computation times for the silhouette stroke generation because often silhouette edges are very short, to which the skipping value cannot be applied. Altering skipping values gives computation times as shown in Table 1. The dependency of computation time for silhouette strokes (including stroke segment generation) on the size of the rendered image is demonstrated in Table 4.

The pre-visibility artifact removal sequentially processes all silhouette edges, hence computation times are dependent on the number of extracted silhouette edges. Post-visibility artifact removal and stylization operate over the generated silhouette strokes, but since strokes are comprised of segments this computation time is dependent on the number of visible silhouette segments.

4.3. Examples

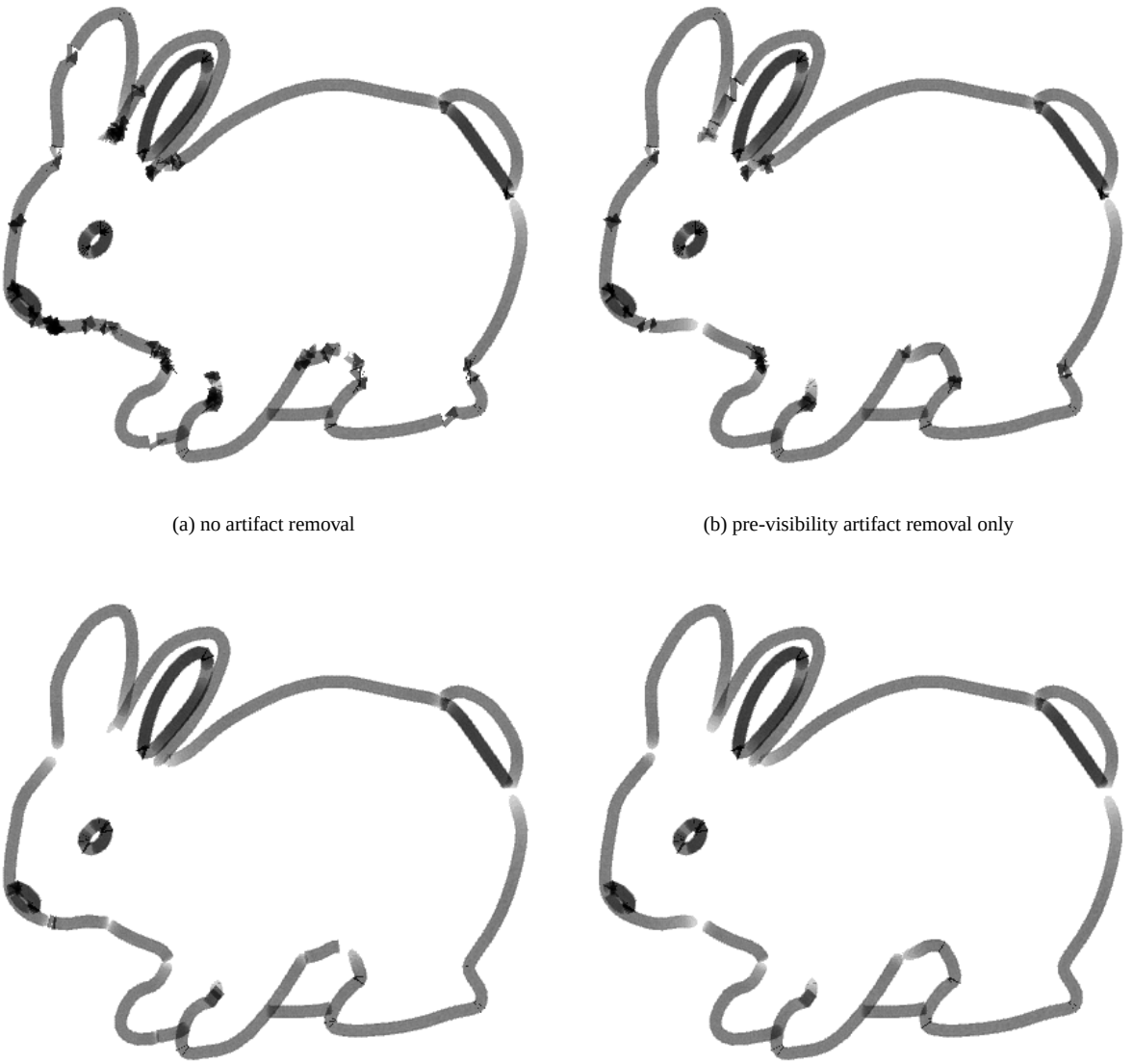
Figure 9 shows the application of different line styles to the silhouette strokes generated from our non-photorealistic rendering system. The line-styles can be modified interactively, such as varying the line width and stroke textures. Perturbation functions (such as applying a sinusoidal wave over the strokes) are also possible, using both image-space or object-space algorithms. Image space algorithms compute styles based on viewport projections whereas object-space algorithms can actually operate in three dimensions, such as varying thickness according to depth.

5. Conclusion

In this paper we have presented an algorithm for visibility culling of silhouette edges and generating silhouette strokes for application in stylized rendering at interactive framerates. Its input is the result of any algorithm that computes silhouette edges from a polygonal mesh. The method introduced enables us to do a fast visibility check by combining analytic edge information available in the form of a winged-edge data structure with the depth information in the z-buffer without having to render an additional image buffer. This also allows us to connect silhouette strokes from only those silhouette segments that are continuous on the object topology. The fact that this yields higher quality images becomes apparent when considering two silhouette segments that are continuous in the image plane (so that their endpoints touch or overlap each other) but are actually disjoint in object space (so that they represent different parts of the object or separate objects). Our approach would render two distinct strokes (one for each object silhouette) whereas former approaches that reconstruct strokes from the image buffer only[14] would draw a single stroke spanning over the two disjoint silhouette segments. Our approach can also be easily extended to allow multiple styles to be rendered across the scene since we can just tag the analytic edges accordingly.

A number of artifact removal techniques have been shown that filter the silhouette strokes before they are sent to the stroke stylization algorithm. These modified silhouettes might differ slightly from the originally computed silhouette, but will yield better images when styles are applied.

Computation times were recorded across a variety of models for each separate stage of our algorithm, demonstrating that interactive rates are achievable. When used to create animations (see accompanying videos), our algorithm performs well in maintaining strokes over successive



(a) no artifact removal

(b) pre-visibility artifact removal only

(c) post-visibility artifact removal only

(d) all artifact removal techniques

Figure 8: *Examples artifact removal. Remaining gaps are due to the applied line style.*

skipping value (pixel)	1	2	3	4	5	6	7	8	9	10
time (ms)	13.1	12.1	12.0	11.7	11.5	10.8	10.4	10.7	10.5	10.6

Table 1: *Computation times for the silhouette stroke generation (including stroke segment generation) depending on the skipping value for the bunny model with 3608 polygons and an approximate rendered size of 295 x 223 pixels.*

frames, although no explicit frame-coherent measures are taken. Sometimes popping occurs in stroke textures due to sudden varying lengths or start and endpoints of silhouette strokes. This occurs most noticeably when new silhouette

model	PN	SED		PrVAR		SSG		PoVAR		STY		total		fps	
		1P	2P	1P	2P	1P	2P	1P	2P	1P	2P	1P	2P	1P	2P
bunny	902	2	2	1	0	11	7	2	1	7	4	23	14	25.1	42.7
bunny	3608	7	7	1	1	11	6	3	2	7	4	29	20	21.7	32.0
bunny	14432	29	27	4	2	11	7	4	3	11	6	59	45	11.7	15.5
bunny	57728	114	108	8	6	14	9	9	5	11	6	156	134	4.8	5.8
knot	2880	4	4	1	0	12	7	2	1	6	3	25	15	24.9	42.7
knot	11520	21	21	2	1	12	7	4	2	11	6	50	37	14.3	21.4
foot	11244	24	21	8	7	22	13	15	7	35	17	104	65	7.5	10.7

Table 2: *Computation times in milliseconds tested on a 450 MHz PentiumIII machine with a GeForce2MX graphics board and 256 MB RAM (1P, left columns) and a 800 MHz Dual-PentiumIII machine with a GeForce1 graphics board and 384 MB RAM (2P, right columns), both running MS Windows 2000 (PN = polygon number, SED = silhouette edge detection, PrVAR = pre-visibility artifact removal, SSG = silhouette stroke generation (including stroke segment generation), PoVAR = post-visibility artifact removal, STY = stylization). The skipping value was set to six pixels and the approximate rendered size was 295 x 223 pixels. The frame-rates given are total tested rates and less than to be expected from the computation times because they also include other activity like the standard OpenGL rendering of the model.*

model	PN	SED		PrVAR		SSG		PoVAR		STY	
		1P	2P	1P	2P	1P	2P	1P	2P	1P	2P
bunny	902	8.7	14.3	4.3	0.0	47.8	50.0	8.7	7.1	30.4	28.6
bunny	3608	24.1	35.0	3.4	5.0	37.9	30.0	10.3	10.0	24.1	20.0
bunny	14432	49.2	60.0	6.8	4.4	18.6	15.6	6.8	6.7	18.6	13.3
bunny	57728	73.1	80.6	5.1	4.5	9.0	6.7	5.8	3.7	7.1	4.5
knot	2880	16.0	26.7	4.0	0.0	48.0	46.7	8.0	6.7	24.0	20.0
knot	11520	42.0	56.8	4.0	2.7	24.0	18.9	8.0	5.4	22.0	16.2
foot	11244	23.1	32.3	7.7	10.8	21.2	20.0	14.4	10.8	33.7	26.2

Table 3: *Computation times in percent according to the data in Table 2 (abbreviations as in Table 2). Differences from a sum of 100% are due to rounding errors.*

size (pixel)	523 x 393	295 x 223	170 x 130	92 x 72	43 x 34
time (ms)	23.2	10.8	5.8	3.8	2.7

Table 4: *Computation times for the silhouette stroke generation (including stroke segment generation) depending on the (approximate maximal) size of the rendered image for the bunny model with 3608 polygons and a skipping value of six.*

segments are added to strokes in regions where the object comprises long edges in the geometry, and when strokes are split by occlusion from other objects. Popping artifacts are inherent to silhouettes generated from course models and can be improved by subdividing the model into finer triangles. To handle sudden occlusion of strokes to maintain frame-coherency would require more investigation. In practice, however, our results so far show that even with applied perturbations to the strokes the resulting animation maintains a degree of frame-coherency, with only the occasional visual artifact (such as popping lines).

Unfortunately, the algorithm sometimes does not perform well when trying to determine the visibility of sharp concave feature lines. In these cases the 8-neighborhood test often fails because most tested points in the adjacent faces will be closer to the viewer than the edge itself. However, this effect can be reduced by using a polygon offset (`glEnable(GL_POLYGON_OFFSET_FILL)` and `glPolygonOffset()` in OpenGL). Also, in cases where two independent silhouette edges are touching or almost touching, such that they lie at exactly the same z-distance from the viewer and separated from each other by

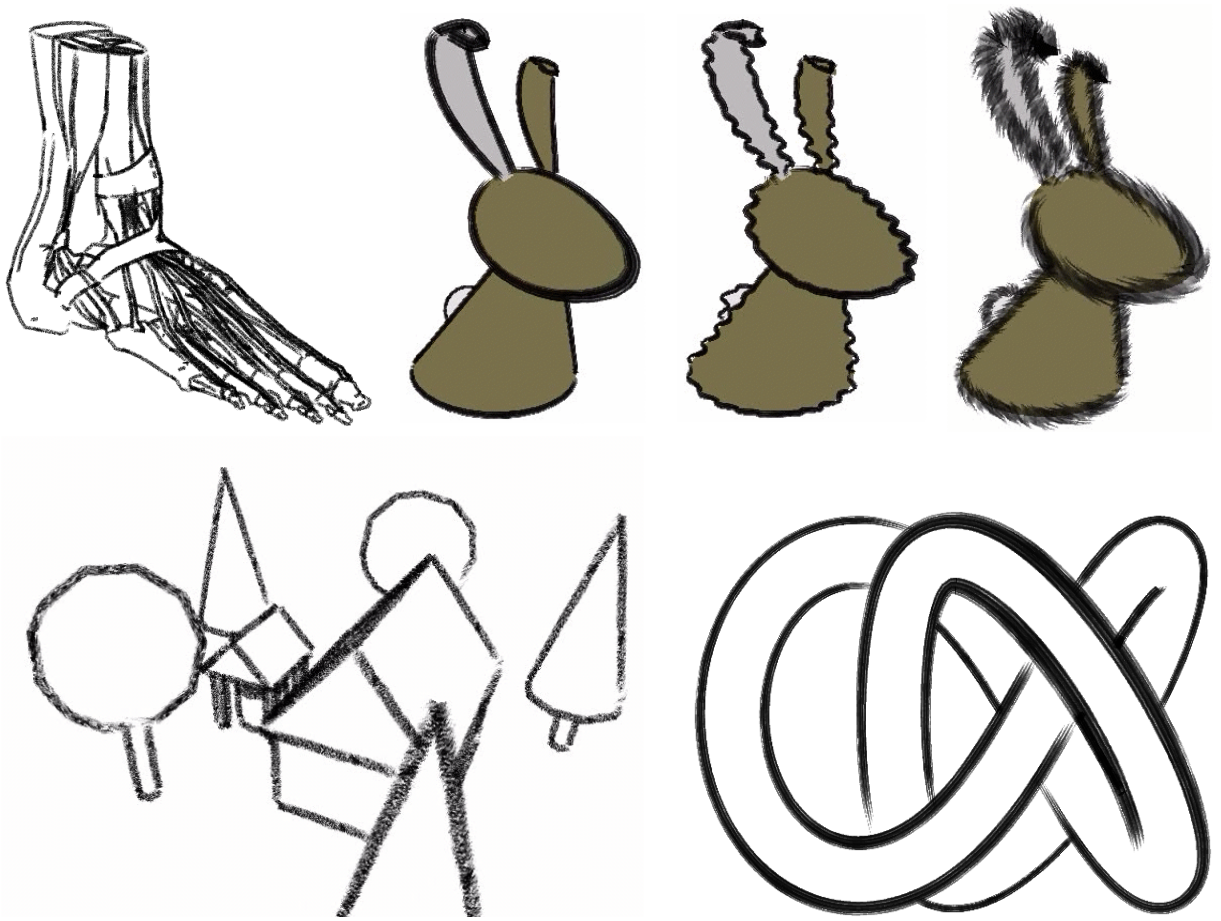


Figure 9: *Examples for applying different line styles to the generated silhouette lines.*

less than one pixel in the frame-buffer, they might not pass the silhouette segment visibility test. The risk of this occurring, however, is small.

6. Future Work

Besides the silhouette edge detection algorithm, the bottleneck in the silhouette stroke generation algorithm is the process of reading of data from the *z*-buffer (note this bottleneck is present only if using graphics hardware for the *z*-buffer generation). Currently, we already make sure not to read the same information twice by storing read information in memory and testing this. However, depending on the size of the *z*-buffer, the amount of information to be allocated can be large (typically 2-4 MB). We could reduce the amount of buffer while also optimizing memory with respect to cache hits by clipping edges to pre-defined regions of the viewport and processing each region in turn.

For high resolution models or sub-divided models a grouping of lines to one bigger line and then checking the bigger line instead could make the algorithm more resolution independent.

Acknowledgments

The authors wish to thank Andreas Raab and their colleagues at the Department of Simulation and Graphics, especially Bert Freudenberg, Henry König, and Stefan Schlechtweg, for various discussions on the topic of this paper.

References

1. Bruce G. Baumgart. A Polyhedral Representation for Computer Vision. In *Proceedings National Computer Conference*, pages 589–596, 1975.
2. F. Benichou and Gershon Elber. Output Sensitive Extraction of Silhouettes from Polygonal Geometry. In *Proc. 7th Pacific Graphics Conference*, pages 60–69, Seoul, Korea, 1999.

3. John W. Buchanan and Mario C. Sousa. The Edge Buffer: A Data Structure for Easy Silhouette Rendering. In *NPAR 2000: Symposium on Non-Photorealistic Animation and Rendering*, pages 39–42, New York, 2000. ACM.

4. Edwin Catmull. Computer Display of Curved Surfaces. In *Procedures of the IEEE Conference on Computer Graphics Pattern Recognition and Data Structures*, 1975.

5. Alon Efrat, Leonidas J. Guibas, Olaf A. Hall-Holt, and Li Zhang. On Incremental Rendering of Silhouette Maps of a Polyhedral Scene. In *Proc. 11th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 910–917, San Francisco, CA, 2000.

6. Bruce Gooch, Peter-Pike J. Sloan, Amy Gooch, Peter Shirley, and Richard Riesenfeld. Interactive Technical Illustration. In *1999 ACM Symposium on Interactive 3D Graphics*, pages 31–38. ACM, 1999.

7. Aaron Hertzmann. *Non-Photorealistic Rendering*, chapter Introduction to 3D Non-Photorealistic Rendering: Silhouettes and Outlines. SIGGRAPH 99 Course Notes. ACM Press, 1999.

8. Aaron Hertzmann and Denis Zorin. Illustrating Smooth Surfaces. In *Proceedings of SIGGRAPH 2000 (New Orleans, LA, July 23–28, 2000)*, *Computer Graphics* Proceedings, Annual Conference Series, pages 517–526. ACM SIGGRAPH, 2000.

9. Elaine R. S. Hodges, editor. *The Guild Handbook of Scientific Illustration*. Van Nostrad Reinhold, New York, 1989.

10. Lee Markosian. *Art-based Modeling and Rendering for Computer Graphics*. PhD thesis, Brown University, May 2000.

11. Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. Real-Time Nonphotorealistic Rendering. In Turner Whitted, editor, *Proceedings of SIGGRAPH 97 (Los Angeles, CA, August 3–8, 1997)*, *Computer Graphics* Proceedings, Annual Conference Series, pages 415–420, Reading, MA, 1997. ACM SIGGRAPH, Addison Wesley Publishing Company.

12. Maic Masuch, Stefan Schlechtweg, and Bert Schönwälder. daLi! – Drawing Animated Lines! In Oliver Deussen and Peter Lorenz, editors, *Simulation und Animation '97*, pages 87–95, Delft, Belgium, 1997. SCS–Society for Computer Simulation Int., SCS Europe.

13. Maic Masuch, Lars Schumann, and Stefan Schlechtweg. Animating Frame-to-Frame-Coherent Line Drawings for Illustrative Purposes. In Peter Lorenz and Bernhard Preim, editors, *Simulation und Visualisierung '98*, pages 101–112, Delft, Belgium, 1998. SCS–Society for Computer Simulation Int., SCS Europe.

14. J. D. Northrup and Lee Markosian. Artistic Silhouettes: A Hybrid Approach. In Jean-Daniel Fekete and David H. Salesin, editors, *Proceedings of First International Symposium on Non Photorealistic Animation and Rendering (Annecy, France, June 5–7, 2000)*, pages 31–37. ACM SIGGRAPH / Eurographics, 2000.

15. M. Pop, G. Barequet, C. A. Duncan, M. T. Goodrich, W. Huang, and S. Kumar. Efficient Perspective-Accurate Silhouette Computation. In *Proc. 17th Ann. ACM Symp. on Computational Geometry*, pages 60–68, Medford, MA, 2001.

16. Ramesh Raskar and Michael Cohen. Image Precision Silhouette Edges. In Stephen N. Spencer, editor, *Proceedings of 1999 ACM Symposium on Interactive 3D Graphics*, pages 135–140, New York, 1999. ACM Press.

17. Takafumi Saito and Tokiichiro Takahashi. Comprehensible Rendering of 3-D Shapes. In Forest Baskett, editor, *Proceedings of SIGGRAPH 90 (Dallas, Texas, August 6–10 1990)*, *Computer Graphics* Proceedings, Annual Conference Series, pages 197–206, New York, 1990. ACM SIGGRAPH, ACM Press.

18. Pedro V. Sander, Xianfeng Gu, Hugues Hoppe Steven J. Gortler, and John Snyder. Silhouette Clipping. In *Proceedings of SIGGRAPH 2000 (New Orleans, LA, July 23–28, 2000)*, *Computer Graphics* Proceedings, Annual Conference Series, pages 327–334, New York, 2000. ACM SIGGRAPH, ACM Press.

19. S. Sechrest and D. P. Greenberg. A Visible Polygon Reconstruction Algorithm. In *Proceedings of SIGGRAPH 81 (Dallas, Texas, August 3–7 1981)*, *Computer Graphics* Proceedings, Annual Conference Series, pages 17–27, New York, 1981. ACM SIGGRAPH, ACM Press.

20. Thomas Strothotte, Bernhard Preim, Andreas Raab, Jutta Schumann, and David R. Forsey. How to Render Frames and Influence People. *Computer Graphics Forum*, 13(3):455–466, September 1994.