To appear in the proceedings of NPAR 2000

Artistic Silhouettes: A Hybrid Approach

J.D. Northrup and Lee Markosian

Brown University, Providence, RI 02912

Abstract

We present a new algorithm for rendering silhouette outlines of 3D polygonal meshes with stylized strokes. Rather than use silhouette edges of the model directly as the basis for drawing strokes, we first process the edges in image space to create long, connected paths corresponding to visible portions of silhouettes. The resulting paths have the precision of object-space edges, but avoid the unwanted zig-zagging and inconsistent visibility of raw silhouette edges. Our hybrid screen/object space approach thus allows us to apply stylizations to strokes that follow the visual silhouettes of an object. We describe details of our OpenGL-based stylized strokes that can resemble natural media, but render at interactive rates. We demonstrate our technique with the accompanying still images and animations rendered with our technique.

CR Categories and Subject Descriptors: I.3.3 [Computer Graphics]: Picture/Image Generation - Display algorithms

Additional Key Words: Strokes, non-photorealistic rendering

1 Introduction

The outline, or silhouette, of a shape is often one of its most striking features. Our work attempts to render attractive silhouette outlines for 3D geometry in real-time, creating brush-strokes resembling natural media along well-chosen paths around each object. This breaks down into three distinct phases. First, we must determine where the silhouette edges are. Because silhouettes are inherently view-dependent, we have to find them every time the scene or camera changes. Second, we need to choose where we want to place each stroke. We want to pick paths that will look good and remain as consistent as possible from one frame to the next. Third, we draw each stroke in a style defined by the user. This last step should be flexible enough to enable the user to achieve almost any artistic goals she may have in mind.

In Section 2, we review several known silhouette detection algorithms to address the first phase. The remainder of this paper focuses on the second two phases. Section 5 describes our algorithm for determining stroke paths, and Section 6 explains our artistic stroke-rendering framework for producing final images.

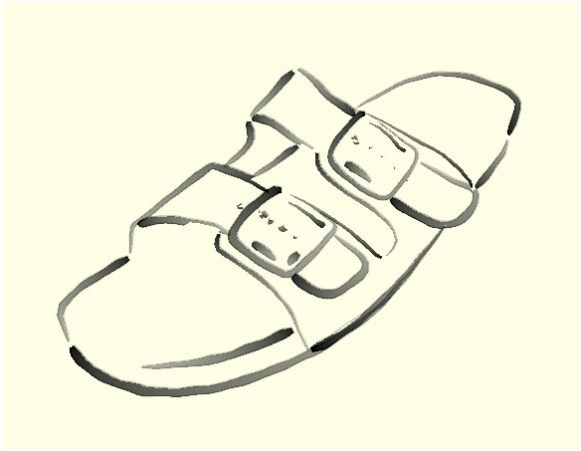


Figure 1 An example of an ink-wash style rendered with our algorithm.

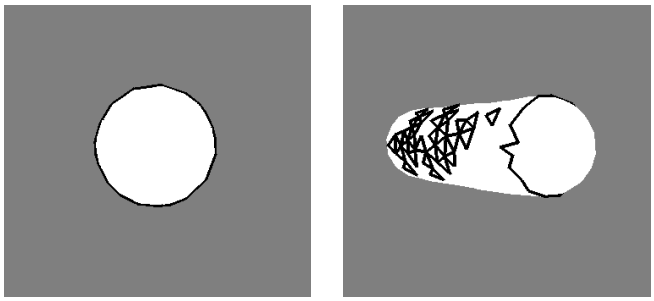
2 Silhouette Edge Detection

Our system begins with 3D scenes consisting of standard triangle meshes. We need to analyze the structure of these models to determine which edges form silhouette outlines. We define a **silhouette edge** to be an edge that connects a front-facing triangle to a back-facing triangle. Because this condition depends both on the camera viewpoint and the state of the model, we must compute these silhouette edges every frame that the world changes.

The brute-force approach to finding the silhouette edges simply checks every edge of the mesh, every frame. This may suffice for high-quality, non-interactive animations which can afford to sacrifice speed for guaranteed results, but causes a major bottleneck in real-time applications like ours.

Several algorithms for rapidly finding the silhouette edges already exist. For a model with n edges and k silhouette edges, the method used by Gooch *et al.* [5] performs the detection of all silhouette edges in $O(k \log n)$ by precomputing a spherical hierarchy data structure, but their algorithm is rather complicated and difficult to implement. Furthermore, reliance on expensive pre-computation makes this algorithm inappropriate for use with changing meshes.

We use the randomized algorithm presented by Markosian *et al.* [11]. Taking advantage of temporal coherence, this algorithm uses the silhouettes found in the previous frame as a starting point for a search of the current frame. For a mesh with n edges, we randomly select a small fraction of edges to test. When a new silhouette is found, its neighbors are also checked for local continuation of the silhouette contour, leveraging the spatial coherence of silhouettes. Further details are supplied in [10]. In our experience, this algorithm has proven efficient, simple to code, and robust enough for our real-time applications.



(a) (b)

Figure 2 Current algorithms render the innocent-looking silhouette of (a) by drawing the complicated mess of overlapping edges shown in (b).

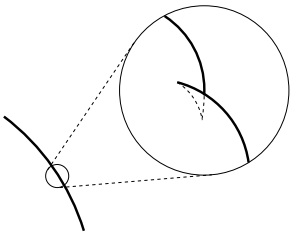


Figure 3 Small zig-zags called swallowtails often occur along silhouette profiles.

3 Problems Rendering Silhouette Edges

The second phase should render the raw silhouette edges found in the previous step in some pleasing way. One straightforward approach for implementations using a traditional API such as OpenGL [1] is to turn on *z*-buffering and render the edges as line strips. This is simple and fast, but does not allow stylization along the silhouettes. When using the *z*-buffer, anything drawn at the silhouette edges will be clipped against the body of the mesh. Ideally, we would disable the *z*-buffer and somehow draw strokes only along edges which are already known to be visible.

Furthermore, a few well-placed strokes often express the shape of a figure much more elegantly than a crowd of shorter marks. With this in mind, we seek to simplify the silhouette edges extracted from our models into paths that we can use to draw long, smooth strokes. Working purely in object space, it's hard to determine where these paths overlap, so we end up over-drawing the silhouette, as shown in Figure 2.

Also, the silhouette edges are connected to each other in ways that often reflect the small-scale structure of the mesh—not the overall shape of the object's outline. Paths created by joining up silhouette edges will contain small but frequent zig-zags that can cause unwanted artifacts when rendering them as strokes. These zig-zags occur at intersections called swallowtails where the silhouette temporarily reverses direction to connect two overlapping edges. This phenomenon is illustrated in Figure 3.

4 Image-Based Solutions

One class of algorithms addresses the problems inherent to working with silhouette edges by ignoring them altogether. Image-based silhouette-rendering algorithms avoid explicitly finding the 3D silhouette edges and instead opt to use 2D image-processing techniques. Notably, the work of Saito and Takahashi [16] renders the outlines of 3D objects by applying edge-detection filters to specially prepared depth and normal maps, and compositing the results with the rest of the scene. However, such approaches suffer from aliasing as the silhouette positions jump from pixel to pixel in the image, because the silhouette positions are not accurately tied to the underlying geometry. Furthermore, these algorithms do not easily allow the use of stylized strokes. Curtis [4] has introduced a technique for generating strokes along these pixel outlines, but using only pixel data sacrifices precision, especially when trying to decide how to join several intersecting curves. Also, techniques requiring multipass filtering and compositing are often too slow for real-time applications. Raskar and Cohen [15] present a geometry-based approach suitable for real-time use which also avoids explicitly finding silhouette edges, but their methods do not allow for stylized strokes.

These image-based techniques inherently focus on depicting only the portions of the silhouettes that contribute to the final 2D image. Like any good artist, these algorithms never even *consider* explicitly depicting every single silhouette edge, so they avoid the problems due to overlap and sub-pixel swallowtails. Furthermore, they do not need *z*-buffering because the visibility is already known, allowing systems like Curtis's to apply stylizations without worrying about strokes being clipped.

5 A Hybrid Algorithm

Our work combines the benefits of the image-based approach with the accuracy of a geometry-based approach. As in the latter, we begin by detecting the silhouette edges of the model, but then we compute visibility and adjacency using a 2D projection of the silhouette edges. This lets us maintain the precision of object coordinates while still working in 2D where that makes most sense. At a high level, the algorithm proceeds as follows:

1. Find silhouette edges.
2. Determine visible segments of each edge.
3. Apply correction for overlaps in segments.
4. Link segments into smooth paths.
5. Render stylized strokes along these paths.

In what follows, the term **edge** refers to a silhouette edge of the mesh; a single visible portion of an edge (such as found in step 2 above) is called a **segment**; and a collection of segments that form a continuous sequence in image space is called a **path**.

Step 1 above is discussed in Section 2. In the remainder of this section we describe steps 2 through 4. The final step is described in Section 6.

5.1 Extracting Visible Segments

Once we have detected the silhouette edges in the current frame, the next step is to determine which portions of the silhouette edges are visible. Our method makes use of the "ID reference image" discussed in a previous paper [9]. Briefly, we create the ID reference image by first rendering the scene with each silhouette edge and each mesh triangle drawn in a color that uniquely identifies it, then reading this image from the framebuffer into memory. Details on how to use the ID reference image to determine visibility can be found in [10].

We next iterate over all the pixels in the reference image and build a list L of edges that contributed at least one pixel. We thus remove from consideration silhouette edges that make no contribution to the pixels of the current frame (such as many of the silhouette edges seen in Figure 2 (b)).

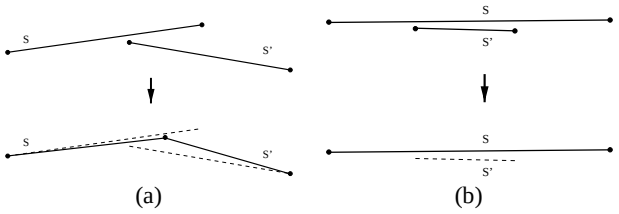


Figure 4 In (a), segments S and S' are corrected by redefining their overlapped endpoints to midpoint between them. The angle between the segments is exaggerated for clarity. An undesired segment is eliminated in (b).

Next we scan-convert along each edge in L to determine which portions of it show up in the ID reference image, and hence are visible.[1] We record each such visible portion, or *segment*. A segment consists of its two image-space endpoints and a pointer to the associated edge. The endpoints need not project exactly onto the edge; for example, a segment may extend across a pixel even when its associated edge occupies just a small fraction of the pixel in the current view.

To determine whether an edge e "shows up" in the ID reference image at a given image space point x on e, we check the ID reference image at x, and also at nearby points (within two pixels in practice) along the image-space line perpendicular to e and passing through x. If e shows up anywhere along this line, we consider e to be visible at x. In addition, we record a list of neighboring edges encountered along this line for use in the next step of the algorithm.

5.2 Correcting For Overlaps

The segments are intended to be linked together to form long, connected image-space paths that will serve as the basis for stylized strokes. Before we perform this linking step, we first carry out two correction steps that promote longer and smoother paths. These are shown in Figure 4.

We "merge" segments that overlap and are nearly parallel; and we eliminate a segment if it is adjacent and nearly parallel to another segment, and the first segment is shorter. In both cases, we consider two edges nearly parallel if the angle between them is less than 1 degree.

5.3 Linking Segments into Paths

At this point we have a collection of segments that together closely approximate the visible silhouettes of the scene. The next step is to link these segments into long chains, or paths, that will form the basis for the strokes. To do this, we first search near each segment's endpoints for potential neighbors. The search is a $n \times n$-pixel local search in the reference image.[2] We perform a series of tests to compute the suitability of each potential match between segment s and neighbor n, outlined in Figure 5.

Whenever we link a pair together, we keep a list of "divorced" segments, i.e., any segments that the new pair had previously been linked to. Once we have tried to find neighbors for each edge, we allow these divorced segments another chance at linking up.

[1] It can happen that a visible portion of an edge does *not* show up in the ID reference image. This might happen for example if the edge is too small to contribute to the rasterization of the image. For our purposes, it is sufficient to find just those visible portions that appear in the ID reference image.

[2] In our system, $n = 3$.

```
CHECK-MATCH(s, n)
    θ ← angle between s and n
    θ_max ← max angle allowed to link
    D ← distance between endpoints of s and n
    D_max ← max distance allowed to link

    if s is already linked to n
        reject n
    if θ ≥ θ_max
        reject n
    if endpoints of s and n don't overlap
    and D ≤ D_max
    and θ ≤ angle of s's current neighbor (if any)
    and θ ≤ angle of n's current neighbor (if any)
        link s and n
```

Figure 5 The CHECK-MATCH function determines whether segments s and n are suitable for being linked to each other. We use $\theta_{max} = 45^\circ$ and $D_{max} = 2$ pixels.

5.4 Rendering Paths

The final phase of our algorithm renders each newly-created path using an "artistic stroke." These strokes are defined in image space, and the visibility of the silhouettes they represent is already assured, so we can disable depth testing and safely draw the strokes with various image-space stylizations. Enabling depth testing would preclude the use of such stylizations, since we cannot reliably assign depth values to parts of the stroke affected by the stylization. We now go into more detail about how these strokes are built and displayed.

6 Fast Artistic Strokes for 3D Scenes

Ultimately the success of any art-based rendering system depends on producing appealing images. Thus the final step of our algorithm takes great care to allow for a wide range of expressive strokes. To convincingly mimic traditional 2D illustration, our strokes should appear foremost as marks on a flat drawing surface—not objects floating in 3D space. From this vantage point, we may choose to selectively reintroduce hints of depth and distance, which we discuss below in Section 6.2.3.

On a practical level, this goal necessitates using a coordinate system in which stroke proportions will reflect screen distance, rather than 3D world distance. We use a variant of what the OpenGL reference manual labels "device coordinates" [1], normalized to preserve the aspect ratio of the drawing area. In other words, our coordinates range from -1 to 1 across the smaller screen dimension, and from $-d$ to d across the larger dimension, where d is the aspect ratio of the window. Unlike a pure screen-space coordinate system, our points retain their z component, allowing us to render strokes using traditional depth buffering if desired. This z-buffered approach is used to render the strokes outlining the graftals of Kowalski *et al.* [9, 12], and the strokes of Cohen, Zeleznik and Hughes' user-drawn world [3]. For the algorithm described in this paper, we only place strokes where we already know they will be visible, so we can draw without z-buffering.

6.1 Creating Basic Strokes

Given a list of vertex/width pairs, we would like to render a stroke that passes through each vertex, smoothly transitions between the given widths, and joins corners to create a continuous path. Unfortunately OpenGL does not support lines of varying width and leaves large gaps between corners of thick line strips. These problems led

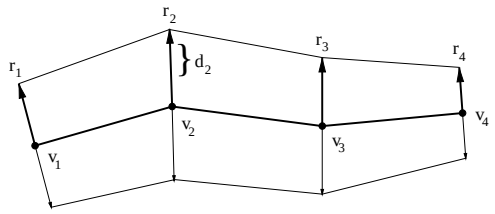


Figure 6 Constructing rib vectors $\vec{r}_i$ to add width to a four-vertex stroke.

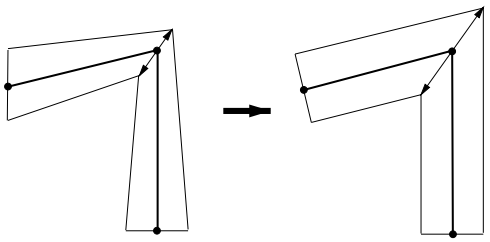


Figure 7 Effect of scaling rib vector to maintain constant path width.

us to create a variable-width line primitive by leveraging OpenGL's speed and flexibility at drawing long triangle strips. We proceed with a method inspired by the work of Hsu *et al.* [7, 8].

For each path vertex v_i and its corresponding pixel width w_i, we generate a vector "rib" $\vec{r}_i$ along the angle bisector which allows us to give breadth to the stroke. This is illustrated in Figure 6. Figure 7 demonstrates the need to scale the length of these ribs to maintain a desired path width. The scale factor for each rib is computed as

$$f = \left| \frac{|\vec{r}_i|}{\vec{r}_i \cdot \vec{n}_i} \right|$$

where $\vec{n}_i$ is the normalized vector perpendicular to the the path direction between v_i and v_{i+1}. Intuitively, this means that the ribs are scaled wider at sharper corners, and left alone at straight segments. We limit the amount of scaling to a factor of 2, since otherwise the miter for a very sharp corner would be too large.

Once we have a suitable set of ribs, we render the stroke as a series of triangle strips. Point pairs for the strips are generated by offsetting each path vertex v_i by its corresponding rib vector $\vec{r}_i$, and its opposite $-\vec{r}_i$. The end result of this phase is a variable-width line strip with nicely joined corners. Now we need to liven up the strokes by adding artistic effects.

6.2 Stylistic Variations

In our system, the designer decides on a combination of stylization operations to apply to the strokes for a given object. All of the style options can be freely mixed and matched, allowing for a wide range of expressiveness. Figure 8 shows how each operation affects a stroke. Since each stroke must be rebuilt every frame, we have tried to make sure that these operations add little overhead to the overall stroke building and rendering burden. We will now describe three categories of stylizations possible with our system.

6.2.1 Resolution-Dependent Stylizations

For the first type of stylization, we perturb the appearance of the stroke along its length. To do this, we need to ensure that the screen distance between adjacent vertices does not exceed some specified maximum (in practice, 2-3 pixels). Ideally, we would resample the stroke to have perfectly even segments using an interpolating spline curve. In practice, we linearly divide each individual path segment,

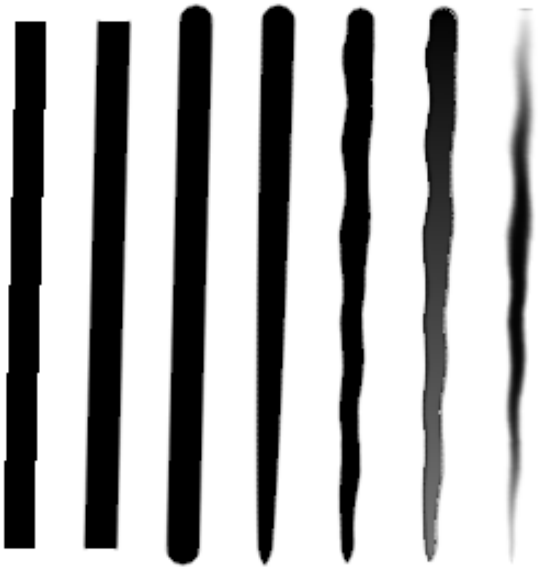


Figure 8 The cumulative effects of adding stroke operations, from left to right: raw stroke, antialiasing, taper, flare, wiggle, alpha fade, and texture-mapping.

adding evenly-spaced vertices to approximate the desired overall spacing. This leaves the original shape of the path undisturbed, and keeps this phase quick.

Once we have a sufficiently fine sampling of stroke vertices, we can perturb their locations and widths to achieve an uneven, hand-drawn look. Our current system applies offsets created by the user with the help of a separate tool. This allows our strokes to reflect the individual character of the user's lines. These strokes are similar to the ones described in [11].

Another style adds an alpha fade in which we linearly increase the transparency along the length of each stroke, creating a simple watercolor or ink-wash feel, as seen in Figure 1. One other operation flares the overall width of the stroke from end-to-end to create a brush-stroke shape. The flare function we use is

$$f = \sqrt{1 - t^2} \qquad \text{where} \qquad t = \frac{\text{current vertex index}}{\text{max vertex index}}$$

6.2.2 Other Stylizations

The second category of operations does not require fixed spacing. These include antialiasing, tapering the ends of each stroke to a point, and applying texture maps along the length of the stroke. Our implementations of OpenGL only support one antialiased primitive: the 1-pixel-wide line. Luckily, we can use this to simulate smoothed triangle strips by placing an antialiased line-strip around the boundary of the stroke. This adds virtually no overhead, and we can know that the antialiasing will register correctly because the line-strips use the exact same coordinates as the body of the stroke.

Tapering the ends of strokes is a bit more complex, but this is an important effect to simulate. First, we ensure a sufficient distribution of vertices to achieve a gradual thinning near the ends of each stroke. We insert vertices along the beginning and ending segments, linearly interpolated in the manner mentioned above. If we are using any of the operations from the previous section, we have already performed an overall interpolation, and may skip this step. Next we scale the rib size of these taper vertices by a function similar to the

Figure 9 A simple architectural rendering that uses wiggle, flare, and tapering.

flare function given above. The result is a smoothly rounded ending for each stroke.

A second style of tapering adds vertices *beyond* the endpoints of the stroke using the direction of the final segment. This creates a rounded extension to the stroke that can be used to hide seams at stroke boundaries. This second taper style is used to render all of the examples in this paper, and addresses the same problem treated by Gooch *et al.* [6], who used fat dots to hide the problem.

One final effect is to stretch a texture map over the length of the stroke. Smoothly faded strokes are easy to draw using this method. We simply apply textures with large regions of transparency (see Figure 12).

6.2.3 Depth and Distance Cue

Often traditional illustration styles give perspective cues by decreasing line weight for distant objects. We mimic this effect by modifying line widths based on two scaling factors: distance cue and depth cue.

The first modifies the overall stroke width based on the distance to an object. As the object recedes, the silhouettes gradually thin; as the object approaches the camera, the silhouettes widen. When the silhouette-rendering algorithm is first applied to a mesh, we compute the value D_i, which is the initial ratio of object-space length to screen-space length at the origin of the mesh. In successive frames, we compute the current scaling factor D_c, and multiply the scaling factor

$$f_D = \sqrt{\frac{D_c}{D_i}}$$

with the width of each stroke used to render that mesh. This is a non-linear scale in order to soften the effects of the distance cue.

The second pass varies the width of the strokes as the depth *within an object* varies. This provides a simple cue to the foreshortening of different parts of the mesh, similar to an effect demonstrated by Gooch [6]. Every frame, we compute the frame-buffer z-value bounds of each mesh, z_{min} and z_{max}. We would like to scale the width of each stroke vertex v based on its depth, z_v, so that the foremost vertices are scaled by a factor $1 + S$, and the rearmost vertices by $1 - S$. We compute the scale f_v for vertex v as

$$f_v = \max(0, 1 + S\frac{z_{max} + z_{min} - 2z_v}{z_{max} - z_{min}}) \qquad \text{where} \qquad 0 \leq S \leq 1$$

and multiply the width of each vertex by it. Unlike f_D, this f_v varies linearly with depth to enhance the localized, intra-object perspective hint.

7 Discussion

In the end, the success of any non-photorealistic rendering system rests on the quality of its final rendered images. We find our view-dependent stroke framework robust, fast, and very flexible for creating effective real-time illustrations. Similar uses of tapering artistic strokes can be found in the pen-and-ink work of Salisbury, Winkenbach, Salesin and others [17, 18], and previous work here at Brown University [11, 9, 3].

The results of using our artistic stroke renderer to depict the paths found by our silhouette-extraction algorithm are shown in Figures 1 and 9 - 12. Figures 11 and 12 can be found in the Color Plates section. These examples range from simple outlines to highly stylized brushwork, giving an idea of the flexibility offered.

Our system is fast enough for interactive use. On a high end Sun workstation, our frame rate varies from about 2 fps for scenes like the house in Figure 9 with many silhouettes, to 10 fps for simpler models like the frog hand or the sandal (Figures 10 and 1). Antialiasing our stroke primitives greatly enhances image quality, especially noticeable in frame-by-frame renderings (see the accompanying animations for examples). However, animations of the scene in Figure 12 reveal small, unwanted strokes that sometimes appear in regions of negative curvature. In these cases, our visibility and placement algorithms have decided to render edges that we find unattractive. It is possible that we could create additional tests to filter out these edges, but this is left as future work.

Another limitation of our algorithm is that it does not make any attempt to remember where it drew silhouettes in previous frames, much less how it parameterized the stylizations used in those strokes. This was a conscious decision in order to simplify and speed our algorithm. We gain some degree of coherence from our reliance on mesh structure, and provide line styles which hide inconsistencies between frames. For styles with uniform line character, such as in Figure 10, we can produce smooth, temporally-coherent animations, but for animations of styles in which each stroke is more noticeably varied, such as in Figure 12, lack of inter-frame coherence is often distracting. For high frame-rates, this randomness becomes much more noticeable, but in some cases it can become a desirable aesthetic effect [4].

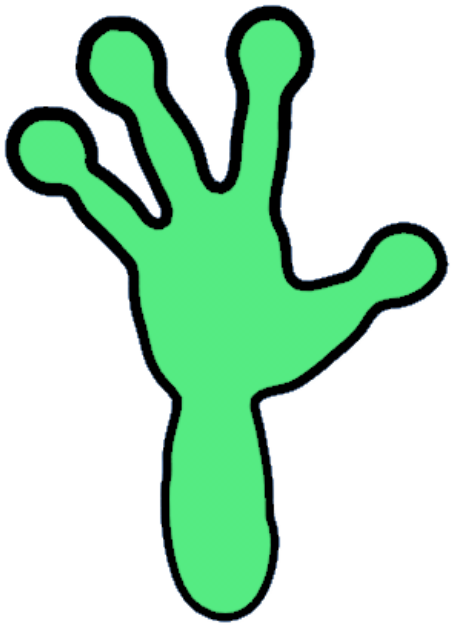


Figure 10 A frog hand. Stroke width varies as a depth cue.

8 Future Work

We know of related algorithms that have attempted to provide control over inter-frame coherence, notably the work of Bourdev [2], and Masuch *et al.* [13, 14]. One possible direction for future work would integrate the control afforded by these algorithms with the aesthetic details of our present work.

One of the advantages of our algorithm is that it works for arbitrary closed meshes, but often we would like to include detail that has been previously annotated by a user. We cannot expect an algorithm to choose the "right" strokes to draw a scene because there is no way to decide what those strokes would be. For a realistic renderer, there *is* a verifiable standard of what the right answer should look like: the real world.

However, a non-photorealistic/art-based rendering system should not impose the standards of the "real world" on our images. Like traditional artistic media, such a system should allow the user the possibility of expressing anything she wants, in any way she wants. It is our hope that the work presented in this paper provides tools that will help realize this goal, but much work still needs to be done before an artist will feel at home using such a system.

9 Acknowledgements

We thank Jun Ohya and Ryohei Nakatsu of ATR Labs for invaluable advice and support. This work is supported in part by the NSF STC for Computer Graphics and Scientific Visualization, Adobe, Advanced Network and Services, Alias/Wavefront, Department of Energy, IBM, Intel, Microsoft, National Tele-Immersion Initiative, Sun Microsystems, and TACO.

References

[1] OpenGL Architecture Review Board. *OpenGL Reference Manual, 2nd Edition*. Addison-Wesley Developers Press, 1996.

[2] Lubomir D. Bourdev. Rendering Nonphotorealistic Strokes with Temporal and Arc-Length Coherence. Master's thesis, Brown University, May 1998.

[3] Jonathan M. Cohen, John F. Hughes, and Robert C. Zeleznik. Harold: A World Made of Drawings. In *Proceedings of the First International Symposium on Non Photorealistic Animation and Rendering (NPAR) for Art and Entertainment*, June 2000. Held in Annecy, France.

[4] Cassidy Curtis. Loose and Sketchy Animation. Siggraph '98 Technical Sketch, 1998.

[5] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A Non-Photorealistic Lighting Model for Automatic Technical Illustration. *Proceedings of SIGGRAPH 98*, pages 447–452, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

[6] Bruce Gooch, Peter-Pike J. Sloan, Amy Gooch, Peter Shirley, and Rich Riesenfeld. Interactive Technical Illustration. *1999 ACM Symposium on Interactive 3D Graphics*, pages 31–38, April 1999. ISBN 1-58113-082-1.

[7] S. C. Hsu, I. H. H. Lee, and H. E. Wiseman. Skeletal Strokes. In *Proceedings of UIST '93*, pages 197–206, November 1993.

[8] Siu Chi Hsu and Irene H. H. Lee. Drawing and Animation Using Skeletal Strokes. *Proceedings of SIGGRAPH 94*, pages 109–118, July 1994. ISBN 0-89791-667-0. Held in Orlando, Florida.

[9] Michael A. Kowalski, Lee Markosian, J. D. Northrup, Lubomir Bourdev, Ronen Barzel, Loring S. Holden, and John Hughes. Art-Based Rendering of Fur, Grass, and Trees. *Proceedings of SIGGRAPH 99*, pages 433–438, August 1999. ISBN 0-20148-560-5. Held in Los Angeles, California.

[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. *Proceedings of SIGGRAPH 97*, pages 415–420, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[12] Lee Markosian, Barbara J. Meier, Michael A. Kowalski, Loring S. Holden, J. D. Northrup, and John F. Hughes. Art-based Rendering with Continuous Levels of Detail. In *Proceedings of the First International Symposium on Non Photorealistic Animation and Rendering (NPAR) for Art and Entertainment*, June 2000. Held in Annecy, France.

[13] Maic Masuch, Stefan Schlechtweg, and Bert Schönwälder. daLi! - Drawing Animated Lines! In *Proceedings of Simulation und Animation '97*, pages 87–96. SCS Europe, 1997.

[14] Maic Masuch, Lars Schuhmann, and Stefan Schlechtweg. Frame-To-Frame-Coherent Line Drawings for Illustrated Purposes. In *Proceedings of Simulation und Visualisierung '98*, pages 101–112. SCS Europe, 1998.

[15] Ramesh Raskar and Michael Cohen. Image Precision Silhouette Edges. *1999 ACM Symposium on Interactive 3D Graphics*, pages 135–140, April 1999. ISBN 1-58113-082-1.

[16] Takafumi Saito and Tokiichiro Takahashi. Comprehensible Rendering of 3D Shapes. *Computer Graphics (Proceedings of SIGGRAPH 90)*, 24(4):197–206, August 1990. ISBN 0-201-50933-4. Held in Dallas, Texas.

[17] Michael P. Salisbury, Sean E. Anderson, Ronen Barzel, and David H. Salesin. Interactive Pen-And-Ink Illustration. *Proceedings of SIGGRAPH 94*, pages 101–108, July 1994. ISBN 0-89791-667-0. Held in Orlando, Florida.

[18] Georges Winkenbach and David H. Salesin. Rendering Parametric Surfaces in Pen and Ink. *Proceedings of SIGGRAPH 96*, pages 469–476, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

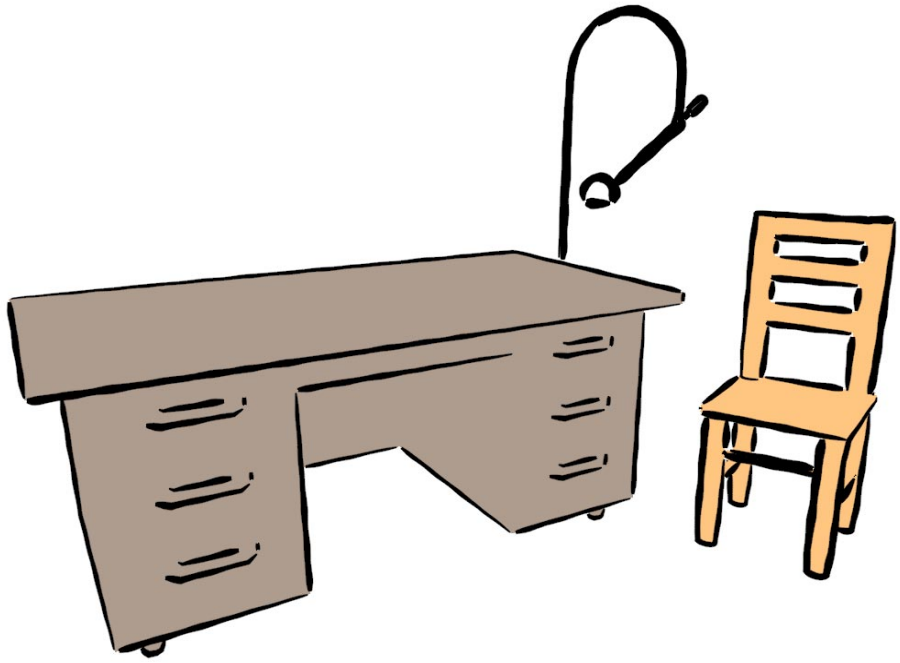


Figure 11 A simple office scene.

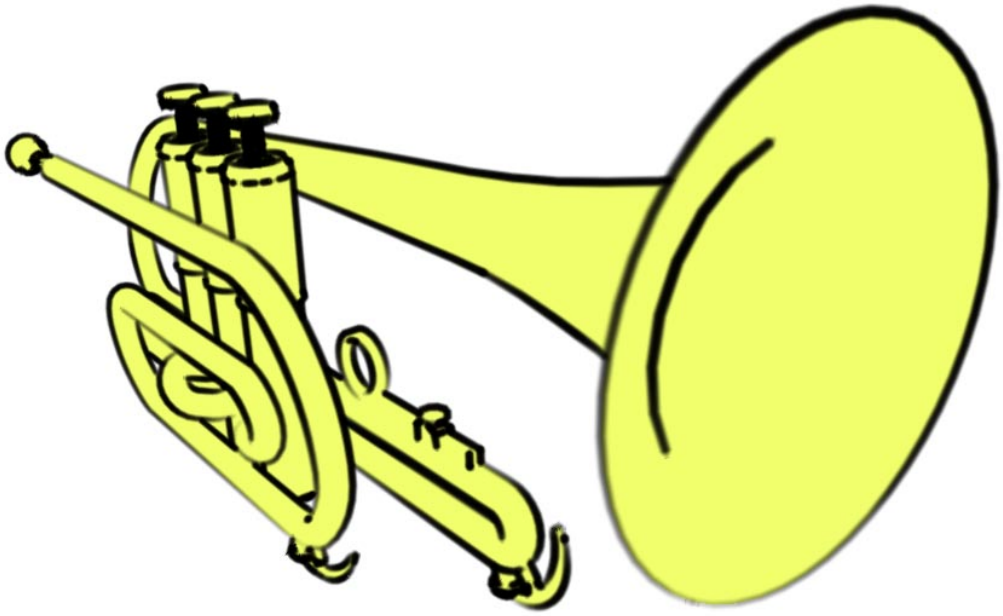


Figure 12 This trumpet demonstrates the use of texture-mapping to create soft strokes.